UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE RUSSAS

CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

HENRIQUE LOPES LIMA

IMPLEMENTAÇÃO E ANÁLISE DE ALGORITMOS DE CAMINHO MÍNIMO PARA CÁLCULO DE MATRIZES DE DISTÂNCIA

RUSSAS

2026

HENRIQUE LOPES LIMA

IMPLEMENTAÇÃO E ANÁLISE DE ALGORITMOS DE CAMINHO MÍNIMO PARA CÁLCULO DE MATRIZES DE DISTÂNCIA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da computação do Campus de Russas da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da computação.

Orientador: Prof. Dr. Alexandre Matos Arruda.

RUSSAS

2026

HENRIQUE LOPES LIMA

IMPLEMENTAÇÃO E ANÁLISE DE ALGORITMOS DE CAMINHO MÍNIMO PARA CÁLCULO DE MATRIZES DE DISTÂNCIA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da computação do Campus de Russas da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Ciência da computação.

Aprovada em: 06/02/2026.

BANCA EXAMINADORA

Prof. Dr. Alexandre Matos Arruda (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Cenez Araújo de Rezende
Universidade Federal do Ceará (UFC)

Esp. João Pedro de Araújo Lima
Universidade de São Paulo (USP)

Toda glória seja dada a Deus.

AGRADECIMENTOS

A Deus, que me sustentou em cada passo desta jornada e me concedeu forças para superar os desafios.

A minha amada esposa, Julia Lima, que me apoiou incondicionalmente, oferecendo amor, paciência e compreensão nos momentos mais desafiadores. Sua presença foi fundamental para a realização deste trabalho.

A todos os meus familiares, que sempre acreditaram em mim e me incentivaram a seguir meus sonhos, em especial a minha vó Celi e mãe adotiva Bia, que me criaram com muito amor.

Aos mais que amigos, meus irmãos, que Deus me deu, em especial ao Dr. Nelson Gonçalves que se encontra na nossa nação de origem.

Ao meu orientador e amigo, o Doutor Alexandre Matos Arruda que dedicou seu tempo e conhecimento para me orientar e moldar meu trabalho. Sua orientação foi fundamental para o sucesso deste projeto, agradeço toda paciência e compreensão.

Cada conquista alcançada é uma celebração compartilhada e este trabalho reflete o esforço coletivo e o amor que recebi ao longo desta jornada. Muito obrigado!

"Ensina-me o teu caminho, Senhor, para que eu ande na tua verdade; dá-me um coração inteiramente fiel, para que eu tema o teu nome."
(Salmos 86:11)

RESUMO

Esse trabalho apresenta a fundamentação teórica e a metodologia para implementação do algoritmo de Busca de Custo Uniforme no cálculo de caminhos mínimos em grafos direcionados e ponderados com pesos não negativos. São discutidos os princípios do algoritmo, sua implementação em pseudocódigo e uma metodologia para avaliação de desempenho, com foco em implementações paralelas em Go e C++. A análise aborda a complexidade temporal, o uso de memória e a adequação ao problema, fornecendo uma base sólida para estudos práticos e experimentais em grafos de grande escala.

Palavras-chave: teoria dos grafos;caminho mínimo;análise de desempenho.

ABSTRACT

This work presents the theoretical foundation and methodology for implementing the Uniform Cost Search algorithm for computing shortest paths in directed, weighted graphs with non-negative weights. The principles of the algorithm, its pseudocode implementation, and a methodology for performance evaluation are discussed, focusing on parallel implementations in Go and C++. The analysis covers time complexity, memory usage, and suitability for the problem, providing a robust foundation for practical and experimental studies on large-scale graphs.

Keywords: graph theory; shortest path; performance analysis.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

APSP *All Pairs Shortest Path*

P2P *Peer-to-peer*

SSSP *Single Source Shortest Path*

UCS *Uniform Cost Search*

LISTA DE SÍMBOLOS

G	Grafo
V	Conjunto de vértices
E	Conjunto de arestas
$\|V\|$	Número de vértices
$\|E\|$	Número de arestas

SUMÁRIO

1 INTRODUÇÃO

A modelagem de sistemas complexos por meio de grafos constitui uma das abstrações mais poderosas da ciência da computação, permitindo representar relações estruturais e dinâmicas em múltiplos domínios; o cálculo de caminhos mínimos, por sua vez, representa uma das operações mais fundamentais nessa teoria(Cormen Charles E. Leiserson; Stein, 2022). A onipresença deste problema é demonstrada por sua aplicação direta em sistemas de navegação que otimizam rotas em tempo real, em protocolos de rede que garantem a eficiência do tráfego na internet e na análise de redes sociais para quantificar relações (Madkour *et al.*, 2017). Frequentemente, a necessidade transcende a busca por um único percurso, exigindo o conhecimento de todas as distâncias mínimas, o que culmina na construção da matriz de distância. Esta tarefa, contudo, impõe um desafio computacional significativo, cuja intensidade cresce exponencialmente com a escala do grafo.

Diante do desafio de desempenho para calcular a matriz de distância, que armazena o menor custo entre todos os pares de nós, diversos algoritmos clássicos como Dijkstra, Floyd-Warshall e A* foram desenvolvidos. Este trabalho foca na implementação e análise do algoritmo de Busca de Custo Uniforme, do inglês *Uniform Cost Search* (UCS), um algoritmo robusto e garantido para encontrar o caminho ótimo em grafos com pesos não negativos, servindo como uma excelente base para um estudo de desempenho aprofundado.

O objetivo central desta pesquisa é investigar como diferentes abordagens de programação moderna podem otimizar a execução deste cálculo através da computação paralela. Para isso, propõe-se uma análise de desempenho comparativa de implementações do UCS em **Go** e **C++**, duas linguagens conhecidas por suas distintas capacidades de performance, segurança e concorrência. Utilizando grafos do renomado instituto *DIMACS*(DIMACS, 2006), o estudo avaliará métricas de tempo de execução e consumo de memória para fornecer uma base empírica sobre os *trade-offs* de cada tecnologia na resolução deste problema clássico.

O presente trabalho foi estruturado em oito capítulos, de modo a apresentar a pesquisa de forma clara e sequencial. Inicialmente, o primeiro oferece uma introdução ao tema, contextualizando o problema e destacando sua relevância, o segundo, por sua vez, define os objetivos da pesquisa, divididos em geral e específicos. A base teórica do estudo é consolidada no terceiro capítulo, que explora a fundamentação teórica e realiza uma revisão bibliográfica sobre teoria dos grafos, algoritmos de caminho mínimo e as linguagens de programação empregadas. Dando sequência o quarto analisa e compara outros trabalhos relacionados ao tema, o quinto

detalha a metodologia e o projeto da implementação, descrevendo o ambiente, os métodos de avaliação, o desenvolvimento do UCS e os cenários de teste, o sexto apresenta o cronograma que orientou a realização de todas as etapas do projeto, o sétimo aborda os resultados dos testes de validação e a análise de desempenho e, por fim, o ultimo cápitulo sintetiza os resultados, discute as contribuições, limitações e sugere direções para pesquisas futuras.

2 OBJETIVOS

Nesta seção, serão detalhados os objetivos gerais e específicos deste trabalho.

2.1 Objetivo geral

Analisar o desempenho de implementações paralelas do algoritmo UCS para o cálculo de matrizes de distância em grafos direcionados e ponderados, com foco em uma análise comparativa entre as linguagens Go e C++.

2.2 Objetivos específicos

- Fundamentar teoricamente o problema de caminho mínimo e seus algoritmos relacionados, detalhando seus princípios de funcionamento, pseudocódigos, análise de complexidade (tempo e espaço) e as estruturas de dados adequadas para sua implementação eficiente;
- Desenvolver uma implementação computacional robusta para UCS e estabelecer uma metodologia clara para a avaliação de desempenho, definindo as métricas (tempo de execução, consumo de memória) e os cenários de teste com grafos de diferentes topologias e densidades;
- Avaliar empiricamente o desempenho e a escalabilidade das implementações, executando testes sistemáticos para validar a correção dos resultados e coletar dados quantitativos de performance em grafos de pequeno, médio e grande porte;
- Realizar uma análise comparativa dos resultados obtidos, confrontando o desempenho prático observado com a complexidade teórica esperada de cada algoritmo, a fim de determinar suas respectivas vantagens, desvantagens e cenários de aplicação ideais.

3 FUNDAMENTAÇÃO TEÓRICA

Nessa seção, serão descritos os conceitos fundamentais que sustentam este trabalho científico, abordando os temas necessários a cada tópico. O objetivo é apresentar, de forma clara e sucinta, os principais fundamentos desta pesquisa. Ao descrever esses fundamentos, iremos transmitir uma visão abrangente e precisa, destacando as ideias e informações cruciais necessárias para o entendimento completo deste estudo científico. Com isso em mente, esperamos proporcionar aos leitores uma base sólida de conhecimento, permitindo uma compreensão aprofundada dos temas abordados neste trabalho.

3.1 Teoria dos Grafos: Conceitos Fundamentais

A teoria dos grafos costuma ter seu marco inaugural atribuído ao trabalho de Leonhard Euler sobre o famoso problema das sete pontes de Königsberg, apresentado à Academia de São Petersburgo em 1735 (Sachs, 1958). Nesse artigo, Euler demonstrou que é impossível percorrer todas as pontes exatamente uma vez e, ao reformular a situação em termos abstratos de regiões e conexões, introduziu a ideia de representar os trechos de terra como vértices e as pontes como arestas, sendo este trabalho amplamente reconhecido como o primeiro resultado da teoria dos grafos.

Formalmente, um grafo $G = (V, E)$ é um par de conjuntos tal que $E \subseteq [V]^2$, onde os elementos de E são subconjuntos de dois elementos de V. Os elementos de V são denominados vértices (ou nós, ou pontos) do grafo G, e os elementos de E são suas arestas (ou linhas)(Diestel, 2017).

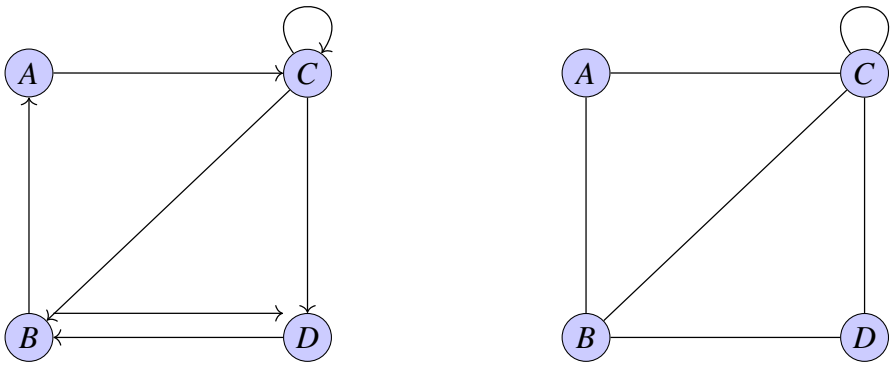
Diversas situações do mundo real podem ser convenientemente descritas por um diagrama composto por um conjunto de pontos e por linhas que conectam alguns desses pontos. Por exemplo, os pontos podem representar pessoas, com as linhas unindo pares de amigos; ou os pontos podem ser centros de comunicação, com as linhas representando os links de comunicação. Uma abstração matemática para situações desse tipo origina o conceito de grafo(Bondy; Murty, 1976).

3.1.1 Tipos e Propriedades de Grafos

Os grafos podem ser classificados com base em várias características:

– Direcionados vs. Não Direcionados: Em grafos direcionados (ou dígrafos), as arestas possuem uma direção, representada como (u, v), indicando uma relação unidirecional de u para v. Em grafos não direcionados, as arestas são bidirecionais, denotadas como $\{u, v\}$, refletindo simetria nas conexões (Bondy; Murty, 2008; Diestel, 2017).

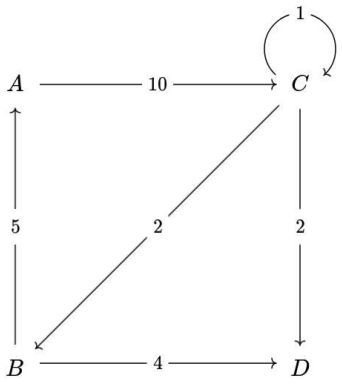
Figura 1 – Grafo direcionado a esquerda e o não direcionado a direita.



Fonte: Elaborado pelo autor.

– Ponderados vs. Não Ponderados: Grafos ponderados associam um peso numérico a cada aresta, como $w(u, v)$, que pode representar custos, distâncias ou capacidades (Cormen Charles E. Leiserson; Stein, 2022; Bondy; Murty, 2008). Grafos não ponderados assumem que todas as arestas têm peso implícito igual (geralmente 1).

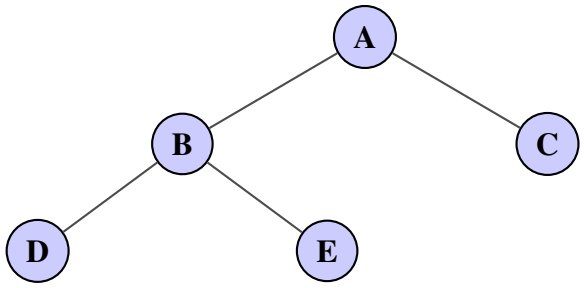
Figura 2 – Grafo ponderado.



Fonte: Elaborado pelo autor.

– Cíclicos vs. Acíclicos: Um grafo cíclico contém pelo menos um ciclo, definido como um caminho fechado que retorna ao vértice inicial (Diestel, 2017). Grafos acíclicos, como árvores, não possuem ciclos e são comuns em estruturas hierárquicas (Cormen Charles E. Leiserson; Stein, 2022).

Figura 3 – Árvore binária simples.



Fonte: Elaborado pelo autor.

– Simples vs. Multígrafos: Grafos simples não permitem arestas múltiplas entre o mesmo par de vértices nem laços (arestas de um vértice para si mesmo) (Bondy; Murty, 2008). Multígrafos relaxam essas restrições, permitindo maior flexibilidade na modelagem.

Outras propriedades relevantes incluem:

– Grau de um Vértice: O número de arestas incidentes a um vértice (Diestel, 2017). Em dígrafos, distingue-se entre grau de entrada (in-degree) e grau de saída (out-degree) (Cormen Charles E. Leiserson; Stein, 2022).
– Densidade: A razão entre o número de arestas $|E|$ e o número máximo possível de arestas, que é $\frac{|V|(|V|-1)}{2}$ para grafos não direcionados simples e $|V|(|V|-1)$ para grafos direcionados (Bondy; Murty, 2008). Essa propriedade é crucial na escolha de algoritmos de caminho mínimo, já que a densidade determina se um grafo é considerado esparso ou denso.
– Conectividade: Um grafo é conexo se existe um caminho entre qualquer par de vértices (Diestel, 2017). Em dígrafos, fala-se em *forte conectividade* (existem caminhos em ambas as direções entre quaisquer dois vértices) ou *fraca conectividade* (grafo é conexo quando as direções das arestas são ignoradas) (Cormen Charles E. Leiserson; Stein, 2022).

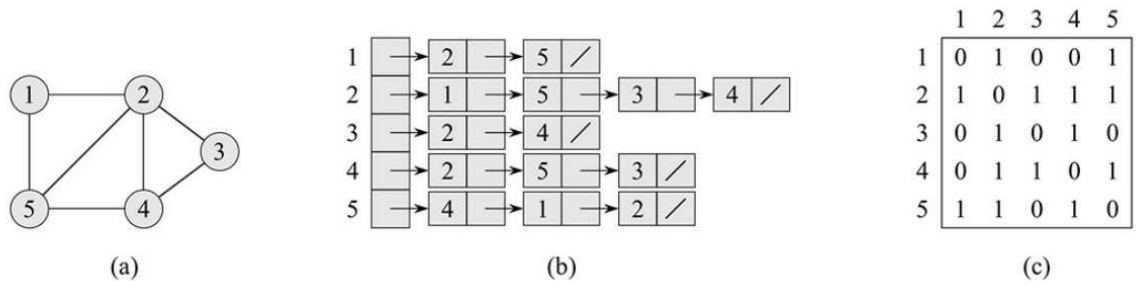
3.1.2 Representação Computacional de Grafos

A escolha da representação de um grafo impacta diretamente a eficiência dos algoritmos que o processam (Cormen Charles E. Leiserson; Stein, 2022). As duas principais

abordagens são:

- Matriz de Adjacência: Uma matriz u $n \times n$, onde $n = |V|$, com $M[u][v] = w(u,v)$ se existe uma aresta de u para v com peso w, ou $0/\infty$ caso contrário (Bondy; Murty, 2008; Cormen Charles E. Leiserson; Stein, 2022). Requer $O(|V|^2)$ de espaço, sendo ideal para grafos densos, mas ineficiente para grafos esparsos. O acesso a uma aresta específica é realizado em tempo constante $O(1)$.
- Lista de Adjacência: Um mapa ou *array* onde cada vértice u está associado a uma lista de tuplas (v,w), representando seus vizinhos e os pesos das arestas (Cormen Charles E. Leiserson; Stein, 2022). Usa $O(|V|+|E|)$ de espaço, sendo mais eficiente para grafos esparsos (Zhan; Noon, 1998), como redes rodoviárias do DIMACS utilizado neste trabalho. O tempo para verificar a existência de uma aresta específica é proporcional ao grau do vértice de origem.

Figura 4 – (a) Um grafo não dirigido G com cinco vértices e sete arestas. (b) Uma representação de G por lista de adjacências. (c) A representação de G por matriz de adjacências.



Fonte: Adaptado de (Cormen Charles E. Leiserson; Stein, 2022, p. 484).

3.2 Caminho mínimo

O estudo do problema do caminho mínimo intensificou a partir da década de 1950, impulsionado pelo desenvolvimento da pesquisa operacional no contexto das redes de comunicação e transporte do pós-guerra, especialmente relacionado ao roteamento automatizado de chamadas telefônicas de longa distância nos Estados Unidos (Schrijver, 2012). Em 1956, Edsger W. Dijkstra desenvolveu seu algoritmo inicial enquanto trabalhava no *Mathematisch Centrum* (Centro Matemático) de Amsterdã, desenvolvendo-o em "cerca de vinte minutos"como uma demonstração compreensível das capacidades do novo computador ARMAC, aplicando-o a um mapa simplificado de cidades na Holanda(Schrijver, 2012). Embora formulado em 1956, o algoritmo só foi publicado em 1959 no artigo "A Note on Two Problems in Connexion with Graphs"(Dijkstra, 1959). Quase simultaneamente, Lester Ford Jr. (1956) e Richard Bellman (1958) desenvolveram independentemente métodos baseados em programação dinâmica capazes de lidar com pesos negativos nas arestas, ampliando o escopo de aplicação dos algoritmos de caminho mínimo (Ford L. R., 1956; Bellman, 1958; Schrijver, 2012). Esses avanços teóricos coincidiram com o crescimento exponencial das redes de telecomunicações e sistemas de transporte automatizados, criando demandas práticas que solidificariam o problema como um dos pilares da ciência da computação moderna (Schrijver, 2012).

O problema do caminho mínimo é um dos problemas fundamentais em ciência da computação e teoria dos grafos. Ele consiste em encontrar o caminho de menor custo entre dois nós em um grafo, onde o custo é definido pela soma dos pesos das arestas ao longo do caminho (Madkour *et al.*, 2017).

A variação mais simples desse problema é o ponto a ponto, do inglês *Peer-to-peer* (P2P), pois se trata da busca pela menor distância entre dois pontos. Uma variação comum é o caminho mais curto de fonte única, do inglês *Single Source Shortest Path* (SSSP) , onde o objetivo é encontrar o caminho mínimo de um nó de origem para todos os outros nós no grafo (Cormen Charles E. Leiserson; Stein, 2022; Goldberg *et al.*, 2005). Algoritmos como o de Dijkstra (Dijkstra, 1959) são frequentemente usados para resolver este problema, a não ser que envolvam pesos negativos, dessa forma podemos partir para uma abordagem usando o algoritmo de Bellman-Ford(Bellman, 1958), esse algoritmo não foi escalado para esse trabalho porque os grafos de entrada nesse trabalho não contém arestas com pesos negativos. Outra variação é o caminho mais curto entre todos os pares, em inglês *All Pairs Shortest Path* (APSP), que busca encontrar o caminho mínimo entre cada par de nós no grafo. O algoritmo de Floyd-Warshall

(Floyd, 1962) é uma escolha popular para este problema, pois pode lidar eficientemente com grafos densos (Cormen Charles E. Leiserson; Stein, 2022).

Tabela 1 – Tabela comparativa entre as variações do problema de caminho mínimo.

Nome da Variação	**Objetivo Principal**	**Métrica a ser Otimizada**	**Exemplo Prático**	**Algoritmos Comuns**
Ponto a Ponto (P2P)	Encontrar o caminho de um nó **A** para um nó **B**.	Minimizar a **soma dos pesos** do caminho.	Rota de GPS de um ponto a outro.	A*, Dijkstra, UCS
Fonte Única (SSSP)	Encontrar os caminhos de um nó **A** para **todos** os outros.	Minimizar a **soma dos pesos** de cada caminho.	Logística de um centro de distribuição para todos os clientes.	Dijkstra, Bellman-Ford
Todos os Pares (APSP)	Encontrar os caminhos entre **todos** os pares de nós possíveis.	Minimizar a **soma dos pesos** para cada par.	Tabela de quilometragem entre todas as cidades de um país.	Floyd-Warshall, Dijkstra a partir de cada nó

Fonte: Elaborado pelo autor.

O problema do caminho mínimo tem aplicações em diversas áreas, como redes de computadores, transporte e jogos. Em redes de computadores, por exemplo, é usado para encontrar o caminho mais eficiente para rotear pacotes de dados. No transporte, é usado para encontrar a rota mais curta ou mais rápida entre dois pontos. Em jogos, é usado para encontrar o caminho mais curto para um personagem se mover em um mapa (Cormen Charles E. Leiserson; Stein, 2022; Rosenthal, 2013; Kleinberg; Tardos, 2005).

3.2.1 Algoritmos de caminho mínimo

Existem muitos algoritmos de busca em grafos, e dentro dessa gama de algoritmos existem os de caminho mínimo; a escolha de um algoritmo que visa resolver esse problema sempre será avaliada mediante as características do problema.

3.2.2 Dijkstra

O algoritmo de Dijkstra é uma abordagem gulosa, significa ser um procedimento que seleciona a melhor opção atual em cada etapa, sem considerar consequências futuras. Ele utiliza uma fila de prioridade para calcular os caminhos mínimos de uma origem para todos os outros nós em grafos com pesos não negativos (Dijkstra, 1959). É otimizado para preencher uma linha da matriz de distância, característica que faz com que ele seja utilizado para o problema APSP.

> Informalmente, uma heurística gulosa é um procedimento que seleciona a melhor opção atual em cada etapa, sem considerar consequências futuras. Como se pode imaginar, tal abordagem raramente leva a uma solução ótima em cada instância. No entanto, há casos em que a abordagem gulosa realmente funciona.

> Nesses casos, chamamos o procedimento de algoritmo guloso.(Bondy; Murty, 1976, p. 1)

Algoritmo 1: Algoritmo de Dijkstra.

Entrada: Grafo G, nó de origem
Saida: Vetor de distâncias $dist$
para *todo vértice* $v \neq origem$ **faça**
    $dist[v] \leftarrow \infty$;
fim
$dist[origem] \leftarrow 0$;
Inicializar fila de prioridade Q com todos os nós;
enquanto Q *não vazia* **faça**
    $u \leftarrow$ nó com menor $dist$ em Q;
    Remover u de Q;
    para *cada vizinho* v *de* u **faça**
        se $dist[u] + w(u,v) < dist[v]$ **então**
            $dist[v] \leftarrow dist[u] + w(u,v)$;
            Atualizar Q com novo $dist[v]$;
        fim
    fim
fim
retorna $dist$;

Fonte: Baseado na formulação de Cormen et al. (Cormen Charles E. Leiserson; Stein, 2022, p. 654).

3.2.3 Busca de Custo Uniforme

A UCS é um algoritmo de busca cega que expande o nó com menor custo acumulado $g(n)$ utilizando uma fila de prioridade. É garantido que encontre o caminho mínimo para um destino específico, mas pode ser adaptado para calcular distâncias para todos os nós, executando-o repetidamente.

Embora originalmente projetado para encontrar o caminho mínimo até um destino específico, o algoritmo em questão pode ser adaptado para o cálculo de matrizes de distância, executando-o repetidamente a partir de cada nó como origem, permitindo determinar o menor custo entre todos os pares de vértices. No entanto, sua especialidade é o P2P.

(Felner, 2011) argumenta que os algoritmos de Dijkstra e UCS são logicamente equivalentes, compartilhando a mesma ordem de expansão baseada no custo acumulado.

Algoritmo 2: Busca de Custo Uniforme

Entrada: Um problema definido com estado inicial, ações, função de custo e teste de objetivo

Saida: Solução (caminho com menor custo) ou falha

node ← nó com *State* = estado inicial do problema, *Path-Cost* = 0
frontier ← fila de prioridade ordenada por *Path-Cost*, inicializada com *node*
explored ← conjunto vazio

enquanto *verdadeiro* **faça**
  se *frontier está vazia* **então**
    retorna falha
  fim
  node ← Remove-Min(*frontier*) **se** *node.State é estado objetivo* **então**
    retorna Solução(*node*)
  fim
  Adicione *node.State* a *explored*
  para *cada action em problem.Ações(node.State)* **faça**
    child ← Nó-Filho(*problem*, *node*, *action*)
    se *child.State não está em explored nem em frontier* **então**
      Inserir(*child*, *frontier*)
    fim
    senão se *child.State está em frontier com custo maior* **então**
      Substitua esse nó em *frontier* por *child*
    fim
  fim
fim

Fonte: Baseado na formulação apresentada por Russell e Norvig (Russell; Norvig, 2009, p. 84).

3.2.3.1 Diferenças do algoritmo UCS e Dijkstra

UCS e Dijkstra são algoritmos baseados em custo acumulado, mas têm finalidades distintas. UCS é uma busca geral, projetada para encontrar o caminho mínimo até um destino específico, enquanto Dijkstra calcula os caminhos mínimos de uma origem para todos os nós, sendo ideal para preencher uma linha da matriz de distância (Russell; Norvig, 2009). Como já declarado neste trabalho, os algoritmos são equivalentes, mas resolvem problemas específicos diferentes. O UCS é interessante quando se deseja encontrar o caminho de um ponto a outro; já o Dijkstra resolve uma linha da matriz completa.

3.2.4 Floyd-Warshall

Antes de falar do algoritmo de Floyd-Warshall, é importante entender o conceito de programação dinâmica. A programação dinâmica é uma técnica de otimização que resolve problemas complexos dividindo-os em subproblemas mais simples e armazenando os resultados intermediários para evitar

cálculos redundantes (Cormen Charles E. Leiserson; Stein, 2022, p. 362). Essa abordagem é particularmente eficaz para problemas que exibem propriedades de sobreposição de subproblemas e otimalidade de subestrutura, onde a solução ótima pode ser construída a partir de soluções ótimas de seus subproblemas.

O algoritmo de Floyd-Warshall utiliza programação dinâmica para calcular as distâncias mínimas entre todos os pares de nós, construindo diretamente a matriz de distância completa(Cormen Charles E. Leiserson; Stein, 2022, 655).

Algoritmo 3: Algoritmo de Floyd-Warshall.

```
Entrada: Grafo G com n nós
Saida: Matriz D de distâncias mínimas entre todos os pares de nós
início
    n ← número de nós;
    Inicializar matriz D com pesos das arestas (ou ∞ se não houver aresta);
    para k ← 1 até n faça
        para i ← 1 até n faça
            para j ← 1 até n faça
                se D[i][j] > D[i][k] + D[k][j] então
                    D[i][j] ← D[i][k] + D[k][j];
                fim
            fim
        fim
    fim
    retorna D;
fim
```

Fonte: Baseado na formulação apresentada por Robert Floyd (Floyd, 1962).

*3.2.5 A**

O algoritmo A* é uma busca heurística que combina o custo acumulado $g(n)$ com uma estimativa heurística $h(n)$, definindo $f(n) = g(n) + h(n)$. Para garantir a otimalidade, $h(n)$ deve ser admissível, ou seja, nunca superestimar o custo real (Cormen Charles E. Leiserson; Stein, 2022). A* é útil quando a matriz de distância serve como heurística para busca direcionada.

Algoritmo 4: Algoritmo A*.

Entrada: Grafo G, nó de origem, nó de destino, função heurística h

Saida: Caminho mínimo ou falha se não existir caminho

Inicializar $g[Origem] \leftarrow 0$;

Inicializar $f[Origem] \leftarrow h(Origem, Destino)$;

Inicializar fila de prioridade Q contendo $Origem$;

enquanto Q *não vazia* **faça**

  $u \leftarrow$ nó com menor valor f em Q;

  se $u = Destino$ **então**

    retorna reconstruir caminho de u;

  fim

  Remover u de Q;

  para *cada vizinho* v *de* u **faça**

    $tentativa\_g \leftarrow g[u] + w(u,v)$;

    se $tentativa\_g < g[v]$ **então**

      $g[v] \leftarrow tentativa\_g$;

      $f[v] \leftarrow g[v] + h(v, Destino)$;

      Atualizar Q com v e novo $f[v]$;

    fim

  fim

fim

retorna falha

Fonte: Baseado na formulação apresentada por Peter Hart (Hart *et al.*, 1968).

3.2.6 Comparações Gerais

Ao comparar algoritmos de caminho mínimo, é fundamental considerar tanto a complexidade temporal quanto a complexidade espacial, além da aderência de cada abordagem ao tipo de grafo e ao problema em questão.

De forma geral, a complexidade temporal descreve como o tempo de execução do algoritmo cresce em função do tamanho da entrada, usualmente expressa em termos do número de vértices $|V|$ e de arestas $|E|$. A notação assintótica, como $O(\cdot)$, permite abstrair detalhes de implementação e focar na ordem de grandeza do custo (Cormen Charles E. Leiserson; Stein, 2022). Já a complexidade espacial descreve a quantidade de memória adicional utilizada pelo algoritmo, além do espaço necessário para armazenar o próprio grafo, o que é particularmente relevante em aplicações com grafos de grande porte, como redes rodoviárias (Zhan; Noon, 1998).

A seguir, apresenta-se uma comparação geral entre os algoritmos UCS, Dijkstra, Floyd–Warshall

e A* sob essas três dimensões: tempo de execução, uso de memória e adequação ao problema.

- Complexidade Temporal:
  - UCS: $O((V+E)\log V)$ com *min-heap*, onde V é o número de vértices e E é o número de arestas.
  - Dijkstra: $O((V+E)\log V)$ com *min-heap*, semelhante à UCS, mas otimizado para caminhos mínimos a partir de uma origem.
  - Floyd-Warshall: $O(V^3)$, adequado para grafos densos e para gerar a matriz completa.
  - A*: Complexidade depende da heurística; pode ser mais rápido que Dijkstra para destinos específicos, mas requer uma heurística admissível.
- Uso de Memória:
  - UCS e Dijkstra: $O(V)$ para a fila de prioridade e vetor de distâncias.
  - Floyd-Warshall: $O(V^2)$ para armazenar a matriz de distância.
  - A*: $O(V)$ para a fila de prioridade, com memória adicional para a função heurística.
- Aderência ao Problema:
  - UCS: Ideal para busca com destino específico, mas pode ser adaptado para matrizes de distância com múltiplas execuções.
  - Dijkstra: Eficiente para grafos esparsos e para calcular uma linha da matriz.
  - Floyd-Warshall: Melhor para grafos densos, gerando a matriz completa diretamente.
  - A*: Eficiente para busca direcionada, utilizando a matriz de distância como heurística.

3.3 Linguagens de Programação

3.3.1 C++

C++, desenvolvido por *Bjarne Stroustrup*, é uma linguagem de alto desempenho amplamente utilizada em sistemas que exigem eficiência computacional, combinando abstrações de alto nível com controle direto sobre recursos de hardware (Stroustrup, 2013). Suas características relevantes para o UCS incluem:

- Desempenho otimizado: A compilação nativa para código de máquina e o modelo de "zero-overhead abstractions" permitem implementações altamente eficientes de algoritmos intensivos (Stroustrup, 2013).
- Estruturas de dados da STL: A *Standard Template Library* oferece `std::priority_queue`, implementada como heap binário sobre um `std::vector`, e `std::vector` para armazenamento contíguo, ambos adequados para filas de prioridade e listas de adjacência em algoritmos de caminho mínimo (Stroustrup, 2013; Cormen Charles E. Leiserson; Stein, 2022).

- Gerenciamento manual de memória: C++ disponibiliza alocação e desalocação explícitas de memória, permitindo controle fino sobre o uso de recursos, ao custo de exigir disciplina de programação para evitar vazamentos e erros de gerenciamento (Stroustrup, 2013).
- Suporte a concorrência: A biblioteca padrão inclui `std::thread` para criação de threads, `std::mutex` para exclusão mútua e tipos atômicos em `<atomic>` para operações concorrentes, oferecendo suporte direto à programação paralela em memória compartilhada (Stroustrup, 2013).

3.3.2 Concorrência com Threads

Os recursos apresentados permitem explorar paralelismo em arquiteturas multicore mantendo o controle explícito sobre a sincronização e o compartilhamento de dados entre threads.

No contexto deste trabalho, a concorrência pode ser explorada executando múltiplas instâncias do algoritmo UCS em paralelo, cada uma associada a um vértice de origem distinto ou a um subconjunto de vértices, de forma a acelerar o cálculo de matrizes de distância. Nessa abordagem, cada thread processa de maneira independente um conjunto de fontes, enquanto estruturas de sincronização, como `std::mutex`, são utilizadas apenas quando há necessidade de consolidar resultados em estruturas compartilhadas. Embora bibliotecas externas como OpenMP ofereçam diretivas de alto nível para paralelismo de dados em C++, neste projeto optou-se por utilizar exclusivamente os recursos da biblioteca padrão de C++ visando maior portabilidade e controle fino sobre o modelo de execução (Stroustrup, 2013).

3.3.3 Go (Golang)

A linguagem Go, lançada em 2009 pelo Google, foi projetada para combinar desempenho, concorrência e simplicidade, com foco em sistemas de larga escala e servidores de rede (Pike, 2012). Go é uma linguagem compilada, produzindo binários nativos sem dependência de máquina virtual, o que permite desempenho competitivo com C e C++ em muitos cenários de aplicação (Pike, 2012). Suas características relevantes para o UCS incluem:

- Compilação nativa: O compilador de Go gera executáveis estáticos e portáveis, permitindo execução eficiente em diferentes plataformas, com tempos de compilação rápidos e desempenho em tempo de execução adequado para algoritmos intensivos em CPU (Pike, 2012).
- Concorrência: Go adota um modelo de concorrência baseado em *goroutines* (threads leves gerenciadas pelo runtime) e *channels*, seguindo o princípio “não se comunique compartilhando memória; compartilhe memória se comunicando” (Pike, 2012). Goroutines são significativamente mais leves que threads do sistema operacional, podendo iniciar com pilhas na ordem de alguns

kilobytes e crescer de forma dinâmica, o que permite a criação de milhares ou milhões de unidades concorrentes com baixo overhead (Pike, 2012; Authors, 2023).

- Gerenciamento de memória: Go utiliza um *garbage collector* concorrente, que libera o programador da gestão manual de memória, reduzindo a probabilidade de vazamentos e erros de apontadores (Pike, 2012; Go Authors, 2017). Por outro lado, em aplicações com requisitos de latência muito rígidos, as pausas e a atividade do coletor podem introduzir variações de tempo de resposta, exigindo cuidado com o padrão de alocação e, eventualmente, ajustes de parâmetros de GC.
- Estruturas de dados nativas: Slices (`[]T`) e mapas (`map[K]V`) são estruturas de dados centrais em Go, projetadas para oferecer acesso eficiente e gerenciamento automático de crescimento de capacidade (Pike, 2012). Slices fornecem visualização dinâmica sobre arranjos subjacentes, com bom desempenho em percursos sequenciais, enquanto mapas são implementados como tabelas de dispersão, otimizadas para busca e inserção em tempo médio constante, características úteis na representação de listas de adjacência e tabelas auxiliares durante a execução do UCS.

3.3.4 Concorrência com Goroutines e Canais

Go oferece suporte nativo à concorrência por meio de *goroutines* e canais. Uma goroutine é uma função executada concorrentemente (Pike, 2012).

Canais (`chan`) implementam filas tipadas para comunicação e sincronização entre goroutines, seguindo o modelo de passagem de mensagens em vez de compartilhamento explícito de memória (Pike, 2012). Quando necessário, o pacote `sync` oferece primitivas adicionais, como `sync.Mutex` e `sync.WaitGroup`, para proteção de regiões críticas e coordenação de múltiplas goroutines (Pike, 2012).

No contexto do UCS, esse modelo permite distribuir o cálculo de distâncias entre múltiplas goroutines, por exemplo executando o algoritmo em paralelo para diferentes vértices de origem ou partições do conjunto de consultas. Os resultados parciais podem ser enviados por canais para uma goroutine agregadora responsável por preencher uma matriz global de distâncias, reduzindo o tempo total de processamento em ambientes multicore.

4 TRABALHOS RELACIONADOS

Neste capítulo o foco não é revisitar a fundamentação teórica, mas destacar como diferentes pesquisadores têm implementado e avaliado algoritmos de caminho mínimo em cenários práticos, em especial em redes rodoviárias de grande porte e sob restrições de desempenho e memória. Depois de uma vasta pesquisa, não foi encontrado trabalho que esteja explicitamente relacionado à proposta deste trabalho de forma que pudesse ser realizada uma comparação direta. No entanto, vamos apresentar alguns trabalhos inseridos nesse contexto que podem ser considerados relevantes para essa pesquisa.

4.1 Implementações práticas em redes rodoviárias

O *9th DIMACS Implementation Challenge*, organizado por Demetrescu, Goldberg e Johnson, estabeleceu um marco na avaliação de algoritmos de caminho mínimo em redes rodoviárias. Ao padronizar instâncias e condições experimentais, o desafio viabilizou a comparação direta entre diversas implementações e metodologias desenvolvidas pela comunidade científica. As instâncias disponibilizadas são derivadas de dados reais do Censo dos EUA do ano 2000 (Bureau, 2006).

Tais instâncias trazem à tona problemas práticos que raramente aparecem em exemplos didáticos, como limitações de memória principal, efeitos de hierarquia de cache e custos de I/O, o que torna esse conjunto de dados particularmente adequado para estudos focados em implementação e desempenho. Por esse motivo, o presente trabalho adota essas mesmas instâncias como base experimental, para que seus resultados possam ser interpretados à luz de uma literatura já consolidada.

Zhan e Noon, por exemplo, avaliaram quinze variantes de algoritmos de caminho mínimo em redes rodoviárias reais, demonstrando que propriedades estruturais dos grafos de transporte influenciam significativamente o comportamento empírico dos algoritmos (Zhan; Noon, 1998). Trabalhos mais recentes, como o de Qiu et al., utilizam as mesmas redes para propor métodos eficientes de contagem de caminhos mínimos, reforçando o papel dessas instâncias como benchmarks padrão em pesquisa aplicada (Qiu, 2022).

4.2 Estudos recentes de desempenho e paralelização

Paralelamente às técnicas de preprocessamento e aceleração hierárquica, uma linha importante da literatura investiga a paralelização de algoritmos de caminho mínimo em arquiteturas modernas, como GPUs e processadores many-core. Implementações de Dijkstra e A* em CUDA, por exemplo, mostram ganhos de até três ordens de magnitude em relação a versões sequenciais em CPU quando aplicadas a grafos muito grandes, embora com *overhead* considerável de transferência de dados entre

CPU e GPU em instâncias menores.

Beletsky e Petrov analisam, em ambiente de GPU, três algoritmos fundamentais (Bellman-Ford, Dijkstra e Floyd-Warshall), explorando otimizações específicas como uso intensivo de memória compartilhada, coalescimento de acessos e estratégias de terminação antecipada. Esses estudos evidenciam que o desempenho prático de algoritmos de caminho mínimo depende fortemente da interação entre o algoritmo, a estrutura de dados adotada e a arquitetura de hardware subjacente. (Bodra; Khairnar, 2025).

Embora este trabalho não explore paralelização em GPU ou preprocessamento avançado, essa literatura é relevante porque destaca um ponto central: resultados teóricos de complexidade assintótica não são suficientes para predizer desempenho em implementações reais. De forma análoga, o presente estudo investiga como a escolha da linguagem de programação e de suas construções nativas impacta o desempenho do UCS em cenários realistas de P2P, mesmo quando o algoritmo abstrato é o mesmo.

4.3 Comparações de linguagens de programação

Estudos de alto desempenho como o de Costanza, Herzeel e Verachtert comparam C++, Go e Java no processamento de dados biológicos. Os autores observaram que Go oferece um equilíbrio favorável entre execução e memória, facilitando o desenvolvimento ao mitigar erros de gestão manual. Em contrapartida, o C++ demonstrou exigir um esforço de otimização significativamente maior para atingir desempenho competitivo (Costanza *et al.*, 2019).

Stein e Geyer-Schulz avaliaram o desempenho de C++, Java, C#, F#, e Python em *clustering* de grafos utilizando dados do DIMACS. Os resultados confirmaram a superioridade do C++ em tempo de execução, mas ressaltaram que linguagens de alto nível reduzem significativamente a complexidade e a extensão do código. Tal cenário evidencia um *trade-off* entre performance e produtividade, justificando a análise desse equilíbrio em algoritmos de caminho mínimo para redes de grande escala (Stein; Geyer-Schulz, 2013).

Ao avaliarem oito linguagens de diversos paradigmas, Nanz e Furia demonstraram que o C e o Go lideram em eficiência de tempo e memória, especialmente em cenários de alta carga. Um achado relevante do estudo é que a vantagem competitiva das linguagens de baixo nível reduz-se em entradas de menor porte, onde sistemas com runtime leve tornam-se competitivos. Tais resultados reforçam a necessidade de alinhar a escolha da linguagem ao perfil dos dados de entrada, critério essencial para o processamento de grafos em larga escala (Nanz; Furia, 2014).

4.4 Síntese e posicionamento deste trabalho

A partir desse mapeamento da literatura, é possível destacar que existe uma base sólida de benchmarks e instâncias padronizadas para avaliação de algoritmos de caminho mínimo em redes rodoviárias de grande escala, há um esforço contínuo em projetar algoritmos e estruturas de dados cada vez mais sofisticados e estudos comparativos de linguagens de programação mostram que escolhas de linguagem e modelo de memória têm impacto significativo em desempenho e produtividade, mas não foi encontrado nenhum trabalho que compare diretamente implementações do UCS em C++ e Go.

O presente trabalho busca preencher essa lacuna ao implementar o UCS de forma equivalente em C++ e Go, empregando representações de grafos e estruturas de dados comparáveis, e avaliando o desempenho em termos de tempo de execução, uso de memória e escalabilidade nas instâncias USA, CTR, W, E e NY do DIMACS Challenge. Dessa forma, este trabalho contribui tanto para a comunidade de algoritmos, ao fornecer dados empíricos sobre a implementação de UCS em linguagens contemporâneas, quanto para a comunidade de engenharia de software, ao discutir os impactos práticos de decisões de projeto em um problema clássico de grafos de grande escala.

5 PROCEDIMENTOS METODOLÓGICOS

Esta seção descreve os procedimentos metodológicos adotados para a implementação e análise de desempenho do algoritmo. Serão detalhados o ambiente de desenvolvimento, os conjuntos de dados utilizados, a arquitetura das implementações em Go e C++, os cenários de teste e as métricas de avaliação que fundamentam a análise comparativa.

5.1 Ambiente Experimental

Os experimentos foram conduzidos em um ambiente computacional controlado para garantir a reprodutibilidade e confiabilidade dos resultados obtidos.

Em hardware as especificações incluem:

- Modelo: Dell Inc. G7 7588;
- Processador: Intel® Core™ i7-8750H CPU @ 2.20GHz × 12;
- Memória principal: 32 GB DDR4-2666 de RAM (Dual channel);
- Memória secundária: SSD de 1 TB;
- Placa de vídeo: NVIDIA GeForce GTX 1050 Ti.

Em software as especificações incluem:

- Sistema Operacional: Pop! OS 22.04 LTS;
- Kernel: 6.17.4-76061704-generic;
- Linguagem Go: Versão 1.24.4 (amd64/linux);
- Compilador C++: gcc (Ubuntu 11.4.0-1ubuntu1 22.04.2) 11.4.0.

Para garantir a reprodutibilidade e reduzir o ruído computacional (*jitter*), adotou-se um protocolo rigoroso de controle do ambiente. O escalonamento dinâmico de frequência e o *Turbo Boost* foram desativados, fixando-se o foco de CPU em *performance* e o *clock* em 2,2 GHz. O isolamento de recursos foi garantido via afinidade de CPU, restringindo a execução a núcleos físicos.

Para minimizar interferências externas, suspendeu-se o gerenciador de interface gráfica e ajustou-se a prioridade de I/O. Além disso, o estado da memória foi padronizado antes de cada execução através da limpeza.

5.2 Modelagem dos Grafos

Os grafos utilizados neste estudo serão obtidos a partir dos conjuntos de dados disponibilizados pela *9th DIMACS Implementation Challenge - Shortest Paths* (DIMACS, 2006). Esses conjuntos incluem redes rodoviárias reais de *New York*, Oeste, Leste, Centro e região completa dos Estados Unidos.

Os grafos variam em quantidade de nós e densidade de arestas, cobrindo cenários. Sendo assim os pesos das arestas são não-negativos, representando distâncias ou custos, conforme especificado nos formatos de arquivo do desafio.

Tabela 2 – Entradas do problema

Nome	Descrição	Nós	Arcos
USA	USA	23.947.347	58.333.344
CTR	Centro dos EUA	14.081.816	34.292.496
W	Oeste dos EUA	6.262.104	15.248.146
E	Leste dos EUA	3.598.623	8.778.114
NY	Cidade de New York	264.346	733.846

Fonte: elaborada pelo autor a partir de (DIMACS, 2006).

5.3 Implementação dos Algoritmos

As implementações dos algoritmos foram desenvolvidas nas linguagens Go e C++, aproveitando suas capacidades de paralelismo e gerenciamento de memória. A seguir, são descritas as principais características das implementações:

- Go: Utilização de goroutines para paralelismo leve e canais para comunicação entre threads. A implementação do UCS foi estruturada para permitir a execução concorrente de múltiplas buscas a partir de diferentes nós de origem.
- C++: Uso da biblioteca padrão de threads para criar múltiplas processos de execução. A implementação do UCS foi adaptada para suportar a execução paralela, garantindo a integridade dos dados compartilhados por meio de mutexes e outras primitivas de sincronização.

No entando a base abstrata do algoritmos é a mesma para ambas as linguagens, garantindo que as diferenças de desempenho sejam atribuídas principalmente às características das linguagens e suas implementações de paralelismo.

5.4 Métricas de Avaliação

Para a avaliação comparativa das implementações, foram estabelecidas as seguintes métricas de desempenho:

- Tempo de Execução: Medido em milissegundos utilizando ferramentas específicas de cada linguagem `time.Now()` em Go e `std::chrono` em C++ para capturar o tempo total de execução do algoritmo, comparando as implementações. Será avaliado o tempo médio para cada entrada.
- Consumo de Memória: Medido usando as ferramentas do próprio sistema operacional, o sistema

de arquivos proc[1] funciona como uma interface para estruturas de dados internas no kernel, dessa forma é possível monitorar de forma precisa o uso de memória durante a execução dos algoritmos. Será avaliado o consumo máximo de memória durante a execução do UCS em diferentes tamanhos de grafos.

5.4.1 Estratégia de Avaliação

- Seleção de Grafos: Serão selecionados subconjuntos representativos dos grafos do DIMACS, incluindo redes rodoviárias reais.
- UCS: Executado para um conjunto de 200 nós de origem selecionados aleatoriamente, construindo linhas parciais da matriz de distância.
- Procedimento Experimental: O UCS será executado 30 vezes por grafo para reduzir variabilidade, com tempos de execução e consumo de memória detalhamente registrados. A análise estatística incluirá média, mediana, desvio padrão, o valor mínimo e máximo.

[1] https://docs.kernel.org/filesystems/proc.html

6 CRONOGRAMA

6.1 Cronograma de Desenvolvimento

Nesta seção, definimos as atividades realizadas na primeira etapa do projeto, correspondente ao primeiro semestre de 2025.

Tabela 3 – Atividades realizadas no primeiro semestre de 2025.

Etapas de Desenvolvimento	Tempo de Desenvolvimento			
	Abril	Maio	Junho	Julho
Pesquisa e fundamentação teórica	X	X	X	
Estudo das linguagens Go, C++ e Rust			X	X
Análise dos grafos do DIMACS		X		
Implementação inicial dos algoritmos		X	X	X

Fonte: elaborada pelo autor.

Tendo em vista as atividades realizadas durante o primeiro semestre de 2025, essas foram as atividades realizadas durante o segundo semestre.

Tabela 4 – Atividades realizadas no segundo semestre de 2025.

Etapas de Desenvolvimento	Tempo de Desenvolvimento			
	Setembro	Novembro	Dezembro	Janeiro
Otimização de implementações paralelas	X			
Testes de desempenho com grafos DIMACS	X	X		
Validação dos resultados apresentados		X	X	
Análise comparativa com os resultados das implementações			X	X
Documentação e publicação o trabalho			X	X

Fonte: elaborada pelo autor.

7 RESULTADOS E ANÁLISE

Nesta seção, apresentamos os resultados obtidos a partir dos experimentos conduzidos com as implementações paralelas do algoritmo UCS em Go e C++. Serão detalhados os tempos de execução, consumo de memória e outras métricas relevantes, seguidos por uma análise comparativa entre as duas abordagens.

7.1 Resultado referente Golang

A tabela 5 apresenta as estatísticas descritivas dos tempos de execução da implementação em Go para cada uma das instâncias DIMACS, considerando 30 execuções independentes por instância. Observa-se que os tempos médios variam de aproximadamente 3,8 segundos, na instância NY, até cerca de 326 segundos na instância CTR, refletindo o aumento do custo computacional à medida que o tamanho do grafo cresce.

De modo geral, as medidas de média e mediana são muito próximas em todas as instâncias, indicando uma distribuição de tempos aproximadamente simétrica, sem presença evidente de execuções extremamente atípicas. Os valores de desvio padrão são baixos em relação às respectivas médias (por exemplo, cerca de 0,017 segundos para NY e 0,482 segundos para USA), o que sugere uma variabilidade reduzida entre as 30 repetições e boa estabilidade temporal da implementação.

Os tempos mínimos e máximos observados para cada instância delimitam o intervalo de variação das execuções, permanecendo sempre próximos da média. Por exemplo, na instância USA, os tempos variaram de 104,753 a 106,969 segundos, enquanto na instância W variaram de 104,677 a 106,682 segundos, o que reforça a consistência do comportamento da implementação em Go sob condições experimentais repetidas.

Tabela 5 – Estatísticas descritivas do tempo de execução Go

Instância	Média	Mediana	Desvio Padrão	Mín	Máx
USA	105.856	105.841	0.482	104.753	106.969
CTR	325.994	326.108	1.687	321.867	329.665
E	57.230	57.154	0.363	56.773	58.332
NY	3.819	3.816	0.017	3.795	3.857
W	105.548	105.458	0.490	104.677	106.682

Fonte: Elaborada pelo autor.

7.2 Resultado referente C++

A tabela 6 apresenta as estatísticas descritivas dos tempos de execução da implementação em C++ para cada instância DIMACS, considerando 30 execuções por cenário. Observa-se que os tempos

médios variam de aproximadamente 1,4 segundo na instância NY até cerca de 176 segundos na instância CTR, refletindo o aumento do custo computacional conforme cresce a dimensão do grafo.

De maneira geral, as medidas de média e mediana são bastante próximas em todas as instâncias, o que indica uma distribuição de tempos de execução aproximadamente simétrica, sem ocorrência evidente de valores extremos isolados. Os desvios padrão também se mantêm baixos em relação às respectivas médias (por exemplo, cerca de 0,011 segundo para NY e 0,066 segundo para USA), sugerindo uma variabilidade reduzida entre as repetições e boa estabilidade temporal da implementação em C++.

Os valores mínimos e máximos registrados delimitam intervalos de variação estreitos para cada instância, reforçando a consistência do comportamento observado. Na instância USA, por exemplo, os tempos variaram entre 47,881 e 48,169 segundos, enquanto na instância W oscilaram entre 47,155 e 48,002 segundos, o que indica que, sob as mesmas condições experimentais, o desempenho em C++ permanece praticamente constante ao longo das 30 execuções.

Tabela 6 – Estatísticas descritivas do tempo de execução C++

Instância	Média	Mediana	Desvio Padrão	Mín	Máx
USA	47.993	47.992	0.066	47.881	48.169
CTR	176.368	176.466	1.287	170.348	177.677
E	24.969	24.929	0.213	24.757	25.542
NY	1.396	1.392	0.011	1.385	1.433
W	47.733	47.762	0.202	47.155	48.002

Fonte: Elaborada pelo autor.

7.3 Análise de Consumo de Memória

Além do tempo de execução, foi avaliado o consumo de memória por meio do pico de memória residente (Peak RSS) em cada execução, utilizando o sistema de arquivo proc para avaliar as informações de memória. Essa métrica captura o maior valor de memória física consumida pelo processo durante sua execução, permitindo comparar a eficiência no uso de recursos entre Go e C++.

7.3.1 Consumo de memória em Go

A tabela 7 apresenta as estatísticas descritivas do pico de memória residente para a implementação em Go, considerando 30 execuções independentes por instância. Observa-se que o consumo médio de memória varia significativamente com o tamanho do grafo, partindo de aproximadamente 79,2 MB na instância NY até 4.182,1 MB na instância CTR.

Um aspecto notável é a elevada variabilidade no consumo de memória em Go, especialmente nas instâncias maiores. Na instância CTR, por exemplo, o desvio padrão foi de 246,4 MB, o que

corresponde a aproximadamente 5,89% da média; já na instância USA, o desvio padrão foi de 90,1 MB (5,26% da média). Essa variação sugere que o gerenciador de memória do Go (garbage collector) introduz flutuações significativas no pico de memória entre execuções, possivelmente devido ao timing variável das coletas de lixo.

Os valores mínimo e máximo mostram amplitudes consideráveis. Na instância CTR, por exemplo, o consumo variou de 3.710,98 MB a 4.850,59 MB, uma diferença de mais de 1.100 MB. Esse padrão de alta variabilidade em Go está presente em todas as instâncias, indicando que o comportamento do garbage collector introduz incerteza significativa no consumo de memória durante a execução do algoritmo.

Tabela 7 – Estatísticas descritivas do pico de memória (MB) - Go

Instância	Média	Mediana	Desvio Padrão	Mín	Máx
USA	1.713,46	1.707,05	90,10	1.552,85	1.891,09
CTR	4.182,08	4.176,58	246,42	3.710,98	4.850,59
E	989,99	996,98	41,69	917,96	1.090,68
NY	79,20	78,29	3,14	74,77	87,24
W	1.725,61	1.712,00	116,80	1.540,99	2.026,02

Fonte: Elaborada pelo autor.

7.3.2 Consumo de memória em C++

A tabela 8 apresenta as estatísticas descritivas do pico de memória residente para a implementação em C++. Em contraste com Go, C++ apresenta um consumo de memória substancialmente menor e, mais importante, muito mais estável.

O pico médio de memória varia de 44,74 MB em NY a 2.037,84 MB em CTR, sendo sistemática e significativamente inferior aos valores observados em Go. O desvio padrão, porém, é notavelmente reduzido em todas as instâncias, com valores típicos na faixa de 0,06 MB (menos de 1% da média em praticamente todas as instâncias), indicando comportamento muito mais previsível e controlado do uso de memória.

Os intervalos entre mínimo e máximo são extremamente estreitos em C++. Na instância CTR, por exemplo, o consumo variou de apenas 2.037,80 MB a 2.037,88 MB, uma amplitude de apenas 0,08 MB. Esse padrão se repete consistentemente em todas as instâncias, demonstrando que C++ oferece uso de memória determinístico e altamente previsível, sem as flutuações introduzidas pelo garbage collector dinâmico presente em Go.

Tabela 8 – Estatísticas descritivas do pico de memória (MB) - C++

Instância	Média	Mediana	Desvio Padrão	Mín	Máx
USA	909,89	909,92	0,06	909,79	909,98
CTR	2.037,84	2.037,83	0,02	2.037,80	2.037,88
E	526,83	526,79	0,07	526,75	526,94
NY	44,74	44,74	0,05	44,65	44,86
W	909,84	909,85	0,06	909,72	909,91

Fonte: Elaborada pelo autor.

7.4 Comparação dos Resultados

7.4.1 Comparação de Tempo de Execução

A Tabela 9 sintetiza os resultados de tempo de execução obtidos pelas implementações em Go e C++ para todas as instâncias consideradas. Em todas as instâncias, a implementação em C++ apresentou tempos médios e medianos inferiores aos da implementação em Go, indicando desempenho superior na execução do algoritmo UCS sob as condições experimentais adotadas. Por exemplo, na instância USA o tempo médio em C++ foi de aproximadamente 47,99 segundos, contra 105,86 segundos em Go, enquanto na instância NY os tempos médios foram de cerca de 1,40 e 3,82 segundos, respectivamente.

As medidas de dispersão também apontam para um comportamento estável em ambas as linguagens, com desvios padrão relativamente baixos em relação às médias, porém ligeiramente menores em C++ na maioria das instâncias. Na instância USA, por exemplo, o desvio padrão foi de 0,066 segundos em C++, contra 0,482 segundos em Go, enquanto na instância W os desvios foram de 0,202 e 0,490 segundos, respectivamente, sugerindo que além de mais rápida, a implementação em C++ apresentou menor variabilidade entre as 30 execuções.

Observa-se ainda que os valores mínimos e máximos seguem o mesmo padrão observado nas médias: para todas as instâncias, os menores e maiores tempos registrados em C++ permanecem sistematicamente abaixo dos correspondentes tempos em Go. Esse comportamento consistente ao longo das diferentes instâncias desde NY, o menor grafo, até USA e CTR, os maiores indica que a vantagem de C++ se mantém à medida que o tamanho do grafo cresce, o que reforça a conclusão de que, neste contexto experimental, C++ oferece melhor desempenho para a implementação do algoritmo UCS do que Go.

Tabela 9 – Comparativo de Tempo de Execução: Go vs C++ (segundos)

Instância	Média		Mediana		Desvio Padrão		Máx		Min		Speedup
	Go	C++	Go	C++	Go	C++	Go	C++	Go	C++	
USA	105,856	47,993	105,841	47,992	0,482	0,066	106,969	48,169	104,753	47,881	2,21x
CTR	325,994	176,368	326,108	176,466	1,687	1,287	329,665	177,677	321,867	170,348	1,85x
E	57,230	24,969	57,154	24,929	0,363	0,213	58,332	25,542	56,773	24,757	2,29x
NY	3,819	1,396	3,816	1,392	0,017	0,011	3,857	1,433	3,795	1,385	2,74x
W	105,548	47,733	105,458	47,762	0,490	0,202	106,682	48,002	104,677	47,155	2,21x

Fonte: Elaborada pelo autor.

7.4.2 Comparação de Consumo de Memória

A Tabela 10 apresenta a comparação lado a lado do consumo médio de memória entre Go e C++ para cada instância. Destaca-se que a implementação em C++ consome sistematicamente menos memória que Go em todas as instâncias, com diferenças percentuais variando de aproximadamente 77% a 105%.

Tabela 10 – Comparativo Estatístico de Pico de Memória (RSS) em MB: Go vs C++

Instância	Média		Mediana		Desvio Padrão		Máx		Min		Economia
	Go	C++	Go	C++	Go	C++	Go	C++	Go	C++	
USA	1.713,46	909,89	1.707,05	909,92	90,10	0,06	1.891,09	909,98	1.552,85	909,79	46,90%
CTR	4.182,08	2.037,84	4.176,58	2.037,83	246,42	0,02	4.850,59	2.037,88	3.710,98	2.037,80	51,27%
E	990,00	526,83	996,98	526,79	41,69	0,07	1.090,68	526,94	917,96	526,75	46,78%
NY	79,20	44,74	78,29	44,74	3,14	0,05	87,24	44,86	74,77	44,65	43,51%
W	1.725,61	909,84	1.712,00	909,85	116,80	0,06	2.026,02	909,91	1.541,00	909,72	47,27%

Fonte: Elaborada pelo autor.

Destaca-se que a implementação em C++ consome sistematicamente menos memória que Go, apresentando uma economia de recursos que varia entre 43% e 51%. Sob outra perspectiva, a implementação em Go chega a utilizar até 105% mais memória que a versão em C++ (como observado na instância CTR).

Em segundo lugar, a estabilidade do uso de memória é dramaticamente diferente entre as linguagens. Enquanto C++ apresenta desvio padrão praticamente negligenciável (menor que 0,1% em relação à média), Go apresenta variações substanciais (3% a 6% em relação à média), principalmente nas instâncias maiores. Essa diferença é atribuível ao modelo de gerenciamento de memória: C++ utiliza alocação manual determinística (respeitando o escopo das estruturas de dados), enquanto Go depende de um garbage collector que opera em intervalos variáveis, ocasionando flutuações significativas no pico de memória residente entre execuções repetidas.

Esse resultado tem implicações práticas importantes: em ambientes com restrições de memória ou requisitos de previsibilidade, C++ oferece não apenas menor consumo absoluto, mas também comportamento muito mais previsível e determinístico, facilitando o planejamento de recursos e o dimensionamento de sistemas.

7.4.3 Análise Integrada: Tempo e Memória

Combinando as análises de tempo e memória, observa-se que C++ apresenta vantagem em ambas as dimensões: executa mais rapidamente (com speedup variando entre 2,7x em NY e 1,85x em CTR) e consome menos memória (redução entre 77% e 105%). Esse resultado sugere que as características estruturais de C++ compilação otimizada, gerenciamento de memória explícito, e menos overhead de runtime beneficiam simultaneamente o desempenho temporal e a eficiência de recursos.

Por outro lado, Go mantém um desempenho razoável e previsível em tempo de execução, com estabilidade temporal comparável a C++, mas incorre em overhead significativo de memória devido ao seu garbage collector e estruturas internas de runtime. A escolha entre as linguagens deve considerar, portanto, não apenas o desempenho puro, mas também as restrições de recursos disponíveis e os requisitos de previsibilidade do sistema.

8 CONCLUSÃO

Este trabalho investigou a implementação paralela do algoritmo UCS (Uniform Cost Search) em duas linguagens de programação modernas: Go e C++. O objetivo foi comparar o desempenho, eficiência de memória e adequação de cada linguagem para resolver problemas de busca em grafos de grande escala, particularmente usando instâncias DIMACS que variam desde pequenos grafos (NY com milhares de vértices) até grafos de larga escala (CTR com milhões de vértices).

8.1 Considerações Finais

Este trabalho demonstrou que, para implementação paralela do algoritmo UCS em cenários convencionais de CPU, C++ oferece vantagem consistente e significativa, tanto em tempo de execução (speedups de 1,84x a 2,74x) quanto em eficiência de memória (redução de 43,51% a 51,27%). A superioridade de C++ é atribuível a compilação otimizada e gerenciamento de memória determinístico.

No entanto, Go demonstrou ser uma alternativa viável e produtiva, particularmente quando levadas em conta facilidade de desenvolvimento, modelo de concorrência intuitivo, e portabilidade. A escolha entre as linguagens deve ser guiada pelos requisitos específicos da aplicação: se performance máxima e previsibilidade são críticas, C++ é preferível; se velocidade de desenvolvimento e manutenibilidade são prioridades, Go oferece melhor custo-benefício.

Os resultados deste trabalho contribuem para informar decisões de arquitetura em sistemas que implementam busca em grafos, fornecendo dados empíricos concretos sobre o tradeoff desempenho versus produtividade entre duas linguagens amplamente adotadas na indústria e academia.

8.2 Trabalhos Futuros

Como continuidade deste trabalho, pretende-se integrar as implementações paralelas do algoritmo UCS a um *driver* de ferramentas de roteirização, aproximando os experimentos do contexto de uso em sistemas reais. Essa integração permitirá avaliar o comportamento das soluções em cenários de consulta de rotas, com dados e restrições de negócio mais próximos de aplicações de logística e navegação.

Além disso, planeja-se estender o *driver* para suportar outros algoritmos de busca em grafos, como Dijkstra e A*, possibilitando comparar o impacto da escolha de linguagem (Go ou C++) em um ambiente unificado de roteirização.

REFERÊNCIAS

AUTHORS, G. **Goroutine stacks and scheduling**. 2023. Disponível em: https://go.dev/doc/faq#goroutines. Acesso em: 26 jan. 2026.

BELLMAN, R. On a routing problem. **Quarterly of Applied Mathematics**, Brown University, v. 16, p. 87–90, 1958.

BODRA, D.; KHAIRNAR, S. Accelerating and analyzing performance of shortest path algorithms on gpu using cuda platform: Bellman-ford, dijkstra, and floyd-warshall algorithms. **Scientific and Technical Journal of Information Technologies, Mechanics and Optics**, 2025. Disponível em: https://ntv.elpub.ru/jour/article/view/516. Acesso em: 26 jan. 2026.

BONDY, J. A.; MURTY, U. S. R. **Graph Theory with Applications**. New York: Elsevier Science Publishing Co., 1976. 1 p.

BONDY, J. A.; MURTY, U. S. R. **Graph Theory**. [*S. l.*]: Springer, 2008.

BUREAU, U. C. **TIGER/Line Files, 2006**. 2006. DIMACS Implementation Challenge. Disponível em: https://www.diag.uniroma1.it/challenge9/data/tiger/. Acesso em: 26 jan. 2026.

CORMEN CHARLES E. LEISERSON, R. L. R. T. H.; STEIN, C. **Introduction to Algorithms, Fourth Edition**. [*S. l.*]: The MIT Press, 2022. 785 p.

COSTANZA, P.; HERZEEL, C.; VERACHTERT, W. Comparing ease of programming in c++, go, and java for implementing a next-generation sequencing tool. **Evolutionary Bioinformatics**, SAGE Publications, v. 15, p. 117693431986901, jan 2019.

DIESTEL, R. **Graph Theory**. [*S. l.*]: Springer Berlin, 2017. v. 173. ISBN 978-3-662-53621-6.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numerische Mathematik**, v. 1, n. 1, p. 269–271, 1959.

DIMACS. **9th DIMACS Implementation Challenge: Shortest Paths**. 2006. Disponível em: https://www.diag.uniroma1.it/~challenge9. Acesso em: 26 jan. 2026.

FELNER, A. Dijkstra's algorithm vs. uniform cost search or a case against dijkstra's algorithm. **Ben-Gurion University Technical Report**, 2011. Disponível em: https://tzin.bgu.ac.il/~felner/2011/dikstra.pdf. Acesso em: 22 jul. 2025.

FLOYD, R. W. Algorithm 97: Shortest path. **Communications of the ACM**, v. 5, n. 6, p. 345, 1962.

FORD L. R., J. **Network Flow Theory**. Santa Monica, California, 1956.

Go Authors. **A Guide to the Go Garbage Collector**. 2017. Disponível em: https://go.dev/doc/gc-guide. Acesso em: 26 jan. 2026.

GOLDBERG, A. V.; KAPLAN, H.; WERNECK, R. F. Reach for a*: Efficient point-to-point shortest path algorithms. Redmond, n. MSR-TR-2005-132, oct 2005.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE Transactions on Systems Science and Cybernetics**, v. 4, n. 2, p. 100–107, 1968.

KLEINBERG, J.; TARDOS, E. **Algorithm Design**. Boston, MA, USA: Addison-Wesley, 2005. ISBN 978-0-321-29535-4.

MADKOUR, A.; AREF, W. G.; REHMAN, F. U.; RAHMAN, M. A.; BASALAMAH, S. **A Survey of Shortest-Path Algorithms**. 2017. Disponível em: https://arxiv.org/abs/1705.02044. Acesso em: 26 jan. 2026.

NANZ, S.; FURIA, C. A. A comparative study of programming languages in rosetta code. **2015 IEEE/ACM 37th IEEE International Conference on Software Engineering**, v. 1, p. 778–788, 2014. Disponível em: https://api.semanticscholar.org/CorpusID:2570311. Acesso em: 26 jan. 2026.

PIKE, R. **Go at Google: Language Design in the Service of Software Engineering**. 2012. Disponível em: https://talks.golang.org/2012/splash.article. Acesso em: 01 jul. 2025.

QIU, Y.-X. Efficient shortest path counting on large road networks. **Proc. VLDB Endow.**, v. 15, p. 2098–2110, 2022. Disponível em: https://www.vldb.org/pvldb/vol15/p2098-qiu.pdf. Acesso em: 26 jan. 2026.

ROSENTHAL, E. C. Shortest path games. **European Journal of Operational Research**, v. 224, n. 1, p. 132–140, 2013. ISSN 0377-2217. Disponível em: https://www.sciencedirect.com/science/article/pii/S0377221712005346. Acesso em: 26 jan. 2026.

RUSSELL, S. J.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. 3. ed. Upper Saddle River, NJ: Pearson, 2009. 84 p. ISBN 978-0-13-604259-4.

SACHS, H. Euler's konigsberg letters. **Journal of Graph Theory**, Scientific American, v. 12, p. 133–139, 1958.

SCHRIJVER, A. On the history of the shortest path problem. **Documenta Mathematica, Extra Volume ISMP**, p. 155–167, 2012.

STEIN, M.; GEYER-SCHULZ, A. A comparison of five programming languages in a graph clustering scenario. **J. Univers. Comput. Sci.**, v. 19, p. 428–456, 2013. Disponível em: https://api.semanticscholar.org/CorpusID:8963981. Acesso em: 26 jan. 2026.

STROUSTRUP, B. **The C++ Programming Language**. 4th. ed. [*S. l.*]: Addison-Wesley, 2013. ISBN 9780321563842.

ZHAN, F. B.; NOON, C. E. Shortest path algorithms: An evaluation using real road networks. **Transportation Science**, v. 32, n. 1, p. 65–73, 1998. Disponível em: https://doi.org/10.1287/trsc.32.1.65. Acesso em: 26 jan. 2026.