Universidade Federal da Bahia

Programa de Pós-Graduação em Engenharia Elétrica – PPGEE

Análise e construção de aceleradores em hardware para o cálculo do menor caminho em planejamento de rotas de robôs

Linton Thiago Costa Esteves

Tese de Doutorado do Programa de Pós-Graduação em Engenharia Elétrica (PPGEE)

Universidade Federal da Bahia – Salvador
Outubro de 2024

Linton Thiago Costa Esteves

Análise e construção de aceleradores em hardware para o cálculo do menor caminho em planejamento de rotas de robôs

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica – PPGEE, como parte dos requisitos para obtenção do título de Doutor em Engenharia Elétrica. *VERSÃO REVISADA*

Área de Concentração: Processamento da Informação e Energia

Orientadores: Prof. Dr. Wagner Luiz Alves de Oliveira / Prof. Dr. Paulo César Machado de Abreu Farias

Universidade Federal da Bahia – Salvador
Outubro de 2024

ANÁLISE E CONSTRUÇÃO DE ACELERADORES EM HARDWARE PARA O CÁLCULO DO MENOR CAMINHO EM PLANEJAMENTO DE ROTAS DE ROBÔS

Linton Thiago Costa Esteves

TESE SUBMETIDA AO PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA (PPGEE) DA UNIVERSIDADE FEDERAL DA BAHIA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM ENGENHARIA ELÉTRICA.

Aprovada por:

Documento assinado digitalmente
gov.br WAGNER LUIZ ALVES DE OLIVEIRA
Data: 27/11/2024 12:37:28-0300
Verifique em https://validar.iti.gov.br

Prof. Wagner Luiz Alves de Oliveira, D.Sc. (Orientador)

Documento assinado digitalmente
gov.br PAULO CESAR MACHADO DE ABREU FARIAS
Data: 27/11/2024 11:45:27-0300
Verifique em https://validar.iti.gov.br

Prof. Paulo César Machado de Abreu Farias, D.Sc. (Orientador)

Documento assinado digitalmente
gov.br ANFRANSERAI MORAIS DIAS
Data: 02/12/2024 15:27:29-0300
Verifique em https://validar.iti.gov.br

Prof. Anfranserai Morais Dias, D.Sc. - UEFS

Documento assinado digitalmente
gov.br EDWARD DAVID MORENO ORDONEZ
Data: 29/11/2024 16:36:53-0300
Verifique em https://validar.iti.gov.br

Prof. Edward David Moreno Ordonez, D.Sc. - UFS

Documento assinado digitalmente
gov.br NELSON ALVES FERREIRA NETO
Data: 27/11/2024 08:40:01-0300
Verifique em https://validar.iti.gov.br

Prof. Nelson Alves Ferreira Neto, D.Sc. - SENAI

Documento assinado digitalmente
gov.br TIAGO TRINDADE RIBEIRO
Data: 03/12/2024 07:32:54-0300
Verifique em https://validar.iti.gov.br

Prof. Tiago Trindade Ribeiro, D.Sc. - UFBA

SALVADOR, BA – BRASIL
OUTUBRO DE 2024

Ficha catalográfica elaborada pelo Sistema Universitário de Bibliotecas (SIBI/UFBA),
com os dados fornecidos pelo(a) autor(a).

Thiago Costa Esteves, Linton
Análise e construção de aceleradores em hardware para o cálculo do menor caminho em planejamento de rotas de robôs / Linton Thiago Costa Esteves. -- Salvador, 2024.
110 f.

Orientador: Wagner Luiz Alves de Oliveira.
Coorientador: Paulo César Machado de Abreu Farias.
Tese (Doutorado - Engenharia Elétrica) -- Universidade Federal da Bahia, UFBA, 2024.

1. Dijkstra. 2. Menor Caminho. 3. PRM. 4. FPGA. 5. Robótica. I. Luiz Alves de Oliveira, Wagner. II. César Machado de Abreu Farias, Paulo . III. Título.

Linton Thiago Costa Esteves

Analysis and construction of hardware accelerators for shortest path calculation in robot route planning

Doctoral dissertation submitted to the Post-Graduation Program in Electrical Engineering – PPGEE, in partial fulfillment of the requirements for the degree of the Doctorate Program in Electrical Engineering. *FINAL VERSION*

Concentration Area: Information Processing and Energy

Advisors: Prof. Dr. Wagner Luiz Alves de Oliveira / Prof. Dr. Paulo César Machado de Abreu Farias

Federal University of Bahia – Salvador
October 2024

Meu amado filho,

Esta tese é dedicada a você, que me ensinou o poder do amor incondicional.

Que este trabalho, fruto de tanta dedicação, desafios e sacrifícios sirva como um exemplo de que, com esforço e determinação, qualquer objetivo pode ser alcançado. Cada momento de empenho foi motivado pelo desejo de construir um futuro melhor para você. Sua presença e amor foram minha maior fonte de força e inspiração.

Que você sempre se lembre de seguir seus sonhos com determinação, sabendo que, mesmo nos momentos mais difíceis, o amor e o esforço valem a pena.

Que esta tese seja um testemunho do quanto é possível alcançar com perseverança e paixão.

Com todo o meu amor,

Seu Pai.

AGRADECIMENTOS

Gostaria de agradecer profundamente à minha esposa Samara Ribeiro, por seu amor, paciência e apoio incondicional. Sua compreensão e encorajamento foram fundamentais para que eu pudesse me dedicar a esta pesquisa. Sua presença e suporte emocional foram essenciais nos momentos mais desafiadores.

Também gostaria de expressar minha profunda gratidão ao meu orientador, Prof. Dr. Wagner Luiz Alves de Oliveira, por sua orientação, paciência e suporte ao longo de todo este projeto. Suas valiosas sugestões e incentivo constante foram fundamentais para a conclusão desta tese.

Agradeço também ao meu orientador, Prof. Dr. Paulo César Machado de Abreu Farias, por suas contribuições essenciais, conselhos valiosos e disponibilidade para discutir ideias e solucionar dúvidas. Seu conhecimento e apoio foram inestimáveis para o desenvolvimento desta pesquisa.

Gostaria ainda de agradecer à minha irmã Camila, por sua ajuda nas revisões de última hora. Sua dedicação e apoio foram essenciais.

Um agradecimento especial ao Instituto Federal Baiano, que me proporcionou o suporte necessário para a realização desta tese. O apoio recebido foi crucial para o sucesso deste trabalho.

“De tudo ficaram três coisas:
A certeza de que estamos começando,
A certeza de que é preciso continuar,
A certeza de que podemos ser interrompidos antes de terminar.
Façamos da interrupção um caminho novo.
Da queda, um passo de dança,
Do medo, uma escada,
Do sonho, uma ponte,
Da procura, um encontro.”
(Fernando Sabino)

RESUMO

ESTEVES, L. **Análise e construção de aceleradores em hardware para o cálculo do menor caminho em planejamento de rotas de robôs**. 2024. 110 p. Tese (Doutorado em Engenharia Elétrica) – Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal da Bahia, Salvador – BA, 2024.

Este trabalho propõe uma análise e otimização para o cálculo do menor caminho no planejamento de rotas para robôs móveis. A solução proposta visa apresentar uma alternativa de alto desempenho que possa atender às restrições de tempo necessárias para processamento em robôs. Para isso, foi construída uma arquitetura focada em paralelismo a ser embarcada em hardware dedicado. Através da exploração de paralelismo, a solução visa apresentar, além de uma melhoria de desempenho, uma adaptação dinâmica às mudanças no grafo de possíveis movimentações a ser analisado, uma vez que arestas podem ser inseridas ou removidas de forma temporalmente aleatória conforme as mudanças no ambiente. Este trabalho demonstra a arquitetura desenvolvida juntamente com seus resultados. O grafo da aplicação é atualizado de forma eficiente através de uma matriz de obstáculos, resultando em uma melhoria notável de 120 vezes para grafos com 1.024 nós. Ao utilizar um dispositivo de baixo custo como o Cyclone IV E, é atingido desempenho cerca de 20 vezes superior a de uma aplicação equivalente em software para um grafo com 1024 nós.

Palavras-chave: Dijkstra, Menor Caminho, PRM, FPGA, Robótica.

ABSTRACT

ESTEVES, L. **Analysis and construction of hardware accelerators for shortest path calculation in robot route planning**. 2024. 110 p. Tese (Doutorado em Engenharia Elétrica) – Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal da Bahia, Salvador – BA, 2024.

This work proposes an analisys and optimization for the calculation of the shortest path in route planning for mobile robots. The suggested solution aims to present a high-performance alternative that can meet the time constraints necessary for robots processing . To do so, we propose an architecture focused on parallelism to be embedded in dedicated hardware. Through the exploitation of parallelism, the solution aims to present, in addition to a performance improvement, a dynamic adaptation to changes in the graph of possible movements to be analyzed, since edges could be inserted or deleted in a temporally random manner as the environment changes. This work demonstrates the architecture developed together with its results. This application graph updating process efficiently updates obstacle matrices, resulting in a remarkable 120-fold improvement for 1024-node graphs. When utilizing a cost-effective device like the Cyclone IV E, it achieves approximately 20 times the performance of an equivalent software applications.

Keywords: *Dijkstra, PRM, Shortest Path, FPGA, Robotics .*

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS-FONTE

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AMBA	Advanced Microcontroller Bus Architecture
APSP	menor caminho entre todos os pares (do inglês *All-Pairs Shortest Path*)
ASIC	Circuitos Integrados de Aplicação Especifica (do inglês *Application-Specific Integrated Circuit*)
BGL	*Boost Graph Library*
CA	Classificador de Ativos
CAE	Controle de Acesso Externo
CAM	Controle de Acesso às Memórias
CME	Controlador de Máquina de Estados
CPU	Unidade Central de Processamento (do inglês *Central Processing Unit*)
DOF	grau de liberdade (do inglês *Degree of Freedom*)
DUT	*Design Under Test*
DUV	*Design Under Verification*
ENA	Expansor de Nó Aprovado
FPGA	*Field Programmable Gate Array*
FSM	máquina de estados finita (do inglês *Finite State Machine*)
GA	Gerenciador de Ativos
GLE	Gerenciador de Leitura e Escrita
GMAE	Gerenciador de Memórias com Acesso Externo
GNA	Gerenciador de Nós Ativos
GPU	Unidade de Processamento Gráfico (do inglês *Graphics Processing Unit*)
HPC	Computação de alto desempenho (do inglês *High-Performance Computing*)
IE	Interface Externa
LUT	Look-Up Table
LVV	Localizador de Vizinho Válido
MA	Memória de Anteriores
ME	Memória de Estabelecidos
MO	Memória de Obstáculos
MPI	*Message Passing Interface*
MR	Memória de Relacionamentos
Mutex	*Mutual Exclusion*

NA	Nó Ativo
OpenCL	*Open Computing Language*
OpenMP	*Open Multi-Processing*
OSPF	*Open Shortest Path First*
PCIe	*Peripheral Component Interconnect Express*
PRM	*Probabilistic Roadmaps*
RRT	*Rapidly exploring Random Tree*
RTL	Linguagem de Descrição de Hardware (do inglês *Register-Transfer Level*)
SDC	*Synopsys Design Constraints*
SSSP	problema do menor caminho a partir de uma fonte (do inglês *Single Source Shortest Path Problem*)
STA	*Static Timing Analysis*
UVM	Metodologia de Verificação Universal (do inglês *Universal Verification Methodology*)

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

O planejamento de caminhos é uma das atividades fundamentais na robótica, ele permite que um robô se desloque com segurança de um ponto de origem A para um ponto de destino B. Para realizar essa tarefa é necessário que o robô, através dos seus diversos sensores, mapeie o ambiente no qual está inserido, detectando obstáculos e possíveis caminhos livres. A partir desse mapeamento do ambiente, o robô passa então a ter à sua disposição diversas possibilidades diferentes de alcançar o destino.

De um modo geral, um caminho pode ser definido como uma sequência de passos entre o ponto inicial e todos os pontos que ligam o início ao fim do caminho. Para realizar cada um desses passos, o robô aloca uma quantidade de energia proporcional ao esforço necessário. Esse esforço, ou peso do deslocamento, pode variar de acordo com diversos fatores, como a posição atual do robô e a configuração do ambiente no qual ele está inserido no momento. Em uma ladeira, por exemplo, subir é mais difícil do que descer.

Desse modo, o esforço total de uma determinado caminho pode ser calculado como a soma do esforço de todos os passos intermediários entre a origem e o destino. Considerando isso, ao realizar o planejamento do caminho a ser seguido, é mais interessante que aquele com o menor esforço total seja utilizado, visto que um melhor caminho resultará em um menor custo energético e, provavelmente, menor tempo de execução.

A construção do menor caminho está inserida em um contexto em que várias etapas devem ser realizadas para que um planejamento eficiente de caminhos possa ser realizado. Inicialmente, um grafo representando o espaço de configuração do robô é construído em uma etapa de pré-processamento (Figura 1A). Este grafo captura todas as posições de configuração possíveis e suas relações, fornecendo uma representação das capacidades de movimentação do robô.

Logo após, o algoritmo *Probabilistic Roadmaps* (PRM) realiza a amostragem das configurações válidas e as conecta através de caminhos livres de colisões. Durante essa etapa é

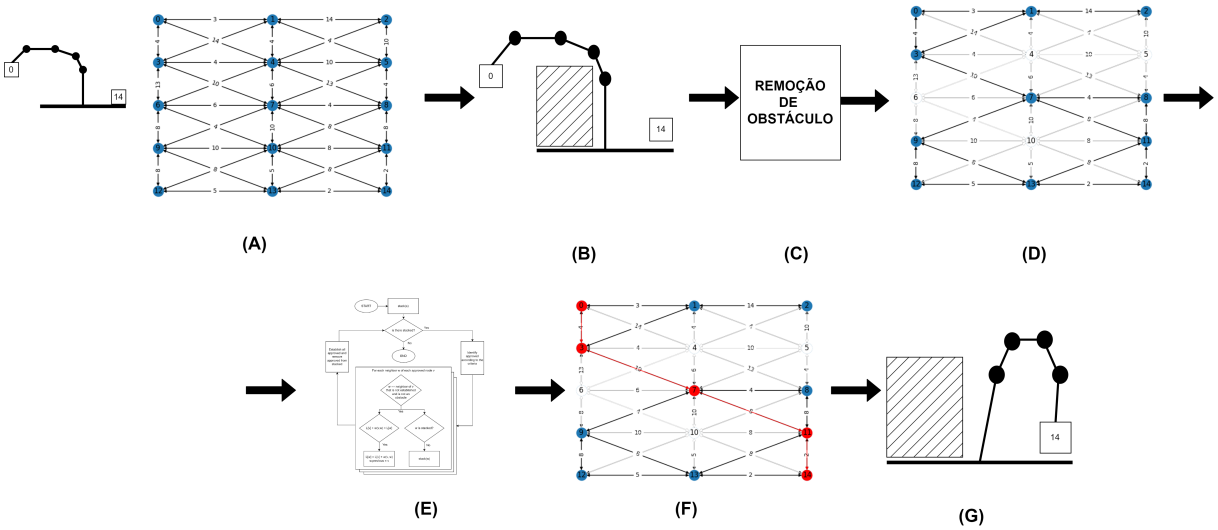


Figura 1 – (A) Um robô e seu espaço de configuração representado em um grafo. (B) Inserção de um obstáculo no ambiente. (C) Aplicação de técnica de remoção de obstáculos. (D) Espaço livre após remoção de obstáculos. (E) Usando um algoritmo de menor caminho. (F) Identificação do caminho que o robô deve seguir. (G) Aplicação pelo robô da sequência de configuração.

identificada a presença de obstáculos e o espaço livre disponível para a navegação do robô é construído. Quando obstáculos são introduzidos no ambiente durante a operação do robô (Figura 1B), o algoritmo PRM atualiza dinamicamente o grafo existente (Figura 1C), removendo os nós comprometidos devido ao obstáculo, objetivando criar caminhos livres que permitem o movimento contínuo do robô. Esta adaptação dinâmica garante uma navegação livre de obstáculos e otimiza o caminho do robô (Figura 1D).

Para determinar o caminho ideal entre os pontos de origem e destino, algoritmos de menor caminho, como o algoritmo de Dijkstra, são empregados (Figura 1E). O problema do menor caminho a partir de uma fonte (do inglês *Single Source Shortest Path Problem*) (SSSP) pretende encontrar o menor caminho entre um nó origem e um nó destino, considerando o esforço existente no processo de deslocamento do robô. Esses algoritmos consideram os relacionamentos e restrições no grafo atualizado dinamicamente. Ao analisar a estrutura do grafo e as mudanças no ambiente, são alcançadas soluções robustas e confiáveis de planejamento de caminhos. Isso permite que o robô navegue da configuração inicial até a configuração de destino de forma eficiente e segura (Figuras 1F e 1G). Ao otimizar o cálculo do menor caminho, o objetivo deste trabalho é atuar nas etapas E e F da Figura 1.

Neste contexto, este trabalho apresenta uma proposta que solucione o SSSP de forma rápida e eficiente, viabilizando sua utilização em aplicações que necessitam de um baixo tempo de processamento. Além disso, a solução deve realizar o cálculo do menor caminho considerando que o ambiente não é estático, ou seja, obstáculos podem surgir ou desaparecer durante a sua execução, necessitando assim de uma representação dinâmica do ambiente, com possíveis caminhos também dinâmicos. Tais requisitos exigem que a solução trabalhe de forma coordenada e otimizada com os outros componentes do processo, como sensores e algoritmos de mapeamento

do ambiente.

Quando se deseja realizar otimizações de determinados algoritmos, uma das alternativas que se destaca é a utilização de dispositivos do tipo *Field Programmable Gate Array* (FPGA). Aplicações com baixa intensidade operacional e com maior tratamento de valores escalares (em oposição à vetorização) dificultam a exploração do desempenho de pico oferecido por soluções baseadas em Unidade de Processamento Gráfico (do inglês *Graphics Processing Unit*) (GPU), abrindo caminho para uma melhor exploração de desempenho por FPGAs. Atrelado a isso, os FPGAs, por apresentarem uma arquitetura especializada para a aplicação, possuem uma melhor eficiência na utilização dos recursos e, consequentemente, apresentam um menor consumo energético.

Uma das principais distinções desta proposta está no processo de atualização do grafo (Figura 1F). Diferentemente das alternativas tradicionais que geram um novo grafo a cada iteração, esta proposta apenas atualiza uma matriz contendo obstáculos. Como resultado, há uma melhoria notável de 120 vezes no processo de atualização para grafos compreendendo 1.024 nós.

Para viabilizar o paralelismo, foi realizado um levantamento bibliográfico do estado da arte das soluções de busca do menor caminho. Como resultado dessa busca foi implementada uma solução utilizando um critério de seleção para remoção e atualização de nós em paralelo.

O projeto busca obter uma solução que não necessite de muitos recursos computacionais, resultando em um produto que possa ser empregado em dispositivos de baixo custo e consumo enérgico. Nesse sentido, como prova de conceito, foi utilizado um FPGA econômico como o Cyclone IV E, onde se atingiu um desempenho aproximadamente 12 vezes melhor em comparação com uma aplicação equivalente em software.

1.1 Objetivos Gerais

O objetivo deste trabalho é apresentar uma contribuição para a resolução do problema do menor caminho entre dois nós de um grafo direcionado com foco no baixo tempo de resposta. Considera-se que as arestas do grafo possam ser removidas ou adicionadas durante o processamento, conforme a inserção ou remoção de obstáculos no espaço de configuração do robô.

Como resultado se obtêm uma solução que pode ser integrada em um projeto de movimentação de Robôs, seja em um mesmo circuito integrado ou através da conexão com controlador localizado em um dispositivo externo.

1.2 Objetivos Específicos

- Realizar levantamento bibliográfico sobre o estado da arte na busca do menor caminho;

- Definir um algoritmo que permita paralelismo para calcular o menor caminho entre dois pontos em um grafo direcionado;
- Desenvolver uma solução que possa ser utilizada na movimentação de robôs;
- Construir uma arquitetura para essa solução;
- Testar e validar a arquitetura construída, obtendo resultados positivos.

1.3 Organização do documento

Este trabalho está organizado conforme descrito na sequência.

No Capítulo 2 é apresentada a fundamentação teórica necessária para a construção desta proposta. Nele são expostas técnicas utilizadas no planejamento de rotas, formas de representação das possíveis posições de movimentação do robô e alguns algoritmos de menor caminho.

O Capítulo 3 discorre sobre o ambiente científico em que a proposta está inserida, apresentando alguns trabalhos relacionados com o projeto e que serviram como norteadores para o desenvolvimento da solução. São exploradas também técnicas de otimização relatadas na literatura.

As contribuições desta proposta ao planejamento de rotas de robôs móveis são apresentadas no Capítulo 4 através da proposição de uma arquitetura. Nesse capítulo são expostas também as motivações e justificativas da proposta.

Em seguida, no Capítulo 5, são mostrados alguns resultados de simulações do algoritmo utilizado.

A partir dos resultados obtidos em simulação e da construção do modelo de referência, no Capítulo 6 é apresentada a microarquitetura do projeto que implementa a proposta. Os resultados de síntese e simulação da Linguagem de Descrição de Hardware (do inglês *Register-Transfer Level*) (RTL) construída a partir da arquitetura proposta são discutidos no Capítulo 7.

No Capítulo 8 são apresentadas as considerações finais sobre o que foi desenvolvido e os resultados obtidos, além da discussão de perspectivas futuras para a extensão da proposta.

Por fim, o Apêndice B aborda o fluxo de projeto digital com enfoque no desenvolvimento em FPGA que foi utilizado para a confecção desta proposta .

CAPÍTULO

2

FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são abordados conceitos importantes utilizados durante a análise e elaboração da proposta. Inicialmente serão descritos os fundamentos teóricos necessários ao planejamento de rotas para robôs móveis e à utilização de grafos para mapear as opções de movimentação de um robô. Por fim, são apresentados alguns algoritmos que calculam o menor caminho entre dois nós de um grafo.

2.1 Planejamento de rotas

Da mesma forma que um motorista de ônibus precisa construir um mapa mental da rodovia que está trafegando, identificando rotas possíveis e obstáculos, um robô necessita também mapear o ambiente no qual está inserido. Desse modo, considerando seu grau de liberdade (do inglês *Degree of Freedom*) (DOF), ele busca primeiramente identificar como pode se movimentar, ou seja, quais são suas posições possíveis em um cenário livre de obstáculos. Após esse conhecimento, posições ocupadas por obstáculos são removidas e então se obtém o real espaço livre para movimentação.

Desta maneira, o conjunto de todas as posições possíveis que um robô pode apresentar em um espaço com n dimensões é denominado espaço de configuração (Q). Nesse espaço, cada estado possível do robô é representado por uma posição de configuração (q). Posições que coincidem com algum tipo de obstáculo que possa comprometer ou até mesmo impedir a movimentação do robô fazem parte do conjunto de obstáculos (Q_{obst}). Ao subtrair do espaço de configuração todas as configurações comprometidas pelos obstáculos, obtém-se o conjunto do espaço livre ($Q_{livre} = Q - Q_{obst}$), que é a região responsável pela definição das posições transitáveis de um robô móvel (KLANCAR *et al.*, 2017).

O planejamento de rotas consiste na tarefa de encontrar um caminho contínuo que levará o robô de uma configuração de início até a de destino através do conjunto do espaço livre (ver

Figura 2). Como existem vários caminhos possíveis até o destino, os seguintes critérios podem ser utilizados para identificar um caminho ótimo (KLANCAR *et al.*, 2017):

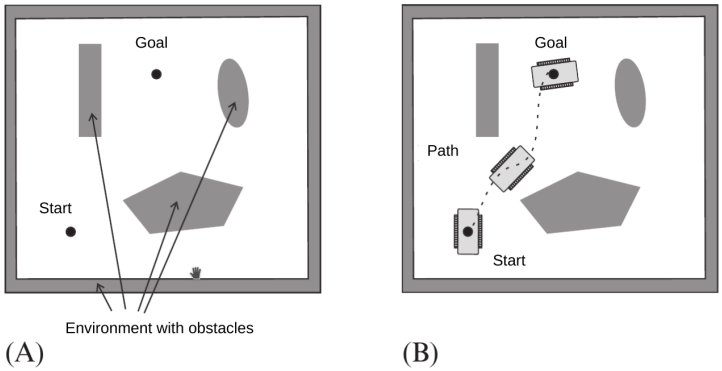


Figura 2 – (A) Ambiente com obstáculos, inicio e objetivo, (B) um dos possíveis caminhos entre o início e o objetivo (Adaptado de Klancar *et al.* (2017)).

1. O tamanho do caminho deve ser o menor;
2. Deve ser aquele em que o robô consegue atravessar no menor intervalo de tempo;
3. Deve estar o mais longe possível dos obstáculos;
4. Deve ser suave, sem transições bruscas;
5. Deve considerar limitações de movimento.

2.2 Utilização de grafos no planejamento de rotas

Para realizar o planejamento de caminhos, é necessário representar computacionalmente o espaço de configuração com seus obstáculos e espaço livre. Em aplicações onde a área de movimentação do robô está limitada a um plano 2D, é comum a utilização de uma grade como método de representação. No entanto, robôs que possuem uma quantidade maior de articulações necessitam ser representados por estruturas mais complexas, como os grafos, devido à heterogeneidade de suas relações. Essas estruturas permitem uma maior flexibilidade de relacionamentos entre as posições de configuração.

Um grafo de transição de estados para um determinado robô pode ser formado reduzindo o espaço livre a um determinado número de configurações intermediárias e suas transições. Cada uma das configurações representa um nó do grafo, e as conexões são simbolizadas por linhas ou arestas, identificando ações necessárias para a movimentação do sistema entre os estados ou nós.

Um grafo $G = (V, E)$ é formado por um conjunto de vértices ou nós $V = v_1, v_2,...,v_n$ e um conjunto de arestas ou arcos $E = e_1, e_2,...,e_m$. Cada aresta é formada por um par (v_i, v_j), onde v_i e $v_j \in$ V (DEO, 2017). Em alguns casos, as arestas possuem um terceiro componente c_{ij} que define seu peso ou custo. Grafos que possuem esse componente são denominados grafos

ponderados (WEISS, 2012; MEHLHORN, 2012). Arestas cujo par possui o mesmo vértice (v_i, v_i) são denominadas laços (ver aresta e_1 da Figura 3). É possível também a existência de duas arestas diferentes formadas por um mesmo par (ver arestas e_5 e e_4 da Figura 3), que nesse caso são chamadas de arestas paralelas.

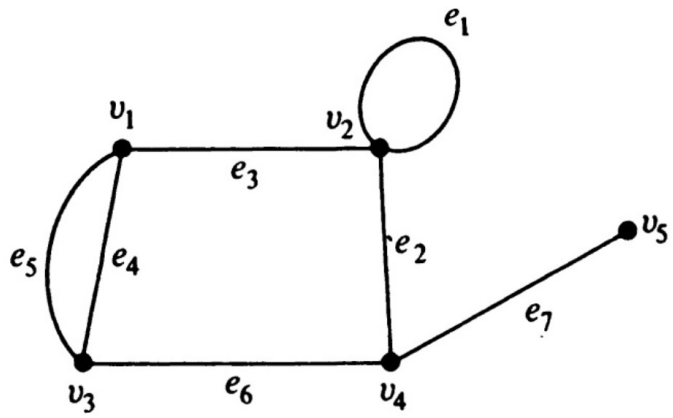


Figura 3 – Exemplo de grafo com 5 vértices e sete arcos (Extraído de Deo (2017)).

Grafos que não possuem nem arestas paralelas nem laços são denominados grafos simples (DEO, 2017). Quando uma aresta é formada por um par ordenado, obtém-se um grafo direcionado ou dígrafo, cujo sentido do arco é fixo (ver Figura 4) (WEISS, 2012; PAPADIMITRIOU; STEIGLITZ, 1998).

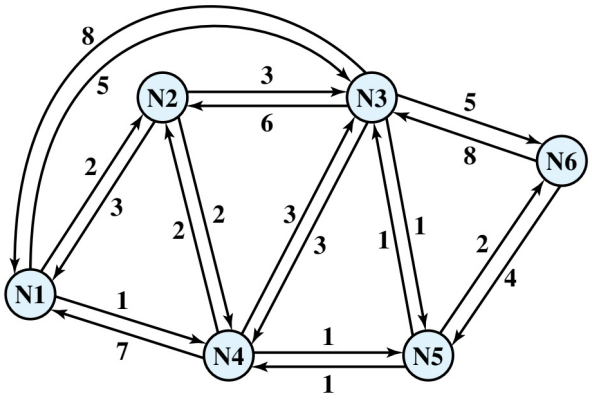


Figura 4 – Exemplo de um grafo direcionado com 6 vértices (Extraído de Stallings (2007)).

Um caminho em um grafo é formado por uma sequência de vértices v_1, v_2, v_3, ..., v_N, onde (v_i, v_{i+1}) $\in V$ para 1 $\leq$ i < N. O tamanho de um caminho é mensurado de acordo com a quantidade de arestas presentes no mesmo. O custo de um caminho v_1, v_2, v_3, ..., v_N é calculado a partir dos seus pesos, sendo definido por $\sum_{i=1}^{N-1} c_{i,i+1}$. Em um grafo podem existir caminhos simples, compostos por vértices distintos, com exceção dos vértices iniciais e finais que podem ser os mesmos (WEISS, 2012).

Grafos também podem ser classificados de acordo com as suas conexões. Nesse contexto, encontram-se os dígrafos fortemente conexos, aqueles que, para cada vértice, existe um caminho que se conecta com todos os outros. Quando cada par de vértices possuir uma aresta, obtém-se um grafo completo (WEISS, 2012).

2.2.1 Formas de representar um grafo

Comumente os grafos podem ser representados como uma matriz de adjacência ou como listas de adjacência. Na figura 5, vemos um dígrafo (a) representado como listas de adjacência (b) e matriz de adjacência (c).

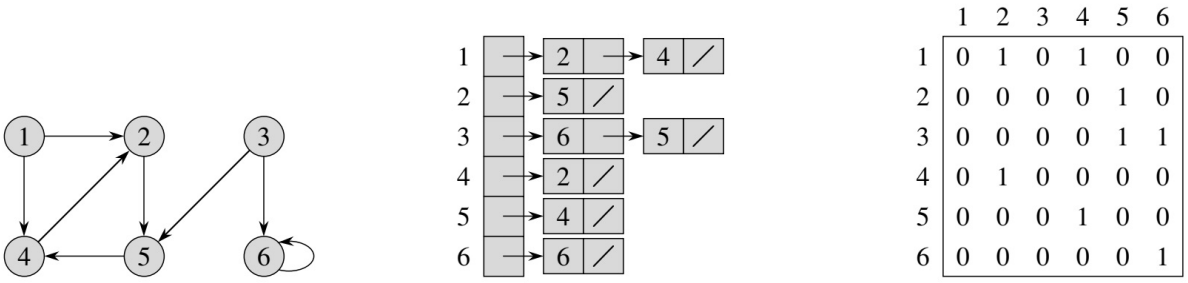


Figura 5 – Representações de um grafo direcionado. (a) Dígrafo G com seis vértices e oito arcos. (b) Representação de G com listas de adjacência. (c) Representação de G com matriz de adjacência (Extraído de Cormen *et al.* (2009)).

A matriz de adjacência é formada através de uma matriz booleana $A_G = (a_{ij})_{1 \leq i,j \leq n}$ de dimensões $|V|$ x $|V|$. Os arcos existentes em um grafo direcionado são identificados na matriz de acordo com a equação 2.1. Nesse tipo de abordagem, os requisitos de armazenamento são de $\Theta(n^2)$ (MEHLHORN, 2012). A utilização da matriz de adjacência é uma boa escolha quando se utilizam grafos densos, $|E| = \Theta(V^2)$.

$$a_{ij} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{se } (i,j) \notin E \end{cases} \tag{2.1}$$

Uma matriz de adjacência pode ser utilizada para armazenar o peso dos arcos em grafos com pesos. Nessa situação, o peso c_{ij} de um arco (i,j) pode ser armazenado na linha i e coluna j da matriz de adjacência, juntamente com a identificação da ligação entre os vértices (CORMEN *et al.*, 2009). Pode-se também definir um valor muito pequeno ou muito grande de peso, quando não existir a ligação entre os vértices, ou o valor do peso quando a ligação existir (WEISS, 2012).

Em situações em que o grafo não é denso, ou seja, é esparso, o método de representação mais recomendado é a lista de adjacência (Adj) de $|V|$, que permite a criação de uma lista para cada vértice de V. A lista de adjacência representando o vértice $u \in V$, por exemplo, aponta para todos os nós vizinhos a u no grafo. Nessa abordagem, as listas são normalmente representadas

de forma não ordenada e necessitam de um armazenamento com $\Theta(V+E)$ dimensões, sendo E a soma do tamanho de todas as listas adjacentes (CORMEN *et al.*, 2009).

Listas de adjacência podem ser adaptadas para permitir a representação de grafos com pesos. Nessa situação, o peso c_{ij} de um arco (i, j) pode ser armazenado nas listas de adjacência dos nós i e j juntamente com seus pares. Uma das desvantagens da utilização de listas de adjacência é devido a uma maior complexidade em identificar se uma aresta (u, v) está presente no grafo. Para isso, é necessário investigar na lista de adjacência de um dos vértices ($Adj[u]$ ou $Adj[v]$) se o outro par está presente (CORMEN *et al.*, 2009).

A partir das informações expostas, é importante ressaltar dois pontos: a maioria dos grafos utilizados em aplicações são esparsos, ou seja, $|E| \leq |V|^2$; a escolha da representação pode ter uma influência expressiva no desempenho do algoritmo (MEHLHORN, 2012). Como apresentado em (MEHLHORN, 2012) muitos problemas que podem ser resolvidos em tempo linear, $O(|V|+|E|)$, utilizando listas de adjacência, podem apresentar um tempo de execução de $\theta(|V|^2)$ quando a matriz de adjacência é utilizada.

2.3 Mapa de navegação

De um modo geral, o planejamento de rotas pode ser dividido em duas fases: i) uma fase inicial de planejamento, que realiza a construção de um mapa do ambiente; e ii) uma fase de consulta, que utiliza os resultados da fase de planejamento para criar um caminho entre dois pontos. Vale ressaltar que a fase de planejamento é consideravelmente mais custosa (ROBOTICS, 2011).

A fase de planejamento gera como resultado um mapa de navegação formado a partir do mapeamento das conexões possíveis entre os nós presentes no espaço livre. Ele é construído de acordo com a geometria do ambiente no qual o robô está inserido, sendo seu principal desafio encontrar a quantidade mínima de caminhos que possibilitem ao robô alcançar todas as configurações contidas no conjunto do espaço livre.

Para sua construção, diferentes técnicas podem ser utilizadas. Os campos potenciais (ver Figura 6), por exemplo, criam pesos imaginários para representar o ambiente. O destino apresenta o menor peso, sendo que quanto mais distante do destino um ponto está, maior será o peso aplicado, além disso, obstáculos também possuem pesos maiores. Dessa forma, o robô deve seguir o gradiente negativo a fim de alcançar o objetivo.

Outra técnica utilizada é a baseada em amostras, na qual pontos aleatórios são coletados e métodos de detecção de colisão são aplicados visando identificar se esses são pontos pertencentes ao espaço livre. A partir do conjunto desses pontos, um caminho do início até o destino pode ser formado. Nessa abordagem não é necessário calcular o espaço livre, etapa que geralmente consome grande quantidade de recursos computacionais em robôs com maiores graus

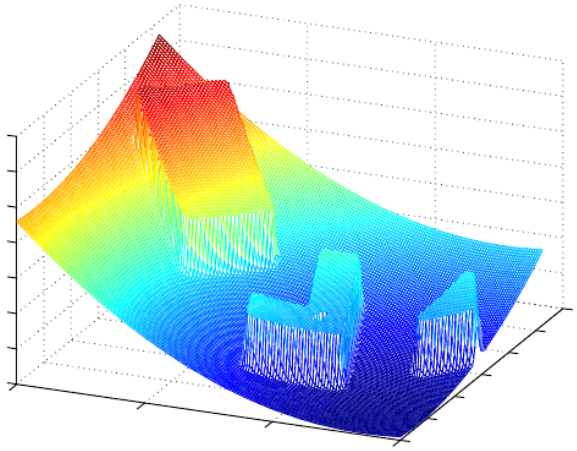


Figura 6 – Demonstração da aplicação de pesos em um campo potencial (Adaptado de Klancar *et al.* (2017)).

de liberdade. Uma variante desse método é o *Rapidly exploring Random Tree* (RRT), que busca o caminho a partir de um ponto inicial até um ponto destino, adicionando a cada iteração uma nova conexão próxima ao ponto atual selecionado aleatoriamente (ver Figura 7).

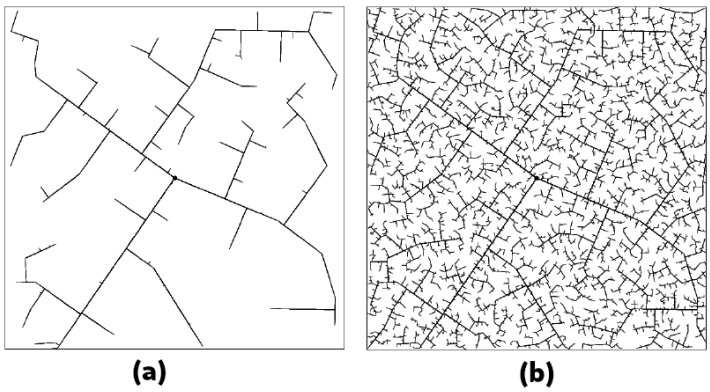


Figura 7 – Evolução de um algoritmo RRT com 45 (A) e 2345(B) iterações (Extraído de LaValle (2006)).

Apesar da fase de consulta ser consideravelmente mais rápida que a fase de planejamento, caso o objetivo mude, a fase de planejamento deve ser executada novamente. Além disso, em um cenário que necessite solicitar diversos pares origem e destino (q_i, q_d) para um conjunto de robôs e obstáculos fixos, as técnicas apresentadas anteriormente devem ser reprocessadas diversas vezes, possivelmente até repetindo os mesmos pares, gerando assim um retrabalho.

Desse modo, a disparidade nos custos de planejamento e consulta levou ao desenvolvimento de métodos de mapa de navegação nos quais a busca pudesse incluir diversas posições de início e objetivo (ROBOTICS, 2011). Assim, em vez de tratar cada par individualmente, se investe em um pré-processamento mais completo, de modo a simplificar o processo de buscas posteriores ao se incluir diversos pontos na análise.

Nesse contexto, primeiramente introduzidas por (KAVRAKI PETR SVESTKA; OVER-

MARS, 1996), surgem as PRM como um método para realizar a construção de um mapa de navegação quando se possuem diversos pontos de início e destino. Essa técnica realiza também as fases de pré-processamento e busca. A fase de pré-processamento (ver Figura 8a), consiste na construção do mapa de navegação de modo a viabilizar a resolução rápida de buscas entre nós presentes na região do espaço livre. A fase de busca (ver Figura 8b), realizada após o pré-processamento, consiste em conectar dois pares q_i e q_d ao grafo e, através de um algoritmo de busca do menor caminho, formar o caminho ótimo entre esses pares (LAVALLE, 2006).

É importante ressaltar que o processo de reconstrução do mapa de navegação é realizado sempre que ocorre uma mudança na configuração dos obstáculos do ambiente, e exclusivamente nesse tipo de situação. Em termos de consumo computacional, o processo de detecção de colisões na fase de aprendizagem é o que mais utiliza recursos.

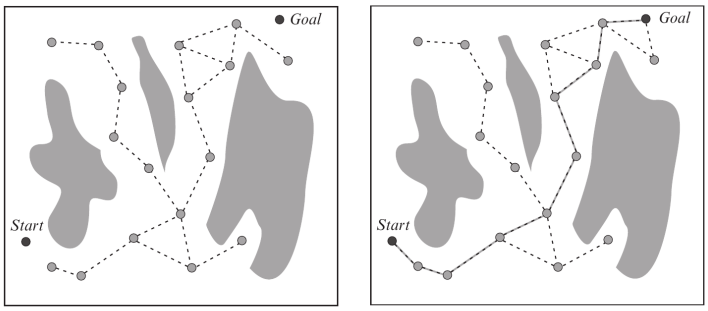


Figura 8 – Método PRM: a) fase de aprendizagem e b) fase de busca. (Extraído de Klancar *et al.* (2017)).

2.4 Algoritmos de menor caminho

Como apresentado em (DEO, 2017) a utilização de algoritmos de busca do menor caminho é um dos temas mais abordados na teoria dos grafos. Existem diversas implementações possíveis para esse tipo de problema, cada uma com suas peculiaridades e focadas em resolver problemas específicos através da utilização de soluções customizadas para cada problema. De um modo geral, esses algoritmos podem ser divididos em cinco grupos (DEO, 2017):

1. Menor caminho entre dois nós específicos;
2. Menor caminho de um vértice para todos os outros;
3. Menor caminho entre todos os pares de nós;
4. Menor caminho entre vértices específicos que passam por determinados vértices;
5. Menor caminho entre diversos vértices.

Como o objetivo desta proposta é o desenvolvimento de algoritmos de menor caminho para aplicações com robôs móveis, nas quais se deseja deslocar o robô de um nó i para um

nó j no grafo, os três últimos grupos não serão discutidos nesta tese. Além disso, em algumas situações, o primeiro grupo é igual ao segundo, pois, durante o processo de busca do menor caminho entre dois vértices, é possível que seja necessário encontrar o menor caminho entre todos os outros vértices, Dessa forma, será suficiente discutir apenas o primeiro grupo (WEISS, 2012).

O problema de encontrar o menor caminho entre dois nós específicos s e t em um dígrafo G com n vértices pode ser representado como uma matriz de adjacência $D = [d_{ij}]$ com dimensão $n\ x\ n$, onde d_{ij} representa o peso do arco, sendo: $d_{ij} \geqslant 0$ o custo do movimento do vértice i para j; $d_{ii} = 0$; e $d_{ij} = \infty$ quando não existir arco entre i e j. Nesse tipo de problema, a desigualdade triangular não precisa ser satisfeita, ou seja, $d_{ij} + d_{jk}$ pode ser menor do que d_{ik}. Desta forma, nem sempre o arco direto d_{ik} entre dois vértices será o menor caminho.

Os algoritmos de busca em grafos podem ser classificados como informados ou não informados. Os algoritmos não informados são aqueles que não utilizam informações além da definição do problema. Eles realizam buscas sistemáticas no grafo e não distinguem nós mais promissores dos menos promissores. Já os algoritmos de busca informados possuem informações adicionais sobre os nós e, sendo assim, conseguem identificar nós mais promissores, gerando uma busca mais eficiente. Estes algoritmos também podem ser classificados como completos ou incompletos. Eles são completos quando conseguem achar uma solução, se ela existir, e incompletos quando, mesmo existindo uma solução, podem não encontrá-la (KLANCAR *et al.*, 2017).

Nas seções a seguir, a título de comparação e melhor entendimento, dois dos principais algoritmos utilizados na busca do menor caminho em um grafo serão expostos, sendo eles: o algoritmo de Dijkstra e o A*.

2.4.1 Algoritmo Dijkstra

O algoritmo de Dijkstra (DIJKSTRA *et al.*, 1959) foi definido em (STALLINGS, 2007) como uma busca do menor caminho a partir de um fonte para todos os outros nós. Caso todos os pesos das conexões entre os nós do grafo possuam um valor maior do que zero, o algoritmo de Dijkstra se apresenta como completo e ótimo. Nesse algoritmo, o tamanho do caminho analisado aumenta gradativamente a cada iteração. Na iteração k, o menor caminho para os k nós mais próximos da fonte são determinados e agrupados em um conjunto T. Na iteração $k+1$, o nó que não está em T e que possui o menor caminho para a fonte é adicionado a T. À medida que cada nó é adicionado em T, o seu caminho para a fonte é definido. Na Figura 9 é mostrado o fluxograma do algoritmo, considerando-se que:

- $N =$ conjunto de nós no grafo;
- $s =$ fonte;

- $T =$ conjunto de nós analisados até o momento;
- $c(i, j) =$ custo do nó i para o nó j, sendo $c(i,i) = 0$, $c(i, j) = \infty$ se os nós não estão diretamente conectados e $c(i, j) \geq 0$ se o nó possui uma conexão direta; e
- $L(n) =$ custo do menor caminho entre a fonte s e o nó n na iteração atual. Ao final do algoritmo será o menor caminho possível entre esses dois nós.

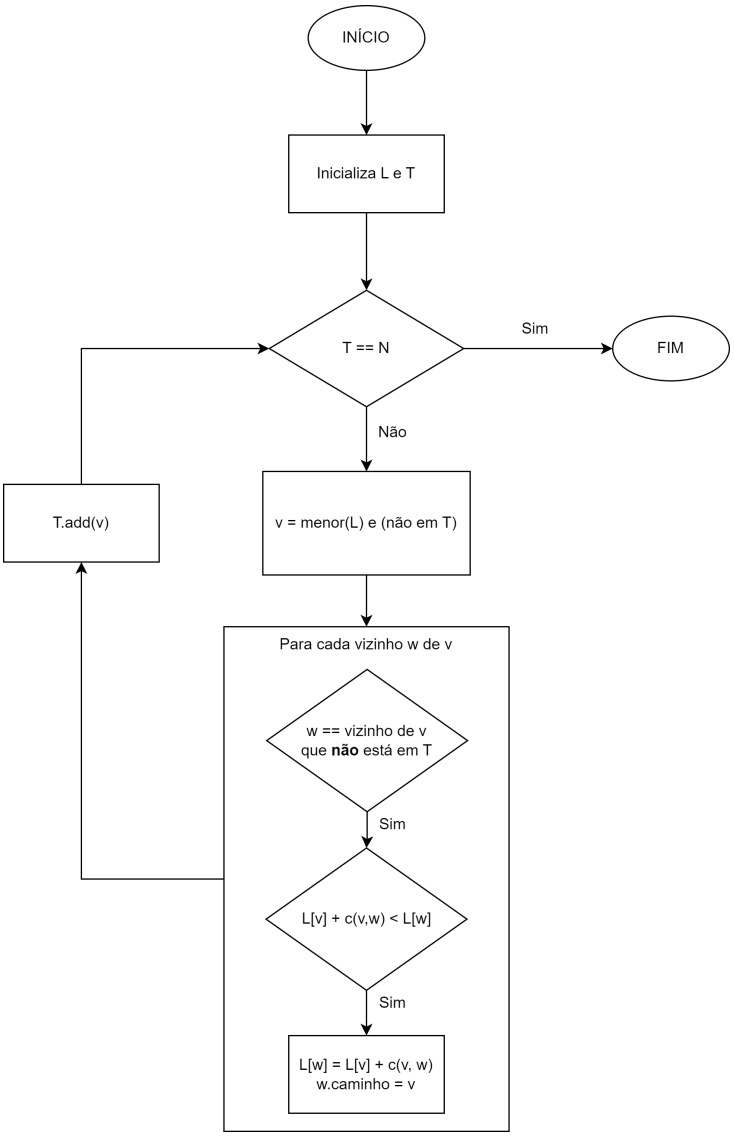


Figura 9 – Fluxograma do algoritmo de Dijkstra.

O algoritmo pode ser dividido em três passos: 1) Inicializar; 2) Coletar próximo nó; e 3) Atualizar os menores caminhos. Os passos 2 e 3 são repetidos até $T = N$.

O primeiro passo inicializa o conjunto T apenas com a fonte ($T = \{s\}$) e inicializa o custo para os vizinhos da fonte (ver Equação 2.2). Após as inicializações, se inicia o segundo passo, onde é encontrado e incorporado a T um nó vizinho que não está em T e que possui a menor distância até a fonte (ver Equação 2.3). Ao escolher sempre o vértice com menor custo até

o momento, pode-se ter certeza de que não existe caminho algum menor que o atual, visto que o algoritmo evolui sempre coletando o vértice com menor distância até a fonte (LEE; HUBBARD, 2015). Nesse momento, é também incorporado o arco que incide neste nó e um nó em T que contribui para o caminho.

$$L(n) = c(s,n) \text{ para n } \neq s \tag{2.2}$$

$$\text{procure } x \notin T \text{ sendo que } L(x) = \underset{j \notin T}{min} L(j) \tag{2.3}$$

O último passo - atualizar os menores caminhos - expresso de acordo com a equação 2.4, atualiza o caminho de s para n quando encontrado um valor menor do que o atualmente armazenado. Satisfeita essa condição, o caminho de s para n é modificado para o caminho de s para x concatenado com o arco de x para n. Quando todos os nós forem analisados o algoritmo é finalizado.

$$L(n) = min[L(n), L(x) + c(x,n)] \text{ para todo } n \notin T \tag{2.4}$$

Na Figura 10 é mostrado o pseudocódigo de uma possível implementação do algoritmo de Dijkstra de acordo com o que foi apresentado nesta seção. Na Tabela 1 e na Figura 11 vemos a evolução do algoritmo Dijkstra no dígrafo da Figura 4, considerando o vértice $N1$ como fonte.

Código-fonte 1 – Pseudocódigo do algoritmo de Dijkstra

```
1  dijkstra(s, N):
2      # Passo 1
3      para cada Vertice n em N:
4          L[n] = c(s,n)
5      s.distancia = 0
6      T = {s}
7      enquanto(T é diferente de N): # Até buscar todos os nós
8          # Passo 2
9          Vertice v = (menor(L)) e (não em T) # vertice com a
       menor distância L[x]
10                                             # e que não faz
       parte de T;
11         # Passo 3
12         para cada w em vizinhos(v):
13             se(w não está em T):
14                 se(L[v] + c(v,w) < L[w]):
15                     L[w] = L[v] + c(v,w) # atualiza w
16                     w.caminho = v # atualiza o caminho de w
17         T.adicionar(v)
```

Figura 10 – Pseudocódigo do algoritmo de Dijkstra.

Tabela 1 – Aplicação do algoritmo de Dijkstra no grafo da Figura 4. (Adaptado de Stallings (2007)).

Iteração	T	L(2)	caminho	L(3)	caminho	L(4)	caminho	L(5)	caminho	L(6)	caminho
1	{1}	2	1-2	5	1-3	1	1-4	INFINITO	-	INFINITO	-
2	{1,4}	2	1-2	4	1-4-3	1	1-4	2	1-4-5	INFINITO	-
3	{1, 2, 4}	2	1-2	4	1-4-3	1	1-4	2	1-4-5	INFINITO	-
4	{1, 2, 4, 5}	2	1-2	3	1-4-5-3	1	1-4	2	1-4-5	4	1-4-5-6
5	{1, 2, 3, 4, 5}	2	1-2	3	1-4-5-3	1	1-4	2	1-4-5	4	1-4-5-6
6	{1, 2, 3, 4, 5, 6}	2	1-2	3	1-4-5-3	1	1-4	2	1-4-5	4	1-4-5-6

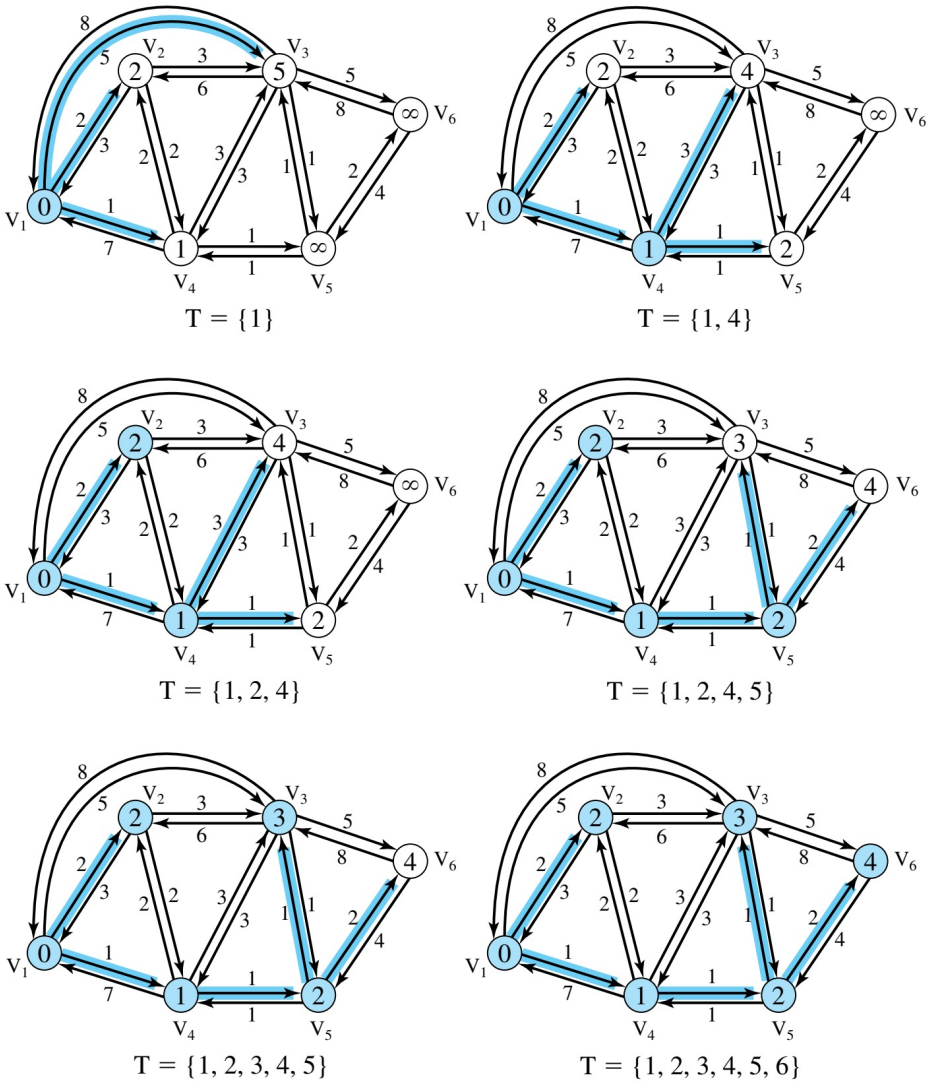


Figura 11 – Evolução do algoritmo de Dijkstra no grafo da Figura 4. (Extraído de Stallings (2007)).

2.4.2 Algoritmo A*

O algoritmo A* (pronunciado A Estrela), proposto por (HART; NILSSON; RAPHAEL, 1968), pertence à categoria dos algoritmos informados, visto que utiliza informação adicional ou função heurística $h(n)$ sobre o grafo durante sua análise. A informação adicional utilizada indica o custo estimado do caminho do nó atual para o destino, localizado na área do grafo ainda não explorada. Essa característica permite distinguir quais nós são mais promissores, viabilizando uma identificação mais eficiente da solução. Por esse motivo, o A* é bastante eficaz na análise

de menor caminho entre dois nós específicos, sendo primariamente utilizado em robótica para análise em *grids* (SIEGWART; NOURBAKHSH; SCARAMUZZA, 2011).

Durante a execução do algoritmo, o custo até o destino (*custo-total*) é calculado para cada nó analisado, sendo composto pela soma do custo da fonte até o nó atual (*custo-aqui*) com o custo estimado até o destino (*custo-destino*).

O algoritmo é iniciado com uma lista de nós abertos contendo apenas o fonte, que possui *custo-aqui* igual a zero, e uma lista (vazia) com os nós fechados. Sua execução pode ser dividida em cinco etapas (KLANCAR *et al.*, 2017):

1. Seleção do primeiro nó da lista de nós abertos (nó atual). Essa lista é ordenada de forma crescente de acordo com o *custo-total*;
2. Para todos os nós que podem ser alcançados a partir do nó atual são calculados o *custo-destino*, o *custo-aqui* e o *custo-total*;
3. São armazenados os valores do *custo-aqui*, do custo da conexão com o nó atual, do *custo-destino* e do *custo-total* para os nós vizinhos que ainda não possuem esses valores. Os nós que já possuem valores armazenados tem seus valores atualizados quando menores valores forem encontrados;
4. Nós cujo valor foi calculado pela primeira vez são adicionados à lista dos nós abertos. Os nós que já se encontravam na lista aberta e foram atualizados são mantidos nesta lista. Os nós que estavam na lista de nós fechados e foram atualizados são movidos para a lista aberta. O nó atual é movido para a lista de nós fechados;
5. Caso ainda existam nós na lista aberta, essa lista é ordenada e o passo 1 é novamente realizado. O algoritmo é finalizado quando o nó destino é adicionado à lista dos nós fechados.

O algoritmo A* é completo e sempre encontra o caminho ótimo no grafo quando se utiliza uma função heurística ótima, que define o *custo-destino* como menor ou igual ao *custo-destino* real. Um dos pontos negativos desse método é o alto uso de memória (KLANCAR *et al.*, 2017).

Na Figura 12, é exibido um exemplo da evolução do algoritmo A*. O nó atual é marcado com um círculo, os nós inseridos na lista de nós abertos recebem a cor cinza, a lista fechada é marcado por cinza escuro e os obstáculos são marcados com preto. A direção do nó pai é definida por uma seta e cada nó visitado possui o custo do caminho. Na figura foi utilizada a distância Manhattan[1] como função de custo .

[1] Distância entre dois pontos medidos ao longo de eixos em ângulos retos (BLACK, 1998)

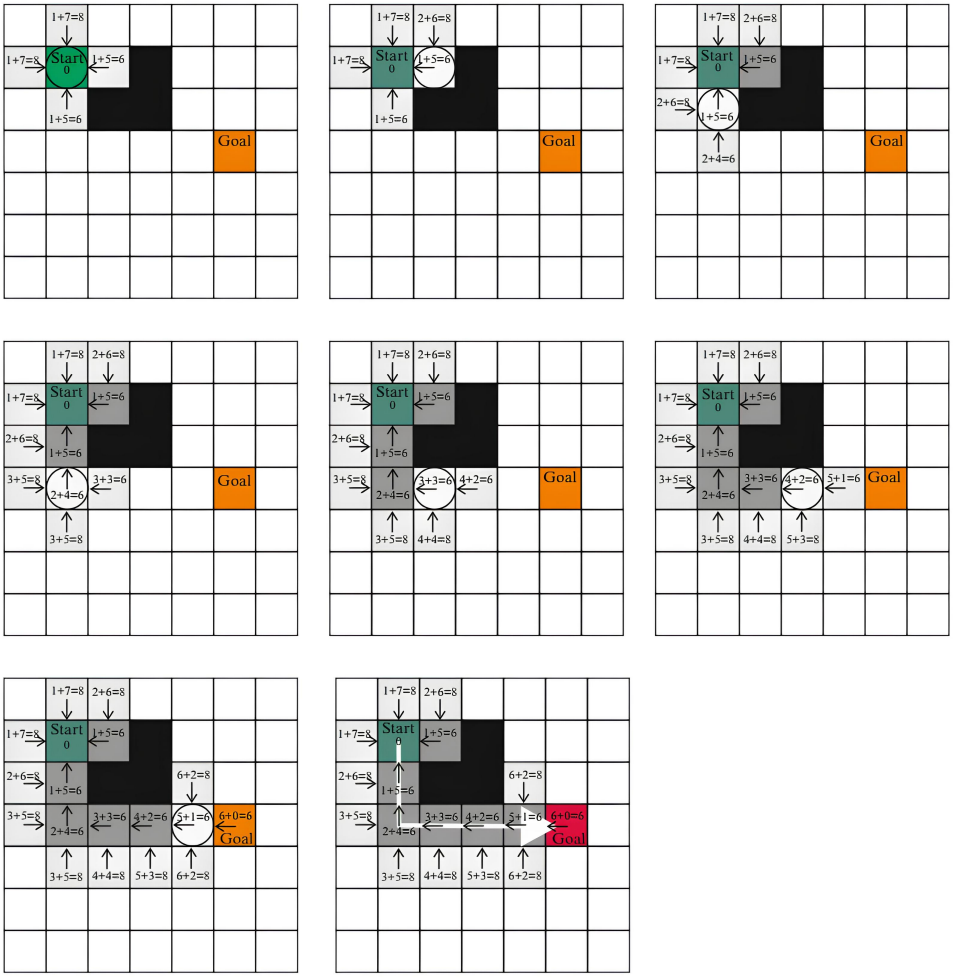


Figura 12 – Evolução do algoritmo A* (Extraído de Klancar *et al.* (2017)).

Uma peculiaridade é que se todos os *custos-destino* forem definidos como zero, o A* funciona igual ao algoritmo Dijkstra. Na Figura 13 é mostrada uma comparação de desempenho entre os dois algoritmos. Na imagem é possível identificar que o algoritmo A* apresenta uma melhor eficiência, pois para encontrar a menor distância entre dois nós, analisa uma quantidade menor de nós quando comparado ao algoritmo de Dijkstra. Por outro lado, devido à adoção de uma função heurística, sua utilização pode acaber sendo inviabilizada a depender da aplicação. Assim, seu uso é mais difundido em aplicações que utilizam grade, uma vez que nesse modelo o cálculo é simplificado quando comparado com aqueles que possuem uma estrutura mais heterogênea.

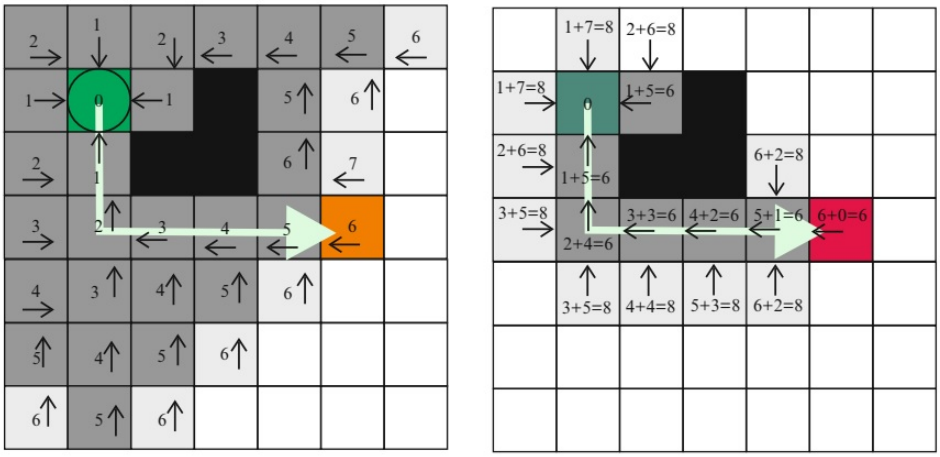


Figura 13 – Comparação do A* (direita) com o Dijkstra (esquerda) (Extraído de Klancar *et al.* (2017)).

CAPÍTULO

3

REVISÃO DA LITERATURA

Neste capítulo são apresentados os principais trabalhos relacionados com esta pesquisa, alguns dos quais contribuíram na construção da proposta e outros que surgiram como referencial teórico. Os trabalhos são agrupados por tema e suas relações com este estudo são também exploradas.

Inicialmente é apresentado o problema do planejamento de rotas com a utilização de PRM, demonstrando os resultados existentes, bem como pontos que podem ser melhorados. Por fim, são apresentados diversos estudos sobre otimização do cálculo do menor caminho.

3.1 Planejamento de rotas

Uma das atividades mais críticas e fundamentais na área de robótica é o planejamento de rotas. Apesar de atualmente existirem diversas formas de se resolver esse problema, a complexidade das soluções tende a aumentar conforme são incluídos mais graus de liberdade no robô. Atualmente, apesar de existirem robôs com alto nível de liberdade de movimento, muitas vezes não é possível empregar todo o seu potencial devido à limitação dos algoritmos utilizados (LIU *et al.*, 2021).

A utilização de técnicas de paralelismo para melhorar o desempenho nesse tipo de aplicação já se mostrou acertada, como demonstrado, por exemplo, em (CABODI *et al.*, 2019) - uma abordagem paralela para resolver RRT em problemas de tempo real.

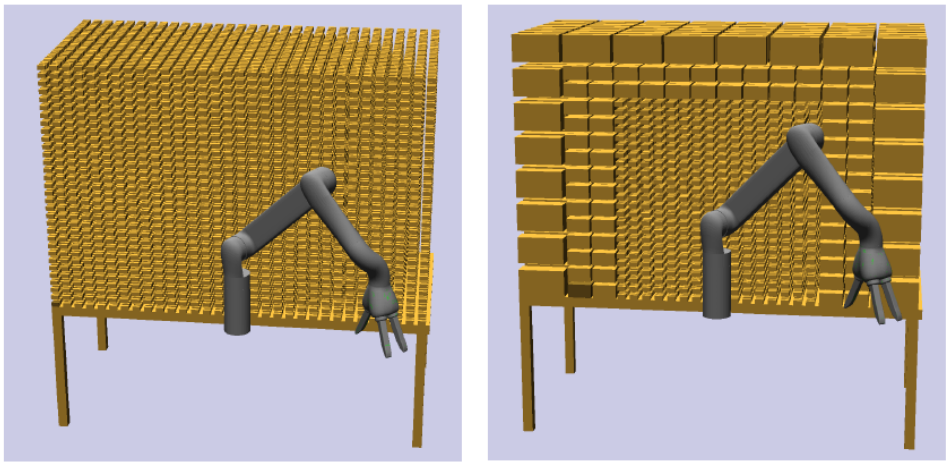
Trabalhos como (LIU *et al.*, 2023) aplicam uma variante da técnica RRT* que adiciona uma operação de religação (do inglês *rewiring*) para melhorar a qualidade das soluções alcançadas. Além disso, eles usam técnicas de compressão de dados para reduzir o impacto das transferências de informações em uma plataforma heterogênea de CPU/GPU. No entanto, apesar dos resultados favoráveis, como seu tempo de processamento é da ordem de segundos, eles não são adequados para aplicações em tempo real.

Em (HORTELANO *et al.*, 2023), a utilização de GPU produz uma notável aceleração de 5× em média em comparação com implementações C++ brutas para cálculos de previsão de movimento. Além disso, para robôs com mais de seis graus de liberdade, muitas soluções que oferecem tempos de resposta na ordem de centenas de milissegundos são implementadas em GPUs, como mostrado por (PAN; MANOCHA, 2012) e (PAN; LAUTERBACH; MANOCHA, 2010).

Como alternativa, a utilização de um FPGA já foi explorada assertivamente em alguns trabalhos como em (ATAY; BAYAZIT, 2006), onde se constrói um planejador de rotas e detector de colisões em FPGA com resultado 25 vezes mais rápido do que o obtido em uma CPU.

Em (MURRAY *et al.*, 2016a) e em (MURRAY *et al.*, 2016b) é apresentada uma proposta de solução para o problema de planejamento de caminhos também com a utilização de um hardware dedicado como o FPGA. A técnica utilizada é aplicada em um robô Kinova Jaco2. Esses trabalhos exibiram uma melhora no desempenho em três ordens de magnitude e redução do consumo de energia em mais de uma ordem quando comparado com outras pesquisas, assim apresentando valores promissores, ao se considerar que, além de reduzir significativamente o tempo de resposta, também alcançam uma redução do consumo de energia. Esse último é um aspecto crítico quando se consideram aplicações que utilizam robôs autônomos alimentados por baterias.

Para alcançar esse ganho, foram empregadas diversas técnicas de otimização, como a utilização de uma discretização não uniforme do ambiente, aplicando maior resolução em áreas que necessitam maior precisão (ver Figura 14). Essa estratégia reduziu em 27% o uso de espaço no FPGA. Também foram utilizadas ferramentas adicionais para minimizar a lógica a ser sintetizada, fazendo uma análise prévia à síntese com a ferramenta *espresso*, o que possibilitou reduzir a lógica necessária em 25%.



(a) (b)

Figura 14 – a) Discretização uniforme. b) Abordagem mais sofisticada ao aplicar uma maior resolução em regiões mais críticas. (Adaptado de Murray *et al.* (2016a)).

Os autores utilizaram a construção de um PRM como base para o desenvolvimento da

solução. Trabalhos recentes como (SAKAMOTO; HARADA; WAN, 2020) e (FRANCIS *et al.*, 2020) confirmaram as vantagens da utilização de PRMs na redução do tempo de processamento necessário para o planejamento de rotas. Na etapa de pré-processamento do PRM, realizada apenas uma vez, o *rodmap* é construído considerando apenas as colisões permanentes no ambiente e as auto-colisões do robô. Sequencialmente, o espaço de configuração é discretizado e conjuntos dos vértices que colidem com cada arco são identificados e armazenados. Por fim, as arestas são representadas como circuitos lógicos sintetizáveis.

A fase de consulta, realizada a cada novo planejamento, utiliza os dados coletados dos sensores para popular uma grade de ocupação, utilizando o mesmo nível de discretização em que o mapa de navegação foi construído. As colisões são encontradas ao analisarem-se os vértices presentes na grade de ocupação, e então, o mapa de navegação é atualizado. Por fim, com um novo mapa de navegação, o algoritmo de menor caminho é aplicado e, caso exista, uma rota livre de obstáculos a ser utilizada pelo robô é gerada.

Como resultado, o processo de construção do mapa de navegação e de obtenção do menor caminho com a utilização de FPGAs, utilizando um *clock* com frequência de 125 MHz, pôde ser realizado em 650 microssegundos. No entanto, apesar de uma melhora significativa de desempenho no processo de criação do *rodmap*, o resultado final foi comprometido pelo algoritmo de menor caminho. Desse modo, dos 650 microssegundos totais necessários ao processamento, 425 eram utilizados apenas pelo algoritmo de menor caminho. Além disso, por não ter sido implementado em FPGA, o resultado do PRM precisa ser exportado e processado na CPU. Essa necessidade, além de acrescentar um custo de comunicação, também promove uma redução de eficiência, pois não utiliza um hardware especializado no cálculo do menor caminho.

O resultado comparativo de desempenho desse trabalho pode ser visualizado na Tabela 2, onde se encontram: a abordagem em FPGA, apresentando um tempo de processamento de 650 microssegundos; os resultados de uma técnica que se utiliza de conjuntos *hash* pré-computados em Unidade Central de Processamento (do inglês *Central Processing Unit*) (CPU) e GPU com 10.000 e 1.600 microssegundos, respectivamente; e, por fim, os resultados de uma implementação em CPU com a utilização de PRM e RRT, apresentando 815.000 e 756.000 microssegundos, respectivamente.

Tabela 2 – Tempo necessário em microssegundos para o planejamento de movimento em diferentes abordagens. (Adaptado de Murray *et al.* (2016a)).

Hardware Dedicado	**Conjuntos Hash Pré computados**		**Abordagem via Software(CPU)**	
FPGA	**CPU**	**GPU**	**PRM**	**RRT**
650	10.000	1.600	815.000	756.000

Os principais trabalhos citados nesta seção, juntamente com suas contribuições, podem ser vistos na Tabela 3.

Tabela 3 – Comparação dos trabalhos relacionados.

Técnica	**Referência**	**Vantagens**	**Tipo de Hardware**
RRT* em paralelo	(CABODI *et al.*, 2019)	Permite soluções otimizadas para problemas computacionalmente complexos	CPU
RRT* com *Rewiring*	(LIU *et al.*, 2023)	Melhora a qualidade da solução e reduz a sobrecarga de transferência de dados (até 10 vezes sobre o algoritmo RRT*)	CPU/GPU
Previsão de movimento com aceleração em GPU	(HORTELANO *et al.*, 2023)	Aceleração de 5 vezes em comparação com implementações em C++	GPU
Previsão de movimento para robôs com alto DoF	(PAN; MANOCHA, 2012)	Tempo de resposta em milissegundos	GPU
Planejamento de rota e detecção de colisão baseados em FPGA	(ATAY; BAYAZIT, 2006)	Atinge velocidade 25x mais rápida em comparação com a CPU	FPGA
Planejamento de caminho baseado em PRM com FPGA	(MURRAY *et al.*, 2016a)	Oferece desempenho significativo (melhoria de 3 ordens de magnitude)	FPGA

3.2 Otimização do cálculo do menor caminho

A evolução da capacidade de processamento nos diversos dispositivos envolvidos na movimentação do robô, viabilizou uma melhora significativa das suas aplicabilidades nos últimos anos. Do mesmo modo, por ser um método crucial na atividade de planejamento de rotas, a identificação do menor caminho também precisou evoluir, se adequando às novas necessidades para que resultados melhores pudessem ser obtidos.

Na área de algoritmos e processamento de dados, o problema do menor caminho é um dos mais antigos e fundamentais. Com o surgimento de novos ramos da tecnologia, como a robótica, sua aplicabilidade aumentou ainda mais e novos estudos passaram a ser desenvolvidos. Dentre as soluções utilizadas para a resolução desse problema, a que mais se destaca é o algoritmo de Dijkstra. Sua característica de sempre encontrar o menor caminho ótimo, quando houver, fez com que essa técnica pudesse ser explorada por décadas. Com o passar dos anos, diversas implementações e técnicas de paralelismo foram desenvolvidas com o intuito de aprimorar seu desempenho.

Como apresentado na seção 3.1, alguns estudos, apesar de apresentarem resultados promissores, poderiam obter resultados ainda melhores, caso optassem por melhores técnicas de cálculo de menor caminho. Nesse contexto, a solução adequada para o menor caminho deveria apresentar, além de uma melhora de desempenho, uma adaptação dinâmica à mudança no grafo a ser analisado, uma vez que arestas podem ser inseridas ou excluídas de modo temporalmente aleatório conforme as mudanças no ambiente.

Desse modo, ficou evidente a necessidade de uma nova abordagem para o tratamento do menor caminho, levando em conta essas condições especiais, permitindo assim atender demandas semelhantes que possam surgir em contextos diferentes. Assim, foi realizado um estudo sobre o estado da arte desse tipo de algoritmo, priorizando o desempenho e as possibilidades de exploração de paralelismo.

Por ser um método bastante utilizado, existem diversos trabalhos que buscam melhorar o desempenho desse tipo de algoritmo. Trabalhos recentes, como (PRASAD; KRISHNAMURTHY; KIM, 2018), propõem soluções de melhoria com a utilização de CPUs. Nesse caso, foi possível atingir um desempenho de até 51% utilizando uma estratégia com multiprocessadores para um grafo com 10^8 nós.

Em (JASIKA *et al.*, 2012) é exposto um estudo sobre os ganhos na paralelização do algoritmo base de Dijkstra com a utilização de *Open Multi-Processing* (OpenMP) e *Open Computing Language* (OpenCL) aplicados a CPUs. Apesar de resultados melhores na alternativa paralelizada, o estudo utiliza a estrutura base do algoritmo que, por ser essencialmente sequencial, prejudica o desempenho da paralelização, Na média, os testes obtiveram uma melhora de 10% no desempenho.

Existem também estudos que exploram a natureza paralela das placas gráficas, como (THOUTI; SATHE, 2013). A partir dos trabalhos realizados por (HARISH; NARAYANAN, 2007) e (BULUÇ; GILBERT; BUDAK, 2010), em que técnicas de busca do menor caminho são exploradas, (THOUTI; SATHE, 2013) desenvolveu uma solução para GPU utilizando *Message Passing Interface* (MPI) e OpenCL. Essa solução obteve um ganho de desempenho entre 10 e 15 vezes quando comparada aos resultados obtidos de modo sequencial em CPU.

Há também trabalhos que realizam otimizações com a utilização de hardware dedicado, como os FPGAs. Em (TOMMISKA; SKYTTÄ, 2001), por exemplo, foi desenvolvida uma solução em FPGA que apresentou para uma aplicação com 64 nós, um desempenho 67 vezes melhor (aproximadamente) do que a versão em processadores.

Uma arquitetura dedicada em FPGA para resolver o problema de construção de tabela de roteamento em redes *Open Shortest Path First* (OSPF) foi proposta por (ABDUL; ALWAN; AL-EBADI, 2012). O estudo obtém uma melhoria de desempenho de até 76x em relação à implementação padrão do Dijkstra em uma CPU para um grafo com 128 nós. No entanto, a complexidade da arquitetura limitou a solução para até 128 nós no dispositivo escolhido.

O trabalho conduzido por (CHIRILA *et al.*, 2022) introduz uma técnica de algoritmo híbrido - de acordo com (BADR; MOUSSA, 2020), um algoritmo $O(n^3)$ - projetado especificamente para abordar problemas de menor caminho entre todos os pares (do inglês *All-Pairs Shortest Path*) (APSP) em grafos com mais de 4.096 nós. O estudo apresenta resultados favoráveis quando comparado a soluções similares. Contudo, é importante notar que o problema APSP acarreta maior complexidade comparado ao problema SSSP. Isto indica que um projeto dedicado exclusivamente ao problema SSSP pode potencialmente alcançar ganhos e otimizações ainda maiores.

Como demonstrado, o algoritmo de Dijkstra vem sendo constantemente testado e aprimorado. Nas seções a seguir, serão expostas quatro técnicas diferentes encontradas na literatura que possibilitam, com suas peculiaridades, um ganho de desempenho na análise do menor

caminho. As técnicas que serão apresentadas contribuíram para um melhor entendimento das possibilidades de melhoria existentes, bem como das suas limitações. Os conhecimentos aqui levantados, como será possível observar nos próximos capítulos, formaram o alicerce para a construção de uma nova proposta de solução.

3.2.1 Particionamento do Grafo

Uma solução para viabilizar o paralelismo através do particionamento do grafo é apresentada por (SCHÜTZ, 2005) (Ver Figura 15). O autor explora a característica estática do grafo ao realizar um pré-processamento visando a aceleração da análise. Essa técnica pode ser utilizada em grafos estáticos devido ao pré-processamento *offline* necessário. O método consegue identificar e ignorar arestas que não levam ao nó destino, reduzindo a quantidade de nós visitados para menos de 0,2%. Para realizar isso, divide os nós do grafo em regiões e, durante o pré-processamento, identifica as regiões que compõem o menor caminho.

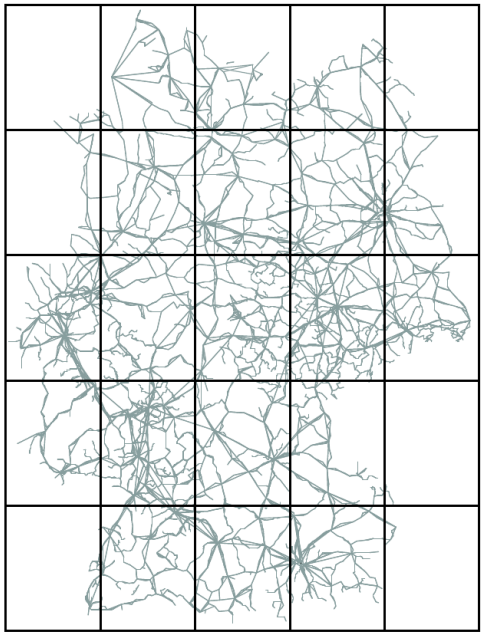


Figura 15 – Um grid 5x5 das estradas da Alemanha (Extraído de Schütz (2005)).

O autor destaca que podem ocorrer problemas quando se combinam técnicas de otimização em grafos com mais de um menor caminho e sugere, como alternativa para eliminar esse problema, a inclusão de valores fracionários pequenos aos pesos do grafo. Ele aborda também a utilização de filas de Fibonacci para a lista de prioridade dos algoritmos de Dijkstra, pois elas apresentam um pior caso de $O(m + n \log n)$, sendo m o número de bordas e n o número de nós.

Vale destacar que um sub-caminho de um menor caminho também é um menor caminho, ou seja: se $s - n_1 - \ldots - n_k - t$ é o menor caminho de s a t então $n_i - \ldots - n_k - t$ é o menor caminho de n_i até t. Então, ao calcular o menor caminho entre todos os pares de nós é possível identificar quais nós fazem parte do menor caminho até o destino. Com o objetivo de reduzir o

custo computacional dessa operação, utiliza-se uma técnica de particionamento do grafo, a qual possibilita a identificação dos nós que não fazem parte do menor caminho até o destino. Essa técnica consiste em dividir o grafo em partições e, com o auxílio de um vetor de bits, mapear, para cada nó, em quais regiões esse nó possui um menor caminho até o destino.

As partições são representadas por um vetor com n posições (uma para cada nó) e com p bits, sendo p a quantidade de regiões. Para cada nó do grafo, são marcados os bits das regiões em que aquele nó faz parte de um menor caminho. Dessa forma, é possível identificar os nós que não possuem um menor caminho para a região pertencente ao nó destino, e, a partir disso, ignorar esses nós durante o processamento.

Para a implementação dessa técnica, é necessária a realização de um pré-processamento, no qual o vetor de bits é alimentado com as informações de menor caminho de todos os nós. Isso pode ser calculado ao encontrar o menor caminho entre todos os nós.

Todos os menores caminhos de um caminho s para uma região R, de número p_r, precisam entrar na região R. Caso s não faça parte da região R, existe um arco $e = (u,v)$ com $r(u) \neq p_r$ e $r(v) = p_r$. Levando isso em consideração, é suficiente que o algoritmo de preprocessamento apenas considere menores caminhos para os nós v que se encontram em regiões de fronteira, não sendo necessário calcular o menor caminho entre todos os nós. Para implementar essa técnica, utiliza-se um grafo reverso D_{rev}, que permite a propagação do menor caminho no sentido inverso. Nesse caso, é calculado o menor caminho entre os nós localizados na fronteira da região de interesse e os nós pertencentes às outras regiões.

Uma análise sobre o tamanho das partições também é realizada, indicando que dividir o grafo em partições menores pode gerar um melhor desempenho, porém com um aumento significativo no tamanho do vetor de bits. Para contornar esse problema, sugere-se a criação de partições em dois níveis para otimizar a busca. Apesar desse método ainda gerar um aumento da quantidade de bits, tal aumento é significativamente inferior ao da abordagem anterior.

Os autores apresentam também uma técnica de busca bidirecional, que permite uma redução de um fator de 4 em relação ao algoritmo padrão. Essa técnica pode ser utilizada com o método de particionamento, no entanto é necessário realizar o pré-processamento em ambos os sentidos.

Com relação ao método de particionamento, os autores apresentam três tipos de particionamento do grafo:

1. Grid - método mais simples e com os piores resultados. Não considera as características do grafo (ver Figura 15);
2. Quadtrees - reflete a geometria do grafo, de modo que regiões mais densas terão mais divisões. Divide cada sub-região em quatro (ver Figura 16);

3. kd-Trees - para o tipo de grafo utilizado no estudo (rede de estradas), apresentou os melhores resultados. Também divide o grafo em quatro regiões com tamanho equalizado (ver Figura 17).

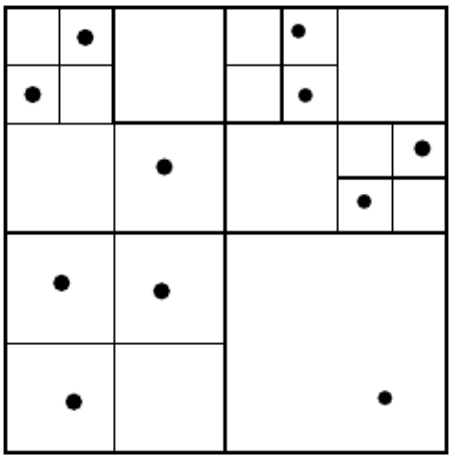


Figura 16 – Exemplo de uma *quadtree* (Extraído de Schütz (2005)).

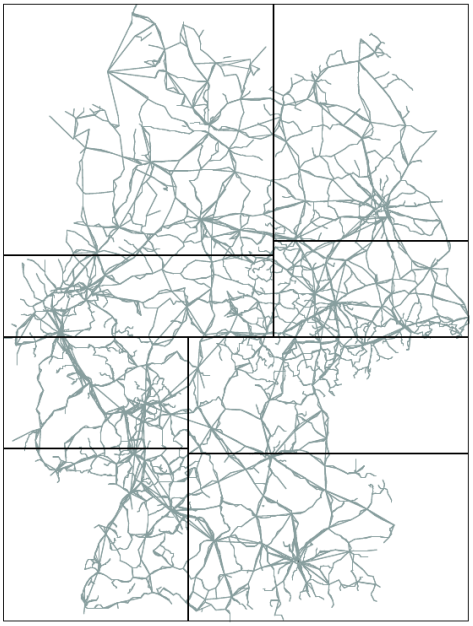


Figura 17 – Exemplo de uma *kd-tree* com 8 regiões (Extraído de Schütz (2005)).

Nas etapas iniciais de desenvolvimento de nosso trabalho, considerou-se investir no desenvolvimento de uma solução com a utilização da técnica de particionamento do grafo. Essa abordagem poderia ser eficiente ao reduzir o tempo de cálculo do menor caminho com a utilização do paralelismo através do particionamento. No entanto, devido à necessidade de um pré-processamento *offline*, esse método deve ser preferencialmente utilizado em robôs cujos obstáculos não variam durante sua operação, visto que, sempre que fosse necessário alterar o espaço de configuração, um novo pré-processamento deveria ser realizado. Essa desvantagem

acabou desfavorecendo a utilização dessa técnica, pois, como será possível observar nas próximas seções, foram encontradas outras técnicas que permitem a utilização do paralelismo e que, ao mesmo tempo, podem ser adaptadas para espaços de configurações mutáveis.

3.2.2 Encontrando o menor caminho de forma bidirecional

Previamente utilizada em (SCHÜTZ, 2005), a abordagem bidirecional permite o paralelismo ao aplicar o algoritmo de Dijkstra em dois sentidos, da origem para o destino e do destino para a origem. Isso permite que o processamento possa ser realizado em paralelo. No entanto, essa técnica nem sempre obtém o menor caminho.

Como alternativa, (VAIRA; KURASOVA, 2011) explora a utilização de um método bidirecional focado em obter sempre o menor caminho. Para isso, altera-se o método bidirecional original utilizado por (BERRETTINI; D'ANGELO; DELLING, 2009; SCHÜTZ, 2005; GOLDBERG, 2007) combinando os dois fatores que definem a condição de parada. Apesar de ser mais lento que o método original, ele se mantém mais rápido que a abordagem tradicional do algoritmo de Dijkstra.

A técnica possui duas condições de parada: i) quando o nó w que está sendo classificado também já foi classificado pela outra parte da busca ($Q_s \cap Q_b = \{w\}$); ii) quando o nó w que está sendo visitado também já foi visitado pela outra parte da busca ($S_s \cap S_b = \{w\}$) (VAIRA; KURASOVA, 2011).

Utiliza-se de *Mutual Exclusion* (Mutex) para realizar o paralelismo em multiprocessadores, garantindo que apenas um processo esteja executando uma operação protegida pelo objeto Mutex em regiões críticas da memória. Essas regiões sensíveis correspondem às áreas de armazenamento dos conjuntos dos nós visitados (S) e de possíveis respostas (R).

Objetivando validar a proposta, os autores realizam simulações de busca do menor caminho com os grafos carregados diretamente na memória, pois, segundo eles, a leitura constante dos dados em disco poderia distorcer os resultados. As simulações são realizadas com três soluções diferentes: i) o método de Dijkstra padrão; ii) o método bidirecional modificado; iii) e o método bidirecional modificado com paralelismo.

Os resultados desses testes indicam que o algoritmo bidirecional modificado é cerca de 2 vezes mais rápido do que o algoritmo Dijkstra padrão. O algoritmo paralelo apresenta-se até 2,9 vezes mais rápido do que o Dijkstra padrão, e 1,4 vezes mais rápido do que o algoritmo Dijkstra bidirecional. Por fim, os autores realizaram testes com o tempo total de processamento, cujas médias dos resultados obtidos podem ser vistos na Tabela 4.

Um dos pontos negativos dessa abordagem é que ela não pode ser utilizada para uma grande quantidade de processadores, sendo limitada a apenas 2 processos em paralelo. Além disso, é necessário armazenar duas vezes o grafo de relações: a primeira vez na forma direta, normal do grafo; a segunda vez com a direção dos arcos invertida, para permitir a propagação

Tabela 4 – Média de tempo de execução e média de nós processados. (Adaptado de (VAIRA; KURASOVA, 2011)).

Abordagem	Tempo de Execução (μs)	Nós processados
Dijkstra Padrão	0,536	53.738,97
Bidirecional	0,350	41.530,77
Paralelo	0,219	41.671,60

a partir do destino até a fonte. Essa característica contribui para um aumento na quantidade de memória necessária para o processamento. De modo simples, pode-se afirmar que tal abordagem utiliza o dobro de recursos de memória para se obter o dobro de desempenho.

3.2.3 Utilização de um critério para remoção dos nós

A utilização de um critério de classificação para viabilizar o paralelismo é apresentada por (CRAUSER *et al.*, 1998). A técnica divide o algoritmo de Dijkstra em fases e realiza as operações de cada fase de forma paralela. Classificam-se os nós em 3 tipos: i) estabelecido; ii) empilhado; e iii) inalcançado.

Para cada nó v é calculada uma tentativa de distância $tent(v)$, medida entre v e a fonte s. Para nós inalcançados $tent(v) = \infty$. Inicialmente a fonte s é marcada como empilhada, $tent(s) = 0$, e todos os outros nós são inalcançados. Para cada iteração, o nó empilhado com a menor tentativa de distância é selecionado e declarado estabelecido, e para todas as suas bordas é calculado o valor de $tent(w) = min\{tent(w), tent(v) + c(v,w)\}$, sendo w o nó vizinho e $c(v,w)$ o custo entre v e w.

Nesse caso, deve-se observar os seguintes pontos:

1. Se o nó w estava marcado como inalcançado ele é agora marcado como empilhado;
2. $tent(v) = dist(v)$ quando v é selecionado da pilha, pois ele é aquele que possui o menor $tent(v)$ e, consequentemente, menor distância até a fonte, ou seja, sua menor distância não irá mudar; e
3. A pilha pode conter mais de um nó v com $tent(v) = dist(v)$.

Todos os nós com $tent(v) = dist(v)$ podem ser removidos da pilha de forma simultânea, sendo objetivo dos autores apresentar um critério para realizar essa identificação. Nesse sentido, sugere-se que o algoritmo possa ser implementado utilizando três análises diferentes: *IN*, *OUT* e *INOUT*.

No tipo *OUT*, o ponto de corte é definido de acordo com o peso das bordas de saída, onde $L = min\{tent(u) + c(u,z) : u \ \ empilhado \ e \ (u,z) \in E\}$. Todos os nós com $tent(v) \leq L$ são removidos, nestes casos $tent(v) = dist(v)$. O limiar para essa versão pode ser calculado também

na forma de uma segunda lista de prioridades com $o(v) = tent(v) + min\{c(v,u) : (v,u) \in E\}$, ou até mesmo em tempo real durante a remoção dos nós.

No tipo *IN*, o ponto de corte é definido de acordo com as bordas de entrada, sendo $M = min\{tent(u) : u \ empilhado\}$ e $i(v) = tent(v) - min\{c(u,v) : (u,v) \in E\}$ para qualquer vértice v. Os nós com $i(v) \leq M$ podem então ser removidos. Por fim, a versão *INOUT* aplica os dois critérios de forma conjunta.

As pilhas são implementadas como filas relaxadas (DRISCOLL *et al.*, 1988), pois elas possuem as seguintes configurações no pior caso: *ENCONTRAR_MINIMO*, *INSERIR* e *DIMINUIR_CHAVE* são executadas em $O(1)$ e *REMOVER_MINIMO/REMOVER* em $O(\log q)$, sendo q o tamanho da fila.

O algoritmo de Dijkstra padrão utilizando filas Fibonacci possui um tempo de execução de $O(n \log n + m)$, sendo n a quantidade de nós e m a quantidade de arestas. Com a técnica sugerida, é possível atingir um tempo de execução de $O(n^{1/3} logn)$ e $O(n \log n + m)$ para tarefas com maior probabilidade.

Para a variante OUT, aplicada em grafos aleatórios com pesos também aleatórios, o método obteve $2,5\sqrt{n}$ fases. A versão refinada da variante *INOUT* obteve $2,5n^{1/3}$ fases na média. Já uma versão modificada da variante *INOUT* obteve $8,5n^{1/3}$ fases.

3.2.4 O algoritmo de Dijkstra impaciente

Duas opções de paralelismo para o Algoritmo de Dijkstra foram desenvolvidas para a biblioteca *Boost Graph Library* (BGL) por (EDMONDS *et al.*, 2006).

A primeira alternativa consiste em expandir todas as arestas que saem do vértice ativo u em paralelo, paralelizando assim o laço interno do algoritmo de Dijkstra. No entanto, a melhora de desempenho que se pode obter com essa abordagem é limitada pela quantidade de vizinhos de saída do vértice. Além disso, a maioria dos grafos do mundo real não apresentam uma grande quantidade de vizinhos. Portanto, paralelizar o relaxamento das bordas de saída provavelmente não resultará em boa escalabilidade.

A segunda alternativa de paralelização consiste em remover vários vértices da fila de prioridade simultaneamente, relaxando suas bordas em paralelo. A cada iteração do algoritmo de Dijkstra, o vértice u ativo com a menor distância global até a fonte é selecionado e tem seus vizinhos expandidos. (EDMONDS *et al.*, 2006) mostra inicialmente que seria possível remover e expandir todos os vértices u com $d(u) = \mu$. Dessa forma, a escalabilidade do algoritmo seria relacionada ao número de vértices que poderiam ser removidos da fila de prioridade. Idealmente, cada iteração removeria um grande número de vértices uniformemente distribuídos entre os processadores. Contudo, os grafos do mundo real raramente têm um grande número de vértices com a mesma distância do vértice de origem, portanto, o grau de paralelismo que se pode extrair dessa paralelização direta acabaria se tornando limitado em aplicações reais.

Uma alternativa para aumentar o paralelismo seria através da remoção de mais vértices a cada iteração, por meio da adoção de um limiar μ como parâmetro de remoção, ao invés de remover apenas o vértice com a menor distância. Desse modo, vértices cuja distância até a fonte fossem maiores do que μ poderiam ser removidos. No entanto, podem existir casos de vértices removidos com $d(u) > \mu$ que necessitem ser reinseridos na fila de prioridades, caso uma distância menor fosse encontrada em iterações futuras. Nesses casos, os vizinhos do nó deveriam ser novamente expandidos com o valor atualizado de $d(u)$. Portanto, é necessário haver um equilíbrio entre explorar mais paralelismo e evitar trabalho desnecessário.

Considerando o exposto, (EDMONDS *et al.*, 2006) apresentam uma variante do algoritmo de Dijkstra denominada Dijkstra impaciente (tradução de "*eager*" Dijkstra). Essa versão utiliza um fator γ constante para selecionar os nós a serem processados em paralelo, considerando $d(u) \leq \mu + \gamma$, ordenado por valores crescentes de $d(u)$.

Uma variante da solução apresentada em (EDMONDS *et al.*, 2006) foi desenvolvida em FPGA por (LEI *et al.*, 2015). Resultados experimentais mostraram que a solução em FPGA apresentou uma melhoria de 5x com relação à implementação em CPU com apenas 1/4 do consumo.

Como descrito na Seção 3.2.3, a solução criada por (CRAUSER *et al.*, 1998) usa heurísticas mais precisas para aumentar o número de vértices removidos em cada iteração sem causar nenhuma reinserção. O algoritmo usa dois critérios separados, o critério OUT e o critério IN, que podem ser combinados para determinar quais vértices devem ser removidos em uma determinada etapa. Ao contrário do algoritmo impaciente, não há parâmetros que precisem ser ajustados, pois as remoções são realizadas de acordo com os parâmetros de distâncias e custos dos vizinhos.

Além disso, a necessidade de manter o armazenamento das distâncias dos nós em memória na implementação do Dijkstra impaciente, devido às possibilidades de reinserção, contribui para um aumento na utilização de memória. Com a solução apresentada por (CRAUSER *et al.*, 1998), é necessário armazenar em memória apenas as distâncias dos nós ativos, os quais, uma vez estabelecidos, não precisam mais ter suas distâncias analisadas, o que resulta na liberação de espaço.

CAPÍTULO

4

PROPOSTA PARA O ALGORITMO DE MENOR CAMINHO

O problema do menor caminho a partir de uma fonte é um tema bastante discutido na literatura, com a existência de diferentes implementações em inúmeras aplicações. Neste contexto, a grande maioria das soluções são implementadas em CPUs. De modo geral, isso não é um problema, pois boa parte das aplicações que utilizam este tipo de algoritmo não possuem requisitos críticos de tempo de resposta.

No entanto, existem soluções que possuem maiores restrições de tempo de resposta, como é o caso das aplicações que exigem um processamento em tempo real. Na área de robótica, por exemplo, considerando um cenário com obstáculos dinâmicos, o robô necessita de um constante processo de mapeamento da área ao seu redor, assim, ele consegue identificar o surgimento e desaparecimento de obstáculos para que suas decisões de deslocamento possam ser tomadas. Estas decisões precisam ser rápidas o suficiente para conseguir acompanhar as mudanças no ambiente.

Uma forma de melhorar o desempenho de uma aplicação é através da utilização de aceleradores em hardware especializado. Essa abordagem permite que uma parte ou o todo de um algoritmo seja processada nestes aceleradores viabilizando, assim, um maior ganho de desempenho.

Os aceleradores podem ser classificados como fortemente ou fracamente acoplados, dependendo do tipo de conexão com o processador principal. No primeiro tipo, o acelerador é implantado como parte do computador, diretamente ligado ao barramento do sistema ou ao chip do microprocessador. Já no modelo fracamente acoplado, o acelerador está ligado ao sistema através de uma interface de expansão, como a *Peripheral Component Interconnect Express* (PCIe).

FPGAs e GPUs são os dois principais tipos de dispositivos utilizados como aceleradores,

com maior ou menor adequação conforme a carga de trabalho. FPGA é um hardware altamente configurável, de temporização determinística e interfaces de projeto diversificadas. Já GPUs fornecem unidades de execução paralela maciça e alta largura de banda de memória, através de uma estrutura regular, fortemente acoplada.

Devido à sua reconfigurabilidade, FPGAs permitem implementar apenas os circuitos necessários para uma aplicação específica, em vez de apenas replicar todas as unidades de processamento (como ocorre nos múltiplos núcleos de GPU). O principal problema das GPUs é sua baixa eficiência energética, quando comparadas a FPGAs. Os FPGAs atuais podem fornecer desempenho igual ou até melhor do que GPUs, enquanto consomem uma média de 28% da energia da GPU (CONG *et al.*, 2018). Trabalhos como (GUO *et al.*, 2019) evidênciam os diferentes ganhos possíveis obtidos entre GPUs e FPGA, neste caso, se atingiu uma aceleração de 7 vezes com GPU e cerca de 28 vezes com FPGA quando comparados a uma implementação de um algoritmo implementado em CPUs com multithread.

Outra vantagem na utilização de FPGAs é a possibilidade de migração para Circuitos Integrados de Aplicação Especifica (do inglês *Application-Specific Integrated Circuit*) (ASIC), caso seja necessária uma solução com ainda mais desempenho. Apesar de abordagens diferentes, um projeto em FPGA pode ser traduzido em ASIC sem que muitas alterações na microarquitetura sejam necessárias. A similaridade é tamanha que os projetos em ASIC muitas vezes se utilizam de FPGAs na etapa de testes físicos e validação da arquitetura construída. O ASIC possui duas características principais que são versatilidade e alto desempenho, uma vez que não existe de fato uma limitação na definição de uma arquitetura para um ASIC. Por outro lado, tanto o FPGA quanto a GPU são dispositivos com recursos definidos e finitos, o que força a adequação do projeto a tais limitações.

Como exposto no início do Capítulo 3, existem diversas implementações tanto para CPU, quanto para GPU e FPGA do algoritmo de menor caminho. No entanto, os resultados apresentados pelos estudos com FPGA se destacam com relação aos demais devido à sua característica de fornecer uma solução altamente especializada.

Em Robótica, o cálculo do menor caminho é apenas uma das etapas necessárias ao planejamento de rotas. Inicialmente, é necessário mapear o ambiente, discretizá-lo, identificar os obstáculos e criar um mapa de navegação, obtendo, como resultado, a representação do ambiente navegável em um grafo. Somente após a obtenção desta representação é que a técnica de identificação do menor caminho é aplicada. Desse modo, para que uma solução em tempo real se torne viável, é necessário que todo o processo seja otimizado. Nossa proposta busca apresentar uma solução em *hardware* dedicado que possibilite melhorar o ganho no processo de identificação do menor caminho entre dois nós de um grafo com obstáculos dinâmicos. Além disso, visando um melhor desempenho, nossa proposta foi construída de modo que permita uma fácil integração com os módulos responsáveis por realizar as outras etapas do processo de planejamento de rotas.

Nesse contexto, diversas técnicas foram analisadas, dentre as quais destacam-se a realização de particionamentos dos grafos (SCHÜTZ, 2005), a aplicação de um processamento bidirecional (VAIRA; KURASOVA, 2011) e duas formas de aplicação de critérios para remoção de nós ativos: a primeira através da utilização de limiares calculados a partir da expansão dos nós ativos (CRAUSER *et al.*, 1998); e a segunda baseada na utilização de um limiar constante aplicado ao nó ativo com menor distância à fonte (EDMONDS *et al.*, 2006).

Dentre as técnicas analisadas, a solução proposta por (CRAUSER *et al.*, 1998) foi a que apresentou uma maior compatibilidade com as necessidades de nossa proposta, uma vez que, além de apresentar um potencial de paralelismo, possui também uma perspectiva de consumo de recursos computacionais menor que as demais. Além disso, até o momento, não foram encontradas implementações em FPGA desse modelo, o que contribui para o aspecto inovador desta tese. Desse modo, o processo de adaptação dessa solução para FPGA foi iniciado. Trabalhos recentes, como os apresentados em (PRASAD; KRISHNAMURTHY; KIM, 2018), (SHEN *et al.*, 2022) e (CHI; GUO; CONG, 2022) validam a eficácia dessa abordagem ao alcançar paralelismo eficiente usando de forma parcial ou total o método proposto.

Considerando o fluxo de projeto em FPGA (ver Anexo B), o processo de especificação foi iniciado com a escolha do método de otimização a ser desenvolvido, o que derivou a implementação de uma primeira versão do algoritmo pretendido em uma linguagem de alto nível, criando assim um modelo de referência para que a técnica pudesse ser analisada e validada. A seguir, diversas versões de microarquitetura foram estudadas, almejando um equilíbrio entre o desempenho e a utilização dos recursos do FPGA, o que gerou a arquitetura atual do projeto. Logo após finalizada, essa microarquitetura foi transcrita para uma linguagem de descrição de hardware, o que permitiu a realização de testes e validações a partir dos resultados obtidos no modelo de referência. Por fim, foram realizadas a sintese lógica e as etapas de *placement* (mapeamento de funções lógicas aos recursos físicos do FPGA) e roteamento, onde problemas encontrados foram corrigidos.

Neste capítulo, são discutido o desenvolvimento dos modelos de referência cujos resultados de simulação são apresentados no Capítulo 5. A arquitetura atual do projeto é exibida no Capítulo 6, enquanto os resultados dos testes implementados após a construção da arquitetura são expostos no Capítulo 7.

4.1 Construção de modelos de referência

Objetivando viabilizar a análise e validação dos resultados obtidos a partir dos sistemas desenvolvidos, uma ferramenta de auxílio foi criada. Uma das suas funcionalidades consiste na construção de grafos com pesos aleatórios, dentro de uma faixa previamente configurada. As relações do grafo também podem ser configuradas de acordo com a necessidade da simulação, possibilitando aumentar ou diminuir a quantidade máxima de relações por nó, além da sua

direção. Esses grafos seriam então utilizados para a propagação do menor caminho entre uma fonte e um destino.

Sequencialmente, uma versão base do algoritmo de Dijkstra foi desenvolvida e testada utilizando a linguagem Python. Essa versão foi construída com o intuito de atuar como modelo de referência, validando os resultados obtidos nas versões otimizadas que viriam a ser concebidas em momentos posteriores. Por esse motivo, seus resultados foram amplamente testados e validados.

Uma das saídas da ferramenta é expressa através de imagens representando o grafo construído, bem como o menor caminho encontrado entre os nós indicados. Essas imagens puderam ser geradas através da biblioteca *matplotlib* e foram utilizadas como método de apoio no processo de validação dos resultados obtidos. A Figura 18 exibe um grafo com 30 nós, com pesos entre 1 e 15 e com até 8 relações por nó, juntamente com o menor caminho calculado (destacado em vermelho) entre os nós 0 e 29 - os nós que são obstáculos são identificados em branco.

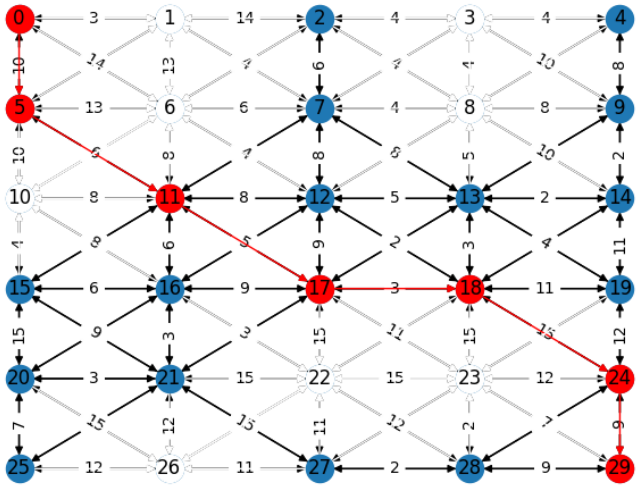


Figura 18 – Demonstração do grafo gerado com a ferramenta construída. Os nós brancos são nós marcados como obstáculos e os em azul são os válidos que não fazem parte do menor caminho. O menor caminho encontrado entre os nós 0 e 29 está destacado em vermelho.

4.2 Especificação do algoritmo de menor caminho

Como descrito anteriormente, a técnica escolhida para otimização foi a apresentada por (CRAUSER *et al.*, 1998), adaptada para uma abordagem FPGA - o fluxograma gerado a partir dessa transição pode ser visto na Figura 19. Neste fluxo, os nós do grafo são categorizados em cinco classificações distintas:

1. **Inativo**: nós que ainda não receberam análise alguma e aguardam processamento;

2. **Empilhado** ou **Ativo**: nós inativos que foram identificados como vizinhos de nós previamente estabelecidos;

3. **Aprovado**: nós que estão empilhados e passaram com sucesso no processo de classificação;

4. **Estabelecido**: nós aprovados cujo caminho mais curto até a origem já foi identificado e que foram encaminhados para o processo de expansão - etapa em que os vizinhos de um nó aprovado são identificados e analisados no sentido de atualizar sua distância até a fonte juntamente com a identificação do seu nó anterior;

5. **Obstáculo**: nós marcados como obstáculos recebem esta classificação, indicando que apresentam barreira ou obstrução no grafo e devem ser ignorados.

O algoritmo de Dijkstra segundo a proposta de otimização em (CRAUSER *et al.*, 1998) utiliza dois critérios para selecionar quais nós empilhados podem ser estabelecidos, o critério *IN* e o *OUT*. Tais critérios podem ser utilizados de forma separada ou conjunta (*INOUT*) no processo de avaliação dos nós.

O primeiro passo realizado pelo algoritmo é marcar o nó fonte (s) como empilhado (*empilhar(s)*). Logo após, o critério de identificação dos nós a serem aprovados é aplicado, resultando em uma lista de nós. Para cada nó (v) aprovado, é realizada a expansão dos seus vizinhos (w) que ainda não foram estabelecidos e que não fazem parte de um obstáculo. Essa etapa pode ser executada de forma paralela, pois não existem dependências entre as operações.

A expansão de um vizinho consiste em comparar sua distância atual até a fonte ($L(w)$) com a possível nova distância, sendo esta formada pela distância do nó aprovado até a fonte, somada ao custo de deslocamento entre o nó aprovado e o seu vizinho ($L(v) + c(v,w)$). Caso a nova distância seja menor do que a atualmente armazenada, a distância atual do vizinho juntamente com o seu ponteiro anterior ($w.anterior$) serão atualizados, ou seja, $L[w] = L[v] + c(v,w)$ e $w.anterior = v$. Além disso, cada vizinho do nó aprovado é marcado como empilhado ($empilhar(w)$), desde que o mesmo não esteja marcado como empilhado, estabelecido ou obstáculo.

Com o objetivo de garantir uma maior independência entre essas operações, cada resultado da expansão é armazenado em uma estrutura de vetores locais. Apenas quando todos os vizinhos da iteração atual são processados é que seus resultados são inseridos em vetores globais. Além disso, como é possível que um mesmo nó seja vizinho de diferentes nós e, consequentemente, esteja sendo expandido mais de uma vez na mesma iteração, a distância e o ponteiro para o nó anterior serão atualizados na memória global apenas quando a nova distância for menor do que a atualmente armazenada, garantindo que apenas a menor distância será atualizada.

Por fim, os nós aprovados são removidos da pilha e marcados como empilhados. O algoritmo termina quando não houver mais nós empilhados. Nesse momento, o menor caminho pode ser obtido ao se formar, a partir do nó destino, o conjunto de nós anteriores até a fonte.

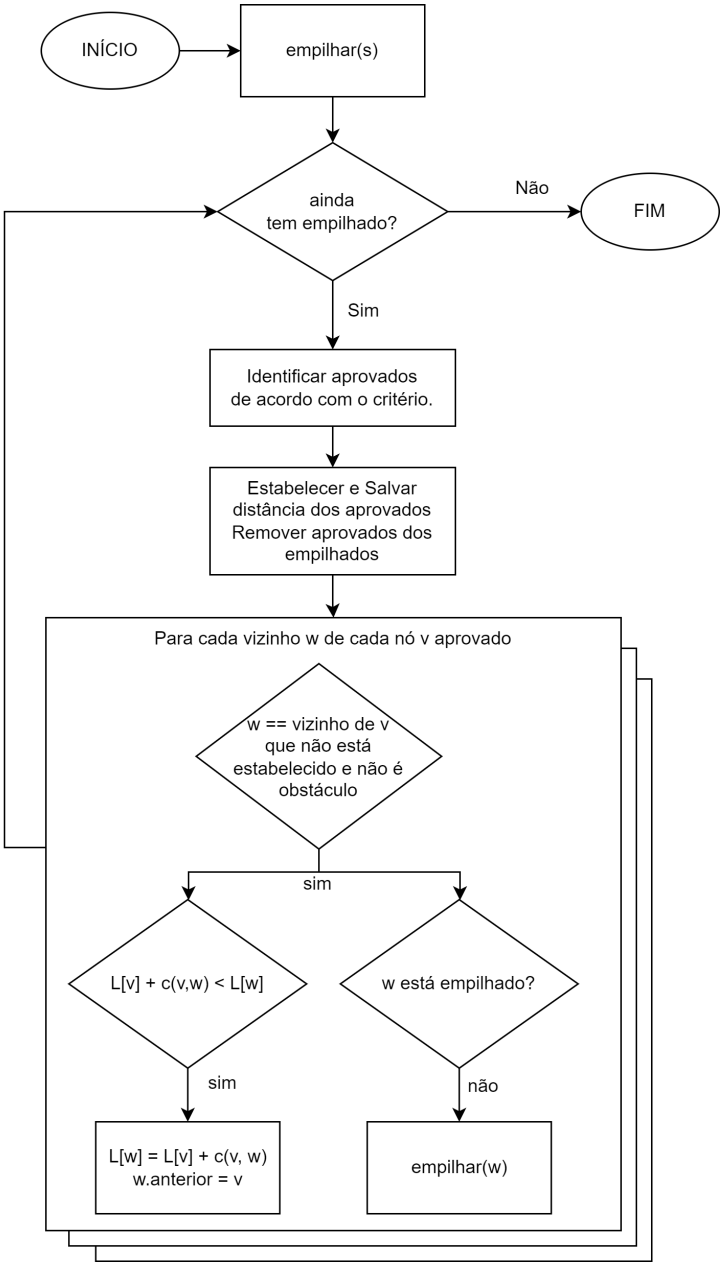


Figura 19 – Fluxograma do algoritmo de Dijkstra com a remoção de nós.

Durante a análise dessa implementação, observou-se que não é necessário armazenar a distância até a fonte de todos os nós: apenas daqueles que se encontram marcados como empilhados. Isso é possível pois, nessa abordagem, a leitura da distância até a fonte só será necessária em duas situações: i) durante a expansão de um nó aprovado - nesse caso, o nó já estará marcado como empilhado; e ii) quando o nó empilhado se encontrar na condição de vizinho inativo, visto que vizinhos não empilhados terão sempre uma distância igual ao infinito.

4.2.1 Análise dos critérios de remoção

No critério de remoção do tipo *OUT*, para cada nó empilhado é calculado um peso de acordo com as bordas de saída $o(v) = tent(v) + min\{c(v,u) : (v,u) \in E\}$. Neste caso, o menor

valor obtido definirá o limiar de remoção. Nós que possuem sua distância menor ou igual a esse limiar serão considerados aprovados, uma vez que sua distância até a fonte não poderá mais diminuir.

Como é possível analisar, no cálculo do peso das bordas de saída para cada um dos nós, é necessário buscar em memória a sua distância atual, juntamente com o custo do seu vizinho de menor custo. Para facilitar a coleta desses dados, as relações de cada nó são inseridas na lista de adjacência de forma ordenada, conforme o custo da relação (ver Figura 20). Dessa forma, o vizinho com menor custo sempre será a primeira relação válida, o que facilitará o processo de busca (considerando-se uma relação válida aquela cujo vizinho não é marcado como obstáculo).

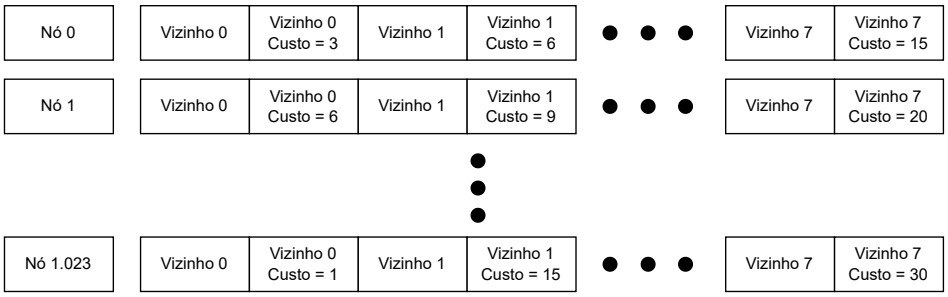


Figura 20 – Estrutura da lista de adjacências com vizinhos ordenados de acordo com o custo do relacionamento.

No critério de remoção do tipo *IN*, para cada nó empilhado é calculado um peso de acordo com as bordas de entrada $i(v) = tent(v) - min\{c(u,v) : (u,v) \in E\}$. O ponto de corte é definido de acordo com a menor distância até a fonte entre os nós empilhados ($M = min\{tent(u) : u \ \ empilhado\}$). Os nós com $i(v) \leq M$ podem então ser classificados como aprovados.

Deste modo, no cálculo do peso das bordas de entrada, para cada um dos nós será necessário buscar em memória a sua distância atual e, dentre os vizinhos que chegam neste nó, aquele que possui o menor custo. Essa característica faz com que sua implementação seja mais complexa do que o tipo *OUT*, uma vez que é necessário buscar na lista de adjacência todas as relações que incidem no nó em análise. Seria possível otimizar essa busca através da criação de uma segunda lista de adjacência, com todas as relações que chegam em um nó. Contudo, tal abordagem aumentaria o uso de memória.

Por fim, a versão *INOUT* aplica os dois critérios de forma conjunta, realizando a expansão dos nós aprovados pelos dois critérios.

CAPÍTULO

5

RESULTADOS DA SIMULAÇÃO DO MODELO DE REFERÊNCIA

Com o objetivo de comparar o desempenho dos três tipos de critérios de seleção, foram realizadas diversas simulações para cada um deles, variando a quantidade de nós do grafo e o número máximo de relacionamentos entre os nós.

Para essas simulações, foi considerado que uma iteração no algoritmo de Dijkstra, tanto para a versão padrão quanto para a otimizada, corresponde aos passos de identificação dos nós aprovados, expansão dos seus vizinhos e estabelecimento destes nós.

Na implementação padrão do algoritmo de Dijkstra, como não existe um sistema de classificação de nós aprovados, os nós são processados de forma sequencial, deste modo, a quantidade de iterações é sempre igual ao número de nós do grafo. Esse conceito é utilizado aqui como parâmetro de comparação entre os critérios de seleção: quanto menor seu número, maior é a paralelização do algoritmo.

Vale ressaltar que a versão otimizada do algoritmo, tem como resultado do processo de classificação uma lista de nós a serem expandidos em paralelo. Deste modo, realiza menos operações de identificação dos nós do que a versão padrão. Assim, apesar de possuir uma latência maior na etapa de identificação dos nós aprovados, pois necessita implementar um sistema de seleção mais complexo, o impacto do acréscimo de latência é diluído pela maior quantidade de nós aprovados. Por esse motivo, para os testes aqui expostos, essa latência será desconsiderada.

Dessa forma, para cada simulação foram extraídos cinco parâmetros diferentes: i) quantidade de iterações necessárias; ii) ganho com relação à abordagem padrão, gerado a partir da divisão da quantidade de iterações do modelo padrão pela quantidade de iterações do critério; iii) mínimo de nós aprovados em cada iteração; iv) máximo de nós aprovados em cada iteração; e v) média de nós aprovados em cada iteração.

Além disso, objetivando verificar se existe alguma relação entre os nós aprovados dos

critérios *IN* e *OUT*, nas simulações utilizando o critério *IN*, o critério *OUT* também foi executado. Nesse caso, os nós aprovados por este último não foram inseridos no processamento do algoritmo, sendo utilizados apenas para realizar a comparação. O mesmo ocorreu, só que de maneira inversa, nas simulações do modelo *OUT*. Com estes testes, foi possível identificar quantas vezes os nós aprovados por um critério estavam contidos nos nós aprovados pelo outro critério. Vale ressaltar que esta análise não contabiliza situações em que os dois critérios apresentavam os mesmos nós aprovados, apenas quando divergem na quantidade de elementos.

Os resultados das simulações do critério *IN* para grafos com até 8 e até 4 relações podem ser visualizados nas Tabelas 5 e 6, respectivamente. A partir dos resultados expostos, é possível observar que para o grafo com até 8 relações não ocorreu caso algum em que os resultados do critério *IN* estavam contidos no conjunto de resultados do critério *OUT*. No entanto, esse comportamento não é observado nos resultados da Tabela 6, que apresenta situações em que o conjunto dos nós aprovados pelo critério *OUT* conteve todos os nós aprovados pelo critério *IN* e alguns nós a mais, apesar de serem poucos casos.

Tabela 5 – Resultados de simulação do critério *IN* com até 8 relações por nó.

Número de Nós	**Iterações**			**Nós ativos por iteração**			**IN contido no *OUT***
	IN	**Padrão**	**Ganho**	**Mín.**	**Máx.**	**Média**	
64	16	64	4,00	1	8	4	0
128	22	128	5,82	1	13	6	0
256	31	256	8,26	1	19	8	0
512	48	512	10,67	1	30	11	0
1.024	73	1.024	14,03	1	34	14	0
2.048	103	2.048	19,88	1	48	20	0
4.096	147	4.096	27,86	1	79	28	0
8.192	215	8.192	37,10	1	120	38	0

Tabela 6 – Resultados de simulação do critério *IN* com até 4 relações por nó.

Número de Nós	**Iterações**			**Nós ativos por iteração**			**IN contido no *OUT***
	IN	**Padrão**	**Ganho**	**Mín.**	**Máx.**	**Média**	
64	21	64	3,05	1	5	3	2
128	32	128	4,00	1	10	4	0
256	52	256	4,92	1	17	5	1
512	76	512	6,74	1	17	7	1
1.024	128	1024	8,00	1	24	8	0
2.048	169	2.048	12,12	1	40	12	1
4.096	261	4.096	15,69	1	41	16	0
8.192	391	8.192	20,95	1	70	21	0

A partir destes resultados é possível observar também que o algoritmo apresenta um melhor desempenho para grafos com maior quantidade de relações, visto que a quantidade de iterações para um mesmo número de nós é menor no grafo com até 8 relações. Isso ocorre pois a

quantidade de relações afeta o processo de expansão de um nó aprovado: com mais relações ele possuirá mais vizinhos e, consequentemente, irá empilhar mais nós a cada iteração.

Os resultados da simulação do critério *OUT* para grafos com até 8 e até 4 relações podem ser visualizados nas Tabelas 7 e 8, respectivamente. De forma semelhante aos resultados do critério *IN*, as análises dos grafos com maior número de relações também apresentaram um melhor desempenho. Observou-se também que existe uma grande quantidade de casos em que todos os nós aprovados pelo critério *OUT* também seriam aprovados no critério*IN*.

Tabela 7 – Resultados de simulação do critério *OUT* com até 8 relações por nó. (Fonte próprio autor).

Número de Nós	**Iterações**			**Nós ativos por iteração**			***OUT* contido**
	OUT	**Padrão**	**Ganho**	**Mín.**	**Máx.**	**Média**	**no *IN***
64	17	64	3,76	1	10	4	12
128	23	128	5,57	1	11	6	18
256	32	256	8,00	1	20	8	27
512	50	512	10,24	1	24	10	42
1.024	73	1.024	14,03	1	39	14	61
2.048	102	2.048	20,08	1	55	20	94
4.096	147	4.096	27,86	1	83	28	141
8.192	216	8.192	37,93	1	115	38	209

Tabela 8 – Resultados de simulação do critério *OUT* com até 4 relações por nó.

Número de Nós	**Iterações**			**Nós ativos por iteração**			**OUT contido**
	OUT	**Padrão**	**Ganho**	**Mín.**	**Máx.**	**Média**	**no *IN***
64	21	64	3,05	1	6	3	14
128	33	128	3,88	1	8	4	26
256	52	256	4,92	1	12	5	42
512	78	512	6,56	1	19	7	70
1.024	130	1.024	7,88	1	20	8	115
2.048	170	2.048	12,05	1	37	12	164
4.096	262	4.096	15,63	1	43	16	244
8.192	393	8.192	20,84	1	61	21	386

Os resultados da simulação do critério *INOUT* para grafos com até 8 e até 4 relações podem ser visualizados nas Tabelas 9 e 10, respectivamente. Por ser a junção da aplicação dos dois critérios, esperava-se que os resultados do critério *INOUT* iriam se destacar com relação aos demais, no entanto, isso não ocorre. Como é possível observar, os resultados são bastante próximos dos obtidos quando se aplicam os critérios de forma individual.

Considerando todos os resultados expostos, é possível observar que quanto maior for a quantidade de nós do grafo, maior será o ganho com relação ao modelo padrão, partindo de um ganho aproximadamente 3 vezes maior com grafos de até 64 nós e chegando a um ganho 38 vezes maior para grafos com até 8192 nós. Esse ganho também é maior quando se utiliza um grafo com mais relações.

Tabela 9 – Resultados de simulação do critério *INOUT* com até 8 relações por nó.

Número de Nós	**Iterações**			**Nós ativos por iteração**		
	INOUT	**Padrão**	**Ganho**	**Mín.**	**Máx.**	**Média**
64	16	64	4,00	1	8	4
128	22	128	5,82	1	13	6
256	31	256	8,26	1	19	8
512	47	512	10,89	1	30	11
1.024	72	1.024	14,22	1	34	14
2.048	102	2.048	20,08	1	48	20
4.096	147	4.096	27,86	1	71	28
8.192	215	8.192	38,10	1	120	38

Tabela 10 – Resultados de simulação do critério *INOUT* com até 4 relações por nó.

Número de Nós	**Iterações**			**Nós ativos por iteração**		
	INOUT	**Padrão**	**Ganho**	**Mín.**	**Máx.**	**Média**
64	19	64	3,37	1	6	3
128	32	128	4,00	1	10	4
256	51	256	5,02	1	17	5
512	75	512	6,83	1	17	7
1.024	128	1.024	8,00	1	24	8
2.048	169	2.048	12,12	1	40	12
4.096	259	4.096	15,81	1	41	16
8.192	390	8.192	21,01	1	70	21

Objetivando identificar de forma mais clara as diferenças nos resultados obtidos, foram construídos alguns gráficos comparativos. O primeiro parâmetro comparado foi a quantidade máxima de nós aprovados por iteração para nós com até 8 e até 4 relações. Essa comparação pode ser observada nas Figuras 21 e 22, respectivamente. Estes gráficos demonstram que, apesar dos resultados não serem sempre iguais, existe uma semelhança grande entre eles. Observa-se também que o critério *INOUT* e *IN* geralmente apresentam resultados parecidos, sendo a única exceção a análise de um grafo com 4096 nós e 8 relações. Além disso, o critério *OUT* apresenta em oito situações um melhor resultado que os demais na análise com até 8 relações, tendo este número reduzido para um no gráfico da Figura 22, demonstrando que esse método se destaca para aplicações com maior número de relações.

Em seguida, foi realizada uma comparação das médias de nós aprovados por iteração para grafos com até 8 e até 4 relações (ver Figuras 23 e 24, respectivamente). Nestes resultados, é possível identificar que todos os critérios apresentaram resultados semelhantes. A única exceção foi o resultado obtido para grafos com 512 nós e 8 relações, no qual o critério *OUT* teve uma média menor.

Por fim, a quantidade total de iterações de cada um dos modelos também foi comparada para grafos com 8 e até 4 relações (ver Figuras 25 e 26, respectivamente). Nesses resultados, é possível observar que existe uma grande similaridade entre os resultados obtidos. Isso prova-

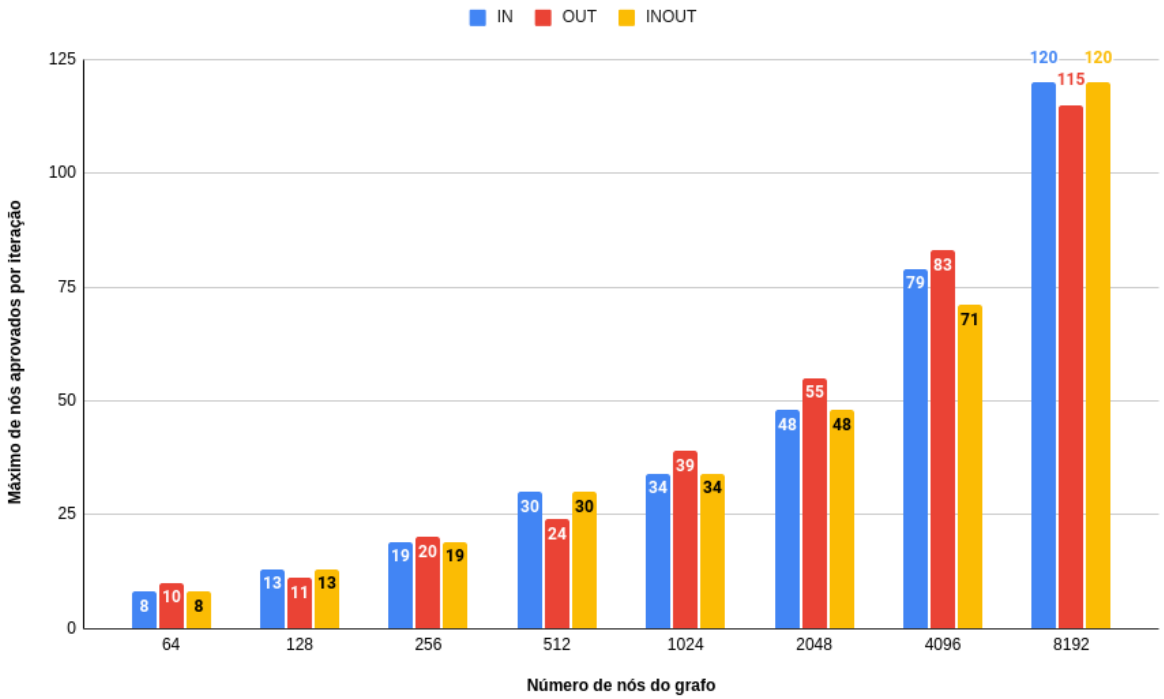


Figura 21 – Comparação da quantidade máxima de nós aprovados por iteração para nós com até 8 relações

velmente reflete os resultados obtidos nas Figuras 23 e 24, que demonstram que os diferentes modelos apresentam média de nós aprovados por iteração semelhante, apesar da ocorrência de alguns picos de performance, como apresentado nas Figuras 21 e 22.

A partir destes resultados é possível observar que para os modelos de grafo utilizados não há uma grande diferença de impacto no desempenho entre os diferentes critérios. Considerando esse fator e a maior complexidade de desenvolvimento do critério *IN*, e consequentemente, do *INOUT*, ficam evidentes as vantagens de se utilizar apenas o critério *OUT*, método mais simples e que apresenta os mesmos resultados. No entanto, apesar dos resultados conclusivos, a fim de alcançar um maior nível de entendimento, trabalhos futuros ainda podem ser realizados com o objetivo de explorar diferentes estruturas de grafo.

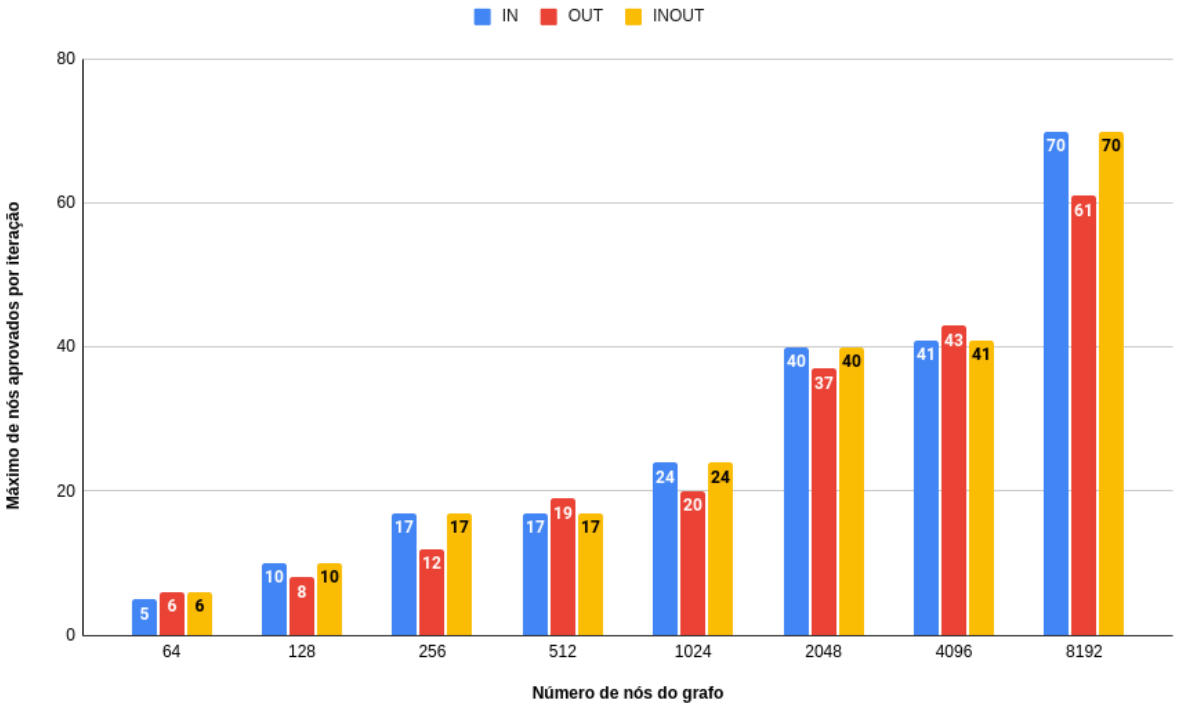


Figura 22 – Comparação da quantidade máxima de nós aprovados por iteração para nós com até 4 relações.

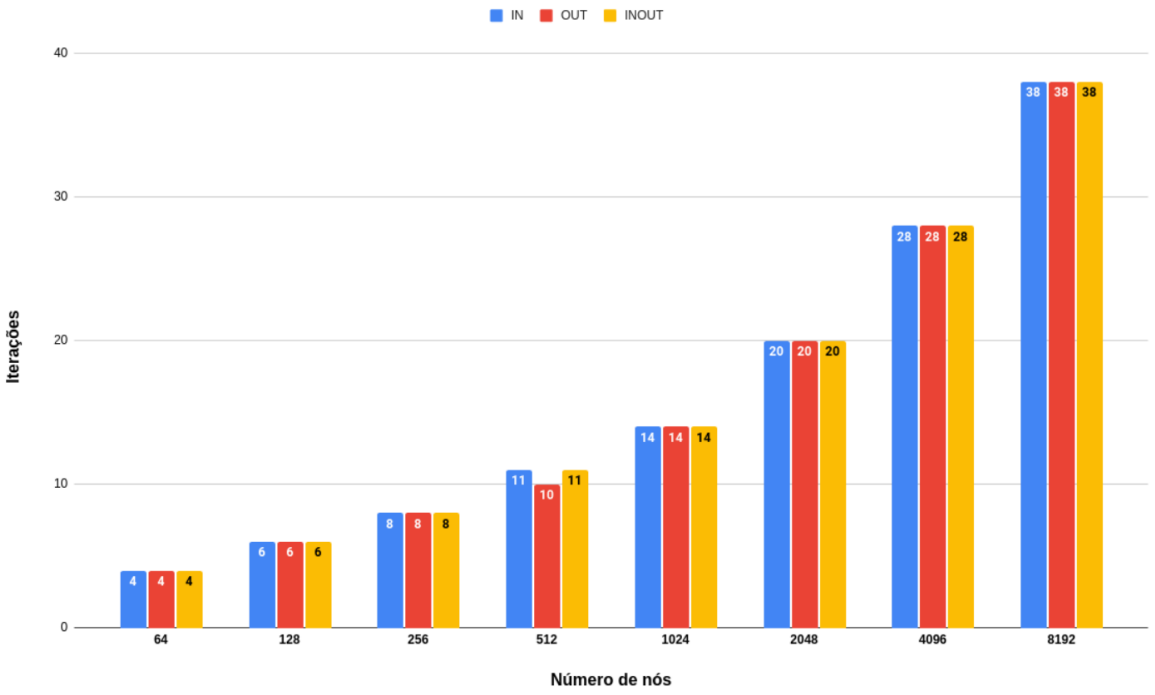


Figura 23 – Comparação da média de nós aprovados por iteração para nós com até 8 relações.

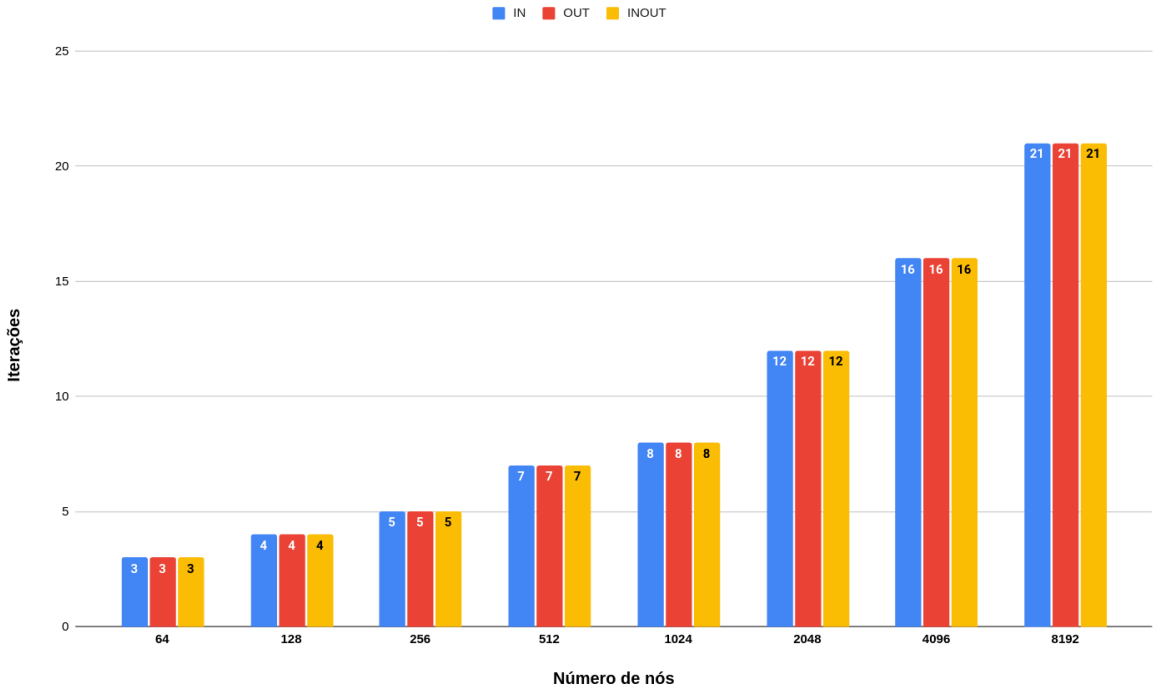


Figura 24 – Comparação da média de nós aprovados por iteração para nós com até 4 relações.

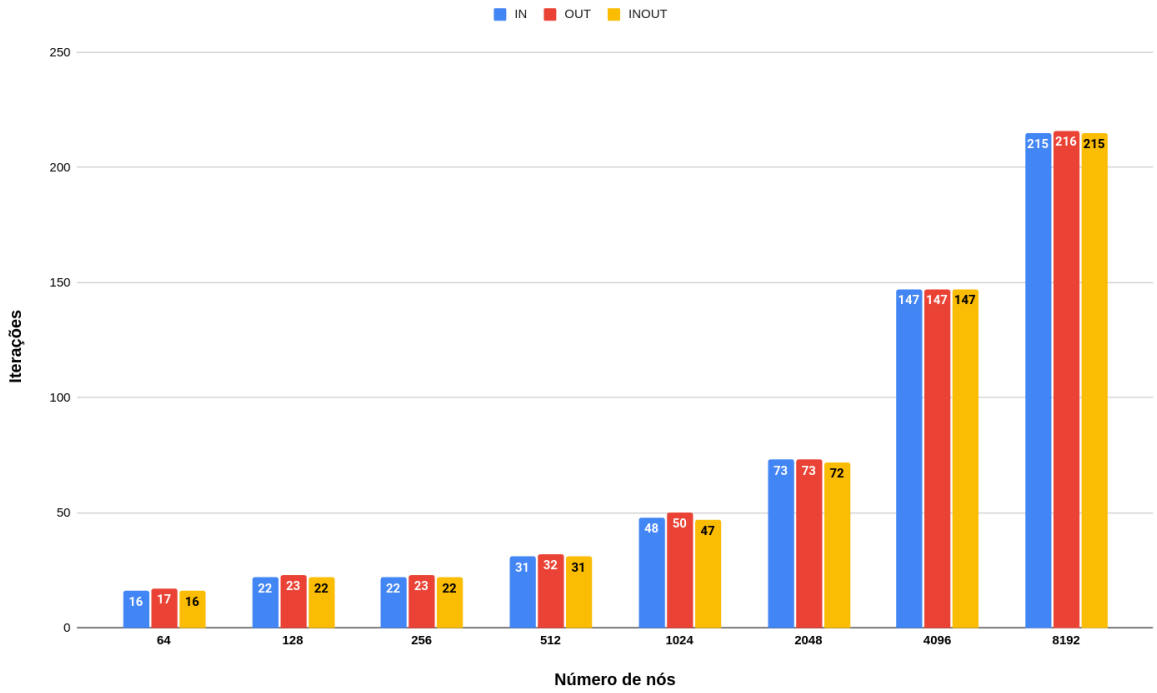


Figura 25 – Comparação da quantidade total de iterações para nós com até 8 relações.

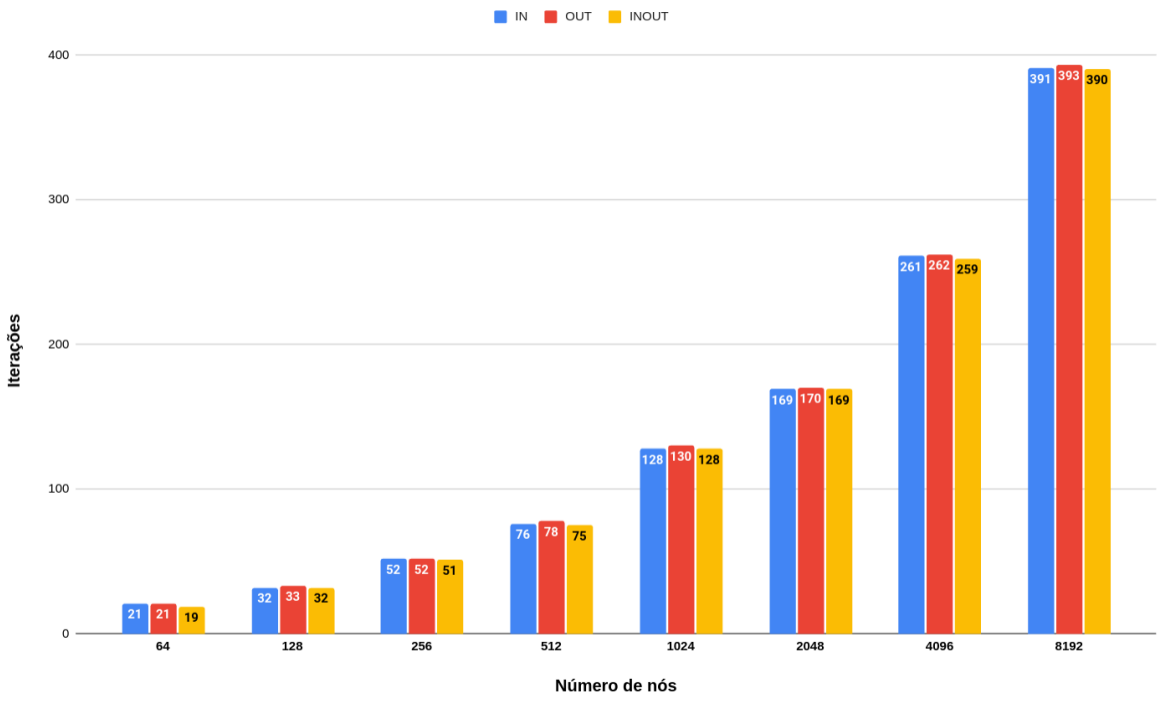


Figura 26 – Comparação da quantidade total de iterações para nós com até 4 relações.

CAPÍTULO

6

ARQUITETURA DA PROPOSTA

O objetivo desta tese é desenvolver uma solução embarcada em FPGA que resolva o problema do SSSP de forma rápida e eficiente, possibilitando sua adoção em robôs que operam em tempo real. Esta solução considera que o grafo de movimentação do robô é fixo e pré-estabelecido e que seus nós possam ser removidos ou reinseridos durante sua operação.

A construção do grafo de movimentação do robô é realizada durante uma fase de configuração *offline*, que precede sua operação. Esta proposta apresenta uma abordagem inovadora para armazenar o grafo, substituindo a utilização de uma única matriz de representação pela adição de uma matriz auxiliar que identifica os nós obstáculos.

Dessa forma, o grafo de movimentação do robô é formado pela junção da matriz contendo todos os relacionamentos do grafo, juntamente com a matriz de identificação dos nós obstáculos a serem ignorados. Essas matrizes são armazenadas em memórias distintas e apenas a memória contendo os nós obstáculos precisa ser atualizada.

O sistema também poderia funcionar de forma tradicional, utilizando uma única memória para armazenar o grafo e seus relacionamentos, caso a memória de obstáculos não fosse utilizada e as atualizações no ambiente fossem traduzidas diretamente na memória de relações. Porém, como neste caso a quantidade de informações que será transmitida será maior, haveria uma redução de desempenho, conforme será mostrado no Capítulo 7.

Durante a execução do robô, o sistema responsável pelo reconhecimento de obstáculos e gerenciamento de movimento atua como controlador da aplicação, acionando o módulo proposto (controlado) sempre que um novo menor caminho precisa ser encontrado (Figura 27). O controlador atualiza as configurações dos obstáculos e fornece informações sobre os nós atuais de origem e destino para os quais o menor caminho precisar ser identificado. Uma vez encontrado este caminho, um sinal é ativado indicando sua disponibilidade. Para viabilizar esta comunicação, a solução proposta se conecta a um barramento de comunicação juntamente com o controlador - um protocolo *Advanced Microcontroller Bus Architecture* (AMBA) poderia ser utilizado por

exemplo, porém, a escolha do protocolo pode ser diferente a depender dos requisitos da aplicação em que o projeto esteja inserido.

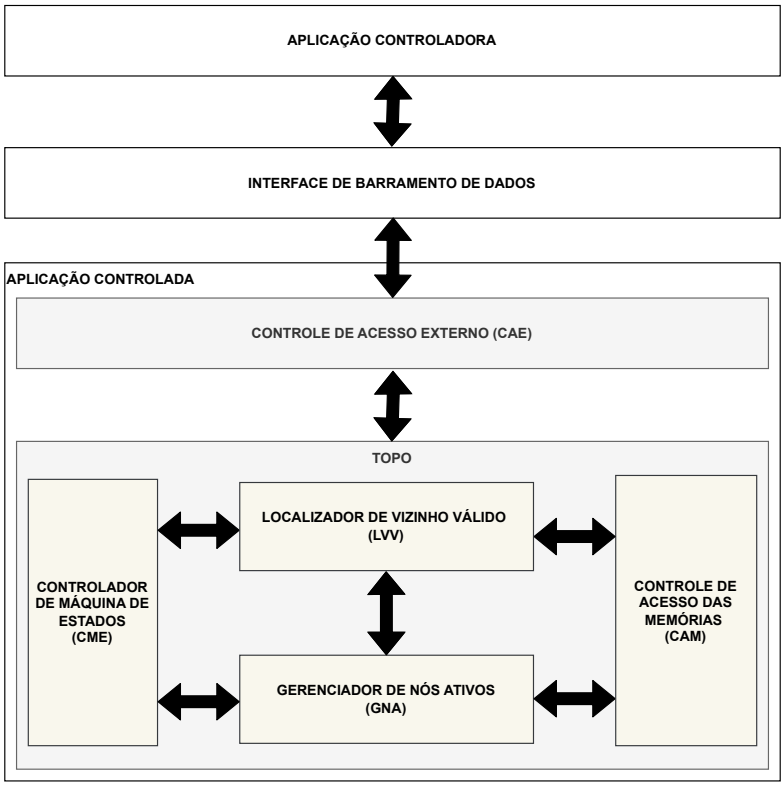


Figura 27 – Arquitetura do topo do projeto.

Internamente, o controlado implementa uma versão adaptada da solução apresentada por (CRAUSER *et al.*, 1998). Com base nos resultados de simulação apresentados no Capítulo 5, há pouca diferença entre os critérios de seleção *IN*, *OUT* e *INOUT* em relação ao número de nós aprovados por iteração, considerando o cenário no qual este projeto está inserido. Além disso, como explicado no Capítulo 4, devido às peculiaridades do seu método de classificação, o critério *IN* necessita de uma estrutura mais complexa com um acesso adicional à memória. Assim, objetivando uma solução que busque um melhor consumo de recursos do FPGA, a proposta de arquitetura aqui apresentada utilizará apenas o critério *OUT* como método classificador.

Deste modo, foi desenvolvida uma arquitetura que pudesse embarcar o algoritmo de Dijkstra otimizado de modo a priorizar o paralelismo e, ao mesmo tempo, manter um consumo eficiente dos recursos do FPGA utilizado. Como resultado, se obteve uma arquitetura dividida em cinco blocos principais:

1. *Controle de Acesso Externo* (*CAE*) - controle de fluxo de comunicações externas;
2. *Controle de Acesso às Memórias* (*CAM*) - gerenciamento de memória interna, armazena e controla informações como as matrizes de relacionamento e obstáculos, identificação dos

nós estabelecidos e anteriores;

3. *Gerenciador de Nós Ativos* (*GNA*) - gerenciamento e armazenamento de nós ativos, classificação e identificação de aprovados;

4. *Localizador de Vizinho Válido* (*LVV*) - realiza o procedimento de expansão dos nós aprovados;

5. *Controlador de Máquina de Estados* (*CME*) - controle do fluxo de operação do algoritmo, gerenciando os demais módulos.

O fluxo de operação desta proposta se inicia na etapa de configuração, onde a matriz de adjacência que representa o grafo é recebida via *CAE* e armazenada internamente no *CAM*. Logo após, o sistema se encontra apto a identificar um menor caminho. Caso ocorram alterações no ambiente e a configuração de obstáculos necessite ser atualizada, uma nova matriz de configuração é enviada pelo controlador, essa informação é então armazenada em um sub-módulo do *CAM*.

O cálculo do menor caminho se inicia quando o controlador, via *CAE*, envia uma solicitação de cálculo do menor caminho, juntamente com os nós de origem e destino. Neste momento, o *CME* é acionado pelo *CAE*, dando início às etapas necessárias para encontrar o menor caminho: ativar o *GNA* (onde os nós aprovados são identificados) e solicitar a expansão dos nós aprovados no *LVV*. Quando não existirem mais nós ativos no *GNA* o processo de encontrar o menor caminho é finalizado. Nesse momento, o *CME* informa ao controlador que o caminho está pronto e aguardando leitura. Quando o menor caminho é lido, o processo é finalizado e o sistema retorna para o modo de espera.

Esse fluxo de operações é descrito com mais detalhes nas próximas seções, onde cada um dos blocos desenvolvidos é explicado.

6.1 *Controle de Acesso às Memórias*

O gerenciamento das memórias necessárias ao correto funcionamento do projeto é realizado pelo *Controle de Acesso às Memórias* (*CAM*) (Figura 28). Este bloco define o tamanho tanto das memórias que são utilizadas quanto dos seus barramentos de dados. Além disso, direciona as solicitações de leitura e escrita recebidas entre as memórias e os demais módulos do projeto. Internamente, gerencia quatro memórias com diferentes propósitos:

1. *Memória de Obstáculos* (MO) – identificação dos nós que são obstáculos;

2. *Memória de Relacionamentos* (MR) – identificação dos relacionamentos de cada nó, juntamente com seus custos;

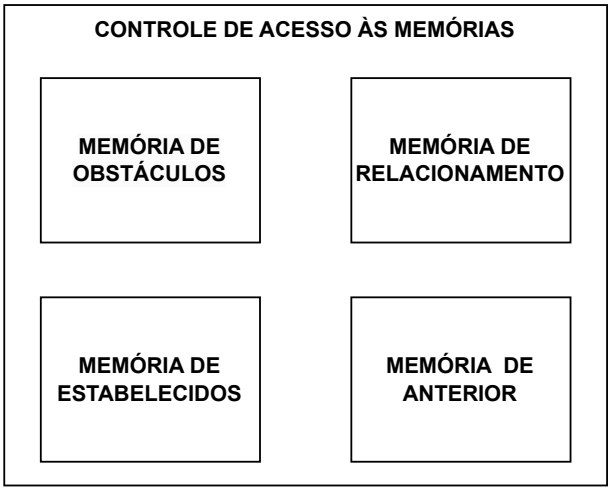


Figura 28 – *Controle de Acesso às Memórias*.

3. *Memória de Estabelecidos* (ME) – identificação dos nós que foram estabelecidos; e
4. *Memória de Anteriores* (MA) – identificação de nós anteriores (vizinho do nó atual mais próximo do nó de origem - a sequência dos nós anteriores forma o menor caminho).

Tanto a *Memória de Obstáculos* quanto a *Memória de Estabelecidos* armazenam apenas 1 bit para cada nó do grafo, uma vez que funcionam como uma representação de um estado booleano do nó. Objetivando construir uma arquitetura mais otimizada para um cenário específico, neste projeto foi definido que cada nó pode ter até oito relações - como consequência, para aumentar o paralelismo durante os processos de expansão (este processo é explicado em detalhes na seção 6.4) ambas as memórias possuem oito portas de leitura, o que torna possível ler as informações referentes a oito vizinhos de um nó de forma paralela no tempo de apenas uma leitura.

Vale ressaltar que a adoção de uma estrutura com oito relações não limita o grafo utilizado a possuir apenas nós com exatamente oito relações. Com a utilização da *Memória de Relacionamentos* é possível representar nós que possuem menos de oito relações, apenas inserindo custos elevados para as relações inexistentes. Assim, as mesmas são ignoradas durante a busca do menor caminho. Além disso, a implementação de nós com mais de oito relações também é possível com a inclusão de nós gêmeos: ao se definir o custo de deslocamento entre dois nós igual a zero, permite-se que a quantidade excedente de relações sejam distribuídas entre tais irmãos. Para otimizar a leitura, a *Memória de Relacionamentos* também possui oito portas de leitura.

A *Memória de Relacionamentos* é responsável por armazenar todas as relações de um nó com seus respectivos custos. O tamanho da palavra (*tamPalavra*) desta memória depende do número de bits necessários para representar todos os nós do grafo (*bitsEndereco*) juntamente

com o custo dos relacionamentos de cada nó (*bitsCusto*), multiplicados pelo número de relações no grafo (*numRelacoes*), conforme a Equação 6.1.

$$tamPalavra = (bitsEndereco + bitsCusto) * numRelacoes \tag{6.1}$$

A Figura 29, demonstra como seria organizada a lista de adjacências da memória de relações considerando os seguintes parâmetros:

1. grafo com até 1024 nós, $bitsEndereco = 10$;
2. custo de relacionamento de no máximo 31 unidades, $bitsCusto = 5$; e
3. cada nó possuindo até 8 relações, $numRelacoes = 8$.

Aplicando os valores obtidos na Equação 6.1 teríamos uma palavra com $(10+5)*8 =$ 120 bits por nó.

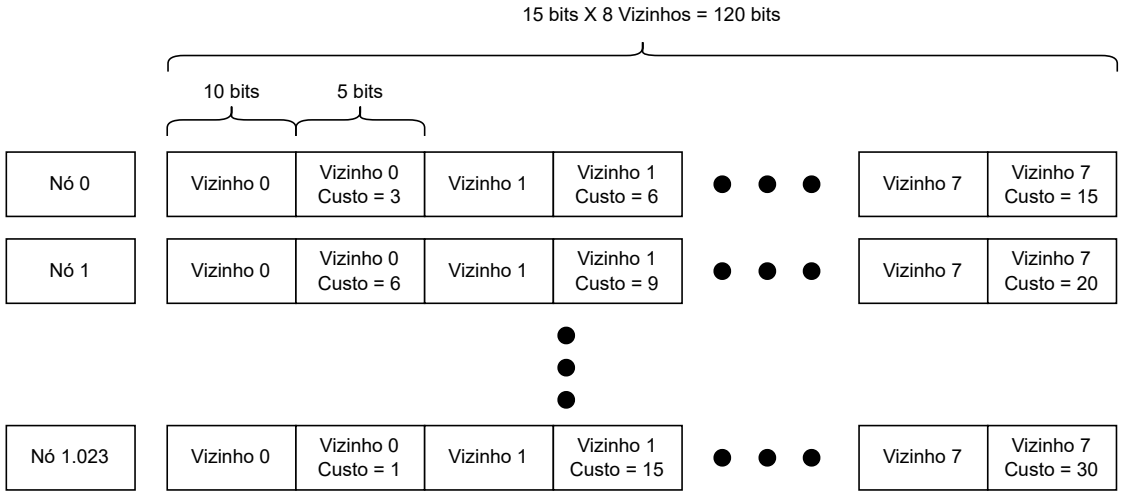


Figura 29 – Lista de adjacências da *Memória de Relacionamentos*. Neste exemplo foi criada uma lista com 1.023 nós (representação em 10 bits), cada um contendo até 8 vizinhos com custo máximo de 31 (5 bits). Cada entrada na lista será uma palavra de 120 bits.

Para construir o menor caminho, é essencial armazenar o nó anterior de cada nó. Começando pelo nó de destino, o menor caminho é então construído seguindo a sequência de nós anteriores até o nó de origem. Como cada nó tem apenas um nó anterior, para um grafo com 1024 nós, por exemplo, é suficiente armazenar apenas 5 bits por nó.

As informações da *Memória de Obstáculos* são atualizadas pelo módulo controlador a cada mudança no ambiente em que o robô está inserido. Por outro lado, a *Memória de Relacionamentos* é atualizada apenas na etapa de configuração inicial, realizada em um processo anterior à execução do projeto. As demais memórias operam dinamicamente durante o cálculo do menor caminho e são reinicializadas a cada nova busca.

As informações dos sinais que fazem interface com o módulo de *Controle de Acesso às Memórias* (a saber direção, bloco de origem e significado) para um grafo com 1024 nós e custo de 5 bits podem ser visualizadas na Tabela 11.

Tabela 11 – Sinais do bloco *Controle de Acesso às Memórias* para um grafo com 1024 nós.

Sinal	**Direção**	**Conexão**	**Bits**	**Descrição**
clk	entrada	-	1	clock do sistema
rst_n	entrada	-	1	reset do sistema
cme_soft_reset_n	entrada	CME	1	reset lógico controlado pelo CME
lvv_anterior_write_en	entrada	LVV	1	habilita a escrita do nó anterior
lvv_anterior_data	entrada	LVV	10	nó anterior a ser salvo
lvv_anterior_write_addr	entrada	LVV	10	endereço do nó que se deseja salvar o anterior
cam_anterior_read_data	saída	CAE	10	nó que faz parte do menor caminho
cam_pronto_out	saída	CME	1	menor caminho construído
cme_construir_caminho	entrada	CME	1	comando para construir o menor caminho
cme_fonte	entrada	CME	10	endereço do nó fonte
cme_destino	entrada	CME	10	endereço do nó destino
cae_relacoes_wr_enable	entrada	CAE	1	habilita a escrita de uma relação
cae_relacoes_wr_addr	entrada	CAE	10	endereço de escrita da relação
cae_relacoes_write_data	entrada	CAE	10	relação a ser armazenada
cae_obstaculos_wr_enable	entrada	CAE	1	habilita a escrita de um obstáculo
cae_obstaculos_wr_addr	entrada	CAE	10	endereço de escrita do obstáculo
cae_obstaculos_wr_data	entrada	CAE	1	obstáculo a ser armazenado
lvv_relacoes_read_addr	entrada	LVV	8*10	endereço de leitura da relação
gma_relacoes_read_data	saída	LVV	8*120	relação lida no endereço fornecido
lvv_obstaculos_read_addr	entrada	LVV	8*10	endereço de leitura do obstáculo
gma_obstaculos_read_data	saída	LVV	8	obstáculo lido no endereço fornecido
lvv_estabelecidos_write_en	entrada	LVV	1	habilita a escrita do nó a ser estabelecido
lvv_estabelecidos_write_addr	entrada	LVV	10	endereço do nó a ser estabelecido
lvv_estabelecidos_read_addr	entrada	LVV	8*10	endereço do nó a ser lido na ME
cam_estabelecido_read_data	saída	LVV	8	situação do nó lido na ME

6.2 *Controle de Acesso Externo*

O *Controle de Acesso Externo* (*CAE*) (Figura 30) é o bloco responsável por gerenciar a comunicação externa do projeto. Este bloco oferece um método confiável de comunicação entre a Interface Externa (IE) e os blocos internos, garantindo consistência entre os sistemas. Ao utilizá-lo, não há necessidade de modificar os sinais internos de entrada e saída para corresponder aos padrões de comunicação externos. Esta abstração simplifica o desenvolvimento da arquitetura interna, permitindo maior flexibilidade.

Deste modo, caso seja necessária uma mudança no protocolo de comunicação, o *CAE* pode ser facilmente adaptado sem que as outras partes do sistema sejam afetadas. Ele opera de forma similar a um demultiplexador, interpretando as solicitações de escrita/leitura do barramento de comunicação externa e as encaminhando para os módulos desejados de acordo com o endereço fornecido. Internamente, o barramento de comunicação é dividido em seis endereços base:

1. *BASE_FONTE* - endereço do nó fonte;

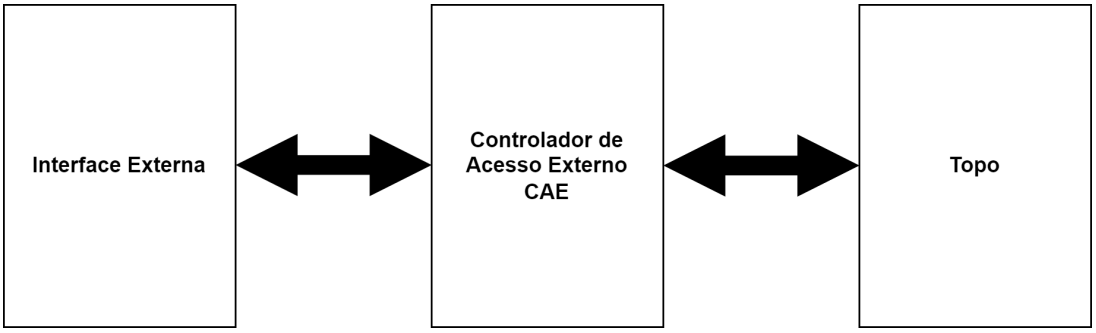


Figura 30 – Interface externa com o *CAE*.

2. *BASE_DESTINO* - endereço do nó destino;
3. *BASE_OBSTACULO* - início do endereço da *Memória de Obstáculos*;
4. *BASE_RELACAO* - início do endereço da *Memória de Relacionamentos*;
5. *BASE_PRONTO* - sinal indicador que o menor caminho está formado;
6. *BASE_CAMINHO* - endereço do menor caminho.

Esses endereços, por serem parametrizáveis, suportam diferentes tamanhos de grafos e configurações. Os valores de tais endereços são gerados automaticamente por uma ferramenta de configuração de acordo com o tipo de grafo a ser utilizado. Vale destacar que essa funcionalidade permitiu uma maior flexibilidade durante o período de testes, também viabilizando uma abordagem mais ampla em aplicações com diferentes propósitos.

As informações provenientes do módulo controlador através do barramento de comunicação são distribuídas internamente de acordo com os endereços recebidos, permitindo o acesso das mesmas pelos demais blocos do projeto. Um exemplo dessa funcionalidade ocorre logo na fase inicial de configuração, quando as relações dos nós são recebidas e encaminhadas para a *Memória de Relacionamentos*. Da mesma forma, sempre que mudanças no ambiente ocorrem e novos obstáculos são inseridos ou removidos, as novas configurações de obstáculos recebidas são direcionadas para a *Memória de Obstáculos*.

Quando um novo caminho deve ser formado, a Interface Externa realiza uma escrita nos endereços *BASE_FONTE* e *BASE_DESTINO*. Logo após identificar essas escritas, o *CAE* encaminha os endereços dos nós fonte e destino recebidos para o *CME* sinalizando que o processo de cálculo de um novo caminho deve ser iniciado. A partir desse momento, a aplicação controlador monitora o valor armazenado no endereço *BASE_PRONTO* - quando a informação contida no mesmo transiciona de 0 para 1, há a indicação que o menor caminho está pronto para ser coletado. Neste momento, o menor caminho se encontra disponível no endereço *BASE_CAMINHO* e, a cada nova leitura neste endereço, o próximo nó do caminho é disponibilizado de forma sequencial.

As informações dos sinais que fazem interface com o módulo *Controle de Acesso Externo* podem ser visualizadas na Tabela 12 para um grafo com 1024 nós.

Tabela 12 – Sinais do bloco *Controle de Acesso Externo* para um grafo com 1024 nós.

Sinal	**Direção**	**Conexão**	**Bits**	**Descrição**
clk	entrada	-	1	clock do sistema
rst_n	entrada	-	1	reset do sistema
ie_data_in	entrada	IE	32	barramento de entrada dados
ie_addr	entrada	IE	32	barramento de endereço
ie_data_out	saída	IE	32	barramento de saída de dados
ie_wr_enable	entrada	IE	1	solicitação de escrita
ie_rd_enable	entrada	IE	1	solicitação de leitura
cme_addr_fonte	saída	CME	10	endereço do nó fonte
cme_addr_destino	saída	CME	10	endereço do nó destino
cme_wr_fonte	saída	CME	1	solicitação de escrita da fonte e destino
anterior_read_data	entrada	CAM	10	endereço do nó anterior
anterior_pronto	entrada	CAM	1	indica a finalização da construção do caminho
relacoes_wr_enable	saída	CAM	1	solicitação de escrita na MR
relacoes_wr_addr	saída	CAM	10	endereço do nó
relacoes_wr_data	saída	CAM	8*15	dados da relação
obstaculos_wr_enable	saída	CAM	1	solicitação de escrita na MO
obstaculos_wr_addr	saída	CAM	10	endereço do nó obstáculo
obstaculos_wr_data	saída	CAM	1	valor do obstáculo

6.3 *Gerenciador de Nós Ativos*

Quando um nó que está atualmente no estado inativo e não é um obstáculo é identificado como vizinho de um nó aprovado, ele é definido como um nó ativo, permanecendo neste estado até ser aprovado na etapa de classificação.

A etapa de classificação por sua vez é formada pela realização de três etapas. Em um primeiro momento, é encontrado o critério de classificação de cada nó, calculado através da soma do relacionamento entre seus vizinhos de menor custo que não são obstáculos com a sua distância atual até a fonte, de acordo com o definido no Capítulo 4. Logo após, o menor critério de classificação entre os nós ativos existentes é encontrado, defindo o critério geral de classificação. Por fim, os nós ativos passam por um processo de avaliação: aqueles cuja distância atual é menor ou igual ao critério geral são considerados aprovados e devem ser encaminhados ao *Localizador de Vizinho Válido* para que o processo de expansão seja realizado.

O papel de gerenciar os nós ativos, realizar o processo de classificação e armazená-los é realizado pelo *Gerenciador de Nós Ativos* (*GNA*). Além disso, o *GNA* é responsável também por guardar as seguintes informações de cada nó ativo, necessárias ao cálculo do menor caminho:

- endereço do nó ativo;
- distância atual até a fonte;

- endereço do nó anterior; e
- menor custo entre seus vizinhos que não são obstáculos.

Internamente o *GNA* é subdividido em três sub-blocos (Figura 31):

- *Nó Ativo* (*NA*) - armazena informações críticas sobre cada nó ativo, como distância atual, menor vizinho, nó anterior atual e endereço. Também é responsável por calcular o critério de classificação do nó individualmente;
- *Classificador de Ativos* (*CA*) - identifica o menor critério de classificação entre os nós ativos; e
- *Gerenciador de Ativos* (*GA*) - gerencia os processos de escrita para nós ativos e de desativação de nós estabelecidos.

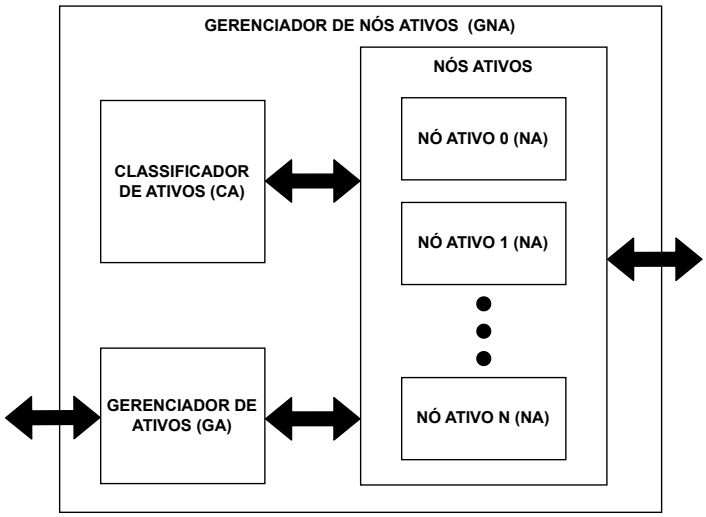


Figura 31 – *Gerenciador de Nós Ativos*.

O bloco *NA* (Figura 32) armazena informações críticas sobre cada nó ativo, como distância atual, menor vizinho, nó anterior atual e endereço. Também é responsável por calcular o critério de classificação do nó individualmente. No início do cálculo do menor caminho, todos os *NA* se encontram desativados, sendo ativados conforme a evolução do algoritmo.

A quantidade de *NA* a ser implementada dependerá das características do grafo utilizado: quanto maior for a quantidade de nós do grafo e relacionamentos, maior será a quantidade de nós que serão ativados ao mesmo tempo. Para determinar a quantidade máxima de nós ativos, testes são realizados no grafo em análise com diferentes configurações de obstáculos, nó destino e origem. O maior valor obtido é empregado como parâmetro para definir a quantidade de *NA*. Vale ressaltar que o número de nós ativos é consideravelmente inferior à quantidade total de nós do grafo - para um grafo com 1024 nós e 8 relações, por exemplo, são necessários 88 *NA*.

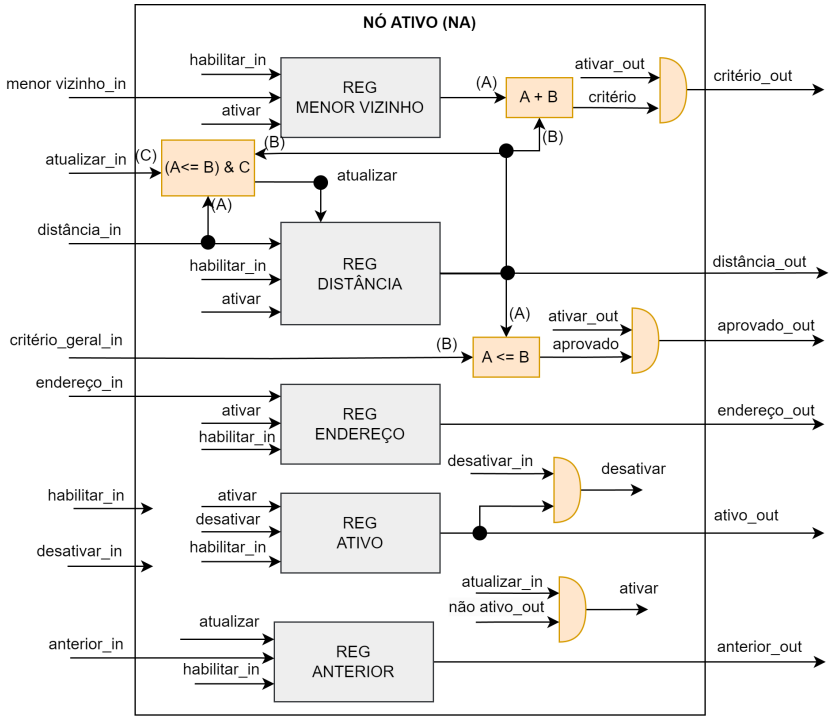


Figura 32 – Nó Ativo.

Quando um novo nó precisa ser ativado, é tarefa do *GA* (Figura 33) identificar um *NA* disponível. Para tal, o *GA* primeiramente verifica dentre os *NA* atualmente ativos se o nó a ser ativado já se encontra armazenado: essa verificação é realizada através da comparação do endereço do nó recebido com o endereço armazenado em cada um dos nós atualmente ativos. Caso já exista um *NA* com tal endereço, o mesmo é habilitado para atualização; caso contrário, um *NA* inativo é utilizado.

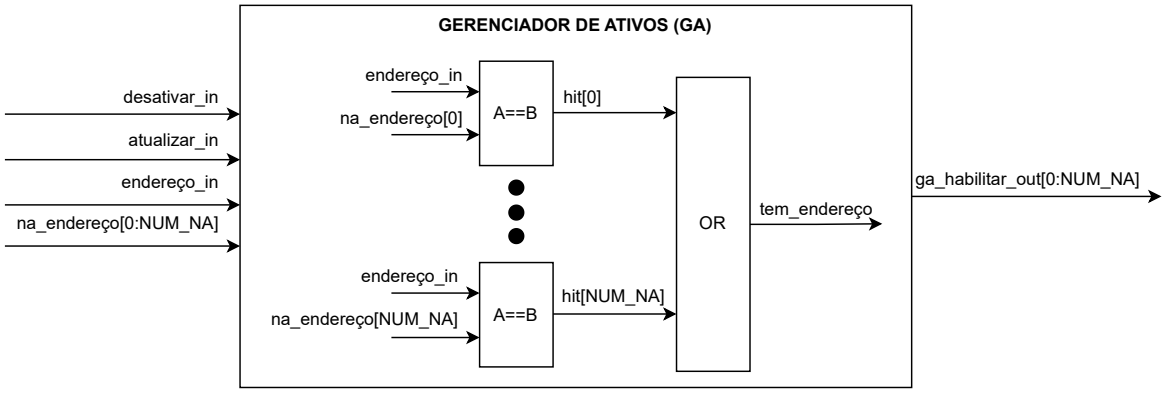


Figura 33 – *Gerenciador de Ativos*.

A busca por um nó inativo é implementada como demonstrado na FSM da Figura 34, sendo realizada sempre que ocorre uma ativação ou desativação de um nó ativo, como resultado se obtêm um vetor que aponta para os *NA* inativos. Quando um novo nó precisa ser ativado, uma posição desse vetor é utilizada.

Para nós inativos que serão ativados, as informações recebidas são armazenadas sem a

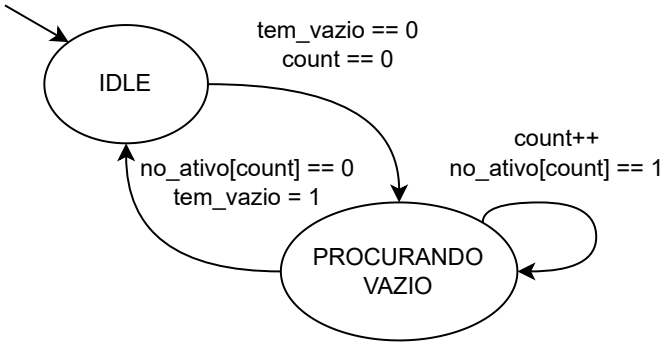


Figura 34 – FSM do *Gerenciador de Ativos*.

necessidade de validação. No entanto, para nós que já se encontram ativos, é necessário verificar se a nova distância recebida é menor do que a atualmente armazenada: os valores da distância e do nó anterior só serão atualizados caso esse critério seja cumprido. Essa regra só não se aplica para o armazenamento do custo do menor vizinho, uma vez que seu valor não se altera com a evolução do algoritmo (o mesmo não é atualizado, apenas inserido na ativação do nó).

Cada *NA* realiza também a identificação dos nós aprovados, comparando o critério geral de classificação recebido com o critério de classificação do nó atualmente armazenado. Para cada *NA* aprovado é gerado um sinal a ser encaminhado para o *Localizador de Vizinho Válido*, identificando os aprovados. Após recebidos no *Localizador de Vizinho Válido* e processados, um sinal de desativação é recebido no *GNA*, fazendo com que os *NA* correspondentes sejam desativados, liberando espaço para receber novos nós.

Para determinar o critério de classificação geral, o *CA* utiliza um comparador que recebe os critérios dos nós ativos e realiza a comparação. O objetivo desta comparação é encontrar o menor critério entre todos os nós ativos. Em um cenário ideal, o mais interessante seria que todas essas comparações fossem realizadas de forma combinacional em apenas um ciclo de clock, aumentando assim o desempenho do sistema, porém, a depender do número de nós que devem ser comparados, isso pode não ser possível devido às limitações de temporização do dispositivo empregado.

Como alternativa, uma estrutura sequencial de comparação em blocos de dados foi construída, como exposto na Figura 35. Nessa abordagem, um módulo controlador é responsável por encaminhar grupos de dados de classificação ao comparador, esses grupos são então comparados entre si e com um registrador base (*DATA_BASE REG*). Na primeira iteração, *DATA_BASE REG* é inicializado com um valor infinito e a cada iteração, o menor valor obtido é armazenado nele. Após todos os grupos de dados serem analisados, o menor critério de classificação pode ser coletado em (*DATA_BASE REG*).

O número máximo de comparadores que podem ser utilizados de forma combinacional está relacionado à tecnologia e ao período de clock utilizado. Assim, para obter o melhor desempenho, esta quantidade máxima de comparadores deve ser configurada de acordo com o FPGA

escolhido. No capítulo 7 são exibidas algumas configurações explorando esta funcionalidade. Como exemplo, na Figura 35 é mostrada uma estrutura com oito comparadores combinacionais - neste caso, para uma estrutura com 16 nós ativos, apenas 02 pulsos de clock são necessários para encontrar o critério geral.

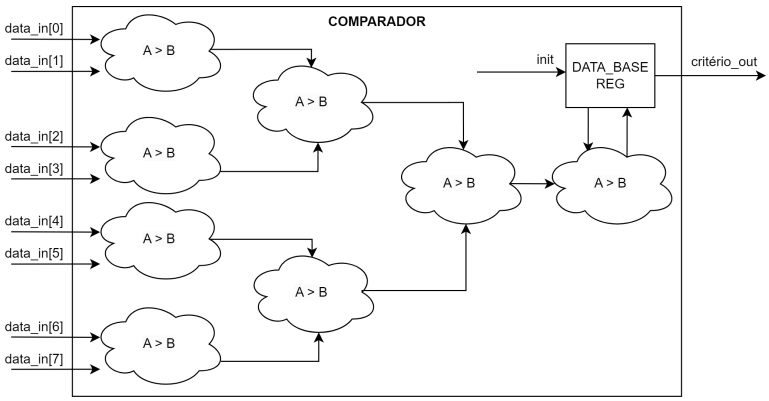


Figura 35 – Bloco comparador do critério *OUT*.

As informações dos sinais que fazem interface com o módulo *Gerenciador de Nós Ativos* para um grafo com 1024 nós e 88 *Nó Ativo* podem ser visualizadas na Tabela 13.

Tabela 13 – Sinais do bloco *Gerenciador de Nós Ativos* para um grafo com 1024 nós e 88 *Nó Ativo*.

Sinal	**Direção**	**Conexão**	**Bits**	**Descrição**
clk	entrada	-	1	clock do sistema
rst_n	entrada	-	1	reset do sistema
cme_atualizar_fonte	entrada	CME	1	insere a fonte como nó ativo na primeira iteração
cme_endereco_fonte	entrada	CME	10	endereço da fonte
cme_atualizar_classificacao	entrada	CME	1	comando para atualizar a classificação
lvv_pronto	entrada	LVV	1	indica que o lvv está pronto para receber novos nós
lvv_desativar	entrada	LVV	1	comando de desativar um nó
lvv_atualizar	entrada	LVV	1	comando de atualizar um nó
lvv_vizinho_valido	entrada	LVV	8	indica qual o vizinho válido
lvv_endereco	entrada	LVV	8*10	endereço do nó a ser atualizado/desativado
lvv_menor_vizinho	entrada	LVV	8*5	custo do menor vizinho do nó
lvv_distancia	entrada	LVV	8*10	distância do nó
lvv_anterior	entrada	LVV	10	anterior atual do nó
gna_aprovado	saída	LVV	88	indica os nós que estão aprovados
gna_endereco	saída	LVV	88*10	endereço dos nós
gna_distancia	saída	LVV	88*15	distância dos nós
gna_anterior_data	saída	LVV	88*10	anteriores dos nós
gna_atualizar_ready	saída	LVV	1	indica a conclusão de uma atualização
gna_tem_aprovado	saída	LVV	1	indica que existe pelo menos um aprovado
gna_tem_ativo	saída	CME	1	indica que existe pelo menos um ativo
gna_ocupado	saída	LVV, CME	1	indica que o bloco está ocupado
gna_pronto	saída	LVV, CME	1	indica que o bloco está livre para receber comandos

6.4 *Localizador de Vizinho Válido*

Para cada nó aprovado é realizado um processo de expansão no *Localizador de Vizinho Válido* (*LVV*), o que objetiva atualizar as distâncias de seus nós vizinhos até a fonte juntamente

com a identificação do seu nó anterior. Neste processo, apenas os vizinhos que não são obstáculos e ainda não foram estabelecidos serão atualizados: vizinhos com essas características são considerados validos e serão ativados ou atualizados no *GNA*, sendo os demais ignorados.

Internamente, o *LVV* é formado por dois sub-blocos (Figura 36):

- *Expansor de Nó Aprovado* (*ENA*), responsável por realizar o processo de expansão de forma individual de cada nó aprovado;
- *Gerenciador de Leitura e Escrita* (*GLE*), que gerencia as trocas de informações entre os *ENA* e as mémorias externas ao *LVV*.

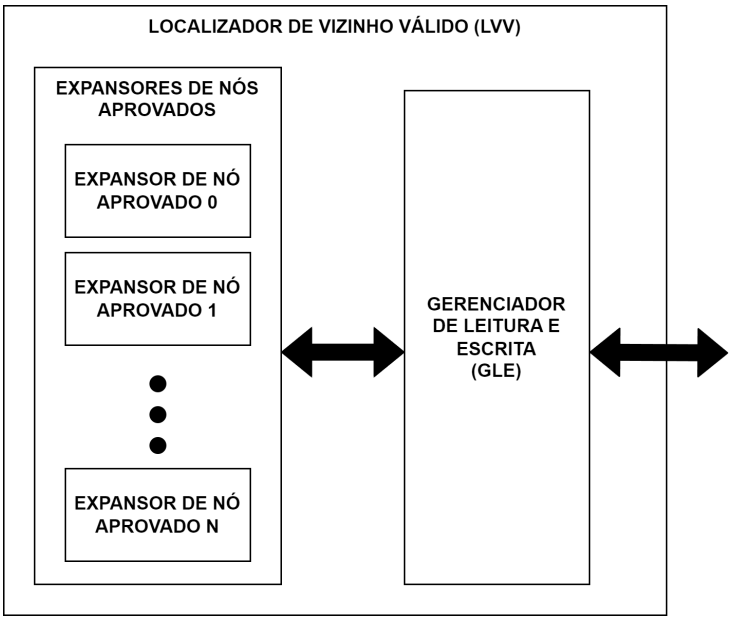


Figura 36 – *Localizador de Vizinho Válido*.

As informações dos sinais que fazem interface com o módulo *Localizador de Vizinho Válido* para um grafo com 1024 nós podem ser visualizadas na Tabela 14.

A primeira etapa realizada no processo de expansão é a identificação dos vizinhos válidos do nó aprovado. Logo após, deve-se identificar o vizinho do vizinho (sub-vizinho) com menor custo, pois esta informação é utilizada no cálculo do critério *OUT* na etapa de classificação. É importante ressaltar que esse vizinho de menor custo também não deve ser um obstáculo, o que exige uma verificação na MO. Assim, para cada nó aprovado, são realizadas as seguintes leituras nas memórias, considerando *MAX_N* o número máximo de vizinhos de um nó:

- 1x *Memória de Relacionamentos* - identificação das relações de um nó;
- *MAX_N* x *Memória de Obstáculos* - identificação das relações que são obstáculos;
- *MAX_N* x *Memória de Estabelecidos* - identificação das relações estabelecidas;

Tabela 14 – Sinais de saída do bloco *Localizador de Vizinho Válido* para um grafo com 1024 nós.

Sinal	**Direção**	**Bits**	**Conexão**	**Descrição**
clk	entrada	-	1	clock do sistema
rst_n	entrada	-	1	reset do sistema
cme_expandir_in	entrada	CME	1	comando para iniciar o processo de expansão
gna_pronto	entrada	GNA	1	indica que o bloco GNA está livre para receber comandos
gna_ocupado	entrada	GNA	1	indica que o bloco GNA está ocupado
gna_anterior_data	entrada	GNA	88*10	anterior atual do nó aprovado
gna_aprovado	entrada	GNA	88	indicação dos nós que estão aprovados
gna_endereco	entrada	GNA	88*10	endereço do nó aprovado
gna_distancia	entrada	GNA	88*15	distância do nó aprovado
gna_atualizar_ready	entrada	GNA	1	indica que a ultima solicitação de atualização/desativação foi concluída
lvv_desativar	saída	GNA	1	comando de desativar um nó
lvv_atualizar	saída	GNA	1	comando de ativar um nó
lvv_vizinho_valido	saída	GNA	8	indicação dos vizinhos válidos
lvv_endereco	saída	GNA	8*10	endereço do nó
lvv_menor_vizinho	saída	GNA	8*5	custo do menor vizinho do nó
lvv_distancia	saída	GNA	8*15	distância do nó
lvv_anterior	saída	GNA	10	anterior atual do nó
lvv_estabelecidos_write_en	saída	CAM	1	escreve em MA e ME
lvv_estabelecidos_write_addr	saída	CAM	10	endereço do nó a ser estabelecido
lvv_anterior_data	saída	CAM	10	endereço do nó anterior
lvv_pronto	saída	CME, GNA	1	indica que o LVV terminou o processo de expansão
lvv_relacoes_read_addr	saída	CAM	8*10	endereço das relações a ser lido
cam_relacoes_read_data	entrada	CAM	8*120	relações lidas
lvv_obstaculos_read_addr	saída	CAM	8*10	endereço dos obstáculos a ser lido
cam_obstaculos_read_data	entrada	CAM	8	obstáculos lidos
lvv_estabelecidos_read_addr	saída	CAM	8*10	endereço dos estabelecidos a ser lido
cam_read_data	entrada	CAM	8	estabelecidos lidos

- *MAX_N* x *Memória de Relacionamentos* - identificação de relacionamentos dos vizinhos para identificar o sub-vizinho com menor custo;
- *MAX_N* x *MAX_N* x *Memória de Obstáculos* - identificação dos menores sub-vizinhos que são obstáculos .

Após o processo de expansão, a distância do nó aprovado até a fonte é adicionada ao custo de deslocamento do nó atualmente aprovado até seu vizinho, gerando assim uma possível nova distância deste vizinho até a fonte - essa informação é então encaminhada para o *Gerenciador de Nós Ativos*, onde poderá ser armazenada. Esta nova distância só é armazenada em duas situações: i) quando o vizinho é um novo nó a ser ativado, pois ainda não possui uma distância válida; e ii) quando a nova distância for menor que a atualmente armazenada. Sempre que o valor da distância de um nó é alterado, o nó anterior desse nó também é alterado para o nó aprovado que causou a mudança.

Quando não houver mais nós ativos no *GNA*, o processo de análise é finalizado e o menor caminho pode ser construído. A partir do nó anterior do nó destino, é realizada uma sequência de leitura dos anteriores até que o nó de origem seja alcançado. Esta etapa é executada sob demanda

após o *Controlador de Máquina de Estados* sinalizar ao bloco *Controle de Acesso Externo* que o processo de expansão foi concluído.

Conforme mostrado, o processo de expansão envolve diversos procedimentos de acesso à memória que, se realizados sequencialmente, podem gerar períodos de inatividade da memória, reduzindo a eficiência do sistema. Para resolver esse problema, a arquitetura do *LVV* foi projetada com foco no paralelismo durante o processo de expansão dos nós, permitindo um acesso escalonado às memórias ao realizar a expansão de diversos nós de forma concorrente.

O *LVV* possui uma máquina de estados finita (do inglês *Finite State Machine*) (FSM) central (Figura 37) que gerencia o processo de expansão de todos os nós aprovados. Cada nó aprovado tem seu processo de expansão realizado de forma independente por instâncias do bloco *ENA*. Leituras e escritas externas são gerenciadas pelo *GLE*.

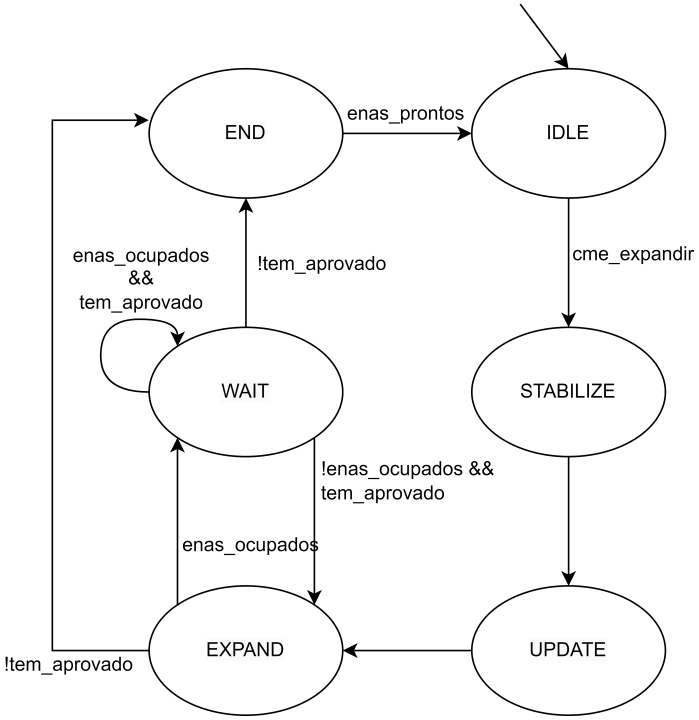


Figura 37 – Máquina de estados do *Localizador de Vizinho Válido*.

A FSM do *LVV* inicia no estado *IDLE* e permanece nele até ser ativado pelo *Controlador de Máquina de Estados*. Quando ativada, muda para o estado *STABILIZE*, onde os nós aprovados são marcados como estabilizados no *Controle de Acesso às Memórias*, e desativados no *Nó Ativo* correspondente. É nesse momento que a *Memória de Anteriores* é atualizada com o endereço do nó anterior do nó aprovado - nesse caso sendo o anterior definitivo que será utilizado na construção do menor caminho.

Em seguida, a FSM transita para o estado *UPDATE*, com o objetivo de aguardar a realização da atualização das memórias de estabelecidos e anteriores. Essa espera é necessária, devido às consultas posteriores que serão realizadas na *Memória de Estabelecidos* durante o processo de expansão - desta forma, se garante a consistência da informação armazenada com o

estado atual do nó. Sequencialmente, a FSM transiciona para o estado *EXPAND*, onde, a cada iteração, um nó aprovado é selecionado e enviado para processamento por um *Expansor de Nó Aprovado* disponível.

A FSM permanece no estado *EXPAND* enquanto houver instâncias do *ENA* disponíveis e existirem nós a serem expandidos. Quando todas as instâncias do *ENA* estão ocupadas, a FSM muda para o estado *WAIT* e permanece nele até que um *ENA* livre apareça, ou não existam mais nós aprovados a serem analisados. Se algum *ENA* ficar disponível, a FSM retorna ao estado *EXPAND*.

Quando não houver mais nós aprovados a serem analisados, a FSM transiciona para o estado *END*, permanecendo nele até que todos os *GNA* finalizem os processos de expansão. Finalmente, quando as atualizações forem concluídas, a FSM retorna ao estado *IDLE*.

Todas as solicitações de leitura e escrita do *Expansor de Nó Aprovado* são gerenciadas pelo *Gerenciador de Leitura e Escrita*. Para viabilizar seu funcionamento, foi especificado um protocolo de solicitação e resposta (Figura 38). Nesta comunicação, cada *Expansor de Nó Aprovado* possui um barramento de comunicação dedicado, viabilizando solicitações simultâneas. Isto permite um melhor aproveitamento do acesso às memorias, pois enquanto um *ENA* está ocupado outro tem a possibilidade de acessar uma interface de memória ociosa.

Deste modo, sempre que uma solicitação de leitura é realizada por um dos *ENA*, um sinal do tipo *habilitar_leitura* é enviado do módulo solicitante para o *Gerenciador de Leitura e Escrita*; quando o *Gerenciador de Leitura e Escrita* obtêm a informação desejada ele responde com um sinal do tipo *leitura_pronta* indicando que a informação solicitada está pronta para ser coletada. O processo de escrita ocorre de forma similar, com um sinal de solicitação e um de resposta de conclusão.

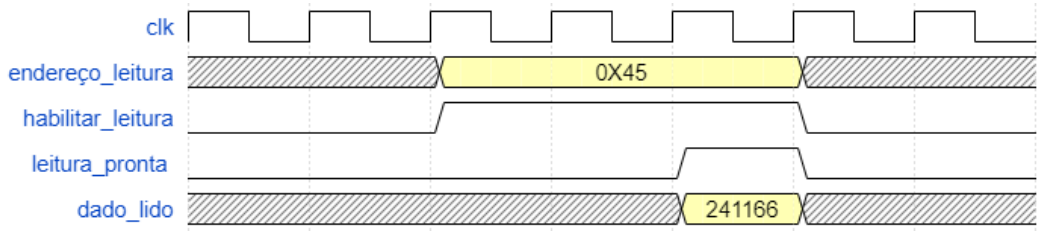


Figura 38 – Protocolo de comunicação do *Gerenciador de Leitura e Escrita*.

Para cada nó aprovado é necessário realizar um processo de expansão, objetivando identificar os vizinhos dos nós atualmente aprovados, juntamente com informações necessárias para a construção do menor caminho, como o nó anterior do vizinho, sua distância até a fonte e custo a ser utilizado no processo de classificação. Para obtenção dessas informações, como demonstrado no fluxograma da Figura 39 o processo de expansão de um nó envolve várias etapas de processamento, leitura de memórias e decisões lógicas.

Primeiramente, é necessário ler e salvar localmente as relações do nó aprovado, identifi-

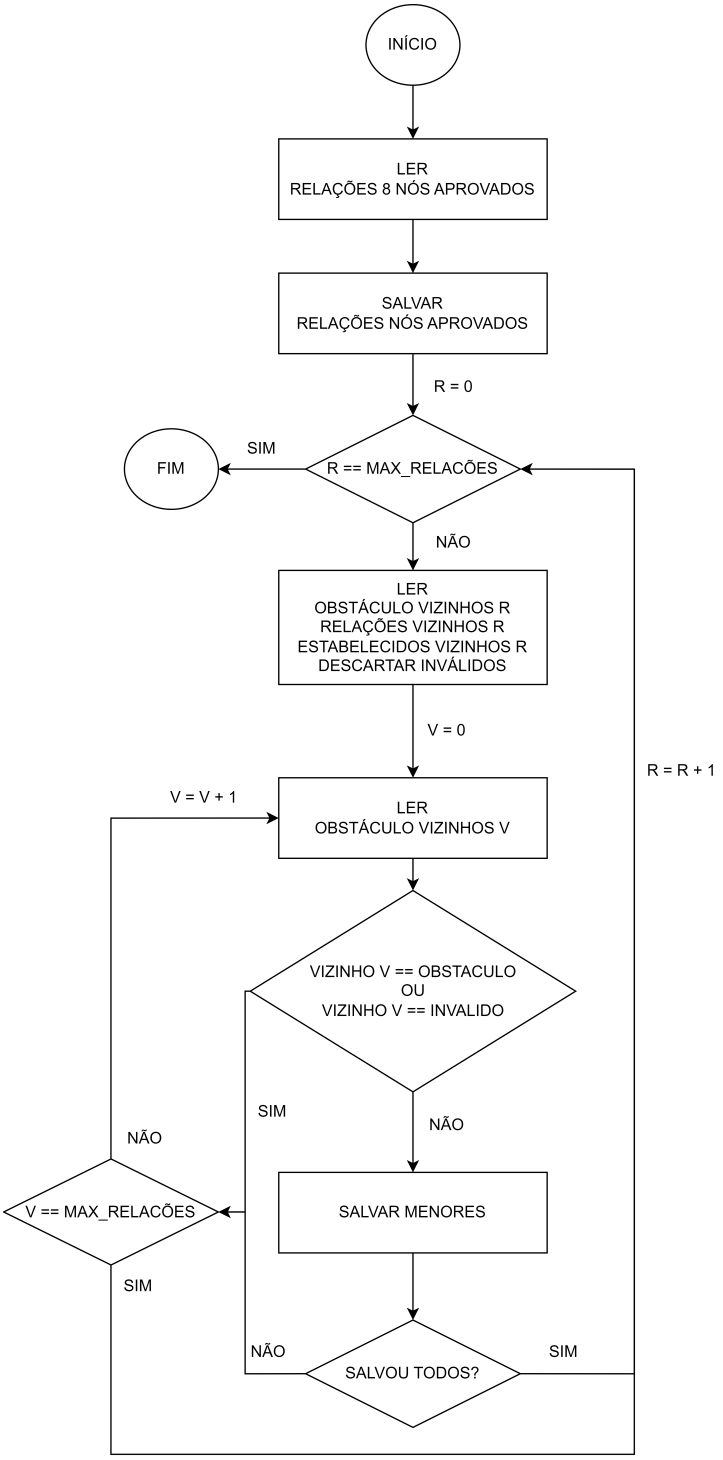


Figura 39 – Fluxograma do processo de expansão de um nó.

cando seus vizinhos. Após, para cada vizinho (R), identificar aqueles que são válidos, ou seja, não são obstáculos nem estabelecidos. Além disso, para identificar o sub-vizinho (V) é necessário também ler as relações do vizinho e verificar se é um obstáculo - caso não seja, se deve salvar o custo deste vizinho, uma vez que o mesmo será utilizado no critério de classificação.

Para viabilizar esse fluxo de processamento, o *Expansor de Nó Aprovado* implementa uma FSM com nove estados, conforme mostrado na Figura 40. A FSM é inicializada no estado

IDLE e permanece no mesmo até que o sinal *vnl_write_approved_in* seja ativado, indicando que o *Localizador de Vizinho Válido* enviou um novo nó aprovado para análise.

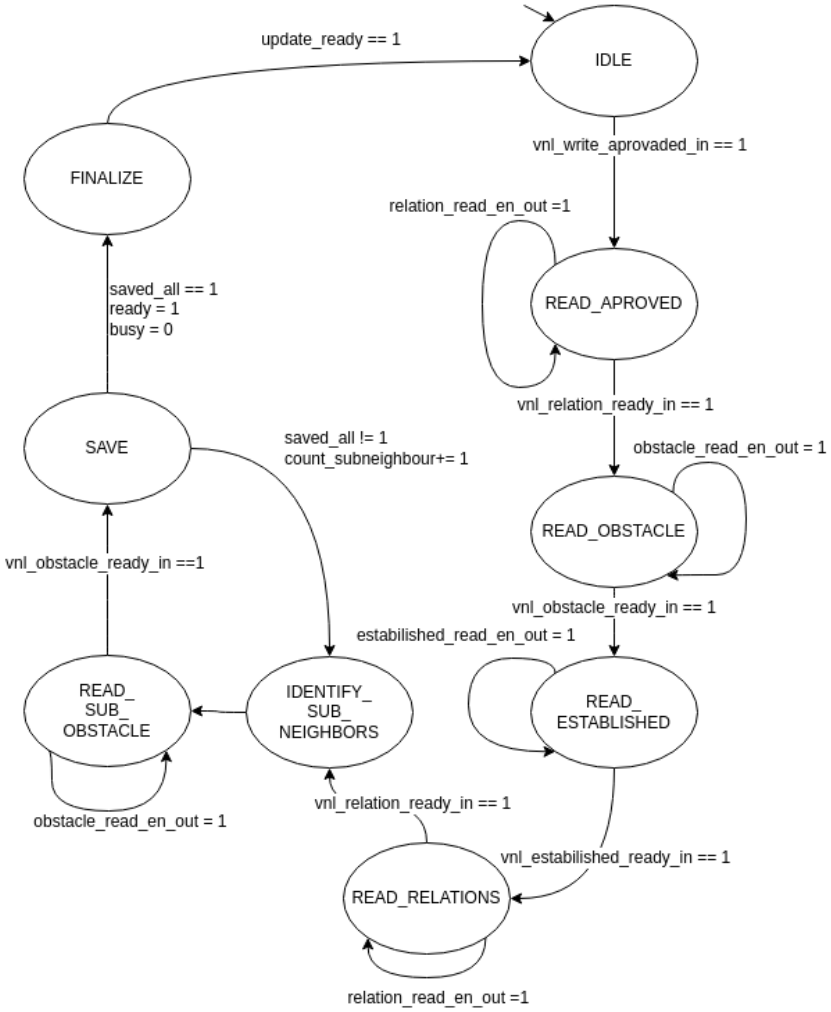


Figura 40 – Máquina de estados do *Expansor de Nó Aprovado*.

Após a ativação, a FSM transita para o estado *READ_APROVED*, onde gera uma solicitação para ler as relações existentes no nó aprovado. Esta solicitação é recebida pelo *Gerenciador de Leitura e Escrita* que a encaminhada ao *Gerenciador de Memórias com Acesso Externo*. Quando a informação solicitada está disponível, o sinal *vnl_relation_ready_in* é ativado, e a FSM transita para o estado *READ_OBSTACLE*, onde verifica se os vizinhos pertencem aos obstáculos. Em seguida, transita para o estado *READ_ESTABLISHED*, onde verifica se os vizinhos já foram estabelecidos.

Uma vez identificado o obstáculo e os vizinhos estabelecidos, a FSM muda para o estado *READ_RELATIONS*, onde realiza o processo de expansão de cada vizinho. Isso envolve identificar os sub-vizinhos do nó aprovado (*IDENTIFY_SUB_NEIGHBORS*), descartar sub-vizinhos que são obstáculos (*READ_SUB_OBSTACLE*) e encontrar o custo do menor sub-vizinho. Então, é solicitada a atualização das informações do vizinho no *NA* no estado *SAVE*. Caso ainda existam vizinhos para analisar, a FSM retorna ao estado *READ_RELATIONS* para realizar a

análise do próximo vizinho. Este ciclo continua até que todos os vizinhos válidos tenham sido avaliados. Após isso, a FSM entra no estado *FINALIZE* e permanece nele até que o sinal *update_ready* seja ativado, indicando que todas as informações foram atualizadas com sucesso no *NA* correspondente.

6.5 *Controlador de Máquina de Estados*

O *Controlador de Máquina de Estados* (*CME*) desempenha o papel de supervisão e controle de todo o processo de identificação do menor caminho, enviando sinais de controle para os demais blocos do projeto, ditando quais etapas do algoritmo devem ser executadas e determinado quando a análise deve ser concluída. Para realizar suas atividades, o *CME* implementa uma FSM com sete estados (Figura 41):

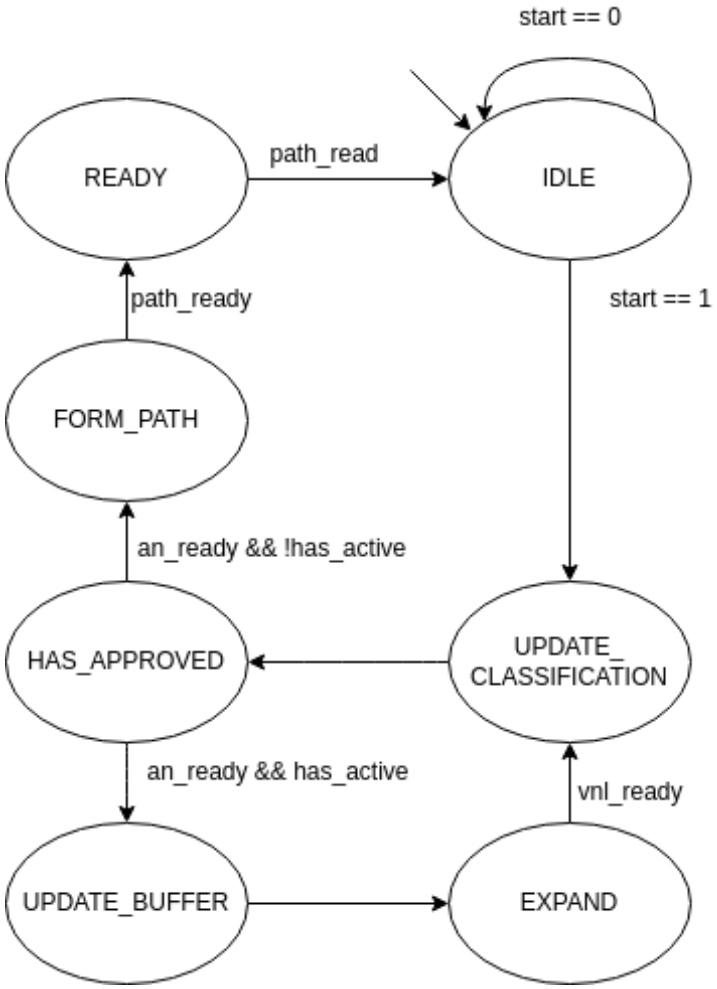


Figura 41 – Máquina de estados finita do *Controlador de Máquina de Estados*.

A FSM do *Controlador de Máquina de Estados* inicia no estado *IDLE* e permanece nele até receber um sinal do *Controle de Acesso Externo* indicando uma solicitação de busca por um novo menor caminho. Ao receber este sinal, a FSM transita para o estado *UPDATE_CLASSIFICATION*, onde solicita a atualização do critério geral de classificação dos nós atualmente ativos

no *Gerenciador de Nós Ativos*. Nesta primeira iteração, o nó de origem se encontra neste estado, os demais são ativados de acordo com a evolução do processo de expansão dos nós aprovados.

A seguir, no estado *HAS_APPROVED*, é verificado se há algum nó aprovado. Caso exista, a FSM muda para o estado *UPDATE_BUFFER*, onde armazena as informações dos nós aprovados em um *buffer*. Essa etapa é necessária uma vez que os nós aprovados são desativados no *Gerenciador de Nós Ativos* no início do processo, objetivando liberar espaço para escrita de novos nós, porém, as informações de distância e endereço do nó aprovado ainda são utilizadas durante o processo de expansão no *LVV*.

Logo após, ocorre a transição para o estado *EXPAND*, onde o *Localizador de Vizinho Válido* lê e analisa os nós aprovados armazenados no buffer. Depois que todos os nós aprovados serem lidos e analisados, o *LVV* ativa o sinal *vnl_ready*, fazendo com que a FSM retorne ao estado *UPDATE_CLASSIFICATION*. Este ciclo continua até que não existam mais nós aprovados, indicando a conclusão do processo de expansão.

Quando não há mais nós aprovados, a FSM muda para o estado *FORM_PATH*, onde o caminho a partir dos nós anteriores armazenados no *CAM* é construído. Assim que a aplicação controlador finalizar a leitura do caminho formado, há a indicação que o processo de construção e leitura do menor caminho foi concluído - deste modo, a FSM retorna ao estado *IDLE* para aguardar uma nova requisição.

As informações dos sinais que fazem interface com o módulo *Controlador de Máquina de Estados* para um grafo com 1024 nós podem ser visualizadas na Tabela 15.

Tabela 15 – Sinais de saída do bloco *Controlador de Máquina de Estados* para um grafo com 1024 nós.

Sinal	**Direção**	**Conexão**	**Bits**	**Descrição**
clk	entrada	-	1	clock do sistema
rst_n	entrada	-	1	reset do sistema
soft_reset_n	saída	CAE	1	reinicializa as MA e ME
gna_tem_ativo_in	entrada	GNA	1	indica que existe pelo menos um ativo no GNA
gna_pronto_in	entrada	GNA	1	indica que o GNA está livre para receber comandos
gna_ocupado_in	entrada	GNA	1	indica que o GNA está ocupado
atualizar_buffer_out	saída	GNA	1	atualiza o buffer do GNA com as informações atuais
atualizar_classificacao_out	saída	GNA	1	atualiza a classificação dos nós ativos
smc_fonte	saída	GNA, CAM	10	endereço da fonte registrado
smc_destino	saída	GNA, CAM	10	endereço do destino registrado
smc_iniciar	saída	GNA	1	ativa a fonte no GNA
lvv_pronto_in	entrada	LVV	1	indica que o LVV está pronto para receber comandos
expandir_out	saída	LVV	1	solicita um novo processo de expansão no LVV
cam_caminho_pronto_in	entrada	CAM	1	indica que o processo de construção do caminho no CAM foi concluído
construir_caminho_out	saída	CAM	1	solicita a construção do caminho no CAM
cae_fonte	entrada	CAE	10	fonte recebida pelo CAE
cae_destino	entrada	CAE	10	destino recevido pelo CAE
cae_wr_fonte_in	entrada	CAE	1	indica que um novo menor caminho deve ser construído

CAPÍTULO

7

AVALIAÇÃO DA ARQUITETURA

Seguindo o fluxo de projeto digital apresentado no Anexo B, após o desenvolvimento da microarquitetura se iniciou a etapa de projeto lógico. Deste modo, a arquitetura construída foi traduzida para uma representação em RTL usando a linguagem Verilog, e o processo de verificação foi iniciado. A verificação funcional do RTL criado foi realizada através de simulações na ferramenta Vivado 2023.1 da Xilinx, onde os resultados obtidos foram comparados com um modelo de referência implementado em Python.

Vale ressaltar que, devido a grande quantidade de simulações necessárias para validar o projeto, foi realizada uma integração entre a ferramenta de simulação e o modelo de referência, o que permitiu a elaboração dos testes de forma automática. Essa integração engloba a geração dos parâmetros de configuração do grafo de forma aleatória, considerando um escopo de aplicação, simulações do modelo de referência e do RTL, coleta e comparação dos resultados das simulações. Durante a implementação dos testes foram realizadas etapas de readequação do código Verilog até que uma taxa de acertos de 100% fosse atingida .

Após a validação do projeto, foi realizada a etapa de síntese lógica, onde o hardware gerado em RTL foi traduzido para componentes estruturais correspondentes a portas lógicas, flip-flops e fios. Posteriormente, foi executada a etapa *Place and Route*, onde os componentes gerados na síntese foram alocados de acordo com os recursos do FPGA.

Após a etapa de *Place and Route*, como os componentes já estão alocados fisicamente, é possível obter uma estimativa precisa do tempo e dos recursos do FPGA utilizados como Look-Up Table (LUT) e registradores, gerando assim medidas de desempenho do projeto. Neste momento, foi necessário também realizar ajustes em algumas partes do código RTL que apresentavam caminhos combinacionais muito grandes. Isto envolveu, por exemplo, o redimensionamento dos comparadores no *Classificador de Ativos* e a reestruturação da forma como o *Gerenciador de Ativos* identificava os *NA* desativados, uma vez que o processo de busca dos nós estava muito grande e precisou ser refatorado para atingir os requisitos de temporização.

Após o sucesso nas etapas de simulação e síntese, se definiu o EP4CE115F29C7, um FPGA Altera da família Cyclone IV E, como aquele que abarcaria a maioria dos testes aqui apresentados. Esta escolha foi realizada devido às vantagens oferecidas por esta família de dispositivos como o seu baixo custo e reduzido consumo de energia, além de permitir a inclusão do NIOS II em alguns testes, para propósitos de comparação de resultados.

Durante a realização dos testes, para nós de origem e destino foram utilizados aqueles com maiores distâncias entre si, como utilizado na Figura 18. Tal figura apresenta um exemplo com 30 nós, sendo os nós brancos representantes de obstáculos e os nós vermelhos os constituintes do menor caminho. Deste modo, os resultados aqui obtidos consideram o pior cenário e o maior tempo de processamento possível para cada grafo, pois, como o algoritmo termina quando o nó de origem é estabelecido, nós origem e destino próximos têm um tempo de execução menor.

Ao longo da etapa de simulação e testes, observou-se que o número máximo de nós ativos depende de algumas características do grafo utilizado, como a quantidade de nós e o número máximo de relações. Como os nós ativos são armazenados nos módulos *NA*, se o número de módulos *NA* utilizados for insuficiente, ocorrerá um erro. Isso acontece porque haveria situações em que um vizinho de um nó aprovado não pôde ser armazenado por não haver espaço disponível, causando o travamento do sistema. Por outro lado, uma quantidade maior de módulos *NA* consome recursos que não serão utilizados.

Assim, é necessário utilizar um valor próximo ao número máximo de nós ativos simultaneamente. Nesta proposta de arquitetura, um grafo com 1.024 nós e até 8 relações foi utilizado na maioria dos testes a seguir. Nestes casos, foi possível identificar nas simulações que o número máximo de nós ativos é 88, sendo este o número de módulos *NA* utilizados. Para grafos com tamanhos diferentes, o mesmo procedimento foi adotado.

7.1 Encontrando a configuração ideal

Foram realizados testes para medir o impacto da utilização de diferentes quantidades de comparadores durante a etapa de classificação no *Gerenciador de Nós Ativos*, conforme mostrado na Tabela 16. O experimento demonstrou que aumentar o número de comparadores resulta em uma redução nos pulsos de clock necessários para calcular o menor caminho. Porém, com esse melhor desempenho ocorre também um aumento na complexidade da lógica envolvida, sendo necessário, nos casos mais complexos, diminuir o período de clock para aderir às restrições de tempo do FPGA. Portanto, ao considerar o tempo de simulação baseado no melhor período de clock alcançado para cada teste (coluna Melhor Clock da tabela), a configuração utilizando 3 comparadores apresentou a relação custo-benefício mais favorável. Esse resultado é evidenciado ao se calcular o ganho com relação ao pior tempo, calculado ao dividir o pior tempo pelo tempo obtido com cada clock.

Também foram realizados testes variando a quantidade de *Expansor de Nó Aprovado*.

Tabela 16 – Resultados com diferentes números de comparadores para um grafo com 1.024 nós e até 8 relações

NUM CA	Melhor Clock (ns)	Pulsos de Clock	Tempo com melhor clock(ns)	Ganho sobre o pior
1 CA	12,0	20.654	247.842	1,67
2 CA	12,0	16.784	201.402	2,06
3 CA (melhor)	**12,0**	**15.434**	**185.202**	**2,24**
4 CA	14,0	14.804	207.249	2,00
5 CA	20,0	14.444	288.870	1,43
6 CA	22,0	14.174	311.817	1,33
7 CA	25,5	139.94	356.834	1,16
8 CA (pior)	**30,3**	**13.814**	**414.405**	**-**

Como é possível observar na Tabela 17, há uma estabilização do tempo de processamento a partir de 8 *ENA*. Isso ocorre porque os barramentos de dados das memórias acabam sendo saturados, não havendo espaço para inserção de mais requisições. Por outro lado, também se observa uma redução no desempenho, com quantidades maiores de *ENA*, devido ao aumento da complexidade do processo de gestão de trocas de informações. Além disso, há um aumento progressivo na utilização de recursos do FPGA como Registradores e LUTs. Assim, para o tipo de grafo em análise, a utilização de 8 *ENA* foi identificada como a configuração ideal.

Tabela 17 – Resultados com diferentes *ENA* para um grafo com 1.024 nós e até 8 relações

NUM *ENA*	Melhor Clock (ns)	Pulsos de Clock	Tempo com melhor clock(ns)	Ganho sobre o pior
1 *ENA* (pior)	**12,00**	**34.829**	**417.942**	**-**
2 *ENA*	12,00	23.367	280.398	1,491
4 *ENA*	12,00	16.579	198.942	2,101
8 *ENA* (melhor)	**12,00**	**15.434**	**185.202**	**2,257**
9 *ENA*	12,00	15.437	185.238	2,256
10 *ENA*	12,00	15.446	185.346	2,255
16 *ENA*	12,00	15.691	188.286	2,220
32 *ENA*	12,00	15.737	188.838	2,213

7.2 Resultados para diferentes tamanhos de grafos

Conforme mostrado na Tabela 18, testes com diferentes tamanhos de grafos demonstraram que foi possível incorporar grafos com até 2.048 nós no FPGA EP4CE115F29C7. Além disso, observou-se, na maioria dos casos, que o tempo de processamento e o consumo de recursos permaneceram proporcionais ao aumento do número de nós. Para grafos com 1.024 nós, o consumo de recursos ficou abaixo de 40%, o que indica a possibilidade de substituição por um FPGA com menos recursos e, consequentemente, menor custo.

A tabela 19 demonstra os resultados de estimação da dissipação de energia do dispositivo de acordo com o tamanho do grafo. Observa-se que a dissipação estática e das portas de entrada e

Tabela 18 – Resultados de síntese com diferentes grafos para o FPGA EP4CE115F29C7

Nós	Pulsos de clock	Melhor clock (ns)	Tempo de simulação (ns)	LUTs (%)	Registradores (%)	Memória (%)
2.048	27.102	15	406.523	43	19	58
1.024	15.434	12	185.202	37	14	26
512	7.859	12	94.302	34	13	13
256	4.078	11	44.853	19	9	8
128	2.212	11	24.327	15	7	8
64	1.468	11	16.143	11	6	8

Tabela 19 – Dissipação de energia térmica de acordo com o tamanho do grafo em mW

	Dissipação de energia (mW)			
Nós	**Dinâmico**	**Estático**	**E/S**	**Total**
2048	1168,3	124,5	32,9	1325,7
1024	911,4	123,3	42,2	1076,9
512	531,3	121,2	35,6	688,1
256	386,4	120,5	42,1	549,0
128	296,6	120,0	24,3	441,0
64	245,2	119,7	22,1	387,1

saída (E/S) se mantêm semelhante para diferentes tamanhos de grafo, sendo o consumo dinâmico o mais afetado com essa modificação

7.3 Ganhos com a memória de obstáculos

Uma das principais características desta proposta é a implementação de memórias separadas para relacionamentos e obstáculos. Esta característica facilita as modificações na configuração do grafo: após mudanças no ambiente apenas a memória de obstáculos necessita ser alterada.

Os requisitos de dados a serem armazenados para um grafo com 1.024 nós, 8 relações e um custo máximo de 31 unidades são demonstrados na Figura 29. Para representar as relações de cada nó são necessários 120 bits de armazenamento por nó, resultando em um total de 122.880 bits para representar todo o grafo. Em uma aplicação convencional, seria necessária a transferência de 122.880 bits para cada nova configuração do grafo. No entanto, o uso de memória de obstáculos simplifica significativamente esse processo, pois é necessário apenas 1 bit por nó. Consequentemente, os dados a serem transmitidos são reduzidos para apenas 1.024 bits.

A eficácia desta abordagem é ilustrada na Tabela 20, que demonstra sua aplicação em um grafo com 1.024 nós para tamanhos variados de barramento de dados entre este projeto e a aplicação Controladora. Esta abordagem diminui o tempo necessário para transferir uma nova

configuração. Para grafos com 1.024 nós, essa melhoria chega a 120 vezes quando comparada à abordagem tradicional, independente do tamanho do barramento de dados. Para grafos maiores os resultados são ainda melhores, chegando a um ganho de até 480 vezes para grafos com 4096 nós, por exemplo.

Tabela 20 – Resultados de transferências com diferentes tamanhos de barramento de dados para um grafo com 1.024 nós e até 8 relações

	Resultados de acordo com tamanho do barramento de dados				
Tamanho do barramento de dados	**32**	**64**	**128**	**256**	**512**
Número de transferências para 1.024 bits	32	16	8	4	2
Número de transferências para 122.880 bits	3.840	1.920	960	480	240
Tempo para transferir 1.024 bits com 100MHz (ns)	320	160	80	40	20
Tempo para transferir 122.880 bits com 100MHz (ns)	38.400	19.200	9600	4.800	2.400
Ganho	120	120	120	120	120

Ao analisar a contribuição desta melhoria de desempenho em relação ao tempo total de processamento, é possível perceber com mais clareza as vantagens deste tipo de abordagem. Conforme mostrado na tabela 21, o ganho quando comparado ao modelo convencional pode chegar a 16% ao se utilizar um barramento com 32 bits. Os testes mostraram que quanto menor o barramento utilizado, melhor será o ganho obtido. Barramentos com 8 bits, por exemplo, apresentaram ganho de 63,66%; por outro lado, para barramentos maiores como o de 512 bits o ganho foi de apenas 1%. Isto ocorre devido ao maior potencial de transferência de barramentos maiores, reduzindo o impacto do processo de transferência no tempo total de processamento.

Tabela 21 – Ganho de acordo com o tipo de transferência em um barramento de dados de 32 bits e tempo de processamento para diferentes tamanhos de grafos com clock de 125 MHz

Nós	**Tempo de processamento (ns)**	**Relações e obstáculos (ns)**	**Apenas obstáculos (ns)**	**Ganho (%)**
2.048	406.523	472.059	407.035	15,98%
1.024	185.202	215.922	185.458	16,43%
512	94.302	108.638	94430	15,05%
256	44.853	51.509	44.917	14,68%
128	24.327	27.399	24.359	12,48%
64	16.143	17.551	16.159	8,61%

7.4 O efeito dos obstáculos

Simulações foram realizadas para avaliar o impacto de diferentes quantidades de obstáculos em grafos de 64 a 4.096 nós, conforme apresentado na Tabela 22. Na tabela a quantidade de obstáculos foi definida proporcionalmente a quantidade de nós do grafo, a primeira coluna de resultados (1/2), por exemplo, indica que a metade dos nós do grafo foram marcados como obstáculos, já a última (0), indica que não foi inserido nenhum obstáculo. De um modo geral, é possível observar uma diminuição no tempo de processamento à medida que o número de obstáculos aumenta em relação ao número total de nós no grafo. Isso ocorre porque os nós marcados como obstáculos não são ativados no processo de expansão, o que diminui a quantidade de nós a serem expandidos.

Tabela 22 – Resultados com diferentes quantidades de obstáculos.

Nós	**Pulsos de clock de acordo com o número de obstáculos**					
	1/2	**1/3**	**1/4**	**1/5**	**1/6**	**0**
4.096	44.225	46.790	49.278	50.141	50.495	54.748
2.048	22.015	21.972	25.088	24.897	24.443	27.102
1.024	11.744	13.295	13.813	13.447	14.464	15.434
512	6.566	6.321	6.549	6.100	6.239	7.859
256	4.167	3.954	4.031	3.576	4.065	4.078
128	1.969	2.047	1.921	2.159	2.208	2.212
64	1.242	1.458	1.392	1.495	1.397	1.468

7.5 Resultados com diferentes FPGAs

Para facilitar comparações futuras entre esta proposta e outras aplicações, foi realizada a síntese em diferentes famílias de FPGAs cujas licenças estavam disponíveis durante a realização deste projeto. Os resultados destas sínteses são exibidos na Tabela 23.

Tabela 23 – Resultados de síntese com diferentes famílias de FPGA

FPGA	**LUTs (%)**	**Registradores (%)**	**Frequência máxima (MHz)**	**Grafo máximo**
ALVEO U250	40,15	2,58	200	16.384
Kintex-7	56,05	12,71	111	2.048
Artix-7	70,49	15,98	100	1.024
Kintex UltraScale+	1,00	0,59	125	1.024
Arria 10	2,00	<1,00	222	2.048
Cyclone IV E	43,00	19,00	80	2.048
Cyclone IV GX	86,00	26,00	50	4.096

7.6 Comparações com modelos de referência

Para validar a eficiência dos resultados obtidos, foram utilizados dois modelos de referência. O primeiro modelo foi uma aplicação desenvolvida em C, otimizada para rodar em um computador equipado com um processador AMD Ryzen 5 3600 e frequência de clock de 3,59 GHz. Como segundo modelo de referência, foi criada uma aplicação utilizando o processador NIOS II da Altera e o FPGA EP4CE115F29C7 - nesta aplicação foi embarcado o mesmo programa C utilizado no computador.

Os resultados obtidos com os dois modelos de referência, juntamente com os resultados alcançados nesta proposta, utilizando o melhor clock obtido para cada tamanho de grafo (Tabela 18), podem ser vistos na Tabela 24. As duas últimas colunas da tabela demonstram o ganho obtido desta solução comparado aos modelos de referência. Observa-se que, para todos os casos, a solução gerada nesta tese demonstrou desempenho superior.

Tabela 24 – Comparação com modelos de referência

Nós	**Tempo de Simulação (ns)**			**Ganho**	
	Nios II 125 MHz	**PC**	**Esta Solução**	**NIOS II**	**PC**
2.048	368.469.861	5.049.900	407.035	905,3	12,4
1.024	166.728.444	3.776.000	185.458	899,0	20,4
512	75.204.452	2.003.100	94.430	796,4	21,2
256	33.799.636	1.711.600	44.917	752,5	38,1
128	15.597.804	758.900	24.359	640,3	31,2
64	7.548.676	672.700	16.159	467,2	41,6

7.7 Comparação com outras soluções

A otimização do menor caminho em FPGAs apresenta desafios quando se trata de encontrar referências relevantes, principalmente devido à natureza especializada e em constante evolução do campo. A falta de estudos consolidados e atualizados, aliada à natureza interdisciplinar do tema, dificulta a comparação direta das publicações existentes com esta proposta. Fatores como tamanho do grafo, número de arestas por nó, seleção de FPGA e contexto do projeto influenciam a eficácia de tais comparações.

Neste contexto, um esforço de pesquisa aprofundado foi conduzido para identificar publicações que, apesar das diferenças, mais se aproximem do contexto específico de nossa proposta, com foco em encontrar literatura relevante relacionada à otimização do caminho mais curto em FPGAs. O resultado desta pesquisa pode ser visto na Tabela 25. Para viabilizar a comparação, foram realizadas simulações deste projeto com a mesma frequência de clock obtida nos artigos citados e foi utilizado um grafo sem obstáculos.

Tabela 25 – Comparação com outras soluções

Solução	Nós	Frequência	Tempo (ns)	Esta solução (ns)	Ganho
(ABDUL; ALWAN; AL-EBADI, 2012)	128	79 MHz	45.587	28.026	1,63
(FERNANDEZ *et al.*, 2008)	256	139 MHz	42.000	29.375	1,43
(CHIRILA *et al.*, 2022)	4.096	200 MHz	390.000	274.378	1,42

A pesquisa apresentada em (ABDUL; ALWAN; AL-EBADI, 2012) apresenta uma solução baseada em FPGA para calcular o caminho mais curto em redes OSPF. Tal abordagem aumenta a eficiência do algoritmo de Dijkstra, reduzindo seu tempo de processamento de $O(n^2)$ para $O(n-1)$. A nossa proposta processa grafos com até 128 nós em 28.026 ns, 1,63 vezes mais rápido do que o apresentado em (ABDUL; ALWAN; AL-EBADI, 2012).

No estudo (FERNANDEZ *et al.*, 2008), uma solução FPGA é introduzida como co-processador para um software rodando em Linux. Tal solução atinge um tamanho máximo de grafo com 256 nós para o FPGA utilizado, demonstrando um tempo de processamento de 42.000 nanossegundos, mais lento comparado ao tempo de processamento de 29.375 obtido pelo modelo proposto nesta tese. Em (FERNANDEZ *et al.*, 2008) o desempenho do projeto fica comprometido devido à necessidade de carregar todo o grafo no FPGA a cada nova análise. Esta abordagem mostra-se particularmente ineficiente quando são empregados grafos maiores, como ilustrado na Tabela 20. A sobrecarga de transferir o grafo completo para cada nova análise impacta negativamente o desempenho geral do sistema.

No estudo (CHIRILA *et al.*, 2022), uma solução heterogênea CPU-FPGA é introduzida para resolver o problema do APSP. Embora tal solução compartilhe algumas semelhanças com a abordagem apresentada aqui, como realizar a geração dos grafos fora do FPGA, ela aborda o problema mais complexo do APSP e é projetada especificamente para lidar com grafos maiores. Como demonstrado, para grafos com 4.096 nós, esta solução apresenta resultados inferiores do que os apresentados em nossa proposta.

Além disso, a solução aqui proposta demonstra desempenho competitivo em relação à recentes abordagens de planejamento de rotas focadas em baixa latência. A arquitetura ASIC baseada em RRT apresentada em (HUANG *et al.*, 2024), por exemplo, atinge tempos de processamento entre 350 μs e 960 μs a 1000 MHz. Ao operar em uma frequência de clock menor, nossa solução exibe tempos de processamento próximos. Além disso, espera-se que a incorporação de nossa abordagem em ASIC melhore ainda mais a eficiência de nossa proposta.

CAPÍTULO

8

CONCLUSÃO

Esta tese propõe contribuições ao problema de identificação do menor caminho a partir de um fonte, o qual constitui etapa necessária ao processo de deslocamento de um robô em um espaço de configuração. A solução proposta tem como objetivo apresentar um baixo tempo de resposta, viabilizando sua utilização em aplicações que exigem um processamento em tempo real. Para alcançar este alto desempenho, a solução derivada foi embarcada em um FPGA.

Uma das funcionalidades da solução apresentada é a possibilidade de mudança dos obstáculos no ambiente. O projeto prevê que obstáculos possam ser inseridos ou removidos do ambiente a todo momento e, por isso, fornece um modo simples e eficiente de realizar esta operação.

No Capítulo 4 foi descrita a implementação de um modelo de referência em nível de sistema da solução proposta. Testes com essa implementação foram realizados no Capítulo 5, onde se observou que os diferentes critérios de seleção, *IN*, *OUT* e *INOUT*, apresentam resultados semelhantes. Os testes também revelaram que é possível atingir uma melhoria de desempenho de aproximadamente 4 vezes para grafos com 64 nós e 37 vezes para grafos com 8.192 nós.

Por apresentar resultados semelhantes aos demais critérios e ter uma implementação mais simples, o critério *OUT* foi selecionado como método de seleção. Deste modo, uma proposta de arquitetura preliminar foi apresentada no Capítulo 6. A arquitetura proposta consegue realizar o processamento de vários nós em paralelo, utilizando-se das características de paralelização disponíveis no FPGA ao explorar o paralelismo possível no algoritmo de Dijkstra otimizado.

Testes também foram realizados no sentido de identificar a melhor configuração possível para o FPGA escolhido, onde se identificou que a utilização de três *Classificador de Ativos* juntamente com oito *Expansor de Nó Aprovado* apresentou o melhor desempenho.

Com o objetivo de aumentar a flexibilidade do projeto, foi inserido um bloco controlador

de acesso externo, que permite uma integração com diferentes protocolos de comunicação. Caso o protocolo de comunicação necessite ser alterado só será necessário adaptar este bloco.

O projeto incorpora uma memória dedicada para armazenar a identificação de obstáculos dos nós do grafo, resultando em uma melhoria de 120 vezes no processo de atualização do grafo em comparação aos modelos tradicionais para grafos com 1.024 nós. Os testes realizados também evidenciaram a eficiência do sistema quando comparado aos modelos de software de referência. Notavelmente, um ganho significativo de vinte vezes foi alcançado para grafos com 1.024 nós.

8.1 Próximos passos

Um fator que pode impactar o desempenho do projeto é o número de portas de leitura da memória. Na proposta atual, as memórias são equipadas com oito canais de leitura, o que restringe o acesso a apenas um ANE por vez. Consequentemente, um pipeline de solicitações de leitura teve que ser implementado. Esta limitação levou à determinação de que oito *ENA* seria a quantidade ideal com base nos testes realizados. No entanto, acredita-se que empregar um número maior de portas, especificamente múltiplo de oito neste caso, poderia produzir resultados ainda melhores. No entanto, mais testes seriam necessários para verificar esta hipótese.

Um quesito que deve ser considerado quanto à escolha da estrutura de memória das relações é que sempre é alocado espaço para o armazenamento de oito relações por nó. Porém, observou-se que determinados nós não estabelecem conexões com oito nós vizinhos. Como consequência, os recursos alocados para tais conexões inexistentes representam desperdício energético e de espaço no FPGA. Por outro lado, esta padronização desempenha um papel crucial para garantir operações consistentes. Construir uma memória com tamanhos variados pode introduzir complexidade adicional ao processo, prejudicando potencialmente quaisquer ganhos de utilização de recursos alcançados. No entanto, uma análise detalhada seria necessária para avaliar plenamente as vantagens e desvantagens dessas abordagens.

Outro aspecto que apresenta potencial para melhorias é a utilização de comparadores para identificação do critério geral de seleção. No projeto atual foi empregada uma estrutura semelhante a um acumulador comparador. Contudo, vale a pena explorar estruturas alternativas que possam alavancar o paralelismo de forma mais eficaz durante o processo de comparação. Esta abordagem poderia potencialmente produzir melhores resultados. No entanto, a realização de um estudo dedicado é essencial para investigar quais técnicas poderiam ser empregadas, avaliando também as possíveis perdas e benefícios associados a cada técnica.

Devido à complexidade dos projetos em FPGA, não foi possível a integração com uma aplicação real de movimentação de um robô neste trabalho. Deste modo, um próximo passo interessante seria desenvolver uma aplicação em PRM para FPGA e criar uma solução embarcada que pudesse controlar um robô em tempo real, validando assim todo o processo. Para um melhor

desempenho, esta solução deveria embarcar todo o projeto em um único FPGA, o que reduziria os gargalos de comunicação.

REFERÊNCIAS

ABDUL, J. J. M.; ALWAN, M. A.; AL-EBADI, M. A new hardware architecture for parallel shortest path searching processor based-on fpga technology. **Int. J. Electron. Comput. Sci. Eng**, v. 1, p. 2572–2582, 2012. Citado nas páginas 42 e 93.

ATAY, N.; BAYAZIT, B. A motion planning processor on reconfigurable hardware. In: IEEE. **Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.** [S.l.], 2006. p. 125–132. Citado nas páginas 39 e 41.

BADR, E. M.; MOUSSA, M. I. An upper bound of radio k-coloring problem and its integer linear programming model. **Wireless Networks**, Springer, v. 26, p. 4955–4964, 2020. Citado na página 42.

BERRETTINI, E.; D'ANGELO, G.; DELLING, D. Arc-flags in dynamic graphs. In: SCHLOSS DAGSTUHL-LEIBNIZ-ZENTRUM FÜR INFORMATIK. **9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)**. [S.l.], 2009. Citado na página 46.

BHASKER, J.; CHADHA, R. **Static timing analysis for nanometer designs: a practical approach**. [S.l.]: Springer Science & Business Media, 2009. Citado nas páginas 12, 108, 109 e 110.

BLACK, P. E. Dictionary of algorithms and data structures. Paul E. Black, 1998. Citado na página 35.

BULUÇ, A.; GILBERT, J. R.; BUDAK, C. Solving path problems on the gpu. **Parallel Computing**, Elsevier, v. 36, n. 5-6, p. 241–253, 2010. Citado na página 42.

CABODI, G.; CAMURATI, P.; GARBO, A.; GIORELLI, M.; QUER, S.; SAVARESE, F. A smart many-core implementation of a motion planning framework along a reference path for autonomous cars. **Electronics**, MDPI, v. 8, n. 2, p. 177, 2019. Citado nas páginas 38 e 41.

CHI, Y.; GUO, L.; CONG, J. Accelerating sssp for power-law graphs. p. 190–200, 2022. Citado na página 52.

CHIRILA, M.; D'ALBERTO, P.; TING, H.-Y.; VEIDENBAUM, A.; NICOLAU, A. A heterogeneous solution to the all-pairs shortest path problem using fpgas. In: IEEE. **2022 23rd International Symposium on Quality Electronic Design (ISQED)**. [S.l.], 2022. p. 108–113. Citado nas páginas 42 e 93.

CONG, J.; FANG, Z.; LO, M.; WANG, H.; XU, J.; ZHANG, S. Understanding performance differences of fpgas and gpus. In: IEEE. **2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.], 2018. p. 93–96. Citado na página 51.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to algorithms**. [S.l.]: MIT press, 2009. Citado nas páginas 10, 27 e 28.

CRAUSER, A.; MEHLHORN, K.; MEYER, U.; SANDERS, P. A parallelization of dijkstra's shortest path algorithm. In: SPRINGER. **International Symposium on Mathematical Foundations of Computer Science**. [S.l.], 1998. Citado nas páginas 47, 49, 52, 53, 54 e 66.

DEO, N. **Graph theory with applications to engineering and computer science**. [S.l.]: Courier Dover Publications, 2017. Citado nas páginas 10, 25, 26 e 30.

DIJKSTRA, E. W. *et al.* A note on two problems in connexion with graphs. **Numerische mathematik**, v. 1, n. 1, 1959. Citado na página 31.

DRISCOLL, J. R.; GABOW, H. N.; SHRAIRMAN, R.; TARJAN, R. E. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. **Communications of the ACM**, ACM New York, NY, USA, v. 31, n. 11, 1988. Citado na página 48.

EDMONDS, N.; BREUER, A.; GREGOR, D. P.; LUMSDAINE, A. Single-source shortest paths with the parallel boost graph library. In: **The Shortest Path Problem**. [S.l.: s.n.], 2006. Citado nas páginas 48, 49 e 52.

FERNANDEZ, I.; CASTILLO, J.; PEDRAZA, C.; SANCHEZ, C.; MARTINEZ, J. I. Parallel implementation of the shortest path algorithm on fpga. In: IEEE. **2008 4th Southern Conference on Programmable Logic**. [S.l.], 2008. p. 245–248. Citado na página 93.

FRANCIS, A.; FAUST, A.; CHIANG, H.-T. L.; HSU, J.; KEW, J. C.; FISER, M.; LEE, T.-W. E. Long-range indoor navigation with prm-rl. **IEEE Transactions on Robotics**, IEEE, v. 36, n. 4, p. 1115–1134, 2020. Citado na página 40.

GOLDBERG, A. V. Point-to-point shortest path algorithms with preprocessing. In: SPRINGER. **International Conference on Current Trends in Theory and Practice of Computer Science**. [S.l.], 2007. p. 88–102. Citado na página 46.

GUO, L.; LAU, J.; RUAN, Z.; WEI, P.; CONG, J. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu. In: **2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)**. [S.l.: s.n.], 2019. p. 127–135. ISSN 2576-2621. Citado na página 51.

HARISH, P.; NARAYANAN, P. J. Accelerating large graph algorithms on the gpu using cuda. In: SPRINGER. **International conference on high-performance computing**. [S.l.], 2007. p. 197–208. Citado na página 42.

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**, IEEE, v. 4, n. 2, 1968. Citado na página 34.

HORTELANO, J. L.; TRENTIN, V.; ARTUÑEDO, A.; VILLAGRA, J. Gpu-accelerated interaction-aware motion prediction. **Electronics**, MDPI, v. 12, n. 18, p. 3751, 2023. Citado nas páginas 39 e 41.

HUANG, L.; GONG, Y.; SUI, Y.; ZANG, X.; YUAN, B. Moped: Efficient motion planning engine with flexible dimension support. In: IEEE. **2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)**. [S.l.], 2024. p. 483–497. Citado na página 93.

JASIKA, N.; ALISPAHIC, N.; ELMA, A.; ILVANA, K.; ELMA, L.; NOSOVIC, N. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In: IEEE. **2012 proceedings of the 35th international convention MIPRO**. [S.l.], 2012. p. 1811–1815. Citado na página 42.

KAVRAKI PETR SVESTKA, J.-C. L. L. E.; OVERMARS, M. H. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. **IEEE Transactions on Robotics and Automation**, v. 12, 1996. Citado na página 30.

KLANCAR, G.; ZDESAR, A.; BLAZIC, S.; SKRJANC, I. **Wheeled mobile robotics: from fundamentals towards autonomous systems**. [S.l.]: Butterworth-Heinemann, 2017. Citado nas páginas 10, 24, 25, 29, 30, 31, 35, 36 e 37.

LAVALLE, S. M. **Planning algorithms**. [S.l.]: Cambridge university press, 2006. Citado nas páginas 10, 29 e 30.

LEE, K. D.; HUBBARD, S. **Data Structures and Algorithms with Python**. [S.l.]: Springer, 2015. Citado na página 33.

LEI, G.; DOU, Y.; LI, R.; XIA, F. An fpga implementation for solving the large single-source-shortest-path problem. **IEEE Transactions on Circuits and Systems II: Express Briefs**, IEEE, v. 63, n. 5, 2015. Citado na página 49.

LIU, J.; XIAO, G.; WU, F.; LIAO, X.; LI, K. Aapp: An accelerative and adaptive path planner for robots on gpu. **IEEE Transactions on Computers**, IEEE, 2023. Citado nas páginas 38 e 41.

LIU, S.; WAN, Z.; YU, B.; WANG, Y. Planning on fpgas. In: **Robotic Computing on FPGAs**. [S.l.]: Springer, 2021. p. 91–108. Citado na página 38.

MEHLHORN, K. **Data structures and algorithms 2: graph algorithms and NP-completeness**. [S.l.]: Springer Science & Business Media, 2012. v. 2. Citado nas páginas 26, 27 e 28.

MEHTA, A. B. Uvm (universal verification methodology). **ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies**, Springer, p. 17–64, 2018. Citado na página 106.

MURRAY, S.; FLOYD-JONES, W.; QI, Y.; KONIDARIS, G.; SORIN, D. J. The microarchitecture of a real-time robot motion planning accelerator. In: IEEE. **2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)**. [S.l.], 2016. Citado nas páginas 10, 14, 39, 40 e 41.

MURRAY, S.; FLOYD-JONES, W.; QI, Y.; SORIN, D. J.; KONIDARIS, G. Robot motion planning on a chip. In: **Robotics: Science and Systems**. [S.l.: s.n.], 2016. Citado na página 39.

PAN, J.; LAUTERBACH, C.; MANOCHA, D. g-planner: Real-time motion planning and global navigation using gpus. In: **Twenty-Fourth AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2010. Citado na página 39.

PAN, J.; MANOCHA, D. Gpu-based parallel collision detection for fast motion planning. **The International Journal of Robotics Research**, SAGE Publications Sage UK: London, England, v. 31, n. 2, p. 187–200, 2012. Citado nas páginas 39 e 41.

PAPADIMITRIOU, C. H.; STEIGLITZ, K. **Combinatorial optimization: algorithms and complexity**. [S.l.]: Courier Corporation, 1998. Citado na página 26.

PEDRONI, V. A. **Circuit design with VHDL**. [S.l.]: MIT press, 2020. Citado na página 105.

PRASAD, A.; KRISHNAMURTHY, S. K.; KIM, Y. Acceleration of dijkstra's algorithm on multi-core processors. In: IEEE. **2018 International Conference on Electronics, Information, and Communication (ICEIC)**. [S.l.], 2018. p. 1–5. Citado nas páginas 42 e 52.

ROBOTICS, P. C. Vision and control: fundamental algorithms in matlab. **Springer**, p. 251–262, 2011. Citado nas páginas 28 e 29.

SAKAMOTO, T.; HARADA, K.; WAN, W. Real-time planning robotic palletizing tasks using reusable roadmaps. **Journal of Robotics, Networking and Artificial Life**, Atlantis Press, v. 6, n. 4, p. 240–245, 2020. Citado na página 40.

SCHÜTZ, B. Partition-based speed-up of dijkstra's algorithm. Citeseer, 2005. Citado nas páginas 10, 43, 45, 46 e 52.

SHEN, Z.; WAN, Z.; GU, Y.; SUN, Y. Many sequential iterative algorithms can be parallel and (nearly) work-efficient. p. 273–286, 2022. Citado na página 52.

SIEGWART, R.; NOURBAKHSH, I. R.; SCARAMUZZA, D. **Introduction to autonomous mobile robots**. [S.l.]: MIT press, 2011. Citado na página 35.

STALLINGS, W. **Data and computer communications**. [S.l.]: Pearson Education India, 2007. Citado nas páginas 10, 14, 26, 31 e 34.

TARAATE, V. **SystemVerilog for Hardware Description: RTL Design and Verification**. [S.l.]: Springer Nature, 2020. Citado na página 105.

____. **Digital logic design using verilog**. [S.l.]: Springer, 2022. Citado na página 105.

THOUTI, K.; SATHE, S. Performance analysis of single source shortest path algorithm over multiple gpus in a network of workstations using opencl and mpi. **International Journal of Computer Applications**, Citeseer, v. 975, p. 8887, 2013. Citado na página 42.

TOMMISKA, M.; SKYTTÄ, J. Dijkstra's shortest path routing algorithm in reconfigurable hardware. In: SPRINGER. **International Conference on Field Programmable Logic and Applications**. [S.l.], 2001. p. 653–657. Citado na página 42.

VAIRA, G.; KURASOVA, O. Parallel bidirectional dijkstra's shortest path algorithm. **Databases and Information Systems VI, Frontiers in Artificial Intelligence and Applications**, v. 224, 2011. Citado nas páginas 14, 46, 47 e 52.

WEISS, M. A. **Data structures & algorithm analysis in C++**. [S.l.]: Pearson Education, 2012. Citado nas páginas 26, 27 e 31.

APÊNDICE

A

TRABALHO PUBLICADO

Os resultados obtidos nesta tese foram relatados em artigo publicado na revista Electronics (ISSN: 2079-9797) – estrato Qualis/CAPES A2 – em 2 de Junho de 2024.

Título

Analysis and Construction of Hardware Accelerators for Calculating the Shortest Path in Real-Time Robot Route Planning

Autores

Linton Thiago Costa Esteves, Wagner Luiz Alvez de Oliveira e Paulo César Machado de Abreu Farias

Abstract

This study introduces an optimization approach for calculating the shortest path in mobile robot route planning. The proposed solution targets real-time processing requirements by offering a high-performance alternative. This is achieved by embedding in the dedicated hardware an architecture which emphasizes parallelism. Through improvements in parallel exploration techniques, our solution aims to present not only a boost in performance but also a dynamic adaptation to graph changes, accommodating randomly occurring edge insertions or deletions as environmental conditions fluctuate. We present the developed architecture alongside its results. Our method efficiently updates obstacle matrices, resulting in a remarkable 120-fold improvement for 1024-node graphs. When utilizing a cost-effective device like the Cyclone IV E, it achieves approximately 12 times the performance of software applications.

Palavras chave

Dijkstra; FPGA; PRM; *Robotics*; *Shortest path*

APÊNDICE

B

FLUXO DE PROJETO DIGITAL

O fluxo de projeto digital está diretamente vinculado ao tipo de tecnologia que será utilizada na solução proposta, havendo diferentes soluções para uma mesma aplicação, a depender do ramo tecnológico empregado. Dentre as tecnologias existentes há duas que são mais utilizadas: ASIC e FPGA. Esses dois modelos conseguem atender a grande maioria dos cenários existentes. No entanto, é preciso identificar inicialmente para qual tipo de aplicação são mais indicados.

O ASIC possui duas características principais que são: versatilidade (antes do processo de fabricação) e alto desempenho. Não existe de fato uma limitação na definição de uma arquitetura para um ASIC. Não é necessário se preocupar com limitações de componentes como memórias, somadores ou multiplicadores, uma vez que não há um hardware pré-definido como ocorre com o FPGA. O que existe é uma limitação no tamanho final do chip por motivos comerciais, pois os fabricantes de ASIC normalmente cobram por *waffer* produzido. Quanto mais chips forem produzidos com um mesmo *waffer*, menor será o custo de cada chip. Seguindo essa linha, é interessante para um projeto ASIC sempre se preocupar com a área final do chip para que o mesmo possa ter viabilidade comercial. Dessa forma, projetos para esse tipo de aplicação devem ser otimizados em área.

Um fator que deve ser destacado também é o custo de todo o processo de produção de um ASIC, relativamente maior quando comparado a um FPGA. Grande parte desse custo está nas etapas iniciais do processo até a criação das máscaras a serem utilizadas na fabricação do chip, o que faz com que o custo por unidade diminua com o aumento da produção. Geralmente são utilizados para aplicações específicas que terão demanda volumosa, tornando possível assim fracionar os custos envolvidos.

Por outro lado, muitas etapas existentes no processo de fabricação de um ASIC não estão presentes no fluxo de projeto para FPGA, o que gera uma redução significativa na quantidade de capital e no tempo de projeto necessários para a obtenção de um produto. Como o FPGA já é um hardware com arquitetura definida, o projetista deve observar com cuidado os recursos que

deverão ser utilizados no projeto em questão e então direcionar a escolha do FPGA que possa suportar a aplicação, tanto na área quanto no tempo de processamento. Atualmente, existem FPGAs que apresentam velocidade de operação muito próxima da atingida por alguns ASICs, não sendo esse um fator determinante para a escolha de qual modelo seguir.

O fluxo de projeto em FPGA pode ser dividido em três etapas principais: sistema, lógica e física (Figura 42). Nas seções a seguir serão apresentadas as etapas necessárias ao desenvolvimento de projeto para FPGA.

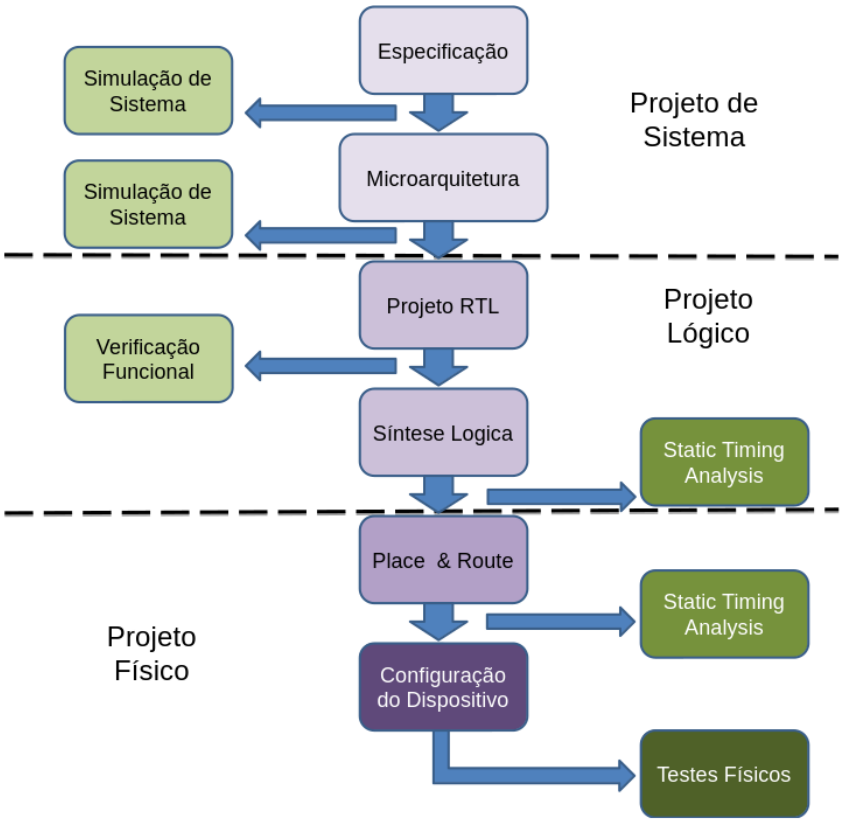


Figura 42 – Fluxo do projeto em FPGA.

B.1 Projeto de Sistema

A fase de projeto em nível de sistema engloba a etapa de prospecção dos requisitos e funcionalidades do projeto, bem como a elaboração de uma solução que atenda aos requisitos levantados.

Nessa etapa, a partir dos requisitos levantados, são identificados os componentes que irão compor a solução como as interfaces que serão utilizadas, consumo máximo permitido, tempo de resposta mínimo necessário, área ocupada e a definição da arquitetura. Essa fase pode ser dividida em duas etapas: especificação e microarquitetura.

B.1.1 Especificação

No fluxo de projeto de um FPGA, após o levantamento de requisitos o primeiro passo a ser realizado é a especificação do sistema.

A especificação pode ser descrita através de diversos tipos de documentos como textos descritivos, normas técnicas ou componentes de software. Esses documentos devem informar os requisitos funcionais e não funcionais do projeto que devem ser atendidos como, velocidade, potência máxima consumida, interfaces de entrada e saída, capacidade de armazenamento, algoritmos a serem utilizados, dentre outros. Dessa forma, a especificação define qual a funcionalidade do projeto e quais são os requisitos para atender tal funcionalidade. Essa etapa deve ser realizada de tal forma que sirva como referência para a confecção do projeto.

A partir das especificações é criado um modelo no nível do sistema do projeto. Geralmente se utiliza uma linguagem de alto nível, que servirá como referência para o processo de verificação a ser executado posteriormente. Esse modelo permite a realização de simulações já nas etapas iniciais do projeto, permitindo assim a realização de uma validação mais eficiente da especificação definida.

B.1.2 Microarquitetura

A partir da especificação desenvolvida, a microarquitetura surge como a etapa responsável por desenvolver uma arquitetura que possibilite a concepção do projeto em hardware.

Sua função é otimizar o desenvolvimento do que foi prospectado durante as etapas de especificação, de modo que se consiga atingir os requisitos do sistema. Como exemplo, o projeto de um filtro pode possuir várias arquiteturas diferentes para um mesmo algoritmo de filtragem: pode-se utilizar uma solução em pipeline para se otimizar o tempo de processamento; memórias auxiliares, para redução de área; entre outras formas.

Além disso, são definidos pontos chaves como quantidade de bits a ser utilizada, barramentos de comunicação, tamanho das memórias, latência de cada operação, entre outros pontos que são necessários ao desenvolvimento dos blocos. Essas informações são então condicionadas em um documento de arquitetura, servindo como referência para as outras etapas do projeto.

Em alguns casos, o modelo do sistema em alto nível, desenvolvido durante a especificação, é adaptado e são inseridas características antes desconhecidas do sistema, como quantidade de bits e consequente método de quantização entre operações, latência entre as interfaces dos blocos, dentre outros. A partir desse novo modelo, são realizadas novas verificações almejando identificar eventuais falhas.

B.2 Projeto Lógico

O projeto lógico (ver Figura 43) envolve a etapa de tradução da arquitetura desenvolvida para uma linguagem de descrição de hardware (HDL, do inglês *Hardware Description Language*) e, posteriormente, tradução da lógica resultante para portas lógicas. O HDL gerado deve ser verificado através de uma verificação funcional para garantir o correto funcionamento do sistema.

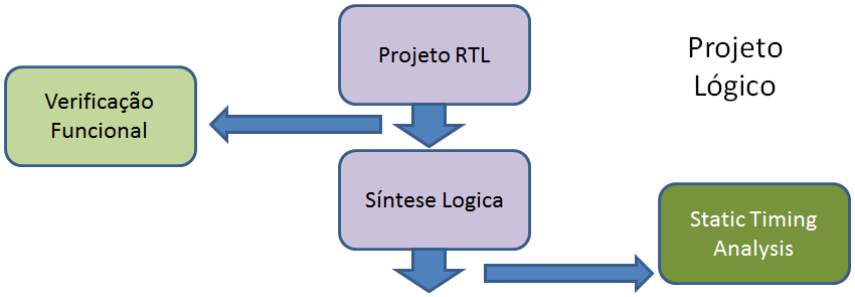


Figura 43 – Projeto lógico.

B.2.1 Projeto RTL

O projeto RTL consiste em representar a arquitetura desenvolvida em uma linguagem de descrição de hardware, correspondendo a uma descrição baseada em circuitos combinacionais intercalados com circuitos sequenciais, os quais são ativados periodicamente por meio de um ou mais sinais de sincronismo (chamados de clock).

Uma das características desse tipo de linguagem é conseguir produzir *hardwares* com características diferentes, como lógicas sequenciais ou combinacionais, *resets* síncronos e assíncronos, dentre outras.

Dentre as diversas linguagens existentes, as mais utilizadas são o VHDL (*VHSIC Hardware Description Language*)(PEDRONI, 2020), SystemVerilog (TARAATE, 2020) e Verilog (TARAATE, 2022).

B.2.2 Síntese Lógica

A síntese lógica é a etapa responsável por traduzir o *hardware* gerado no RTL para um baixo nível de abstração, onde o projeto passa a ser descrito através de componentes mais simples como, portas lógicas, flip-flops e fios.

Devido a complexidade dos projetos atuais, essa etapa é realizada por ferramentas específicas que conseguem analisar o RTL produzido e extrair a representação equivalente. A partir de parâmetros de configuração, é possível indicar à ferramenta diferentes focos de otimização da síntese como área, consumo e velocidade de processamento. Um dos modos de indicar ao projeto algumas restrições relacionadas à temporização é através de um arquivo *Synopsys Design Constraints* (SDC) que deve ser inserido nessa etapa.

B.2.3 Verificação Funcional

A etapa de validação do RTL é realizada pela verificação funcional. Atualmente, existem diversas metodologias de verificação diferentes, sendo a Metodologia de Verificação Universal (do inglês *Universal Verification Methodology*) (UVM) uma das mais populares (MEHTA, 2018).

Com o objetivo de validar em simulação o correto funcionamento do projeto, é construído um ambiente de verificação. Nesse projeto, foram utilizados conceitos da metodologia UVM, onde foram incluídos os seguintes componentes: *scoreboard*, monitor, gerador de sequência, *driver* e cenários (Figura 44).

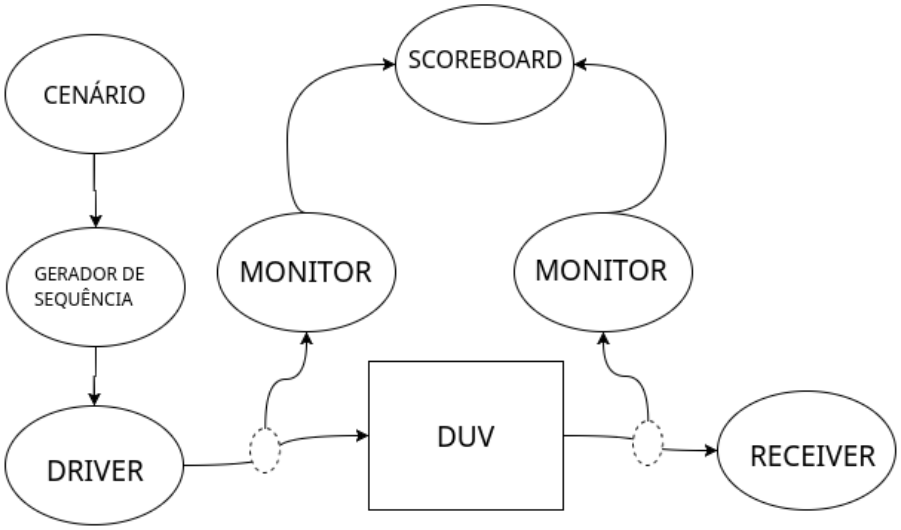


Figura 44 – Ambiente de verificação.

Compostos por vetores de entrada que devem ser aplicados ao *Design Under Verification* (DUV) e das saídas esperadas, cada cenário é responsável por fornecer as informações necessárias para a realização dos testes.

As interfaces de entrada e saída entre o DUV e o ambiente de verificação é realizada pelos *drivers*. Através deles, o gerador de sequência consegue repassar os estímulos de entrada para o DUV.

O monitor opera como uma unidade passiva, com a função de coletar as informações do barramento e repassá-las para o *scoreboard*.

Por fim, a função de verificar a correta operação do projeto é realizada pelo *scoreboard*. Esse componente tem a função de relacionar os valores de entrada com as saídas esperadas, identificando a ocorrência de comportamentos inesperados.

B.3 Projeto Físico

O projeto físico (Figura 45) transforma as portas lógicas em componentes físicos presentes no FPGA. Em projetos para FPGA, essa etapa pode ser dividida em duas: *Place and Route* e *Static Timing Analysis* (STA).

Figura 45 – Projeto físico.

Como resultado do projeto físico é gerado um *bitstream* que permite configurar o FPGA com o projeto construído. Após a configuração, são realizados testes para validar o projeto em um ambiente real.

B.3.1 Place and Route

Como resultado da síntese lógica se obtém uma *netlist* que descreve o projeto através de portas lógicas e fios. No entanto, apesar de já ser possível identificar quais componentes deverão ser utilizados, ainda não existe uma alocação física dos mesmos.

A etapa de transformar a lógica resultante da *netlist* em componentes físicos específicos no FPGA é chamada de *place* and *route*. Dessa forma, os elementos básicos (como portas lógicas) são alocados para componentes físicos através do *placement*, enquanto os fios que conectam os diversos componentes do projeto são ligados através do *route* (ou roteamento).

A qualidade do *placement* afeta diretamente o desempenho do projeto, uma vez que uma distribuição ruim das células lógicas pode dificultar a etapa de roteamento. A depender do resultado do *placement*, o tamanho da trilha que conecta dois componentes pode aumentar, resultando em um aumento do atraso na propagação dos sinais, o que poderá impactar na redução de frequência de clock do domínio associado. Além disso, é possível que a ferramenta de roteamento não consiga encontrar uma rota que conecte os componentes, sendo necessário, nesse caso, a realização de um novo *placement*.

B.3.2 Static Timing Analysis

Após a etapa de *place and route*, como os componentes já estão alocados fisicamente, é possível ter uma estimativa precisa da temporização do projeto. A etapa responsável por verificar

as restrições de tempo do projeto é denominada *Static Timing Analysis*(STA).

Com o auxílio de arquivos contendo informações de temporização da tecnologia utilizada, a STA consegue calcular os atrasos para todas as trilhas e portas do projeto, permitindo assim verificar se o mesmo atende às restrições de tempo do projeto. Uma das medições realizadas durante essa etapa é o cálculo do tempo de *slack*.

O *slack* é calculado como a diferença entre o tempo de propagação requerido de um sinal (*RequiredTime*) e o tempo real (*ArrivalTime*) (Equação B.1). Este cálculo pode ser realizado para diferentes componentes: registrador para registrador (Figura 46); entrada para registrador; e registrador para saída. Nesse caso, podem ser realizadas análises de *slack*, de *setup time* e de *hold time*.

$$Slack = RequiredTime - ArrivalTime \tag{B.1}$$

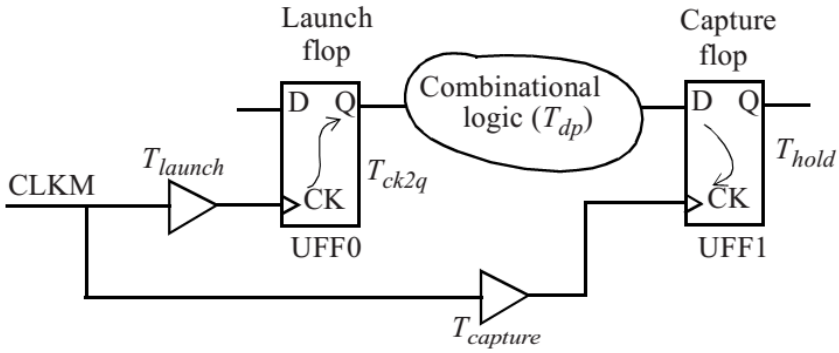


Figura 46 – Caminho registrador para registrador (Extraído de Bhasker e Chadha (2009)).

Para o *RequiredTime* da análise do *slack* de *setup*, considerando que o período do *clock* é um valor conhecido, é necessário encontrar apenas o valor de *setup time* do registrador destino (Equação B.2 e Figura 47).

$$RequiredTime_{Setup} = ClockPeriod - SetupTime \tag{B.2}$$

O tempo de *setup* representa a quantidade de tempo que a informação que se deseja armazenar em um registrador deve estar estável antes da borda ativa do *clock* (ver Figura 48) (BHASKER; CHADHA, 2009). Esse valor pode ser encontrado nas bibliotecas da tecnologia utilizada.

O *ArrivalTime* na análise de *setup* indica o tempo necessário para o dado se estabilizar antes de ser registrado. Seu cálculo é realizado de acordo com a Equação B.3. O *ClockToQ* representa o tempo necessário para o sinal ser registrado na borda ativa do *clock* anterior e ser disponibilizado na saída do registrador. O *CombDelay* indica os atrasos gerados devido à lógica combinacional contida entre o registrador anterior(*launch*) e o seguinte(*capture*).

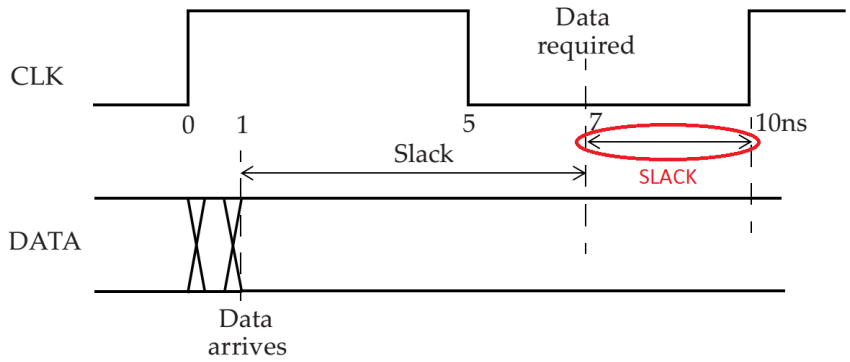


Figura 47 – Análise de tempo de *slack* para *setup time* (Adaptada de Bhasker e Chadha (2009)).

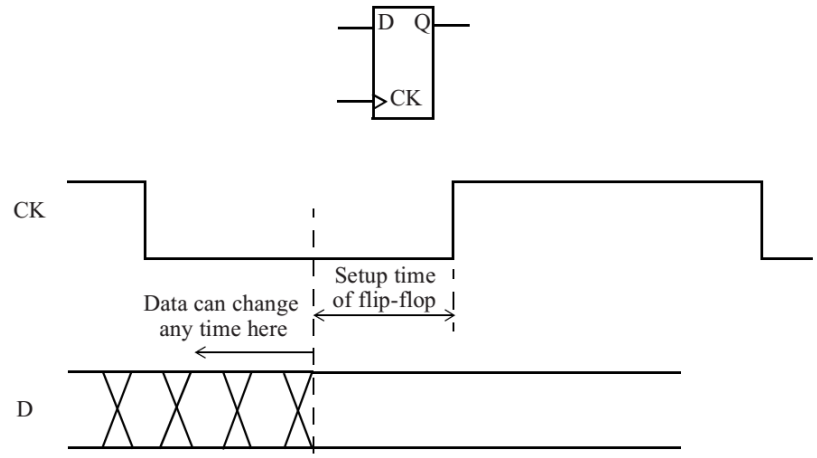


Figura 48 – *Setup time* (Extraído de Bhasker e Chadha (2009)).

$$ArrivalTime_{Setup} = ClockToQ + CombDelay \tag{B.3}$$

O *RequiredTime* para a análise do *hold time* pode ser calculado de acordo com a equação B.4, sendo $T_{Capture}$ o atraso na árvore de *clock* para o registrador de recepção e o tempo de *hold time*, a quantidade de tempo que o dado deve ser mantido estável após a ocorrência da borda de ativação do sinal de clock (borda de subida ou de descida, conforme o projeto), para que seja possível armazená-lo em um registrador sem erros (ver Figura 49) (BHASKER; CHADHA, 2009). Por fim, o *ArrivalTime* para análise de *hold* é igual ao *ArrivalTime* da análise *setup time*.

$$RequiredTime_{Hold} = T_{Capture} + HoldTime \tag{B.4}$$

Tanto o *slack* de *setup time* quanto o de *hold time* devem apresentar valor positivo. O valor final do *slack* de *hold* é igual ao valor negado do encontrado através da equação B.1.

Como problemas de violação do *slack* de *setup time* e *hold time* podem causar comportamentos inesperados no projeto, é necessário utilizar a STA para identificá-los. Dessa forma, a STA opera interativamente com o *place* e *route* ao informar regiões do projeto que ainda não

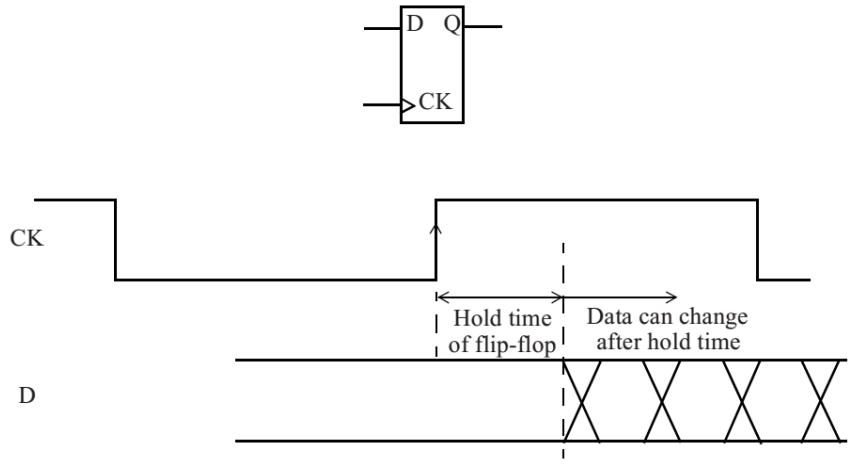


Figura 49 – Análise de tempo de *hold time* (Extraído de Bhasker e Chadha (2009)).

atingiram as restrições de tempo e que necessitam ser analisadas.

Em alguns casos não é possível atender às restrições de tempo do projeto. Nessas situações algumas soluções possíveis são: reduzir a frequência de operação do sistema; inserir pipeline, com a quebra de lógicas combinacionais grandes; voltar no fluxo para as etapas de especificação e microarquitetura.