Universidade Federal
de Campina Grande

Universidade Federal de Campina Grande
Unidade Acadêmica de Sistemas e Computação
Curso de Ciência da Computação

Implementação de busca pela menor distância entre cidades usando algoritmo de Dijkstra

Disciplina:
Inteligência Artificial

Professor:
Eanes Torres Pereira

Aluno	Matrícula	Função
Alana Vanessa Pimentel Toldo de Andrade	123210882	Implementação do código
Camila Rodrigues de Oliveira Rezende	123210240	Criação e execução dos testes de unidade para TDD
José Vitor Barbosa Maciel	120210954	Think Aloud
Lorena Nascimento Carvalho	123211071	Criação e execução dos testes de unidade para TDD
Lukas Soares Nascimento	123210298	Implementação do código
Sérgio de Sousa Duarte	117110814	Implementação do código

Campina Grande – PB
Novembro de 2025

1. Resultados do TDD e Cobertura de Testes

Durante o desenvolvimento utilizou-se a metodologia *Test Driven Development* (TDD), seguindo o ciclo *Red → Green → Refactor*.

Para cada funcionalidade, escreveu-se inicialmente um teste que falhava (fase *Red*), implementou-se o código mínimo necessário para que o teste passasse (fase *Green*) e, finalmente, realizou-se a refatoração mantendo todos os testes verdes (fase *Refactor*). Esse processo garantiu que todas as funcionalidades fossem guiadas diretamente por testes e que a implementação atendesse aos requisitos definidos.

1.1 Conjunto de Testes Implementados

Foram implementados testes para o algoritmo de Dijkstra, no módulo *dijkstra,* para o módulo *build_graph* e para o módulo *validator* utilizando Pytest [1]e Networkx[2], cobrindo tanto cenários de uso comum quanto casos de erro e borda.

a) Tratamento de erros

Os testes incluem validação de condições inválidas, garantindo que uma exceção adequada seja lançada nos seguintes casos:

- Grafo None;
- Nós inexistentes ou não informados;
- Ausência de arcos (grafo vazio);
- Pesos negativos nos arcos;
- Formatação incorreta do JSON;
- Arcos com tamanho inválido (menos de 3 parâmetros);
- Peso não numérico para os arcos;
- Laços inválidos (ex.: A → A com peso diferente de zero).

b) Casos funcionais do algoritmo de Dijkstra

Foram verificados diversos cenários reais de cálculo de caminhos mínimos:

- Caminho inexistente entre origem e destino;
- Existência de múltiplos caminhos com custos diferentes;
- Caminhos equivalentes (mesmo custo);
- Grafos com ciclo;
- Grafos com gargalos (arcos muito caros);
- Comparação entre caminho direto e caminhos alternativos mais baratos;
- Existência de apenas um caminho possível;
- Caso em que o nó inicial é igual ao nó final.

Estes testes garantem que o algoritmo se comporta corretamente mesmo em condições complexas, refinando a implementação ao longo do processo de TDD.

1.2 Cobertura de Testes

A cobertura de testes foi analisada para garantir a qualidade e a completude do conjunto de testes. Para isso, utilizamos o *pytest-cov*[3], um *plugin* do Pytest que integra a biblioteca *coverage.py*. O *pytest-cov* é usado para medir e reportar quais linhas do

---

[1] Disponível em: https://docs.pytest.org/en/stable/
[2] Disponível em: https://networkx.org/en/
[3] Disponível em: https://pypi.org/project/pytest-cov/

código-fonte foram executadas durante a execução dos testes, indicando a porcentagem de código coberto.

Ao final da execução de todos os testes, o relatório gerado indicou uma cobertura de 100%, o que demonstra um alto nível de confiança na robustez e na qualidade da implementação do algoritmo de Dijkstra e da função de construção do grafo.

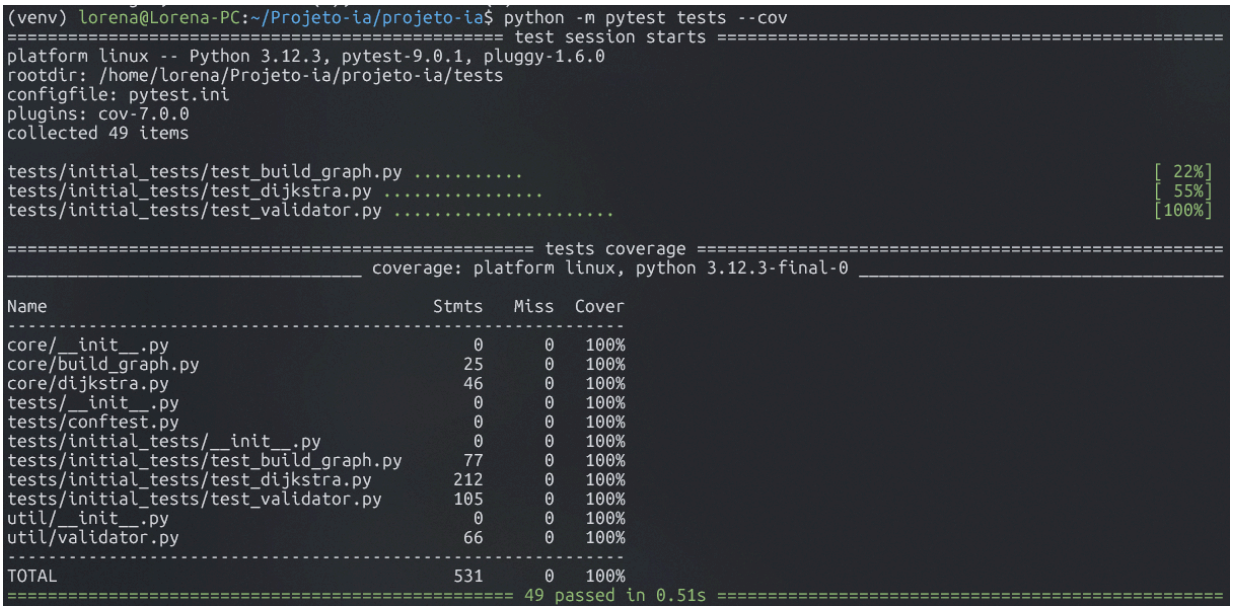

Figura 1: Execução do pytest após a refatoração

1.3 Análise dos Resultados

Os testes implementados demonstram que tanto o algoritmo de Dijkstra quanto a função de construção do grafo apresentam comportamento robusto, mesmo diante de cenários extremos ou entradas inválidas.

O uso de TDD foi fundamental para detectar inconsistências logo nas fases iniciais do desenvolvimento, permitindo que erros fossem corrigidos rapidamente antes de prejudicar funcionalidades dependentes.

2. Relatório Think Aloud

Todos os membros tinham clareza sobre qual o problema que estávamos tentando resolver e como a sua solução ajudava a alcançar o objetivo do projeto. Por conta disso, as respostas para as três perguntas iniciais foram praticamente as mesmas por todos os membros da equipe.

2.1 Transcrição das Entrevistas (resumo)

Entrevistador: **Vitor**
Entrevistado(a): **Alana**
Objetivo do trabalho do entrevistado: **Escrever a função que representa o algoritmo de Dijkstra e realizar a busca em um grafo**
Data da entrevista: **20/11/25**
Transcrição:
Vitor: “Oi, Alana. Boa tarde. Alana, eu queria que você me explicasse, por gentileza, qual é o seu entendimento sobre o nosso problema que a gente está tentando resolver.”

Alana: “Dado a especificação que a gente escolheu, eu entendi do problema que, basicamente, nós teríamos, o usuário, enfim, o algoritmo, o programa teria como entrada duas cidades, uma cidade A é uma cidade B, e o nosso algoritmo deveria conseguir encontrar, a partir do algoritmo de Dijkstra, encontrar o menor caminho entre essas duas cidades.”
Vitor: “Certo. E aí, essas duas cidades, elas estão em que formato? Como é que o algoritmo, ele deveria receber essas cidades?É string? É um objeto? O que é, exatamente?”
Alana: “Inicialmente, o nosso algoritmo, ele vai ser... o nosso programa, ele vai ser rodado no terminal. E aí, como... como entrada do nosso programa, nesta primeira parte, ele recebe um arquivo, um caminho de um arquivo JSON. E ele recebe duas strings. Uma string, a primeira string para a cidade inicial, e a segunda string para o nome da cidade final. E aí, nós temos... aí nós temos o main. Nós vamos ter o main. Nós temos o main. E esse main, a partir do JSON, do arquivo JSON, ele constrói o grafo. E então, ele chama o algoritmo de Dijkstra, em cima desse grafo, para ele conseguir encontrar o menor caminho entre o vértice, entre o nó da cidade início, até o nó da cidade final.”
Vitor: “Então, a gente tem um peso de cada, vamos dizer assim, de cada distância entre as cidades. Aí, de onde veio esses dados?”
Alana: “Então, nesse momento, é o usuário que tem que montar esse JSON e passar para o programa.
A gente pretende, quando finalizarmos toda a implementação do Dijkstra, do Build Graph e do Validator, quando a gente terminar toda a parte bruta, digamos assim, a gente quer evoluir para a parte bônus, para implementar uma das partes bônus, que é, o usuário vai informar apenas Campina Grande, apenas de uma pessoa, apenas, por exemplo, o usuário vai dizer apenas assim. Campina Grande de uma pessoa. Isso. E aí, nós estamos pesquisando alguma biblioteca que manipule dados geográficos para que essa biblioteca gere esse grafo entre a cidade de Campina Grande até a cidade de uma pessoa, e a gente pega esse grafo e rode o nosso Dijkstra em cima para a gente encontrar o menor caminho. Mas, inicialmente, nesse momento agora, quem tem que passar esse JSON completo é o usuário.”
Vitor: “Então, vamos dizer, essa é talvez a maior limitação da implementação atual, né?”
Alana: “O usuário tem que passar esse JSON e aí tem todos aqueles riscos, né? Ele pode ter um peso não condizente com a real distância, a formatação pode estar errada. Ele pode esquecer um arco, ele pode colocar uma distância errada, ele pode escrever o nome da cidade errado, enfim.”

Entrevistador: **Vitor**
Entrevistado(a): **Lorena**
Objetivo do trabalho do entrevistado: **Testar o algoritmo de Dijkstra desenvolvido por Alana e refatorar em funções com base em testes estáticos**
Data da entrevista: **20/11/25**
Transcrição:
Vitor: “No caso, dado o nosso problema, o que é que você ficou responsável de resolver?”
Lorena: “Eu fiquei, junto com o Camila, responsável de fazer os testes para o TDD. E os testes, acredito que os testes estáticos também, para fazer funcionar. Tipo, para testar a aplicação quando já estiver toda pronta, do começo ao fim, sabe?”
Vitor: “Aí, qual ferramenta vocês usaram para fazer os testes? Teve alguma dificuldade para usar ela?”

Lorena: “Foi o PyTest. Eu achei tranquilo. Aí, a minha dificuldade mais foi de mudar o pensamento de que eu tenho que fazer um teste baseado no código, e sim que eu tenho que fazer um teste para as outras pessoas programarem em cima. Aí, isso foi uma complicação minha, mas com relação à ferramenta não foi difícil.”
Vitor: “Então, uma coisa que eu quero que tu faça é só dar uma passada rápida sobre esses testes que tu criou aí. Na verdade, antes disso, eu tenho outra pergunta. Os cenários de testes, quais que tem atualmente aí?”
Lorena: “A gente está com todos aqueles que a gente decidiu, que seriam os testes que eles basearam na função do algoritmo de Dijkstra. Sim. Então, os testes que estão aqui, eles estão cobrindo com relação ao grafo nulo, com relação a se a função está funcionando direito com parâmetros errados, se ele está lançando o que deve lançar, se ele está funcionando direitinho quando tem dois caminhos iguais. Então, foram os testes que a gente decidiu e listou e eu fui fazendo esse.”
Vitor: “Mas, assim, vocês já pensaram em usar algum tipo de LLM só para ela descrever quais cenários seriam interessantes para cobrir nos testes? Ou então, pedir sugestões de algum tipo?Vocês estão pensando em levar isso em consideração?”
Lorena: “não vou negar, agora o que você falou é uma boa ideia, eu acho que é uma boa ideia, porque a gente estava junto naquela reunião de, acho que foi quinta, não, enfim, acho que foi quarta.
Naquela reunião à noite a gente saiu pensando em alguns casos e anotou. Quem estava com o documento escrevendo, acho que era mais eu e Alana, eu não usei e eu não sei se ela usou na hora de pensar em casos de teste, mas eu acho que é realmente uma boa ideia, eu cogitaria. Eu acho que pode ser uma boa ideia, por exemplo, pegar essa lista que eu tenho aqui nas informações que a gente decidiu e dizer, ó, eu vou explicar todo o problema e dizer, eu tenho esses testes aqui e perguntar se tem algo que poderia ser anexado, né, assim, para melhorar. Com certeza tem mais testes para fazer e completar, sabe? Mas acho que é porque o projeto não está terminado ainda.”

Entrevistador: **Vitor**
Entrevistado(a): **Camila**
Objetivo do trabalho do entrevistado: **Testar a função de criação de grafo direcionado e trazer testes mais complexos do algoritmo de Dijkstra e refatoração em funções com base em testes estáticos**
Data da entrevista: **22/11/25**
Transcrição:
Vitor: “O que é que você ficou responsável de fazer?”
Camila: “Eu continuei com os testes junto com a Lorena. Aí, eu continuei com os testes para o Dijkstra, alguns, e também fiz os testes para o BuildGraph, que foi uma função que o Lukas fez, que é para construir o grafo com o arquivo JSON, basicamente.”
Vitor: “E aí, vocês dividiram os testes entre si e aí a questão, como é que estão os teus testes? Eles já estão finalizados? O que é que os teus testes abordaram aí? O que é que eles estão tentando cobrir”
Camila: “Como a Lorena já fez os testes mais básicos que a gente definiu, eu fui atrás de tentar pensar em mais testes não tão básicos para serem feitos. E foram esse daqui, que tem o caminho direto entre o nó e existe outro caminho que é mais barato do que esse caminho direto, aí ele tem que retornar ao outro caminho que é mais barato. Aí, tem esse, que também é basicamente a mesma ideia, só que tem mais outras rotas alternativas do

que o caminho direto, que são mais baratas, aí ele escolhe qual é a mais barata entre essas. Aí, tem esse daqui, que é um grafo ciclo, que entre as cidades formam um ciclo, só que duas cidades que estão dentro desse ciclo, existe um caminho entre elas que não está dentro desse ciclo, que é bem mais barato do que você rodar por esse ciclo aí. E esse daqui, que é tipo um grafo onde tem vários caminhos possíveis, é uma situação bem geral, e entre esses caminhos tem uma aresta com um custo muito alto. Aí, acaba gerando um gargalo no meio desse caminho, faz ele ficar muito caro. Então, foi só para verificar se a função realmente escolheu o mais barato, mesmo tendo esses gargalos no meio de alguns caminhos, como se fosse uma pegadinha, sabe?"

Entrevistador: **Vitor**
Entrevistado(a): **Lukas**
Objetivo do trabalho do entrevistado: **Escrever a função que gera um grafo direcionado a partir de um dataset com um formato pré-estabelecido**
Data da entrevista: **22/11/25**
Transcrição:
Vitor: "O que é que tu ficou responsável dessa solução aí?"
Lukas: "Ah, eu fiquei responsável por criar o código para construir o grafo usando o NetworkX da biblioteca. Ela recebe um path. Esse path é o caminho onde está o dataset que contém os dados das cidades e a distância entre elas. O algoritmo, ele usa a biblioteca NetworkX para construir o grafo que através desse path que é o dataset vai construir os grafos com os nós e o peso entre essas arestas. Um grafo direcionado, inclusive."
Vitor: "A escolha dessa biblioteca foi devido à familiaridade?
Foi a mesma que você trabalhou em grafos, foi?"
Lukas: "Exatamente isso. A NX é o que está representando o NetworkX, no caso. Essa biblioteca foi o que a gente utilizou tanto em grafos, quanto em teoria da computação. A professora Patrícia, que nos apresentou essa biblioteca. E acredito que não tenha uma biblioteca melhor que essa, além da familiaridade que a gente já tem.Acho que acredito que essa seja a melhor biblioteca para trabalhar com grafos, a construção e operações nele."

Entrevistador: **Vitor**
Entrevistado(a): **Sérgio**
Objetivo do trabalho do entrevistado: **Escrever a função que realiza validações nas informações passadas pelo usuário na entrada do sistema**
Data da entrevista: **24/11/25**
Transcrição:
Vitor: "eu entendo que o trabalho foi dividido entre as pessoas, cada pessoa ficou com um objetivo. Qual foi o teu objetivo e como é que ele ajudou a resolver o problema principal do projeto?"
Sérgio: "Meu papel foi de fazer um arquivo, um programa, um pedaço, um código, que valida as entradas do usuário, garantindo o que a gente precisa para calcular. Se ele dá a entrada que não vai válida, o algoritmo já usa e, para ficar com a interface entre a pessoa que está usando o código e o algoritmo em si, a gente tem algumas tentativas e, eventualmente, na parte 2 do projeto, na segunda entrega, enfim, quando evoluir mais o código, a gente vai fazer o melhor uso dessa parte de validação."
Vitor: "Pode me explicar melhor, por favor?"

Sérgio: “Primeiro, a gente vai validar se o caminho existe, se é um arquivo .json, se futuramente a gente precisa ver se está no formato, se as informações dentro do próprio arquivo também estão ok, mas, até então, a gente já consegue validar se é um arquivo .json. Depois, a gente vai validar as entradas do próprio usuário, com relação a se o grafo possível ser gerado, como a função do build lá do pessoal, que quem fez, acho que teve o Lukas, e aí, depois disso, se for validado com relação ao grafo possível ser gerado, a gente valida se os vértices em si fazem parte do grafo gerado. Mais aqui embaixo, a gente vai ter que, a validação do path - como eu tinha falado anteriormente - verifica se existe o arquivo e se ele é um arquivo .json. Caso não, a gente gera a exceção aqui e, para não quebrar na cara do usuário, lá no fluxo principal, a gente trata direitinho. Depois, a gente valida as entradas do grafo em si, verifica se o vértice é vazio, cada vértice, cada aresta montada dentro do grafo, verifica se as arestas são vazias, se os vértices são vazios, se o valor entre um vértice e outro é válido, se ele precisa ser um número e não pode ser um string.”

2.2 Acesso às Entrevistas

Todas as entrevistas foram gravadas através do *software OBS*. Os membros compartilhavam a sua tela e apresentavam o seu código. A transcrição foi feita através da ferramenta Whisper e todas as gravações e transcrições podem ser acessada através dos links abaixo:

Membro	Link da entrevista
Alana	link
Camila	link
Lorena	link
Lukas	link
Sérgio	link

2.3 Top 10 Pontos Mais Importantes Observados

Após a finalização do trabalho, toda a equipe participou de uma reunião de retrospectiva. O objetivo dessa reunião era analisar três pontos principais: a) o que deu certo; b) o que precisava melhorar e c) o que não deu certo. Cada membro precisava levar em consideração o que foi falado nas entrevistas individuais e principalmente, qual era a sua responsabilidade dentro do problema principal. O resultado alcançado com essa conversa foi uma lista de 10 itens apresentados abaixo.

Foi Bom	Pode ser Melhorado	Foi Ruim
Realizar uma reunião inicial para definir os principais testes a serem implementados. Isso acabou sendo um grande norte para o desenvolvimento.	Má gestão do tempo da equipe. Isso acarretou na não implementação do bônus. Ter começado antes teria dado uma folga.	Horários diferentes de disponibilidade. Ter feito mais sessões de trabalho em conjunto teria sido mais benéfico para o projeto.
Utilização da ferramenta *pylint*[4], para identificar *code smells*[5].	Subestimar o problema. Alguns membros subestimaram a sua	

[4] Disponível em: https://pypi.org/project/pylint/
[5] https://pt.wikipedia.org/wiki/Code_smell

Acabou sendo importante durante todo o tempo de trabalho.	parte, o que acabou gerando alguns impeditivos	
Separação clara entre as tarefas de cada membro. Isso permitiu o trabalho paralelo e também o conhecimento sobre o trabalho dos demais.	Falta de documentação de todas as decisões tomadas. Alguns pontos foram esquecidos, o que gerou alguns desentendimentos.	
Sessões de trabalho em conjunto, via *discord*		
Implementação de uma *minHeap*, através da biblioteca *HeapQ*[6] , para auxiliar no algoritmo de Dijkstra		
Forçar os membros a trabalharem em equipe, visto que a maioria prefere trabalhar sozinho.		

3. Testes Estáticos

Para garantir a manutenção, legibilidade e conformidade do código desenvolvido, foi realizada a etapa de Testes Estáticos. Esses testes objetivam analisar a estrutura do código em busca de más práticas de programação (*bad smells*), erros de sintaxe e violações de convenção de estilo. A ferramenta escolhida para esta etapa foi o *Pylint,* um analisador que verifica se o módulo está em conformidade com o PEP 8 e retorna um relatório classificando as mudanças que precisam ser feitas em: *Convention*, *warning*, *refactor*, *error* e *fatal*.

O Pylint foi executado via terminal no ambiente virtual do projeto (*venv*) e o processo seguiu um ciclo iterativo de Análise → Correção → Reavaliação para cada módulo do projeto até que se atingisse a nota máxima.

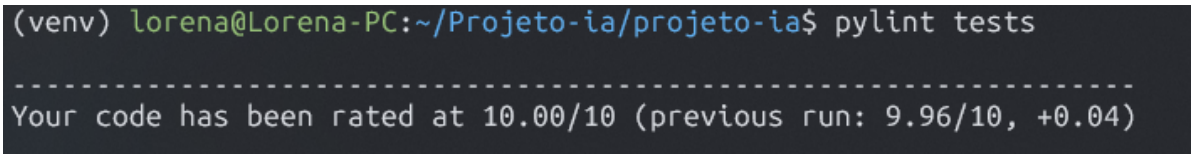

Figura 2: Avaliação final do módulo 'tests' após as correções indicadas

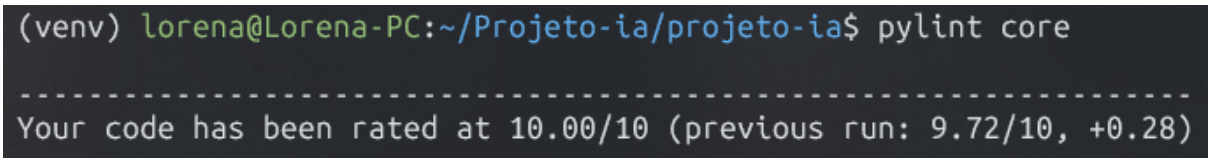

Figura 3: Avaliação final do módulo 'core' após as correções indicadas

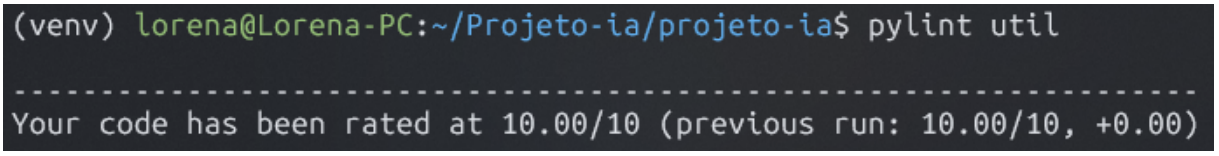

Figura 4: Avaliação final do módulo 'util' após as correções indicadas

Os problemas mais comuns encontrados durante os processos envolvem ausência de documentação (*docstring)* de módulos e funções, melhoria nos nomes atribuídos às

[6] Disponível em: https://docs.python.org/3/library/heapq.html

variáveis e uma distinção clara de variáveis usadas globalmente e localmente. Ajustando todos os avisos requisitados conseguimos um código padronizado, legível e de boa manutenção.

4. Métricas Estatísticas

Como o algoritmo de Dijkstra é determinístico e não-probabilístico, adotamos como métricas principais o tempo de processamento sob diferentes cargas e a complexidade do código. Os experimentos foram executados em uma única máquina com as seguintes especificações: CPU Intel(R) Core(TM) i5-1335U (1.30 GHz) e 32GB de RAM DDR4 3200 MHz (2x16). A metodologia adotada foi a seguinte:

- Geramos dois dígrafos estruturados em grade, variando sua dimensão. O primeiro utiliza pesos uniformes iguais a 1 e o segundo utiliza pesos randômicos, aproximando melhor um cenário real;
- Para cada dimensão, definimos como nó inicial o vértice superior esquerdo e como nó final o vértice inferior direito;
- Para cada configuração, executamos o algoritmo trinta vezes, calculando o tempo médio e o intervalo de confiança de 95%.

A Tabela 1 apresenta, para cada dimensão, a quantidade de nós e arcos envolvidos.

Dimensão	Quantidade de Nós	Quantidade de Arcos
10x10	100	360
20x20	400	1.520
30x30	900	3.480
40x40	1.600	6.240
50x50	2.500	9.800
60x60	3.600	14.160
70x70	4.900	19.320
80x80	6.400	25.280
90x90	8.100	32.040
100x100	10.000	39.600
500x500	250.000	998.000
1000x1000	1.000.000	3.996.000

Tabela 1: Quantidade de nós e arcos por dimensão

Em seguida, a Tabela 2 mostra as médias e intervalos de confiança obtidos para os dígrafos com pesos uniformes (PU) e pesos randômicos (PR).

Dimensão	Média (PU)	Int. Confiança (PU)	Média (PR)	Int. Confiança (PR)
10x10	0,00025	0,0002; 0,0003	0,00027	0,0003; 0,0003
20x20	0,00112	0,0011; 0,0012	0,00121	0,0012; 0,0013

30x30	0,00260	0,0024; 0,0028	0,00270	0,0026; 0,0028
40x40	0,00429	0,0041; 0,0045	0,00471	0,0045; 0,0049
50x50	0,00635	0,0063; 0,0064	0,00696	0,0069; 0,007
60x60	0,00932	0,0092; 0,0095	0,01053	0,0104; 0,0107
70x70	0,01290	0,0127; 0,0131	0,01462	0,0145; 0,0148
80x80	0,01751	0,0172; 0,0179	0,01992	0,0195; 0,0204
90x90	0,02298	0,0226; 0,0234	0,02585	0,0252; 0,0265
100x100	0,02828	0,0277; 0,0288	0,03272	0,0318; 0,0337
500x500	0,89435	0,8897; 0,8990	1,04649	1,0420; 1,0510
1000x1000	3,94004	3,6124; 4,2677	4,59397	4,3743; 4,8136

Tabela 2: Média e intervalo de confiança de 95% para os dois tipos de grafos analisados, por dimensão

A Figura 5 ilustra visualmente o comportamento do tempo de execução em função da dimensão do grafo.

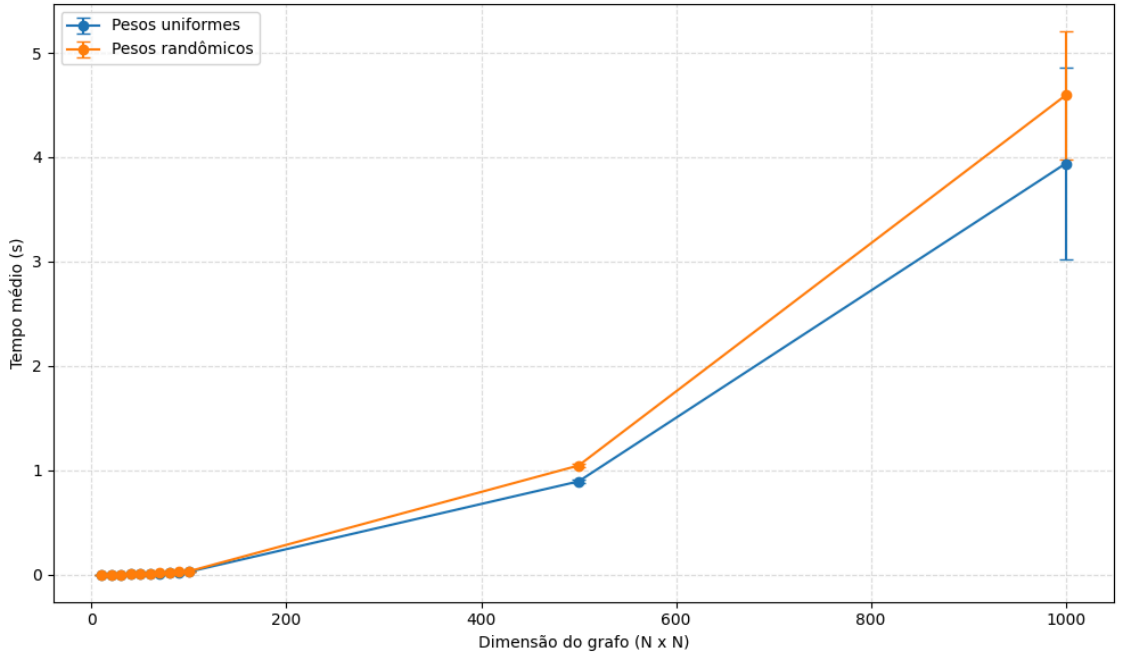

Figura 5: Tempo de execução do Dijkstra por dimensão do grafo

Os resultados mostram que, ao aumentar o número de nós em 100 vezes (de 10×10 para 100×100), o tempo de execução cresce aproximadamente 113 vezes para pesos uniformes e 121 vezes para pesos randômicos. O mesmo aumento de escala (100×) aplicado de 100×100 para 1000×1000 resulta em um crescimento de aproximadamente 140 vezes em ambos os cenários.

Esse comportamento indica crescimento superlinear, acima de O(n), mas ainda muito inferior a um crescimento exponencial. Assim, o desempenho observado confirma que o algoritmo permanece escalável e adequado para dígrafos de grande escala.