

CAMINHO MAIS CURTO ENTRE PONTOS

Luan Gerber Siiss ¹
Lucas Gilmar da Silva ²
Fernando Andrade Bastos ³

RESUMO

Este artigo investiga a aplicação do algoritmo de Dijkstra para otimização de rotas rodoviárias entre os municípios do Alto Vale do Itajaí, em Santa Catarina. Utilizando a Teoria dos Grafos, o estudo modela a rede de transporte da região, onde os municípios são representados como vértices e as estradas como arestas ponderadas. Os pesos das arestas são definidos com base em fatores como distância, tipo de pavimentação e intensidade do tráfego, fornecendo um modelo realista da infraestrutura local. A implementação do algoritmo foi realizada em duas linguagens de programação: Python, com o suporte da biblioteca NetworkX, e JavaScript, para desenvolvimento de uma interface web interativa. Essa abordagem permite não apenas demonstrar a flexibilidade e adaptabilidade do algoritmo em diferentes ambientes de programação, mas também oferece uma solução prática para usuários que desejam calcular rotas de menor custo em tempo real. O processo de desenvolvimento incluiu a construção de matrizes de adjacência e de custo, que foram fundamentais para mapear as conexões entre as cidades e ajustar os parâmetros necessários para o cálculo preciso dos caminhos mais curtos. O artigo também discute os desafios enfrentados durante a implementação, como o tratamento de grandes volumes de dados e a adaptação do algoritmo para diferentes cenários de uso. Ao final, é apresentada uma análise comparativa entre as implementações em Python e JavaScript, destacando a eficácia do algoritmo de Dijkstra em solucionar problemas de roteamento em redes de transporte e comunicação. Assim, este trabalho contribui tanto para o avanço técnico no desenvolvimento de sistemas de otimização de rotas quanto para o aprendizado e compartilhamento da experiência aprendida e desenvolvida no processo.

Palavras-Chave: Algoritmo Dijkstra. Sistemas de informação. Estrutura de dados. Grafos.

¹ Acadêmico em Sistemas de Informação pelo Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí - UNIDAVI. E-mail: luan.siiss@unidavi.edu.br

² Acadêmico em Sistemas de Informação pelo Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí - UNIDAVI. E-mail: lucas.gilmar@unidavi.edu.br

³ Mestre em Sistemas de Informação pela Universidade Federal de Santa Catarina - UFSC. Docente do Centro Universitário para Desenvolvimento do Alto Vale do Itajaí - UNIDAVI. Lattes: <http://lattes.cnpq.br/8854684844938218>, e-mail: fbastos@unidavi.edu.br.

1. INTRODUÇÃO

A conectividade entre municípios e a eficiência no planejamento de rotas são elementos críticos para o desenvolvimento regional, especialmente em regiões com grande dependência de infraestrutura rodoviária, como o Alto Vale do Itajaí, em Santa Catarina. A Teoria dos Grafos oferece um modelo eficaz para representar essas redes de transporte, onde os municípios são representados como vértices e as estradas como arestas. Dentre os algoritmos utilizados para otimização de rotas, o algoritmo de Dijkstra destaca-se por sua capacidade de encontrar o menor caminho em grafos ponderados, sendo amplamente aplicado em problemas de logística, redes de comunicação e transporte.

Neste artigo, será apresentado o desenvolvimento de um protótipo que implementa o algoritmo de Dijkstra para calcular o menor caminho entre cidades da região do Alto Vale do Itajaí. Foi utilizada duas abordagens de implementação: uma em Python, utilizando a biblioteca NetworkX, e outra em JavaScript, para explorar a flexibilidade da solução em diferentes ambientes de programação.

O objetivo é avaliar a eficácia dessas tecnologias na resolução de problemas de roteamento e apresentar uma aplicação prática que possa ser adaptada a outros cenários. A análise das distâncias, condições de pavimentação e intensidade de tráfego foi essencial para ajustar os parâmetros do algoritmo e garantir a precisão dos cálculos.

2. REVISÃO DA LITERATURA

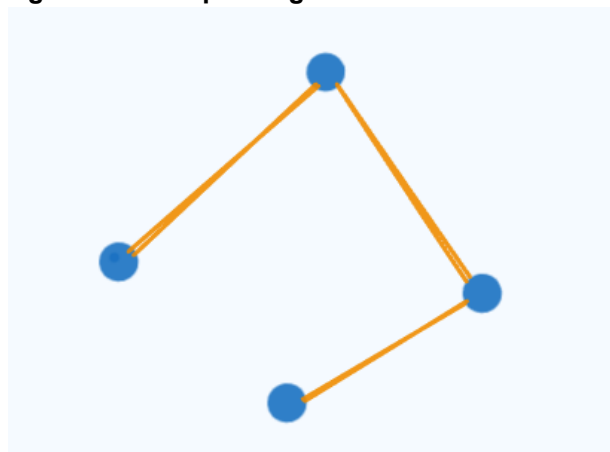
Neste capítulo, são apresentados os conceitos relacionados às ferramentas e linguagens utilizadas no desenvolvimento do protótipo.

2.1 TEORIA DOS GRAFOS

A Teoria dos Grafos tem uma origem relativamente recente na história da Matemática. Embora suas primeiras ideias tenham surgido no século XVIII, foi apenas no século XX que sua definição formal foi consolidada. A teoria desempenha um papel crucial devido a suas inúmeras aplicações no mundo moderno, bem como em diversas áreas da Matemática.

De forma simples, um grafo pode ser descrito como um conjunto de vértices (ou nós) e arestas (ou ligações), onde cada aresta conecta um par de vértices. Os grafos são frequentemente representados visualmente, com vértices como pontos em um plano e arestas como linhas que os conectam. Essa representação facilita a compreensão de relações e estruturas em diversos contextos.

Figura 1 - Exemplo de grafo



Fonte: Acervo do autor (2024).

O estudo dos grafos teve seu marco inicial com o trabalho do matemático suíço Leonhard Euler, que em 1736 solucionou o famoso problema das pontes de Königsberg. Esse evento é considerado o ponto de partida para a formalização da Teoria dos Grafos. Desde então, ela evoluiu significativamente e se tornou uma ferramenta essencial para modelar e resolver problemas complexos em várias áreas do conhecimento.

Os grafos podem ser classificados de diferentes formas, de acordo com suas características: grafos direcionados, onde as arestas possuem uma direção; grafos não direcionados, em que as conexões entre os vértices são bidirecionais; grafos ponderados, nos quais as arestas possuem valores associados; e grafos bipartidos, que possuem vértices divididos em dois subconjuntos distintos, entre outras classificações.

Diversos algoritmos foram desenvolvidos para explorar e analisar grafos. O algoritmo de Dijkstra, por exemplo, é amplamente utilizado para encontrar o caminho mais curto entre dois vértices em um grafo ponderado, sendo aplicado em áreas como redes de comunicação e planejamento de rotas. Já o algoritmo de Kruskal é um método eficiente para encontrar árvores geradoras mínimas, uma estrutura crucial para resolver problemas de otimização.

2.2 MATRIZ DE ADJACÊNCIAS

A matriz de adjacências é uma das representações mais comuns utilizadas na Teoria dos Grafos para descrever a relação entre vértices de um grafo. Trata-se de uma matriz quadrada em que as linhas e colunas representam os vértices do grafo, e os valores em suas células indicam a presença ou ausência de arestas entre esses vértices. Essa abordagem é amplamente utilizada devido à sua capacidade de armazenar informações de forma direta e acessível.

Dada uma matriz de adjacências **A** de um grafo **G** com **n** vértices, os elementos **A[i][j]** da matriz indicam se existe uma aresta entre os vértices **i** e **j**. No caso de grafos não direcionados, a matriz é simétrica, ou seja, **A[i][j] = A[j][i]**. Em grafos direcionados, no entanto, **A[i][j]** pode ser diferente de **A[j][i]**, indicando que as arestas têm direções específicas.

A matriz de adjacências pode ser usada tanto para grafos não ponderados, onde cada célula contém 0 ou 1 (indicando a ausência ou presença de uma aresta), quanto para grafos ponderados, onde as células contêm o peso da aresta que conecta os dois vértices. Essa flexibilidade permite a aplicação da matriz de adjacências em diferentes cenários, como redes de transporte, redes sociais e modelagem de circuitos.

Em resumo, a matriz de adjacências é uma ferramenta fundamental na Teoria dos Grafos e continua a ser uma representação eficaz para certos tipos de grafos, especialmente quando se trabalha com grafos densos ou se necessita de operações rápidas para verificar a existência de conexões entre vértices.

2.3 LISTA DE ADJACÊNCIAS

As listas de adjacências são uma das representações mais eficientes para descrever grafos, especialmente quando se trata de grafos esparsos, ou seja, aqueles com um número relativamente pequeno de arestas em comparação ao número de vértices.

Em uma lista de adjacências, cada vértice é associado a uma lista que contém todos os outros vértices aos quais está conectado por arestas. Essa abordagem permite que a informação sobre as arestas seja armazenada de maneira compacta. Por exemplo, em um grafo com n vértices, a lista de adjacências pode ser representada como um vetor de listas, onde cada posição do vetor corresponde a um vértice e cada lista contém os vértices adjacentes. São amplamente utilizadas em algoritmos relacionados à Teoria dos Grafos.

2.4 ALGORITMO DIJKSTRA

O algoritmo de Dijkstra, desenvolvido pelo cientista da computação holandês Edsger Dijkstra em 1956 e publicado em 1959, é um método fundamental para resolver o problema do caminho mais curto em grafos, sejam eles dirigidos ou não, com arestas de pesos não negativos. Este algoritmo desempenha um papel crucial em diversas aplicações práticas, desde roteamento em redes de computadores até navegação em sistemas de mapas.

2.4.1 Complexidade e Funcionamento

A complexidade ciclomática do algoritmo é de $O((m + n) \log n)$, onde m representa o número de arestas e n o número de vértices do grafo. O algoritmo é projetado para determinar o caminho mais curto até todos os outros vértices a partir de um vértice de origem específico. Também é possível adaptar o algoritmo para encontrar o caminho mais curto entre dois vértices específicos.

2.4.2 Exemplo Prático

Uma aplicação prática do algoritmo de Dijkstra é observada em cenários de navegação, como quando um viajante precisa determinar a rota mais eficiente entre cidades. Nesse contexto, as cidades são representadas como vértices e as estradas como arestas do grafo. Nesse cenário, o algoritmo possibilita identificar a rota mais curta.

2.4.3 Etapas do Algoritmo

O funcionamento do algoritmo pode ser descrito em três etapas principais:

2.4.3.1 Inicialização

- Cria-se uma lista de todos os vértices não visitados (abertos) do grafo.
- Estabelece-se uma lista que guarda as distâncias entre o vértice de origem e todos os outros vértices, onde todas as distâncias são inicialmente definidas como infinito, exceto a do vértice de origem, que é fixada em 0.
- Uma lista de predecessores é criada, na qual todos os predecessores são inicialmente nulos.

2.4.3.2 Execução

- Enquanto houver vértices não visitados, escolhe-se o vértice **U** com a menor distância estimada.
- Para cada vértice **V** adjacente a **U**, a distância é atualizada somando a distância até **U** com o custo da aresta que conecta **U** e **V**.
- Se a nova distância até **V** for menor do que a distância previamente registrada, essa distância é atualizada e o predecessor de **V** é definido como **U**. Esse processo é conhecido como o relaxamento das arestas.

2.4.3.3 Conclusão

- Quando todos os vértices tiverem sido visitados, o algoritmo terá determinado a menor distância até o vértice de destino. A partir da lista de predecessores, é possível reconstruir o caminho mais curto.

Em síntese, o algoritmo de Dijkstra é uma ferramenta poderosa e eficiente para a determinação de caminhos em grafos, sendo amplamente utilizado em diversas áreas, como sistemas de informação, logística e planejamento de rotas.

2.5 HTML

HTML juntamente com CSS é fundamental para a estruturação e apresentação de conteúdos na web.

De acordo com a MDN, o HTML é a base da web, definindo a estrutura e o significado do conteúdo. Enquanto HTML organiza o conteúdo, CSS cuida da aparência e JavaScript da funcionalidade das páginas. O "hipertexto" refere-se aos links que conectam diferentes páginas, permitindo a navegação e interação entre sites na web. Ao contrário de linguagens de programação como JavaScript, HTML não possui lógica de programação, sendo utilizado exclusivamente para descrever o layout e a estrutura dos documentos web.

2.6 JAVASCRIPT

JavaScript é uma das três principais ferramentas para desenvolvimento web, junto com HTML e CSS, sendo amplamente utilizada em praticamente todos os sites modernos.

MDN descreve o JavaScript como uma linguagem leve, interpretada e orientada a objetos, amplamente conhecida por seu uso em páginas web, mas também aplicada em ambientes como Node.js e Adobe Acrobat. Baseada em protótipos, JavaScript suporta múltiplos paradigmas, incluindo orientação a objetos, imperativo e funcional. O padrão da linguagem é o ECMAScript, com suporte nos navegadores modernos. Desde 2015, versões do ECMAScript são lançadas anualmente, sendo a ECMAScript 2024 a mais recente.

JavaScript possui várias características, sendo uma das mais notáveis o fato de ser uma linguagem dinamicamente e fracamente tipada. Isso significa que não há uma definição fixa para o tipo de dados de variáveis ou constantes, permitindo que seus tipos sejam atribuídos e modificados em tempo de execução, além de serem facilmente convertidos. Outro ponto importante é o modelo de herança baseado em protótipos, que difere da herança clássica baseada em classes, comum em outras linguagens.

2.7 PYTHON

Python é uma das linguagens de programação mais populares e versáteis do mundo. Desde seu lançamento em 1991 por Guido van Rossum, ela se destacou por sua simplicidade, clareza e ênfase na legibilidade do código. Essas características tornam Python ideal tanto para iniciantes, que buscam aprender programação sem enfrentar a complexidade de sintaxes rígidas, quanto para desenvolvedores experientes, que desejam criar soluções robustas de maneira ágil e eficiente.

Uma das grandes vantagens de Python é sua flexibilidade e aplicabilidade em diversos domínios. Ela é amplamente utilizada em áreas como desenvolvimento web, automação de tarefas, análise de dados, ciência de dados, inteligência artificial e aprendizado de máquina. Sua sintaxe intuitiva permite que os programadores se concentrem na solução de problemas, em vez de se preocuparem com a estruturação do código, facilitando o desenvolvimento de projetos de todos os tamanhos e complexidades.

Além disso, Python é amplamente adotada por sua enorme coleção de bibliotecas que estendem suas funcionalidades, permitindo que desenvolvedores lidem com tarefas complexas com facilidade.

2.7.1 NetworkX Python

O grande diferencial de Python está justamente em seu ecossistema de bibliotecas, que proporciona soluções prontas para diversas necessidades. Essas bibliotecas são coleções de módulos pré-desenvolvidos, facilitando e acelerando o desenvolvimento de aplicações especializadas. Por exemplo, a biblioteca “Pandas” é amplamente utilizada para análise de dados, “Matplotlib” para visualização gráfica, e “TensorFlow” se destaca em projetos de aprendizado de máquina.

No código apresentado, foi utilizada a biblioteca “NetworkX”, uma ferramenta poderosa para a criação, manipulação e análise de grafos e redes complexas.

NetworkX foi empregada para modelar um mapa de cidades, permitindo calcular o caminho mais curto entre duas localidades. Esse exemplo destaca como o suporte de bibliotecas especializadas em Python, como o NetworkX, pode resolver problemas computacionais avançados de forma eficiente e acessível. A combinação entre a simplicidade da linguagem e a robustez de suas bibliotecas faz de Python uma escolha preferida para desenvolvedores em diversas áreas, do desenvolvimento de aplicações até a pesquisa acadêmica.

3. METODOLOGIA DA PESQUISA

Este artigo caracteriza-se como uma pesquisa aplicada e descritiva, pois aborda o desenvolvimento e a aplicação do algoritmo de Dijkstra para otimização de rotas em um cenário real, envolvendo cidades do Alto Vale do Itajaí, em Santa Catarina.

Após uma revisão bibliográfica abrangente sobre Teoria dos Grafos e algoritmos de caminhos mínimos, foi decidido que o protótipo seria desenvolvido na IDE Visual Studio, utilizando duas linguagens de programação: JavaScript e Python. Essas linguagens foram selecionadas devido à sua flexibilidade e ao suporte oferecido por bibliotecas especializadas, como o NetworkX em Python e as estruturas nativas de manipulação de dados em JavaScript. O uso dessas ferramentas visa não apenas garantir a robustez e a eficiência da aplicação, mas também ampliar seu escopo de estudo, ao possibilitar comparações entre diferentes linguagens e ambientes de implementação.

Para assegurar a relevância do protótipo, foi conduzida uma análise detalhada de cenários reais, selecionando um conjunto representativo de cidades da região. As distâncias e as condições das estradas foram modeladas em uma matriz de adjacência e uma matriz de custo, levando em consideração fatores como pavimentação e intensidade de tráfego. Esse processo de mapeamento foi essencial para ajustar os parâmetros do algoritmo e garantir a precisão dos cálculos realizados.

A validação da solução se deu por meio da implementação de um sistema interativo, no qual o usuário pode inserir as cidades de origem e destino, com o menor caminho sendo calculado em tempo real pelo algoritmo de Dijkstra. O protótipo final apresenta a rota ideal para o usuário.

4. DESENVOLVIMENTO DO PROTÓTIPO

Neste capítulo são apresentados todos os aspectos técnicos do processo de análise e implementação do protótipo.

4.1 CENÁRIO

Algumas cidades do Alto Vale do Itajaí, em Santa Catarina, foram selecionadas para formar um mapa rodoviário, com o objetivo de determinar o menor caminho entre duas cidades específicas.

4.2 MATRIZ DE ADJACÊNCIA E MATRIZ DE CUSTO

Neste artigo, foram desenvolvidos uma matriz de adjacência e também uma matriz de custo com todas as cidades selecionadas para a documentação e cálculo das distâncias e custos.

4.2.1 Lista de Cidades

Segue uma lista das cidades usadas no mapa rodoviário e suas abreviações para uso nas matrizes e implementação no protótipo.

Quadro 1 - Lista de cidades

Cidade	Abreviação	Cidade	Abreviação
Salete	SA	Chapadão do Lageado	CL
Witmarsum	WI	Aurora	AU
Presidente Getúlio	PG	Agrolândia	A
Dona Emma	DE	Trombudo Central	TC
José Boiteux	JB	Agronômica	AG
Ibirama	IB	Pouso Redondo	PR
Lontras	LO	Braço do Trombudo	BT
Presidente Nereu	PN	Laurentino	LA
Vidal Ramos	VR	Rio do Oeste	RO
Imbuia	IM	Taió	TA
Ituporanga	IT	Mirim Doce	MD
Petrolândia	PE	Rio do Sul	RS

Fonte: Acervo do autor (2024)

4.2.2 Matriz de Adjacências

Essa matriz de adjacências tem o objetivo de representar as arestas do grafo, usado o google planilhas para desenvolvimento.

Quadro 2 - Matriz de Adjacências

	SA	WI	PG	DE	JB	IB	LO	PN	VR	IM	IT	PE	CL	AU	AL	TC	AG	PR	BT	LA	RO	TA	MD	RS
SA		1		1																		1		
WI	1		1	1	1																			
PG		1		1	1	1																		1
DE	1	1	1		1																	1		
JB		1	1	1		1																		
IB			1		1		1																	
LO						1		1																1
PN							1		1															
VR								1		1	1													
IM									1		1													
IT									1	1		1	1	1	1									
PE										1		1												
CL										1	1													
AU										1														1
AL										1														
TC															1	1	1	1	1					
AG															1	1		1		1				1
PR															1	1		1				1	1	
BT															1		1		1					
LA																	1				1			1
RO																				1	1	1		
TA	1			1														1				1	1	
MD																		1				1		
RS			1				1							1			1			1				

Fonte: Acervo do autor (2024)

4.2.3 Matriz de Custo

Essa matriz de custos foi projetada para representar os custos das arestas do grafo, utilizando o Google Planilhas para o desenvolvimento. Os valores de custo nesta matriz não correspondem exatamente à realidade, sendo calculados com a fórmula(fornecida pelo docente Fernando Andrade Bastos):

$f(x) = \text{distância} * \text{pavimentação} * \text{tráfego}$, onde:

- distância = valor em km do ponto X ao ponto Y;
- pavimentação = 1 para asfalto, 1,5 para lajota/paralelepípedo, 2,2 para outros tipos de pavimentação;
- tráfego = 1 para pouca movimentação, 2 para tráfego normal, e 3 para tráfego intenso.

Os valores dessas variáveis foram atribuídos de forma aleatória, resultando nos custos apresentados abaixo.

Quadro 3 - Matriz de custos

	SA	WI	PG	DE	JB	IB	LO	PN	VR	IM	IT	PE	CL	AU	AL	TC	AG	PR	BT	LA	RO	TA	MD	RS
SA		54,34		75,46																		19,01		
WI	54,34		28,5	20,5	55,88																			
PG		28,5		24,15	35,64	11,7																	27,4	
DE	75,46	20,5	24,15		39,38																	90,64		
JB		55,88	35,64	39,38		20,5																		
IB			11,7		20,5		34,8																	
LO						34,8		29,3															25,6	
PN							29,3		21,01															
VR								21,01		19,8	31,6													
IM									19,8		38,4													
IT									31,6	38,4		18,8	26,8	13,3	62,04									
PE											18,8		55,44											
CL											26,8	55,44												
AU											13,3												26,4	
AL											62,04													
TC															14,1									
AG															14,1	17,4	55,2	14,4						
PR															17,4		75,03		8,9				10,7	
BT															55,2	75,03		80,96				19,4	37,4	
LA															14,4		80,96							
RO																	8,9				16,6		13,01	
TA	19,01			90,64															19,4		71,06		56,76	
MD																		37,4				56,76		
RS			27,4				25,6							26,4			10,7			13,01				

Fonte: Acervo do autor (2024)

4.3 ANÁLISE E LEVANTAMENTO DOS REQUISITOS

O protótipo tem como objetivo encontrar o caminho de menor custo entre dois pontos quaisquer em um mapa rodoviário. Para isso, foi adotado o formato de grafos, onde as estradas são representadas por arestas e as cidades por vértices. O protótipo, então, deve calcular o menor percurso entre essas localidades.

Com base no cenário e orientações do docente do curso, foram definidos alguns requisitos:

- O protótipo deve de um campo onde será informado a cidade de origem e destino.
- O protótipo deve calcular, com base nos vértices e arestas disponíveis, qual é o menor caminho entre a cidade de origem e de destino.
- O protótipo deve usar o algoritmo Dijkstra para cálculo do menor caminho.
- O protótipo deve mostrar a rota final para o usuário
- A rota final precisa ser exibida como um caminho, onde terá as cidades e uma ligação entre elas, indicando a direção que se deve seguir.

4.4 IMPLEMENTAÇÃO

Nesse trecho, será apresentado duas implementações do sistema, uma feita com JavaScript e HTML e a outra com Python. Dessa forma, há mais flexibilidade para quem um dia quiser também implementar o software com algoritmo Dijkstra.

4.4.1 Desenvolvimento em JavaScript

Aqui temos um protótipo desenvolvido usando principalmente JavaScript para toda a lógica e interatividade, além de HTML para usabilidade.

4.4.1.1 Aplicação

Primeiramente, foi definido um array com todos os vértices. Conforme mostrado na figura 2.

Figura 2 - Declaração do array de vértices

```
const vertices = [  
  'SA', 'WI', 'PG', 'DE', 'JB', 'IB', 'LO', 'PN', 'VR',  
  'IM', 'IT', 'PE', 'CL', 'AU', 'AL', 'TC', 'AG', 'PR',  
  'BT', 'LA', 'RO', 'TA', 'MD', 'RS'  
]
```

Fonte: Acervo do autor (2024)

Em seguida, conforme a figura 3, foi criado um objeto que armazena os vértices adjacentes e seus respectivos custos, essa é a lista de adjacência (com valor).

Figura 3 - Declaração do objeto no JavaScript que representa a lista de adjacências

```
const distancias = {  
  'SA': { 'WI': 54.34, 'DE': 75.46, 'TA': 19.01 }, // Salete  
  'WI': { 'SA': 54.34, 'PG': 28.50, 'DE': 20.50, 'JB': 55.88 }, // Witmarsum  
  'PG': { 'WI': 28.50, 'DE': 24.15, 'JB': 35.64, 'IB': 11.70, 'RS': 27.40 }, // Presidente Getúlio  
  'DE': { 'SA': 75.46, 'WI': 20.50, 'PG': 24.15, 'JB': 39.38, 'TA': 90.64 }, // Dona Emma  
  'JB': { 'WI': 55.88, 'PG': 35.64, 'DE': 39.38, 'IB': 20.50 }, // José Boiteux  
  'IB': { 'PG': 11.70, 'JB': 20.50, 'LO': 34.80 }, // Ibirama  
  'LO': { 'IB': 34.80, 'PN': 29.30, 'RS': 25.60 }, // Lontras  
  'PN': { 'LO': 29.30, 'VR': 21.01 }, // Presidente Nereu  
  'VR': { 'PN': 21.01, 'IM': 19.80, 'IT': 31.60 }, // Vidal Ramos  
  'IM': { 'VR': 19.80, 'IT': 38.40 }, // Imbuia  
  'IT': { 'VR': 31.60, 'IM': 38.40, 'PE': 18.80, 'CL': 26.80, 'AU': 13.30, 'AL': 62.04 }, // Ituporanga  
  'PE': { 'IT': 18.80, 'CL': 55.44 }, // Petrolândia  
  'CL': { 'IT': 26.80, 'PE': 55.44 }, // Chapadão do Lageado  
  'AU': { 'IT': 13.30, 'RS': 26.40 }, // Aurora  
  'AL': { 'IT': 62.04, 'TC': 14.10 }, // Agrolândia  
  'TC': { 'AL': 14.10, 'AG': 17.40, 'PR': 55.20, 'BT': 14.40 }, // Trombudo Central  
  'AG': { 'TC': 17.40, 'PR': 75.03, 'LA': 8.90, 'RS': 10.70 }, // Agronômica  
  'PR': { 'TC': 55.20, 'AG': 75.03, 'BT': 80.96, 'TA': 19.40, 'MD': 37.40 }, // Pouso Redondo  
  'BT': { 'TC': 14.40, 'PR': 80.96 }, // Braço do Trombudo  
  'LA': { 'AG': 8.90, 'RO': 16.60, 'RS': 13.01 }, // Laurentino  
  'RO': { 'LA': 16.60, 'TA': 71.06 }, // Rio do Oeste  
  'TA': { 'SA': 19.01, 'DE': 90.64, 'PR': 19.40, 'RO': 71.06, 'MD': 56.76 }, // Taió  
  'MD': { 'PR': 37.40, 'TA': 56.76 }, // Mirim Doce  
  'RS': { 'PG': 27.40, 'LO': 25.60, 'AU': 26.40, 'AG': 10.7, 'LA': 13.01 }, // Rio do Sul  
}
```

Fonte: Acervo do autor (2024)

Também, conforme a figura 4, foi criado um objeto que associa cada vértice à sua respectiva cidade, para depois ser possível mostrar os nomes das cidades.

Figura 4 - Declaração do objeto no JavaScript que armazena as abreviaturas das cidades

```
const cidadesExtenso = {  
  'SA': 'Salette',  
  'WI': 'Witmarsum',  
  'PG': 'Presidente Getúlio',  
  'DE': 'Dona Emma',  
  'JB': 'José Boiteux',  
  'IB': 'Ibirama',  
  'LO': 'Lontras',  
  'PN': 'Presidente Nereu',  
  'VR': 'Vidal Ramos',  
  'IM': 'Imbuia',  
  'IT': 'Ituporanga',  
  'PE': 'Petrolândia',  
  'CL': 'Chapadão do Lageado',  
  'AU': 'Aurora',  
  'AL': 'Agrolândia',  
  'TC': 'Trombudo Central',  
  'AG': 'Agronômica',  
  'PR': 'Pouso Redondo',  
  'BT': 'Braço do Trombudo',  
  'LA': 'Laurentino',  
  'RO': 'Rio do Oeste',  
  'TA': 'Taió',  
  'MD': 'Mirim Doce',  
  'RS': 'Rio do Sul',  
}
```

Fonte: Acervo do autor

A seguir, foi construído um método responsável por calcular os predecessores dos vértices, bem como registrar os vértices visitados e a soma de seus pesos. Dentro desse método, foi utilizado um loop while, que continua até que todos os vértices tenham sido visitados ou até que o vértice final seja encontrado. Durante essas iterações, o algoritmo identifica o vértice não visitado com o menor valor, calcula a distância estimada para os vértices adjacentes e, caso essa distância seja menor do que a distância atual, atualiza o valor com a nova distância. Finalmente, o vértice é removido da lista de não visitados. Conforme mostrado na figura 5.

Figura 5 - função implementando todo o algoritmo Dijkstra em JavaScript

```
function encontrarRota(atual, fim) {
  // cria um objeto e deixa todos com valor infinito
  const naoVisitado = {};
  vertices.forEach(nos => {
    naoVisitado[nos] = Infinity
  })

  let atualDistancia = 0
  naoVisitado[atual] = atualDistancia
  const visitado = {}
  const predecessores = {}
  // Loop vai se repetir enquanto estiver nós não visitados
  while (Object.keys(naoVisitado).length > 0) {
    //acha o nó não visitado com menor distancia estimada
    let minVertex = Object.keys(naoVisitado).reduce((a, b) =>
      naoVisitado[a] < naoVisitado[b] ? a : b
    )
    // Olha os vizinhos e decido o proximo com menor valor; Está fazendo o "Relaxamento"
    for (const [vizinho, distancia] of Object.entries(distancias[minVertex])) {
      if (!(vizinho in naoVisitado)) {
        continue
      }
      const novaDistancia = naoVisitado[minVertex] + distancia
      if (naoVisitado[vizinho] === Infinity || naoVisitado[vizinho] > novaDistancia) {
        naoVisitado[vizinho] = novaDistancia
        predecessores[vizinho] = minVertex // Guardando os predecessores
      }
    }
    // Remove o elemento dos naovisitado(colocando como "fechado") e também atualiza a distancia estimada
    visitado[minVertex] = naoVisitado[minVertex]
    delete naoVisitado[minVertex]

    if (minVertex === fim) {
      break
    }
  }
  return {predecessores, visitado}
}
```

Fonte: Acervo do autor (2024)

Continuando, foi construído outro método para gerar o caminho mais curto com base no ponto de início, no ponto final e no objeto de predecessores. Conforme mencionado anteriormente, com os predecessores definidos, basta traçar o caminho inverso, indo do ponto final ao ponto inicial. Conforme mostrado na figura 6.

Figura 6 - Geração do caminho e armazenamento no formato de string

```
function geracaoCaminho(predecessores, inicio, fim) {  
    const caminho = [fim]  
  
    while (true) {  
        const key = predecessores[caminho[0]]  
        caminho.unshift(key)  
        if (key === inicio) {  
            break  
        }  
    }  
  
    const caminhoCidades = caminho.map( e => cidadesExtenso[e] )  
    return caminhoCidades.join(' -> ' )  
}
```

Fonte: Acervo do autor (2024)

Depois, conforme figura a 7, foi criado um método para obter as cidades de origem e destino do documento HTML, validar os vértices, chamar a função encontrarRota() e atualizar o documento HTML para exibir o resultado ao usuário.

Figura 7 - Função principal para pegar os inputs e chamar as outras funções.

```
function executarCodigo() {  
    // define o inicio e fim com base nos inputs da pagina  
    const inicio = document.getElementById('cid_origem').value  
    const fim = document.getElementById('cid_destino').value  
  
    const resultado = document.getElementById('melhor_rota')  
  
    if (inicio == fim) {  
        resultado.innerHTML = `O caminho mais curto: ${cidadesExtenso[inicio]}`  
    } else {  
        const {predecessores, visitado} = encontrarRota(inicio, fim)  
  
        console.log(predecessores)  
        console.log(visitado)  
  
        const caminhoFinal = geracaoCaminho(predecessores, inicio, fim)  
        resultado.innerHTML = `O caminho mais curto: ${caminhoFinal}` // Para mostrar no site  
        console.log(`O caminho mais curto: ${caminhoFinal}`)  
    }  
}
```

Fonte: Acervo do autor (2024)

Também foi desenvolvido um documento HTML contendo os campos de entrada necessários para o funcionamento do algoritmo em JavaScript, e posteriormente, exibir o caminho mais curto ao usuário. Conforme mostrado na figura 8.

Figura 8 - Código HTML para definir a estrutura da página e chamar os scripts.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>Sistema para calculo do caminho mais curto entre dois pontos</h1>
  <form action="desenvolvendo.js">
    <p>
      <label for="cid_origem">Cidade de Origem</label>
      <select name="cid_origem" id="cid_origem">
      </select>
    </p>
    <p>
      <label for="cid_destino">Cidade de Destino</label>
      <select name="cid_destino" id="cid_destino">
      </select>
    </p>
    <button type="button" onclick="executarCodigo()">Calcular Rota</button>
  </form>
  <p id="melhor_rota">
  </p>
</body>
<script src="desenvolvendo.js"></script>
```

Fonte: Acervo do autor (2024)

Por fim, conforme a figura 9, dentro do HTML, também foi incluído um pequeno script em JavaScript para construir as opções (option) do elemento select, com base no objeto cidadesExtenso. C

Figura 9 - Script em JavaScript para criar os elementos “option” do “select” das cidades de origem e destino.

```
<script>
  //Criação das opções de cidades no PAGINA HTML com base no objeto cidadesExtenso

  // Obtém a referência ao elemento <select>
  const selectOrigem = document.getElementById('cid_origem')
  const selectDestino = document.getElementById('cid_destino')

  // Cria um array de pares chave-valor e ordena pelo valor
  const cidadesOrdenadas = Object.entries(cidadesExtenso).sort((a, b) => a[1].localeCompare(b[1]))

  // funcao para criar as opcoes
  const criaOp = (key, value, select) => {
    const option = document.createElement('option')
    option.value = key
    option.textContent = value
    select.appendChild(option)
  }

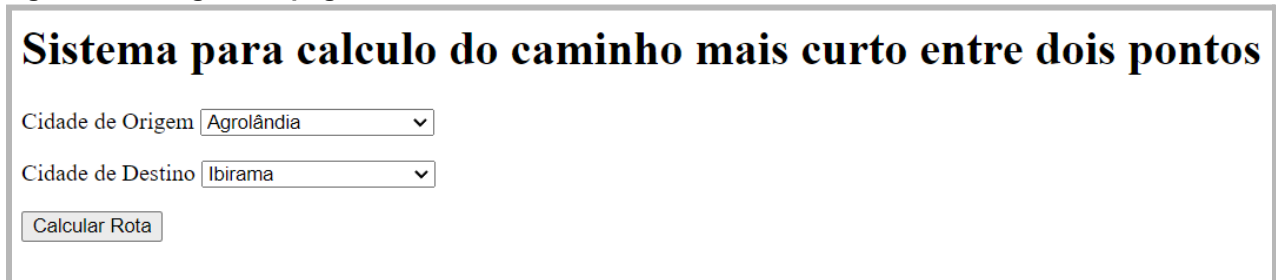
  // Adiciona as opções ao <select>
  cidadesOrdenadas.forEach(([key, value]) => {
    criaOp(key, value, selectOrigem) // opcoes de origem
    criaOp(key, value, selectDestino) // opcoes de destino
  });
</script>
```

Fonte: Acervo do autor (2024)

4.4.1.2 Resultado Final

Imagem de como fica a página ao abrir-lá.

Figura 10 - Imagem da página no chrome, selecionando as cidades.



Sistema para calculo do caminho mais curto entre dois pontos

Cidade de Origem

Cidade de Destino

Fonte: Acervo do autor (2024)

Ao clicar no botão “calcular Rota”, o protótipo rota e mostra na tela a rota de menor custo.

Figura 11 - Imagem do resultado do caminho mais curto depois de clicar no botão.



Sistema para calculo do caminho mais curto entre dois pon

Cidade de Origem

Cidade de Destino

O caminho mais curto: Agrolândia -> Trombudo Central -> Agronômica -> Rio do Sul -> Presidente Getúlio -> Ibirama

Fonte: Acervo do autor (2024)

4.4.2 Desenvolvimento em Python

Neste protótipo, a lógica e a interatividade foram desenvolvidas principalmente em Python, utilizando suas bibliotecas e recursos nativos para processamento e manipulação de dados de forma eficiente e estruturada.

4.4.2.1 Aplicação

O módulo NetworkX foi importado para manipulação de grafos. O dicionário `idades_extenso` associa códigos de cidades aos seus nomes completos, enquanto `idades_abreviado` faz o mapeamento inverso, convertendo nomes de cidades para códigos em maiúsculas para garantir consistência, conforme a figura 12.

Figura 12 - Uso de Networkx e mapeamento bidirecional de códigos e nomes de cidades.

```
import networkx as nx

cidades_extenso = {
    'SA': 'Salette',
    'WI': 'Witmarsum',
    'PG': 'Presidente Getúlio',
    'DE': 'Dona Emma',
    'JB': 'José Boiteux',
    'IB': 'Ibirama',
    'LO': 'Lontras',
    'PN': 'Presidente Nereu',
    'VR': 'Vidal Ramos',
    'IM': 'Imbuia',
    'IT': 'Ituporanga',
    'PE': 'Petrolândia',
    'CL': 'Chapadão do Lageado',
    'AU': 'Aurora',
    'AL': 'Agrolândia',
    'TC': 'Trombudo Central',
    'AG': 'Agronômica',
    'PR': 'Pouso Redondo',
    'BT': 'Braço do Trombudo',
    'LA': 'Laurentino',
    'RO': 'Rio do Oeste',
    'TA': 'Taió',
    'MD': 'Mirim Doce',
    'RS': 'Rio do Sul'
}

cidades_abreviado = {v.upper(): k for k, v in cidades_extenso.items()}
```

Fonte: Acervo do autor (2024)

Um grafo não direcionado G foi criado com NetworkX. A lista edges contém arestas, onde cada aresta é uma tupla (origem, destino, peso), com o peso representando a distância entre as cidades. G.add_weighted_edges_from(edges) adiciona essas arestas ao grafo, incorporando as distâncias como pesos, conforme a figura 13.

Figura 13 - Criação de um grafo não direcionado com Networkx e arestas ponderadas.

```
G = nx.Graph()

edges = [
    ('SA', 'WI', 54.34), ('SA', 'DE', 75.46), ('SA', 'TA', 19.01),
    ('WI', 'DE', 20.5), ('WI', 'PG', 28.5), ('WI', 'JB', 55.88),
    ('JB', 'DE', 39.38), ('JB', 'PG', 35.64), ('JB', 'IB', 20.5),
    ('PG', 'IB', 11.7), ('PG', 'RS', 27.4), ('PG', 'DE', 24.15),
    ('IB', 'LO', 34.8), ('LO', 'RS', 25.6), ('LO', 'PN', 29.3),
    ('PN', 'VR', 21.01), ('VR', 'IM', 21.01), ('VR', 'IT', 31.6),
    ('IM', 'IT', 38.4), ('IT', 'AU', 13.3), ('IT', 'CL', 18.8),
    ('IT', 'PE', 26.8), ('IT', 'AL', 62.04), ('AU', 'RS', 26.4),
    ('CL', 'PE', 55.44), ('AL', 'TC', 14.1), ('TC', 'BT', 14.4),
    ('TC', 'PR', 17.4), ('TC', 'AG', 55.2), ('BT', 'PR', 80.96),
    ('AG', 'PR', 75.03), ('AG', 'RS', 10.7), ('AG', 'LA', 8.9),
    ('PR', 'MD', 37.4), ('PR', 'TA', 19.4), ('MD', 'TA', 19.4),
    ('TA', 'RO', 71.06), ('TA', 'DE', 90.64), ('RO', 'LA', 16.6),
    ('LA', 'RS', 13.01)
]

G.add_weighted_edges_from(edges)
```

Fonte: Acervo do autor (2024)

Conforme a figura 14, a função `converter_para_codigo` converte nomes ou códigos de cidades para um código padrão. Ela remove espaços e transforma o texto em maiúsculas para padronização. Retorna o código da cidade se o nome for encontrado em `idades_abreviado`, ou o nome original se encontrado em `idades_extenso`. Se o nome ou código não for válido, levanta um erro.

Figura 14 - Função converter código para padronização de nomes e códigos de cidades.

```
def converter_para_codigo(nome_ou_codigo):
    nome_ou_codigo = nome_ou_codigo.strip().upper()
    if nome_ou_codigo in cidades_abreviado:
        return cidades_abreviado[nome_ou_codigo]
    elif nome_ou_codigo in cidades_extenso:
        return nome_ou_codigo
    else:
        raise ValueError(f"{nome_ou_codigo} não é um código ou nome de cidade válido.")
```

Fonte: Acervo do autor (2024)

Conforme a figura 15, o código solicita ao usuário que informe a cidade de origem e a cidade de destino, podendo ser o nome completo ou o código. Esses dados são então processados pela função `converter_para_codigo` para obter os códigos padrão das cidades.

Figura 15 - Código que solicita cidade de origem e destino e processa para obter códigos padrão.

```
entrada_source = input("Informe a cidade de origem (por extenso ou código): ").strip()
entrada_destination = input("Informe a cidade de destino (por extenso ou código): ").strip()

try:
    source = converter_para_codigo(entrada_source)
    destination = converter_para_codigo(entrada_destination)
```

Fonte: Acervo do autor (2024)

O código verifica se as cidades de origem e destino estão presentes no grafo. Se ambos os nós estiverem no grafo, usa `nx.shortest_path` e `nx.shortest_path_length` para calcular o caminho mais curto e seu comprimento. Converte os códigos das cidades de volta para seus nomes completos para exibição, formata o comprimento do caminho para duas casas decimais e imprime o caminho mais curto junto com seu comprimento, conforme a figura 16.

Figura 16 - Verificação de cidades no grafo e cálculo do caminho mais curto.

```
if source not in G.nodes:
    print(f"Nó de origem '{source}' não está no grafo.")
if destination not in G.nodes:
    print(f"Nó de destino '{destination}' não está no grafo.")

if source in G.nodes and destination in G.nodes:
    shortest_path = nx.shortest_path(G, source=source, target=destination, weight='weight')
    shortest_path_length = nx.shortest_path_length(G, source=source, target=destination, weight='weight')

    shortest_path_names = [cidades_extenso[cidade] for cidade in shortest_path]

    shortest_path_length_formatted = f"{shortest_path_length:.2f}"

    print("Caminho mais curto entre", cidades_extenso[source], "e", cidades_extenso[destination], ":", ' -> '.join(shortest_path_names))
    print("Comprimento do caminho:", shortest_path_length_formatted)
else:
    if source not in G.nodes:
        print(f"Nó de origem '{source}' não está presente no grafo.")
    if destination not in G.nodes:
        print(f"Nó de destino '{destination}' não está presente no grafo.")
```

Fonte: Acervo do autor (2024)

O código trata exceções que podem ocorrer durante a execução. Se um código ou nome de cidade não for válido, é levantado um `ValueError`. Se não houver um caminho entre as cidades especificadas, é levantado um `NetworkXNoPath`. Se algum nó não for encontrado no grafo, é levantado um `NodeNotFound`, conforme a figura 17.

Figura 17 - Tratamento de exceções para códigos e caminhos entre cidades.

```
except ValueError as e:
    print(e)
except nx.NetworkXNoPath as e:
    print(f"Não há um caminho entre '{entrada_source}' e '{entrada_destination}'.")
except nx.NodeNotFound as e:
    print(f"Erro: {e}")
```

Fonte: Acervo do autor (2024)

4.4.2.2 Resultado Final

Resultado gerado sem abreviações dos municípios da região do Alto Vale do Itajaí, apresentado no terminal.

Figura 18 - Resultado obtido por extenso.

```
Informe a cidade de origem (por extenso ou código): Rio do Sul
Informe a cidade de destino (por extenso ou código): Dona Emma
Caminho mais curto entre Rio do Sul e Dona Emma : Rio do Sul -> Presidente Getúlio -> Dona Emma
Comprimento do caminho: 51.55
```

Fonte: Acervo do autor (2024)

Resultado gerado com abreviações dos municípios da região do Alto Vale do Itajaí, apresentado no terminal.

Figura 19 - Resultado obtido abreviado.

```
Informe a cidade de origem (por extenso ou código): PR
Informe a cidade de destino (por extenso ou código): WI
Caminho mais curto entre Pouso Redondo e Witmarsum : Pouso Redondo -> Taió -> Salete -> Witmarsum
Comprimento do caminho: 92.75
```

Fonte: Acervo do autor (2024)

5. CONSIDERAÇÕES FINAIS

O estudo do algoritmo de Dijkstra se destaca por sua relevância em áreas como ciência da computação e engenharia de software, proporcionando soluções eficientes para problemas de roteamento e otimização de caminhos. Implementado tanto em Python quanto em JavaScript, o algoritmo mostra sua flexibilidade e aplicabilidade em diferentes ambientes de programação, ampliando suas possibilidades de uso em aplicações do mundo real.

O uso da biblioteca NetworkX em Python demonstrou ser uma abordagem robusta para a manipulação e visualização de grafos, facilitando a criação de algoritmos complexos com eficiência e clareza. Já em JavaScript, o desenvolvimento de soluções customizadas utilizando estruturas de dados e funções nativas permite um maior controle e personalização, atendendo às necessidades específicas de projetos web, especialmente aqueles voltados para a interatividade e desempenho.

Em conclusão, o aprendizado e a implementação do algoritmo de Dijkstra servem não apenas como uma prática essencial no estudo de estruturas de dados e algoritmos, mas também como uma ferramenta valiosa em aplicações práticas, como navegação por GPS, redes de comunicação e análise de redes sociais. A adoção de bibliotecas especializadas e de soluções adaptadas às linguagens de programação utilizadas torna o processo de desenvolvimento mais acessível e escalável para diversas plataformas e projetos.

REFERÊNCIAS

DIJKSTRA

PORTELA, A. Algoritmo de Dijkstra. Disponível em: http://www.deinf.ufma.br/~portela/ed211_Dijkstra.pdf. Acesso em: 18 set. 2024.

DIJKSTRA EM PYTHON

AKIRADEV. Algoritmo de Dijkstra em Python. Disponível em: <https://akiradev.netlify.app/posts/algoritmo-dijkstra/>. Acesso em: 18 set. 2024.

DIJKSTRA - CARLOS HERRERA

HERRERA, C. Algoritmo de Dijkstra. Disponível em: <https://www.dio.me/articles/o-algoritmo-de-dijkstra-em-python-encontrando-o-caminho-mais-curto>. Acesso em: 21 set. 2024.

DIJKSTRA NO YOUTUBE

YOUTUBE. Algoritmo de Dijkstra. Disponível em: <https://youtu.be/ovkITlgyJ2s?si=EvVTvpwvj0oNA26s>. Acesso em: 18 set. 2024.

NETWORKX EM PYTHON

MARIANO, D. NetworkX em Python. Disponível em: <https://diegomariano.com/networkx/>. Acesso em: 21 set. 2024.

ESTRUTURAS DE GRAFOS

UFPB. Estruturas de Dados em Grafos. Disponível em: <https://repositorio.ufpb.br/jspui/bitstream/tede/7549/5/arquivototal.pdf>. Acesso em: 15 set. 2024.

DADOS PARA GRAFOS

IME. Estruturas de Dados para Grafos. Disponível em: https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/graphdatastructs.html. Acesso em: 15 set. 2024.

HTML

MOZILLA. HTML: Estrutura e Semântica. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTML>. Acesso em: 15 set. 2024.

JAVASCRIPT

MOZILLA. JavaScript: Guia de Introdução. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em: 15 set. 2024.

TEORIA DOS GRAFOS

OBM. Teoria dos Grafos. Disponível em:

https://www.obm.org.br/content/uploads/2017/01/Nivel1_grafos_bruno.pdf. Acesso em: 15 set. 2024.

MATRIZ DE ADJACÊNCIA

ENSINO E INFORMAÇÃO. Teoria dos Grafos e Matriz de Adjacência. Disponível em:

<https://www.ensinoeinformacao.com/teoria-grafos-matriz-de-adjascencia>. Acesso em: 15 set. 2024.

TUTORIAL NETWORKX

NETWORKX. Tutorial de NetworkX. Disponível em:

<https://networkx.org/documentation/stable/tutorial.html>. Acesso em: 20 set. 2024.