Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Departamento de Engenharia Elétrica

Gabriel Souza Barbosa

Planejamento de Rotas em Ambientes Dinâmicos: Uma comparação entre Algoritmos na RoboCup SSL

São Carlos
2026

Gabriel Souza Barbosa

Planejamento de Rotas em Ambientes Dinâmicos: Uma comparação entre Algoritmos na RoboCup SSL

Trabalho de Conclusão de Curso apresentado à Universidade Federal de São Carlos como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletricista.

Orientadora: Profa. Dra. Tatiana F. P. A. T. Pazelli

Universidade Federal de São Carlos
Centro de Ciências Exatas e de Tecnologia
Departamento de Engenharia Elétrica

São Carlos
2026

Barbosa, Gabriel Souza

Planejamento de rotas em ambientes dinâmicos: Uma comparação entre algoritmos na RoboCup SSL / Gabriel Souza Barbosa -- 2026.
96f.

TCC (Graduação) - Universidade Federal de São Carlos, campus São Carlos, São Carlos
Orientador (a): Tatiana de Figueiredo Pereira Alves Taveira Pazelli
Banca Examinadora: Andre Carmona Hernandes, Rafael Guedes Lang
Bibliografia

1. Robótica autônoma. 2. Planejamento de Rotas. 3. RoboCup SSL. I. Barbosa, Gabriel Souza. II. Título.

Ficha catalográfica desenvolvida pela Secretaria Geral de Informática (SIn)

DADOS FORNECIDOS PELO AUTOR

Bibliotecário responsável: Arildo Martins - CRB/8 7180

FUNDAÇÃO UNIVERSIDADE FEDERAL DE SÃO CARLOS
COORDENAÇÃO DO CURSO DE ENGENHARIA ELÉTRICA (CCEE)
Rod. Washington Luís km 235 - SP-310, s/n - Bairro Monjolinho, São Carlos/SP, CEP 13565-905
Telefone: (16) 33519701 - http://www.ufscar.br

DP-TCC-FA nº 7/2026/CCEE/CCET

Graduação: Defesa Pública de Trabalho de Conclusão de Curso

Folha Aprovação

FOLHA DE APROVAÇÃO

GABRIEL SOUZA BARBOSA

PLANEJAMENTO DE ROTAS EM AMBIENTES DINÂMICOS: UMA COMPARAÇÃO ENTRE ALGORITMOS NA ROBOCUP SSL

Trabalho de Conclusão de Curso

Universidade Federal de São Carlos – Campus São Carlos

São Carlos, 07 de abril de 2026

ASSINATURAS E CIÊNCIAS

Cargo/Função	Nome Completo
Orientador	Tatiana de Figueiredo Pereira Alves Taveira Pazelli
Membro da Banca 1	André Carmona Hernandes
Membro da Banca 2	Rafael Guedes Lang

Documento assinado eletronicamente por **Tatiana de Figueiredo Pereira Alves Taveira Pazelli, Professor(a)**, em 07/04/2026, às 17:06, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Osmar Ogashawara**, **Professor(a)**, em 08/04/2026, às 09:03, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Andre Carmona Hernandes**, **Professor(a)**, em 23/05/2026, às 07:28, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://sei.ufscar.br/autenticacao, informando o código verificador **2234866** e o código CRC **C8552DAB**.

Referência: Caso responda a este documento, indicar expressamente o Processo nº 23112.005310/2026-04

SEI nº 2234866

Modelo de Documento: Grad: Defesa TCC: Folha Aprovação, versão de 02/Agosto/2019

Documento assinado digitalmente
RAFAEL GUEDES LANG
Data: 24/05/2026 11:28:06-0300
Verifique em https://validar.iti.gov.br

Resumo

Souza Barbosa, Gabriel, **Planejamento de Rotas em Ambientes Dinâmicos: Uma comparação entre Algoritmos na RoboCup SSL**. Trabalho de Conclusão de Curso (Graduação em Engenharia Elétrica). Departamento de Engenharia Elétrica, Centro de Ciências Exatas e de Tecnologia, Universidade Federal de São Carlos. 95 p. São Carlos, 2026.

Este trabalho apresenta uma análise comparativa entre algoritmos clássicos e incrementais de planejamento de rotas aplicados à Small Size League (SSL) da RoboCup, um ambiente caracterizado por alta dinâmica, múltiplos robôs e frequentes interações com contato entre os robôs. Nesse contexto, a escolha do planejador de rotas influencia diretamente a segurança da navegação, a eficiência das trajetórias e o desempenho coletivo da equipe, o que justifica a necessidade de uma avaliação estruturada entre diferentes abordagens. Foram implementados e comparados os algoritmos *Dijkstra*, A*, *Lifelong Planning A** (LPA*) e D* Lite, todos operando sobre um mesmo Grafo de Visibilidade, garantindo uma base comum para análise. Os experimentos reproduzem situações típicas de jogo com obstáculos estáticos e dinâmicos, permitindo observar como cada método responde a diferentes níveis de exigência de situações típicas de jogo. A avaliação combinou análises qualitativas visuais e métricas quantitativas de desempenho, além de simulações de partidas completas, evidenciando diferenças claras entre planejadores clássicos e incrementais, especialmente quanto à adaptação a mudanças no cenário e à ocorrência de colisões. Os resultados oferecem uma compreensão consistente das vantagens e limitações de cada abordagem e fornecem subsídios práticos para a escolha de estratégias de planejamento de rotas em sistemas de robótica móvel submetidos a situações dinâmicas.

Palavras-Chave: planejamento de rotas; robótica móvel; navegação autônoma; SSL RoboCup; algoritmos incrementais.

Abstract

Souza Barbosa, Gabriel, **Path Planning in Dynamic Environments: A Comparison of Algorithms in the RoboCup SSL**. Final year thesis (Bachelor degree in Electrical Engineering). Department of Electrical Engineering, Exact and Technology Sciences Center, Federal University of São Carlos. 95 p. São Carlos, 2026.

This work presents a comparative analysis between classical and incremental path planning algorithms applied to the RoboCup Small Size League (SSL), an environment characterized by high dynamics, multiple robots, and frequent physical interactions among them. In this context, the choice of the path planner directly influences navigation safety, trajectory efficiency, and the overall team performance, which justifies the need for a structured evaluation of different approaches. The algorithms Dijkstra, A*, Lifelong Planning A (LPA*), and D* Lite were implemented and compared, all operating on the same Visibility Graph representation to ensure a common basis for analysis. The experiments reproduce typical game situations involving both static and dynamic obstacles, allowing the observation of how each method responds to varying levels of environmental demands. The evaluation combined qualitative visual analyses and quantitative performance metrics, along with full-match simulations, highlighting clear differences between classical and incremental planners, particularly regarding adaptation to environmental changes and collision occurrence. The results provide a consistent understanding of the advantages and limitations of each approach and offer practical support for selecting path planning strategies in mobile robotic systems operating under dynamic conditions.

Keywords: path planning; mobile robotics; autonomous navigation; RoboCup SSL; incremental algorithms.

Lista de Figuras

Lista de Tabelas

Lista de Siglas e Abreviaturas

SSL *Small Size League*

TDP *Team Description Paper*

UFSCar Universidade Federal de São Carlos

VG *Visibility Graph*

LPA* *Lifelong Planning A**

JPS *Jump Point Search*

AGVs *Automated Guided Vehicle*

UAVs *Unmanned Aerial Vehicle*

RRT *Rapidly-exploring Random Tree*

ERRT *Exploration Rapidly-exploring Random Tree*

TDP *Team Description Paper*

PSO *Particle Swarm Optimization*

CBR *Competição Brasileira de Robótica*

DC *Direct Current*

PRM *Probabilistic RoadMap*

ITA Instituto Tecnológico de Aeronáutica

EKF *Extended Kalman Filter*

BPMN Business Process Model and Notation

Lista de Publicações

Gabriel S. Barbosa, Kelen C. T. Vivaldini, Tatiana F. P. A. T. Pazelli, **Path Planning in Dynamic Environments: A Comparison of Algorithms in the RoboCup SSL**, In: 2026 Brazilian Conference on Robotics (CROS), João Pessoa, Brazil, 2026, pp. 1-6.

Sumário

1 Introdução

A robótica autônoma tem se consolidado como uma das principais áreas de pesquisa e desenvolvimento tecnológico da atualidade. O constante progresso em inteligência artificial, sensoreamento avançado e sistemas de controle tem permitido que máquinas assumam tarefas cada vez mais complexas, operando com menor dependência da intervenção humana. Essa evolução representa um passo importante na construção de sistemas capazes de interpretar o ambiente e agir de forma adaptativa, ampliando significativamente o potencial da automação em diferentes contextos (WAGA et al., 2025).

O crescimento dos sistemas autônomos está diretamente ligado à busca por soluções mais inteligentes, seguras e sustentáveis para os desafios do cotidiano. A chamada mobilidade inteligente, impulsionada por veículos autônomos, drones e robôs móveis, reflete uma tendência global de integrar tecnologia, eficiência energética e capacidade de adaptação em tempo real. Esse movimento tem criado um cenário favorável para pesquisa e inovação, oferecendo inúmeras oportunidades de estudo e desenvolvimento de novas abordagens que aproximam a robótica do convívio humano e das demandas reais da sociedade (WAGA et al., 2025).

Diante disso, o tema de planejamento de rotas em robôs autônomos é um dos tópicos centrais da robótica móvel e consiste em determinar um caminho viável que conecte um ponto inicial a um ponto final no espaço de trabalho, evitando colisões e respeitando restrições físicas e cinemáticas do robô. Essa tarefa é essencial para garantir que qualquer robô possa se deslocar de forma segura, eficiente e autônoma, especialmente em ambientes dinâmicos onde decisões precisam ser tomadas em tempo real. A complexidade desse problema cresce à medida que o número de obstáculos aumenta ou que o ambiente se torna parcialmente observável. Dessa forma, a busca por algoritmos capazes de equilibrar qualidade de trajetória, custo computacional e adaptabilidade às variações do ambiente é um dos principais desafios da área e fundamenta o desenvolvimento de novas estratégias para sistemas robóticos autônomos (SÁNCHEZ-IBÁÑEZ; PULGAR; GARCÍA-CEREZO, 2021).

O planejamento de rotas pode ser classificado em duas abordagens principais, global e local. O planejamento global tem como objetivo encontrar um caminho completo entre o ponto de partida e o destino com base em informações do ambiente que já estão disponíveis. Essa abordagem permite gerar trajetórias otimizadas e consistentes, garantindo caminhos eficientes em termos de distância ou custo total. Já o planejamento local atua em uma escala mais imediata, ajustando a trajetória em resposta a mudanças próximas, como obstáculos móveis ou variações momentâneas na rota. Na prática, muitos sistemas eficazes combinam as duas abordagens para unir a previsibilidade de um planejamento

global com a capacidade de adaptação oferecida pelo controle local (SÁNCHEZ-IBÁÑEZ; PULGAR; GARCÍA-CEREZO, 2021).

O desempenho de um planejador também depende do nível de conhecimento disponível sobre o ambiente. Em ambientes completamente estáticos e conhecidos, o robô tem acesso antecipado a todas as informações relevantes e pode traçar um caminho otimizado antes mesmo de iniciar o movimento. Já em ambientes dinâmicos, onde há elementos móveis ou condições que mudam com o tempo, o planejador precisa combinar capacidade de adaptação através de ajustes incrementais da trajetória com eficiência computacional. Para esse tipo de situação, o problema de planejamento se torna ainda mais complexo, pois nesses cenários o planejador precisa ser capaz de recalcular a rota rapidamente e garantir que o movimento seja executado sem interrupções, preservando o desempenho do robô. Esse tipo de ambiente é típico em aplicações reais, como veículos autônomos, onde decisões precisas e rápidas são essenciais para o sucesso e segurança da navegação (SÁNCHEZ-IBÁÑEZ; PULGAR; GARCÍA-CEREZO, 2021).

A representação do ambiente é uma etapa essencial para que o planejador de rotas possa operar de forma eficiente, e o grafo surge como uma estrutura prática para organizar essas informações. Ele transforma o espaço contínuo e complexo em um conjunto discreto de nós e arestas que representam posições e conexões viáveis de deslocamento, possibilitando a aplicação de algoritmos clássicos de busca. Essa modelagem permite considerar restrições geométricas e cinemáticas, como o tamanho do robô e a presença de obstáculos. O grafo também separa de forma clara as etapas de modelagem do ambiente e de cálculo de trajetória, podendo ser reutilizado em diferentes testes e algoritmos. Dessa forma, o grafo atua como a base que conecta a percepção do ambiente ao processo de tomada de decisão (KATONA; NEAMAH; KORONDI, 2024).

Existem diferentes ferramentas e estratégias para construir os grafos e cada uma delas parte de um princípio distinto sobre como representar o espaço livre. Algumas abordagens utilizam estruturas geométricas explícitas como o VG ou o Diagrama de *Voronoi* para explorar a geometria dos obstáculos e gerar arestas que privilegiam comprimento ou distanciamento seguro entre obstáculos. Outras adotam representações regulares como grades ocupacionais, fáceis de implementar e úteis quando se quer garantir cobertura completa do espaço, ainda que isso gere muitos nós. Há também técnicas que discretizam o espaço de configuração em pontos significativos ou vértices de obstáculos e depois aplicam pré-processamento e indexação espacial para acelerar testes de colisão e consultas de vizinhança. Em resumo, cada ferramenta traz um conjunto de características diferentes relacionadas a qualidade do caminho, custo de construção do grafo e custo de consulta durante a execução do planejador (LIU et al., 2024).

Uma família importante de ferramentas baseia-se em amostragem para gerar o grafo de forma estocástica ou guiada, sendo as *Probabilistic RoadMap* (PRM) e as variantes de *Rapidly-exploring Random Tree* (RRT) as mais conhecidas, projetados para contornar as

limitações das abordagens determinísticas. Esses métodos não constroem o grafo a partir de uma discretização fixa, mas sim de pontos amostrados aleatoriamente no espaço livre, conectando-os por meio de arestas válidas segundo os critérios de navegação. O PRM é mais indicado para ambientes estáticos, pois constrói uma malha de conexões reutilizável que representa o espaço de configuração. Já o RRT é mais apropriado para ambientes dinâmicos, pois cresce de forma incremental a partir do ponto inicial, explorando o espaço rapidamente em busca de soluções viáveis. Essa característica torna o RRT especialmente útil em aplicações que exigem replanejamento frequente e respostas rápidas, como navegação de drones ou robôs móveis em áreas não totalmente conhecidas (CHOUDHARY, 2023).

Além das abordagens baseadas em amostragem, o próprio VG também pode ser explorado de diferentes maneiras dependendo da forma como o ambiente é modelado. Uma variação relevante apresentada na literatura utiliza obstáculos circulares em vez de representações poligonais, ajustando o cálculo das arestas de visibilidade para considerar regiões arredondadas em torno dos objetos. Esse formato simplifica a verificação geométrica necessária para determinar se dois pontos podem ser conectados sem interferência do obstáculo e pode tornar a construção do grafo mais estável em situações em que a modelagem poligonal gera vértices muito próximos ou arestas sensíveis à discretização (MAXIMO et al., 2025). Embora essa técnica não tenha sido adotada neste trabalho, ela ilustra como o VG pode assumir diferentes configurações a depender das necessidades da aplicação e das características do ambiente.

Com relação efetivamente aos planejadores de rota, a variedade de algoritmos desenvolvidos ao longo dos anos mostra que esse é um tópico que pode ser abordado de diferentes maneiras. Alguns métodos se baseiam em buscas determinísticas em grafos, capazes de encontrar soluções ótimas quando há conhecimento completo do espaço de navegação, enquanto outros adotam estratégias reativas que permitem reagir rapidamente a mudanças inesperadas. Também existem algoritmos baseados em amostragem e técnicas inspiradas em inteligência artificial, como redes neurais e algoritmos genéticos, que buscam trajetórias viáveis em ambientes complexos ou parcialmente conhecidos. Essa diversidade reflete a evolução da área e o esforço em equilibrar qualidade de caminho, tempo de processamento e capacidade de adaptação, reforçando que a escolha do algoritmo está diretamente ligada ao tipo de ambiente e ao nível de critério crítico que está sendo levado em consideração (SÁNCHEZ-IBÁÑEZ; PULGAR; GARCÍA-CEREZO, 2021).

Além das abordagens clássicas e baseadas em amostragem, há também estratégias mais recentes que exploram a modelagem de grafos voltada para sistemas multi-robôs. Um exemplo é o planejador proposto por (LIU et al., 2024), que combina a estrutura de grafos com restrições de formação e desvio de obstáculos para permitir movimentos coordenados entre múltiplos robôs autônomos. Nesse modelo, o grafo não representa apenas o espaço livre, mas também as relações espaciais entre os robôs, garantindo que a

formação seja preservada mesmo em ambientes complexos. Essa técnica é especialmente útil em cenários colaborativos, onde os robôs precisam manter distâncias relativas fixas ou executar tarefas conjuntas, como transporte cooperativo ou cobertura de área. Além disso, o uso dessa arquitetura permite incorporar facilmente restrições adicionais, como prioridades de trajeto e zonas de segurança, tornando o planejamento mais flexível.

Durante o processo de revisão bibliográfica, foram identificados alguns trabalhos relacionados a este estudo que serviram como referência para delimitar o escopo da comparação entre algoritmos, definir métricas de desempenho e explorar oportunidades ainda pouco exploradas pela literatura. Entre os principais trabalhos relacionados, destaca-se um estudo comparativo entre VG combinado com A* e RRT, que traz evidências sobre diferenças entre qualidade do caminho e custo computacional (COSTA; TONIDANDEL, 2019), o comparativo que contrasta A*, D* e *Particle Swarm Optimization* (PSO) com foco em eficiência e robustez (OKUMUŞ; KOCAMAZ, 2018), e a proposta de melhoria do *D* Lite* voltada a ambientes dinâmicos, que apresenta ajustes heurísticos e estratégias de suavização de caminho (YU et al., 2020).

O estudo apresentado por (COSTA; TONIDANDEL, 2019) investiga o desempenho de um planejador baseado em *Dynamic Visibility Graph* combinado com A* em comparação com a família RRT. Os autores descrevem a construção do grafo a partir dos vértices relevantes do ambiente e a aplicação do A* sobre esse grafo para gerar trajetórias poligonais. Os resultados indicam que a abordagem baseada em VG tende a produzir caminhos mais curtos e com menos quebras de direção enquanto o RRT se mostra mais eficaz na exploração de regiões do espaço não facilmente capturadas pela representação por vértices. O trabalho detalha aspectos de implementação do VG que são úteis para avaliação comparativa, são discutidos critérios para seleção de vértices relevantes a serem incluídos no grafo e estratégias para reduzir verificações redundantes durante a construção das arestas. Também é apresentada uma análise qualitativa de desempenho em estreitamentos e áreas abertas que ajuda a entender como a topologia do ambiente afeta a qualidade do caminho e o tempo de cálculo (COSTA; TONIDANDEL, 2019). Este trabalho serve como referência direta para a escolha do VG como base para geração de grafos neste estudo e orienta a definição de cenários e métricas de comparação.

Já o artigo apresentado por (OKUMUŞ; KOCAMAZ, 2018) compara algoritmos clássicos de busca e uma meta-heurística no contexto de planejamento de trajetórias, colocando A* e D* em comparação com uma versão de PSO aplicada à otimização de caminhos. Os autores descrevem como cada método é formulado para o problema de navegação e discutem diferenças conceituais entre busca baseada em grafos e otimização por enxame. Essa comparação oferece uma visão interessante sobre como abordagens determinísticas e estocásticas se comportam em relação à eficiência e à robustez.

No trabalho é apresentada também uma discussão sobre aspectos práticos de implementação que influenciam o desempenho dos métodos. São abordadas questões como

a escolha de heurísticas para A*, a forma de manutenção de custos em D* e parâmetros de ajuste do PSO que afetam convergência e qualidade das soluções. A ênfase na configuração experimental e na sensibilidade dos parâmetros auxilia na compreensão da importância de efetuar uma parametrização adequada (OKUMUş; KOCAMAZ, 2018). Por fim, o artigo traz observações sobre a relação entre custo computacional e qualidade do caminho quando se compara busca exata e otimização estocástica. Os resultados e as discussões apresentam fundamentos para definir critérios de comparação entre algoritmos de planejamento e análise de desempenho em estudos comparativos.

Apesar da ampla variedade de algoritmos e técnicas propostas na literatura, é por meio de ambientes experimentais que se torna possível avaliar sua aplicabilidade prática e avaliar o desempenho real de cada abordagem. Diante disso, grandes eventos e competições científicas têm se tornado um espaço que abre as portas para a aplicação prática de conceitos teóricos, a validação de novas metodologias e a comparação entre diferentes soluções em condições controladas. Nesse sentido, a *RoboCup* representa um modelo de referência para estudos dessa natureza, pois traz desafios práticos que estimulam o desenvolvimento de soluções cada vez mais eficientes para o planejamento e controle de robôs autônomos.

A *RoboCup* nasceu como uma iniciativa científica com o objetivo de avançar pesquisas em robótica e de promover desafios práticos que acelerem o desenvolvimento de sistemas autônomos capazes de atuar em ambientes complexos. O desafio atual de formar até 2050 uma equipe de robôs humanoides capaz de competir com campeões humanos ilustra a ambição do projeto. Além do aspecto competitivo, a *RoboCup* funciona como uma plataforma global de cooperação que estimula a troca de conhecimentos e a pesquisa científica combinada com engenharia aplicada (ROBOCUP, 2025).

Dentro desse contexto, a SSL se destaca como uma liga historicamente consolidada na competição e é um ambiente de teste ideal para o estudo de robôs móveis autônomos. O formato da liga impõe desafios únicos ao exigir tomada de decisão rápida, controle preciso de movimento e coordenação entre múltiplos robôs que interagem em alta velocidade dentro de um espaço reduzido. Cada equipe precisa desenvolver soluções eficientes para planejamento de rotas, desvio de obstáculos e controle dinâmico, enfrentando condições que simulam de forma prática diversos problemas clássicos da robótica autônoma. Essas características tornam a SSL um campo de pesquisa de grande relevância para o desenvolvimento de algoritmos de planejamento, percepção e controle aplicáveis a sistemas robóticos do mundo real (ROBOCUP, 2025). Para facilitar a visualização e ilustrar a dinâmica, a Figura 1 apresenta uma foto tirada durante uma partida de SSL.

Figura 1 – Partida de Futebol de robôs na SSL

Fonte: (ROBOCUP, 2025)

Durante uma partida da SSL, todo o funcionamento do sistema depende de um fluxo de comunicação coordenado entre as câmeras, o computador que faz o processamento das imagens e envia os pacotes com as informações necessárias para cada time, a tomada de decisão do código de estratégia e o envio do sinal, comumente via rádio ou *Wi-fi*, para o controle embarcado nos robôs. Esse fluxo é o que permite que as equipes recebam informações do ambiente em tempo real, interpretem a situação de jogo e transmitam comandos precisos aos robôs em campo. A estrutura foi projetada para garantir que todos os times tenham acesso às mesmas informações de visão e às mesmas condições de comunicação, assegurando equilíbrio e padronização na competição.

O processo começa com a captura de imagens por câmeras posicionadas acima do campo, responsáveis por observar toda a área de jogo. Essas imagens são processadas pelo sistema *SSL-Vision*, que identifica a posição da bola, a orientação e a localização de todos os robôs. O sistema opera de forma centralizada e envia os dados para as equipes através de uma rede *multicast*. O fluxo de informação segue continuamente, permitindo que as equipes recebam atualizações em tempo real. O *SSL-Vision* é composto por uma sequência modular de etapas que incluem captura, segmentação, extração de regiões e conversão das coordenadas da imagem para coordenadas reais do campo. Cada pacote transmitido contém as posições detectadas, a orientação e um indicador de confiança, garantindo que cada equipe receba as mesmas medições brutas para posterior filtragem e rastreamento segundo os próprios métodos implementados (ZICKLER et al., 2009).

Em paralelo, há também a comunicação com o *RefereeBox*, que atua como o árbitro eletrônico da partida. Ele transmite comandos de jogo, como início, parada, faltas ou reinício, assegurando que todas as equipes reajam de forma sincronizada às decisões de arbitragem. O computador de estratégia de cada equipe recebe simultaneamente as informações do *SSL-Vision* e do *RefereeBox*, processa os dados e toma decisões com base na situação atual do jogo.

A camada de estratégia é responsável por interpretar o estado do jogo e definir as ações de cada robô, enquanto o planejador de rotas calcula os caminhos livres de coli-

são até os objetivos definidos. O controlador de baixo nível, por sua vez, converte as velocidades e comandos em sinais que são enviados para os robôs em campo. Esse ciclo ocorre de forma contínua, garantindo que os robôs reajam dinamicamente às mudanças do ambiente e mantenham um comportamento coordenado. A Figura 2 ilustra esse fluxo de comunicação, destacando o papel de cada componente e as interações que permitem a execução autônoma das partidas na SSL (KALISCH; PANFIL, 2015).

Figura 2 – Fluxo de comunicação entre os principais elementos da SSL

Fonte: Adaptado de (KALISCH; PANFIL, 2015)

Portanto, no caso da SSL, os robôs operam em um ambiente muito denso e dinâmico, onde decisões de trajetória devem ser calculadas em frações de segundo e devem, ao mesmo tempo, evitar colisões, minimizar comprimento e garantir que o robô chegue ao destino na orientação correta. Por isso, o planejamento de rotas não é apenas uma etapa de cálculo de caminho, mas sim um componente fundamental que influencia de maneira determinante o desempenho da equipe em campo. Além disso, o desempenho do planejador impacta outras camadas do sistema, como o controle de baixo nível e a estratégia geral da equipe. Caminhos mal planejados podem sobrecarregar o controlador com correções constantes ou comprometer o tempo de resposta.

Assim, a escolha da SSL como ambiente experimental neste estudo se deve ao fato de que ela oferece um contexto desafiador para o planejamento de rotas. A liga combina elementos de alta complexidade, como múltiplos robôs interagindo em tempo real, limitação de espaço físico e necessidade de decisões rápidas e precisas. Essa combinação cria um ambiente ideal para observar o comportamento dos algoritmos diante de situações dinâmicas e com obstáculos em constante movimento. Além disso, o uso de um simulador que permite reproduzir com fidelidade as condições de jogo, garante que os resultados obtidos tenham relevância prática tanto para o desenvolvimento científico quanto para a aplicação direta em competições reais.

Diante das características da liga, a motivação deste estudo é investigar, de forma justa e esclarecedora, o desempenho de diferentes algoritmos de planejamento de rotas em cenários de teste que reproduzem situações reais de jogo. Cada algoritmo reage de maneira distinta a ambientes dinâmicos e restritos como o da competição, e analisar esse comportamento permite construir uma base científica sólida sobre quais abordagens apresentam melhor desempenho conforme o critério mais crítico de análise adotado pelas equipes, como tempo de resposta, eficácia no desvio de obstáculos dinâmicos ou tempo mínimo de execução da trajetória.

Para tomar a decisão de propor quais algoritmos seriam implementados para servirem de base para este estudo comparativo, foi estudado o trabalho de (YU et al., 2020), que propõe uma melhoria no algoritmo *D* Lite* para aplicação em robôs móveis com ambientes dinâmicos. Os autores apresentam uma versão modificada do algoritmo com ajustes heurísticos voltados a reduzir o tempo de processamento e a melhorar a qualidade do caminho obtido em cenários com mudanças frequentes na configuração dos obstáculos. A proposta busca manter a eficiência do planejamento incremental característico do *D* Lite* ao mesmo tempo em que introduz técnicas para suavizar as trajetórias geradas e adaptá-las melhor às restrições de movimento do robô.

O trabalho discute de forma detalhada como a heurística utilizada no processo de expansão dos nós influencia a velocidade de convergência e a estabilidade do algoritmo em ambientes que mudam constantemente durante o tempo de execução. Além disso são descritas estratégias adicionais de suavização que visam reduzir variações abruptas de direção presentes nos caminhos poligonais resultantes. Essa abordagem permite que as trajetórias planejadas sejam mais adequadas ao controle de robôs reais e menos suscetíveis a oscilações durante a execução (YU et al., 2020).

Portanto, o estudo evidencia que a versão modificada do *D* Lite* apresenta ganhos significativos em cenários de navegação dinâmica quando comparada à formulação original. Esses resultados reforçam a relevância do *D* Lite* e de suas variantes como uma alternativa robusta para planejamento em ambientes sujeitos a alterações constantes e servem de base para a motivação da aplicação do algoritmo neste trabalho.

Cada equipe da categoria desenvolve a sua própria estratégia de planejamento de rotas, adaptando os algoritmos de acordo com as limitações e dinâmicas de cada um. Foi efetuada uma revisão bibliográfica dos *Team Description Paper* (TDP), que é um documento anual elaborado pelas equipes para apresentar as principais evoluções de *hardware* e *software* dos robôs, publicados nos últimos anos para entender para entender quais estratégias as equipes estão adotando. A equipe *RoboIME*, por exemplo, desenvolve seu sistema de planejamento com base em algoritmos da família RRT, bastante difundido entre os times da competição, explorando especialmente variantes como o *Exploration Rapidly-exploring Random Tree* (ERRT). A equipe justifica essa escolha pela capacidade desses algoritmos de gerar soluções viáveis em ambientes complexos e de alto dinamismo, típicos da SSL.

As versões implementadas realizam reamostragem contínua do espaço de configuração e refinamento incremental da árvore de busca, o que melhora a qualidade das trajetórias ao longo do tempo. O TDP destaca que o RRT* oferece um bom equilíbrio entre velocidade de resposta e qualidade do caminho, sendo adequado para replanejamentos frequentes e condições em que o ambiente muda durante o jogo (SILVA et al., 2023).

A *ITAndroids*, equipe do Instituto Tecnológico de Aeronáutica (ITA), inicialmente também utilizava uma versão modificada do algoritmo ERRT. No entanto, os desenvolvedores observaram limitações associadas à natureza estocástica do método, que ocasionalmente resultava em trajetórias excessivamente longas ou demoradas para serem calculadas. Após testar heurísticas para atenuar esse comportamento, a equipe optou por substituir o planejador por uma abordagem determinística baseada em A* aplicado a um VG com obstáculos modelados como círculos. Essa escolha foi influenciada por trabalhos da equipe referências da liga e buscou unir suavidade e estabilidade, características desejáveis para robôs com corpo circular, como os utilizados na SSL. O método resultante permite caminhos mais curtos e suaves, com baixo custo computacional, além de eliminar a aleatoriedade das abordagens por amostragem, o que favorece a previsibilidade e possibilita o replanejamento frequente a cada ciclo de atualização (MARANHÃO et al., 2020).

Já a equipe *TurtleRabbit* adota um planejador de rotas baseado em PRM, por conta da simplicidade de implementação com confiabilidade nos resultados. O grafo é construído a partir de amostragem aleatória de pontos válidos no espaço de navegação, conectados por arestas livres de colisão, e o caminho final é obtido com o algoritmo de *Dijkstra*. Segundo o TDP, a escolha do PRM foi motivada pela necessidade de um método de planejamento rápido e consistente, que funcionasse bem mesmo em cenários com múltiplos robôs e obstáculos. De acordo com a equipe, essa abordagem também facilita a integração com o sistema de controle, permitindo que o módulo de navegação opere de forma independente da camada de decisão, o que torna o sistema mais modular e fácil de ajustar durante o desenvolvimento (TRINH et al., 2024).

Apesar da ascensão do tema e do interesse mútuo em estratégias de planejamento de rotas para robôs autônomos, ainda existem tópicos pouco explorados pela literatura e lacunas que este trabalho busca preencher. Em particular, não foram encontrados estudos ou aplicações práticas que implementem algoritmos como LPA* e *D* Lite* no contexto específico do futebol de robôs. Outro ponto diferencial, é que esse estudo comparativo considera os algoritmos clássicos em sua formulação original e operando sobre o mesmo grafo, o que permite isolar o efeito do planejador das demais variáveis do sistema. Além disso, este estudo se destaca por apresentar uma análise detalhada e conduzida em cenários que reproduzem situações reais de jogo, possibilitando uma avaliação mais próxima das condições práticas enfrentadas em competições da SSL.

Portanto, o objetivo central deste trabalho é implementar e comparar algoritmos clás-

sicos e incrementais de planejamento de rotas a partir do mesmo grafo gerado pelo VG para múltiplos cenários de teste com situações reais e com as mesmas configurações de simulador e parâmetros do controlador de posição. Serão avaliados métodos baseados em busca exata, como *Dijkstra* e A* puro, além de métodos incrementais incluindo LPA* e *D* Lite*. A comparação focará em métricas que medem tanto qualidade geométrica quanto eficiência temporal do planejamento, tais como comprimento do caminho, tempo de cálculo da trajetória, tempo de execução da trajetória, sobreposição entre caminho planejado e executado, capacidade de desvio de obstáculos, entre outros. O objetivo é oferecer uma avaliação clara e prática que sirva de base para que sejam definidos os planejadores para uso na SSL ou em aplicações robóticas com requisitos semelhantes.

Do ponto de vista prático e científico, este trabalho oferece uma contribuição abrangente para o desenvolvimento de planejadores de rotas para robôs autônomos em ambientes dinâmicos. A pesquisa fornece uma base pronta que pode orientar decisões de implementação tanto em equipes da SSL quanto em outros contextos de robótica móvel. No contexto da equipe *Red Dragons*, da Universidade Federal de São Carlos (UFSCar), que até então utiliza o planejador baseado em VG combinado com o algoritmo de *Dijkstra*, o estudo fornece subsídios concretos para avaliar a troca do planejador e a adoção de alternativas mais promissoras, como LPA* e *D* Lite.* Além de apoiar o aprimoramento das soluções empregadas pela equipe, os resultados também oferecem um referencial prático para outros grupos e aplicações que enfrentam desafios semelhantes de navegação autônoma, como robôs de inspeção, veículos autônomos e sistemas colaborativos. Dessa forma, o trabalho contribui não apenas para o avanço técnico da equipe, mas também para o fortalecimento do conhecimento científico sobre o comportamento comparativo desses algoritmos em condições reais de operação.

Este estudo foi organizado de forma a apresentar de maneira clara e progressiva o desenvolvimento do estudo, partindo da fundamentação teórica até a análise dos resultados. A introdução, apresentada neste capítulo, reúne a contextualização geral do tema, alguns conceitos essenciais e o embasamento bibliográfico que sustenta a pesquisa, enquanto os capítulos seguintes descrevem os métodos teóricos e procedimentos adotados para a implementação dos algoritmos e para a definição dos cenários de teste. Em sequência, são apresentados e discutidos os resultados obtidos a partir das simulações, analisando o desempenho dos planejadores segundo critérios definidos previamente. Por fim, o trabalho encerra com as considerações finais, que sintetizam as conclusões alcançadas e apontam possíveis caminhos para trabalhos futuros.

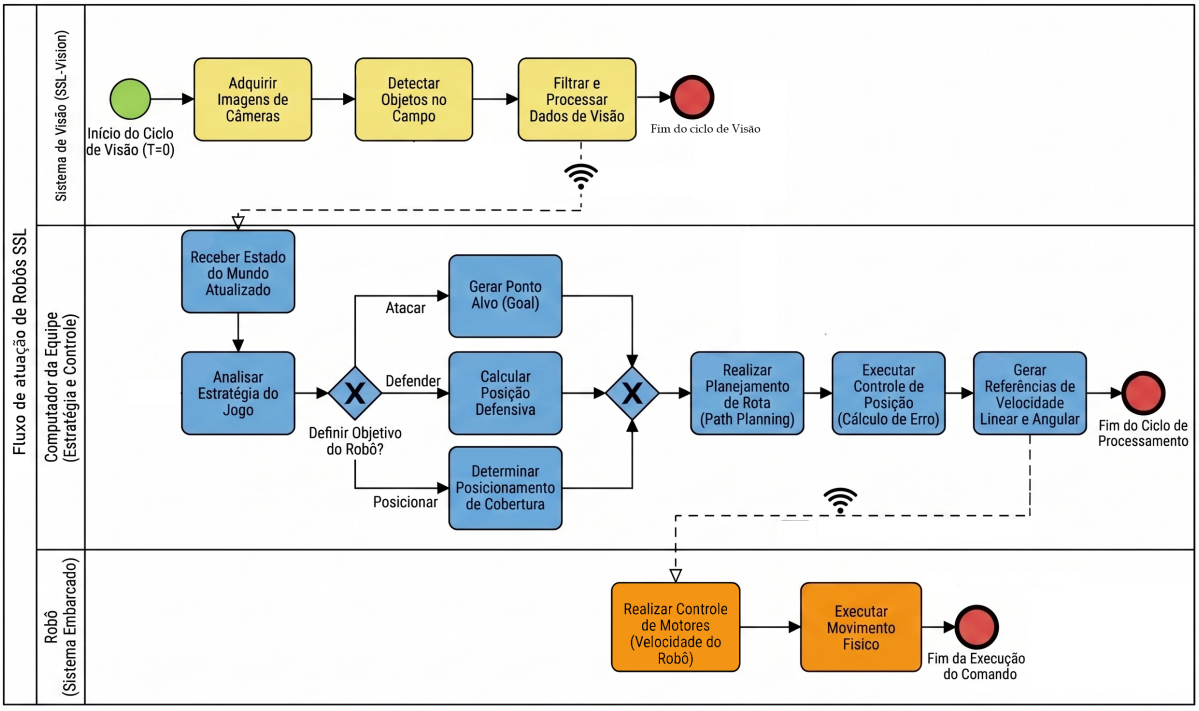
2 Metodologia

Este capítulo apresenta os fundamentos técnicos que sustentam o desenvolvimento deste trabalho, reunindo os elementos necessários para a compreensão dos métodos adotados. São abordadas as características principais do fluxo de execução do processo durante o jogo e dos robôs omnidirecionais empregados na SSL, além de uma apresentação detalhada da forma de operação do VG, bem como a fundamentação teórica dos quatro algoritmos que foram utilizados como base para a implementação.

2.1 Arquitetura do Sistema e Fluxo de Execução

O fluxo de execução geral do sistema pode ser melhor compreendido por meio do diagrama apresentado na Figura 3, estruturado com base na notação Business Process Model and Notation (BPMN). O diagrama organiza as principais etapas do sistema em três módulos distintos: sistema de visão, computador da equipe e robô, evidenciando o ciclo contínuo de percepção, decisão e ação.

Figura 3 – Diagrama de fluxo de processo da categoria SSL



Fonte: Autoria própria

No primeiro módulo, correspondente ao sistema de visão, ocorre a aquisição das imagens do ambiente, seguida da detecção dos elementos relevantes em campo, como robôs e

bola. Esses dados são então filtrados e processados para gerar uma representação consistente do estado do jogo, que é enviada através de um pacote de dados ao computador da equipe.

No computador, inicia-se a etapa de tomada de decisão. A partir do estado atualizado do ambiente, é realizada a análise da estratégia de jogo, permitindo definir o objetivo do robô, como ações ofensivas, defensivas ou de posicionamento. Em seguida, é determinado o ponto alvo a ser alcançado e executado o planejamento de rotas, considerando a presença de obstáculos no ambiente. Com a trajetória definida, é aplicado o controle de posição, responsável por calcular o erro em relação ao objetivo e gerar referências de movimento. Essas referências são então convertidas em comandos de velocidade linear e angular, que são enviados ao robô via rádio ou *Wi-fi*.

No robô, os comandos recebidos são utilizados pelo sistema embarcado para realizar o controle de velocidade dos motores, resultando na execução do movimento físico em campo. Após a movimentação, o novo estado do robô é novamente capturado pelo sistema de visão, fechando o ciclo de operação.

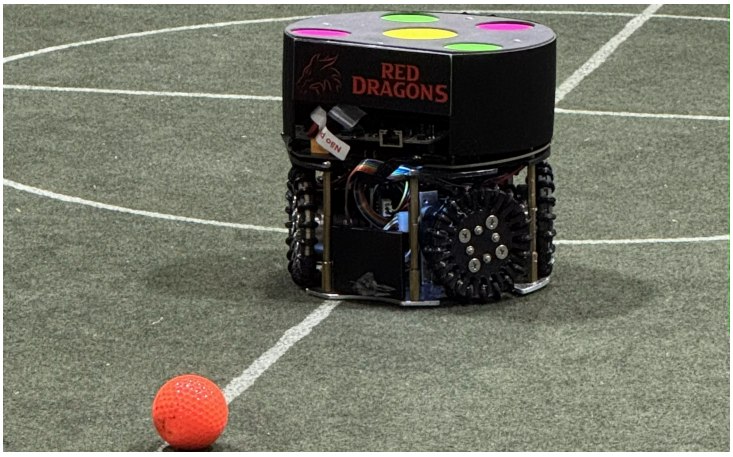
Esse processo ocorre de forma contínua ao longo da partida, permitindo que o sistema reaja dinamicamente às mudanças do ambiente, atualizando decisões e trajetórias em tempo real.

2.2 Robôs da Small Size League

Os robôs da SSL representam a estrutura física de toda a operação de planejamento e controle das equipes que disputam a liga, sendo os elementos que executam na prática as trajetórias calculadas pelos algoritmos de planejamento. Portanto, é fundamental entender suas características porque a estrutura mecânica, os limites de desempenho e a forma de locomoção definem as restrições geométricas e dinâmicas aplicadas nos testes.

A categoria segue especificações oficiais definidas pela *RoboCup*, que padronizam dimensões e área de atuação dos robôs para garantir equilíbrio entre as equipes. Cada robô deve caber em um cilindro de até 180 mm de diâmetro e 150 mm de altura, com massa em torno de 2,5 kg. Esses limites influenciam diretamente a modelagem do simulador e o dimensionamento do espaço livre considerado nos planejadores (ROBOCUP, 2025). Para ilustrar de forma mais clara, a Figura 4 apresenta o robô utilizado pela equipe *Red Dragons*, da UFSCar, na *Competição Brasileira de Robótica* (CBR) em 2025.

Figura 4 – Robô da equipe *Red Dragons*

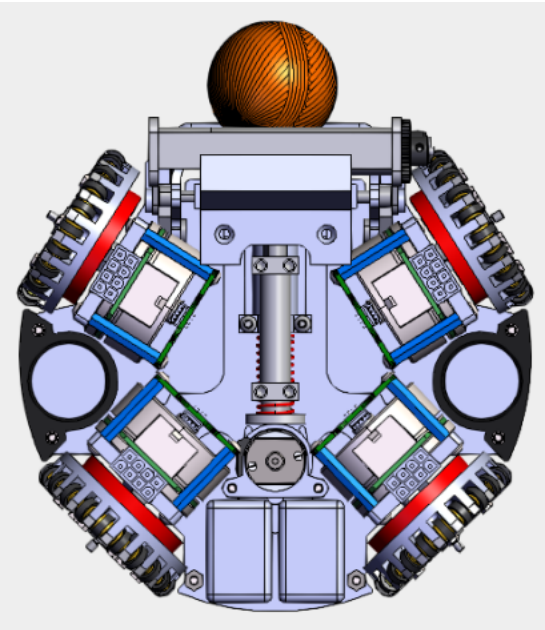


Fonte: Autoria própria

A estrutura mecânica típica dos robôs da SSL é composta por um chassi que pode ser construído com material livre, por exemplo usinado ou impresso, e abriga os módulos necessários para o desempenho das principais atividades, como drible e chute, além das placas com componentes eletrônicos e da bateria. Essa disposição favorece a compactação dos sistemas e o baixo centro de gravidade, aspectos fundamentais para garantir estabilidade durante manobras rápidas. Os robôs utilizam rodas omnidirecionais dispostas radialmente, o que lhes permite se mover em qualquer direção do plano sem necessidade de reorientar o corpo do robô. Essa configuração possibilita o controle independente das velocidades nos eixos x e y, além da rotação em torno do próprio eixo, permitindo deslocamentos suaves e precisos. Essa característica é especialmente importante no planejamento de rotas, pois os algoritmos podem explorar movimentos contínuos em todas as direções do campo, sem limitações de manobrabilidade.

A cinemática dos robôs da SSL é baseada em um modelo que relaciona as velocidades de referência, lineares nos eixos v_x e v_y e angular ω, às velocidades de cada roda, considerando o raio das rodas e o ângulo de montagem em relação ao corpo. Essa formulação é amplamente utilizada pelas equipes da liga e serve de base para o controle de baixo nível implementado no simulador. A Figura 5, extraída do TDP da equipe *ThunderBotz* (DUMITRUF et al., 2020), ilustra o arranjo das rodas representando a configuração cinemática omnidirecional usada pelos robôs.

Figura 5 – Esquema da configuração de rodas omnidirecionais



Fonte: (DUMITRUF et al., 2020)

Os atuadores empregados normalmente são motores *Direct Current* (DC) de alta rotação acoplados a *encoders* de precisão, responsáveis por garantir respostas rápidas e alta capacidade de aceleração. As principais limitações que impactam o desempenho são a velocidade máxima linear e angular, a taxa de aceleração e o tempo de resposta do sistema. O controle de baixo nível embarcado é responsável por converter as referências de movimento v_x, v_y e ω em comandos individuais de velocidade para cada roda.

Por fim, a integração entre o planejador e o controlador reflete o fluxo utilizado em competições reais, em que as camadas de estratégia definem os objetivos e o planejador calcula o caminho até o destino, enquanto o controle de baixo nível executa o movimento. Essa arquitetura é reproduzida no ambiente de simulação *grSim*, garantindo que os testes realizados representem de maneira fiel a dinâmica dos robôs em campo e permitindo que as comparações entre algoritmos considerem apenas as diferenças nos métodos de planejamento de rotas.

2.3 *Visibility Graph*

O VG é uma forma de representação geométrica do espaço de navegação em que os vértices do grafo correspondem aos vértices dos polígonos que representam obstáculos e aos pontos de interesse, como o ponto de partida e o ponto de chegada. Duas posições do grafo são ligadas por uma aresta quando o segmento de reta que as une não intersecta o interior de nenhum obstáculo, o que equivale a haver visão direta entre as posições. Esta construção torna explícitas as conexões geométricas possíveis no espaço livre e permite que

algoritmos de busca encontrem trajetórias formadas por segmentos de reta entre vértices sem depender de uma discretização uniforme do espaço (AZIZI; SULAIMANY, 2024).

Formalmente adota-se um conjunto de obstáculos representado por polígonos fechados e define-se V como o conjunto de vértices livres que inclui os vértices desses polígonos e os pontos de interesse. O conjunto de arestas do VG é dado por

$$E = \{(u, v) \in V \times V \mid \overline{uv} \cap \text{int}(O) = \varnothing \ \forall O\},$$

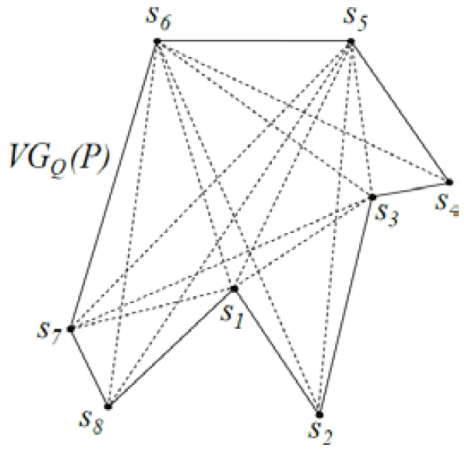
sendo que $\overline{uv}$ representa o segmento que une u e v e $\text{int}(O)$ denota o interior de um obstáculo. A presença de uma aresta significa que existe linha de ligação direta entre as duas posições sem interseção com obstáculos (D'AMATO et al., 2021).

A construção completa do VG pode ser custosa em mapas com muitos vértices reflexos, pois é preciso testar visibilidade para até $O(n^2)$ pares de pontos e cada teste pode exigir varredura sobre arestas do mapa. Por isso implementações práticas usam pré processamento que identifica vértices reflexos relevantes, ordenação angular dos pontos e estruturas que evitem verificações redundantes, reduzindo o custo médio da construção sem alterar a correção do grafo (KALUDER; BREZAK; PETROVIC, 2011).

A definição do ambiente começa pela representação dos obstáculos como polígonos simples. Cada obstáculo é descrito por uma sequência ordenada de vértices que definem sua fronteira e a orientação dessa sequência indica quais vértices são reflexos. Pontos reflexos são os vértices cujo ângulo interno é maior que 180 graus e são eles que determinam, em grande parte, as possíveis conexões relevantes no grafo. A partir do conjunto de vértices dos polígonos formam se os pontos candidatos do VG e, para cada vértice reflexo, verifica se os outros vértices são visíveis a partir dele. A informação de visibilidade comumente é armazenada em uma matriz binária onde cada linha corresponde ao vetor de visibilidade de um vértice reflexo, e como a visibilidade é bidirecional só é necessário computar uma das duas entradas do par (KALUDER; BREZAK; PETROVIC, 2011).

A figura 6 apresentada no artigo de (KALUDER; BREZAK; PETROVIC, 2011) ilustra um polígono Q com os pontos $s_1, \ldots, s_8$ usados para a construção do VG. Os vértices numerados representam os pontos da borda do polígono e as linhas contínuas indicam a fronteira do obstáculo. As linhas tracejadas representam as conexões de visibilidade estabelecidas entre pares de vértices que possuem linha de ligação direta no interior do polígono. Dessa forma, a figura exemplifica a tradução da geometria do ambiente em um conjunto de nós e arestas do grafo, onde cada aresta corresponde a um segmento livre de colisão que conecta dois vértices.

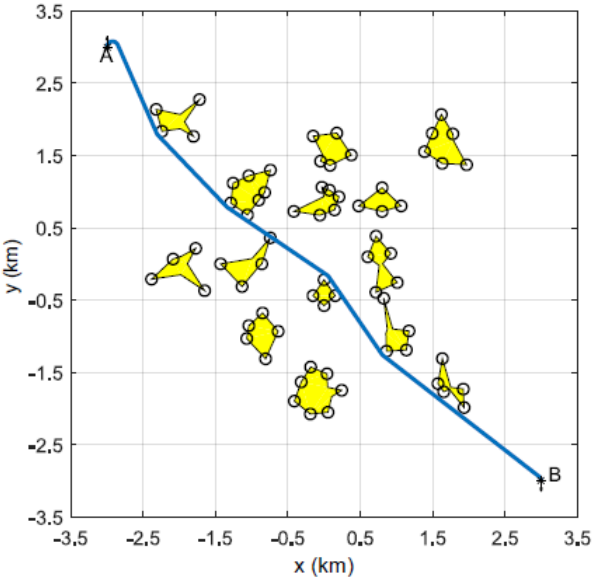
Figura 6 – Polígono Q com conexões de visibilidade do VG



Fonte: (KALUDER; BREZAK; PETROVIC, 2011)

A partir dos vértices e da matriz de visibilidade, constrói-se o conjunto de arestas do VG unindo todo par de vértices que se veem mutuamente. O caminho no grafo é então uma sequência de vértices conectados por arestas visíveis e, quando a métrica de custo é o comprimento euclidiano, o trajeto selecionado tende a minimizar a soma dos comprimentos dos segmentos retos que o compõem. A Figura 7 exemplifica o resultado final desse processo e permite identificar claramente que cada vértice do trajeto é um vértice original do polígono enquanto cada aresta do trajeto é um segmento livre de colisão entre dois desses vértices (D'AMATO et al., 2021).

Figura 7 – VG com caminho poligonal destacado



Fonte: (D'AMATO et al., 2021)

Do ponto de vista prático, o VG formaliza as conexões geométricas do espaço livre ao reduzir o problema de planejamento de trajetórias à busca por caminhos em um grafo cujos

nós são vértices poligonais. Entre os benefícios do VG está a capacidade de representar trajetórias poligonais próximas do comprimento mínimo, resultando em caminhos com menos quebras de direção e com boa qualidade geométrica.

Destaca-se que após a construção do grafo, torna-se possível aplicar algoritmos clássicos de caminho mínimo para obter trajetórias que exigem pouco pós processamento para se ajustarem às exigências de seguimento do robô. Como limitação importante tem se o custo da construção completa em mapas detalhados. Por esta razão, trabalhos da área recomendam não construir o grafo integralmente em mapas de alta resolução sem primeiro reduzir ou filtrar o conjunto de vértices (LEE; CHOI; KIM, 2021).

2.4 Algoritmo de *Dijkstra*

O algoritmo de *Dijkstra* é um dos métodos mais clássicos para cálculo de caminhos mínimos em grafos, proposto originalmente em 1959 e ainda hoje amplamente utilizado em diversas áreas de planejamento de rotas. Seu funcionamento baseia-se na expansão progressiva de vértices a partir de uma fonte inicial, em ordem crescente de custo acumulado, até que as menores distâncias para todos os nós tenham sido determinadas. Essa abordagem garante soluções em grafos com pesos de aresta não-negativos, sendo ao mesmo tempo simples de implementar e conceitualmente robusta. Como vantagens, destacam-se a previsibilidade de comportamento e a independência de heurísticas externas, já que a busca depende apenas dos custos efetivos das arestas. Entre as limitações estão o elevado custo computacional e de memória em grafos grandes, além da ausência de mecanismos incrementais que permitam reutilização de resultados em ambientes dinâmicos, o que exige reexecução completa a cada mudança no mapa (FADZLI et al., 2015).

Em aplicações práticas como *Automated Guided Vehicle* (AGVs), o algoritmo de *Dijkstra* é usado devido à sua confiabilidade e determinismo, sendo adequado para trajetórias em ambientes industriais estruturados. Entretanto, quando o espaço de busca cresce ou há necessidade de replanejamentos frequentes, seu custo pode se tornar proibitivo, o que motiva técnicas auxiliares como redução do grafo de busca ou discretizações otimizadas para preservar viabilidade computacional (LI; NIU, 2014). De forma semelhante, em *Unmanned Aerial Vehicle* (UAVs), adaptações do algoritmo contemplam modelos tridimensionais de espaço e funções de custo específicas, por exemplo incorporando altitude e energia consumida, mas preservando o princípio central de garantir o menor custo acumulado desde a fonte até cada destino (DHULKEFL; DURDU; TERZIOĞLU, 2020).

Formalmente, ao considerar um grafo $G = (V, E)$ com função de peso $w : E \to \mathbb{R}\geq 0$. Para um caminho $p = \langle v_0, v_1, \ldots, v_k \rangle$, define-se o comprimento

$$\text{len}(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}),$$

e a distância ótima entre dois vértices u, v como

$$\delta(u, v) = \min_{p:u \to v} \text{len}(p),$$

com $\delta(u, v) = +\infty$ quando não existe caminho. O objetivo do algoritmo de *Dijkstra*, a partir de uma fonte s, é calcular $\delta(s, v)$ para todo $v \in V$ (LI; NIU, 2014).

O método mantém para cada vértice uma estimativa $d[v]$ do custo mínimo conhecido e, tipicamente, um ponteiro $parent[v]$ para reconstrução da rota. Inicialmente $d[s] = 0$ e $d[v] = +\infty$ para todo $v \neq s$. Em cada iteração, o vértice u com menor valor $d[u]$ é extraído da fila de prioridade e processado por meio da operação de relaxamento:

$$d[v] \leftarrow \min\Big(d[v],\, d[u] + w(u, v)\Big),$$

para cada aresta $(u, v) \in E$. A extração ordenada garante que, no momento em que u é removido da fila, tem-se $d[u] = \delta(s, u)$ (DHULKEFL; DURDU; TERZIOĞLU, 2020).

No contexto deste trabalho, em que o grafo é gerado por VG, o algoritmo de *Dijkstra* representa a solução de referência utilizada pela equipe *Red Dragons*, da UFSCar. Ele oferece uma base sólida de comparação para outros algoritmos, pois sua natureza exata e determinística garante caminhos de custo mínimo global, permitindo avaliar de forma justa as vantagens de abordagens heurísticas ou incrementais.

2.5 Algoritmo A*

O A* é um algoritmo de busca heurística amplamente utilizado para planejamento de trajetórias, caracterizando-se por dirigir a exploração do espaço de estados de forma informada e previsível. Ele combina, de maneira explícita, o custo já incorrido e uma estimativa do custo remanescente para priorizar expansões. Essa combinação confere ao método a capacidade de concentrar esforço de busca nas regiões mais promissoras do grafo. Entre suas vantagens destacam-se a simplicidade conceitual, a facilidade de implementação em grafos genéricos, e a garantia teórica de obter solução de custo mínimo sob as hipóteses adequadas. Como limitações práticas salientam-se o elevado consumo de memória quando muitos nós são gerados, a sensibilidade à qualidade da heurística (que determina diretamente a eficiência) e a ausência, na formulação clássica, de mecanismos incrementais para reutilização de resultados em mapas dinâmicos (NILAVAR et al., 2021).

No contexto de futebol de robôs, o A* tende a ser uma escolha robusta para planejamento inicial devido à clareza das garantias teóricas e à compatibilidade com representações geométricas VG, entretanto, quando replanejamentos frequentes ou requisitos de baixa latência são exigidos, a falta de incrementalidade do A* impõe recálculos completos, o que pode ser impraticável em cenários com mudanças contínuas, nesses casos, costuma-se combinar A* com técnicas complementares para atualizações locais (CHU et al., 2024).

Para cada nó n, o A* mantém o custo acumulado $g(n)$ correspondente ao custo conhecido do nó inicial até n, e uma heurística estimativa $h(n)$ do custo restante de n até o objetivo. A função de avaliação que orienta a expansão é

$$f(n) = g(n) + h(n),$$

e o algoritmo expande iterativamente o nó com menor valor de f na fila de abertura, de modo a equilibrar exploração e custo já incorrido (NILAVAR et al., 2021).

Em representações espaciais discretas, os custos de transição $c(n, n')$ usualmente adotados são $c = 1$ para movimentos ortogonais e $c = \sqrt{2}$ para movimentos diagonais, e as heurísticas mais comuns são a distância euclidiana:

$$h_{\mathrm{EUC}}(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2},$$

a Manhattan:

$$h_{\mathrm{MAN}}(n) = |x_n - x_g| + |y_n - y_g|$$

e variantes diagonais ajustadas à discretização (CHU et al., 2024).

No contexto de grafos gerados por VG, que é caso aplicado neste trabalho, a heurística euclidiana é a escolha natural e preserva admissibilidade quando os custos de aresta correspondem a distâncias euclidianas entre vértices. Para cenários práticos e de maior escala, técnicas de aceleração e poda são recomendadas, como o *Jump Point Search* (JPS), que identifica pontos de salto e pula sequências de nós redundantes mantendo ótimos sob heurísticas consistentes (YAO, 2023).

2.6 Algoritmo LPA*

O LPA* é uma variante incremental do A* concebida para reaproveitar trabalho entre execuções sucessivas quando o grafo sofre alterações locais. Em seu conceito de funcionamento, em vez de recalcular a busca do zero a cada modificação do mapa, o LPA* mantém estruturas auxiliares que permitem propagar apenas as correções necessárias, reduzindo a latência de replanejamento em cenários onde a percepção atualiza parcialmente o ambiente enquanto o agente opera. Essa característica torna o LPA* adequado para aplicações em que mudanças são frequentes, porém localizadas (LU et al., 2011).

Embora o LPA* seja capaz de lidar com obstáculos dinâmicos e diminua o custo de replanejamento em muitos casos práticos, ele apresenta limitações operacionais em relação a eficiência. Em contextos onde o agente se desloca continuamente e muitas chaves precisam ser reajustadas, existem algoritmos, como o *D* Lite,* com mecanismos adicionais de deslocamento de chave que tratam essa situação de forma específica (LU et al., 2011; KOENIG; LIKHACHEV, 2002).

Formalmente, o LPA* mantém para cada vértice s duas quantidades essenciais: $g(s)$, o custo conhecido do estado inicial até s, e $rhs(s)$, a estimativa "um passo à frente" definida por

$$rhs(s) = \begin{cases} 0, & \text{se } s = s_{goal}, \\ \min_{s' \in Succ(s)} \Big(c(s, s') + g(s') \Big), & \text{se } s \neq s_{goal}, \end{cases}$$

em que $c(s, s')$ é o custo de transição entre s e s'. Um vértice é dito consistente quando $g(s) = rhs(s)$. Inconsistências ($g(s) \neq rhs(s)$) sinalizam nós que precisam ser reprocessados após alterações locais.

A ordem de processamento é regulada por uma chave dupla associada a cada vértice:

$$k(s) = \Big(k_1(s),\, k_2(s) \Big), \qquad k_1(s) = \min\{g(s),\, rhs(s)\} + h(s), \quad k_2(s) = \min\{g(s),\, rhs(s)\},$$

no qual $h(s)$ é uma heurística admissível que estima o custo remanescente. O algoritmo extrai repetidamente o vértice de menor chave, aplica uma atualização nas arestas incidentes e invoca a rotina *UpdateVertex* apenas quando necessário, restaurando assim a consistência local. Quando $g(s_{start}) = rhs(s_{start})$, o caminho reconstruído pelos ponteiros de antecessores é ótimo em relação aos custos atuais do grafo, sob a hipótese de admissibilidade da heurística (LU et al., 2011).

Além do algoritmo convencional adotado para comparação neste estudo, a literatura apresenta adaptações do LPA* voltadas ao particionamento do grafo para permitir maior escalabilidade. Nesses trabalhos, o espaço de busca é dividido em subdomínios menores, reduzindo o custo computacional de cada atualização. Com isso, torna-se possível executar buscas incrementais localizadas, limitando o impacto de alterações a apenas uma parte do grafo. A técnica também facilita a paralelização do processamento, distribuindo tarefas entre múltiplos núcleos ou processos. Essa estratégia é particularmente útil em aplicações que lidam com mapas extensos e sujeitos a mudanças frequentes (LIM; SALZMAN; TSIOTRAS, 2021).

Em suma, o LPA* concilia a condição ótima conceitual do A* com mecanismos incrementais que permitem reaproveitar cálculos entre execuções, resultando em respostas rápidas a mudanças locais do mapa quando implementado com estruturas de dados e políticas adequadas (LU et al., 2011).

2.7 Algoritmo D* Lite

O *D* Lite,* proposto por Koenig e Likhachev, constitui uma formulação incremental eficiente para o planejamento de rotas em ambientes sujeitos a alterações durante a execução do percurso. Assim como o LPA*, ele se apoia na ideia de reutilizar cálculos anteriores, evitando recomputações globais sempre que novas informações sobre o ambiente são adquiridas. A diferença central está na forma como essa atualização é conduzida,

pois o *D* Lite* adapta a estrutura incremental para lidar especificamente com problemas de navegação em tempo real (KOENIG; LIKHACHEV, 2002). Além disso, o algoritmo realiza o planejamento a partir do destino em direção à origem, o que se mostra vantajoso em cenários onde o agente está em movimento, pois garante que, ao final do processo de planejamento, a posição atual da origem permaneça consistente em relação ao estado considerado no início do cálculo, reduzindo inconsistências associadas ao deslocamento durante a execução.

A principal melhoria em relação ao LPA* é a introdução de um mecanismo que ajusta dinamicamente a ordem de expansão dos nós, por meio do parâmetro de deslocamento acumulado (k_m). Isso permite ao algoritmo manter consistência mesmo quando o agente se desloca, reduzindo o retrabalho em replanejamentos consecutivos. Com isso, o *D* Lite* propaga apenas as correções necessárias nas áreas realmente afetadas, tornando-se mais adequado para cenários dinâmicos e parcialmente conhecidos, como a navegação de robôs móveis em campo (KOENIG; LIKHACHEV, 2002).

Matematicamente, a estrutura do *D* Lite* apoia-se em duas funções associadas a cada vértice s: $g(s)$, o custo atualmente conhecido de s ao objetivo, e $rhs(s)$, a estimativa "um passo à frente" definida como

$$rhs(s) = \begin{cases} 0, & \text{se } s = s_{goal}, \\ \min_{s' \in Succ(s)} \Big(c(s, s') + g(s') \Big), & \text{se } s \neq s_{goal}, \end{cases}$$

em que $c(s, s')$ é o custo da transição entre s e s'. Um nó é dito consistente quando $g(s) = rhs(s)$; inconsistências ($g(s) \neq rhs(s)$) são o motor do processo de correção incremental, ou seja, somente nós inconsistentes e aqueles que os influenciam precisam ser reprocessados. Essas definições seguem a formulação padrão (KOENIG; LIKHACHEV, 2002).

A ordem de expansão é controlada por uma chave dupla $k(s) = (k_1(s), k_2(s))$ construída a partir de g, rhs e de uma heurística admissível h:

$$k_1(s) = \min\{g(s),\, rhs(s)\} + h(s_{start}, s) + k_m,$$
$$k_2(s) = \min\{g(s),\, rhs(s)\},$$

sendo k_m um deslocamento acumulado que é atualizado quando a referência, por exemplo, o robô se move, preservando a ordem relativa dos nós na fila e evitando reprocessamentos desnecessários. A escolha de h deve respeitar admissibilidade para manter a trajetória ótima (KOENIG; LIKHACHEV, 2002).

A rotina central do *D* Lite* é a operação de atualização local (*UpdateVertex*), que recalcula $rhs(s)$ a partir dos sucessores, decide se deve inserir ou remover s da fila de prioridade e, quando necessário, propaga mudanças para predecessores. Esse mecanismo garante convergência incremental, dessa forma as mudanças locais de custo afetam apenas

um subconjunto de vértices que, via atualizações sucessivas de rhs e ajuste dos g, retornam o sistema ao estado de consistência (todos os nós relevantes com $g = rhs$) (YU et al., 2020).

Do ponto de vista de complexidade, o ganho essencial do *D* Lite* é obtido através do fato de que o custo do replanejamento é proporcional ao número de nós efetivamente afetados pelas mudanças locais, denotado m, enquanto uma reexecução de *Dijkstra* ou A* do zero tem custo associado ao total do grafo. Assim, para alterações pequenas e localizadas espera-se que *D* Lite* seja substancialmente mais eficiente na prática. Entretanto, se uma mudança afeta ampla porção do mapa o esforço tende a se aproximar do custo de uma busca global, justificando políticas híbridas (YU et al., 2020).

Na integração com módulos de percepção e representação do mapa, mudanças detectadas, como por exemplo uma nova ocupação ou alteração de custo em arestas, devem ser traduzidas imediatamente em atualizações locais de custo e em chamadas a *UpdateVertex* somente para os vértices afetados (REYES; BARCZAK; SUSNIAK, 2018).

No contexto multi-robô, o *D* Lite* pode ser empregado por cada agente de forma distribuída, ou seja, os robôs mantêm mapas locais e propagam mudanças relevantes. Essa abordagem funciona de forma eficaz, mas exige mecanismos de coordenação para evitar conflitos de planejamento. Trabalhos que adaptam *D* Lite* para múltiplos agentes mostram ganhos quando há coordenação das mudanças de mapa e políticas de atenção às rotas como obstáculos temporários (AL-MUTIB et al., 2011).

No contexto deste trabalho, adotou-se a filosofia padrão do *D* Lite,* ou seja, mantiveram-se as estruturas g, rhs, a fila priorizada por k, e as detecções de mudança provenientes do módulo de percepção geraram chamadas locais à função *UpdateVertex*.

3 Desenvolvimento

O presente capítulo descreve as etapas de desenvolvimento do trabalho, abrangendo desde a configuração do ambiente de simulação até a implementação e execução dos algoritmos de planejamento de rotas. A construção do trabalho foi guiada pelos conceitos e métodos apresentados no capítulo 2, com foco na aplicação prática e na avaliação comparativa dos planejadores propostos. Inicialmente é detalhado o simulador utilizado para os experimentos, seguido da descrição das implementações realizadas para cada algoritmo, todas estruturadas sobre o mesmo grafo gerado pelo método VG. Por fim, são apresentadas as definições dos testes conduzidos, os critérios adotados para avaliação de desempenho e a forma como os resultados foram obtidos.

3.1 Simulador - grSim

O ambiente de simulação adotado neste trabalho é o grSim, um simulador concebido para competições de futebol de robôs, que permite testar o software de controle e planejamento da equipe sem grandes adaptações entre simulação e hardware real. O grSim reproduz em computador as condições de jogo e os sinais de percepção de modo que os módulos de localização, planejamento e controle consumam os mesmos formatos de dados utilizados em partidas reais. Essa compatibilidade torna o simulador especialmente útil para desenvolver e validar estratégias de alto nível quando o acesso aos robôs físicos é restrito ou quando se deseja realizar um grande número de ensaios controlados (MONAJJEMI; KOOCHAKZADEH; GHIDARY, 2012).

A arquitetura do grSim procura equilibrar fidelidade física e desempenho computacional para permitir execuções em tempo real. O núcleo realiza cálculo de colisões e respostas de corpo rígido por meio de um motor de dinâmica consolidado e a camada de visualização faz uso de aceleração gráfica quando disponível. Para reduzir o custo computacional sem perder comportamento observável relevante, o simulador adota modelos simplificados para componentes críticos da dinâmica dos robôs, por exemplo para as rodas omnidirecionais, em que efeitos de deslizamento e resistência são representados por coeficientes direcionais de atrito em vez de modelagem geométrica detalhada (MONAJJEMI; KOOCHAKZADEH; GHIDARY, 2012).

A integração do grSim com os módulos de controle e estratégia da equipe utilizam o mesmo protocolo e o mesmo formato de dados empregados pelo sistema de visão computacional da liga, de forma que as saídas simuladas informam posição e orientação dos robôs e posição da bola segundo o esquema habitual e os comandos chegam ao simulador por meio de pacotes de rede. Na Figura 8 é possível visualizar a interface inicial do simulador.

Figura 8 – Interface Simulador grSim

Fonte: Autoria própria

Em suma, o grSim reproduz com fidelidade o comportamento dos robôs em campo, oferecendo dinâmica compatível com a operação real e respostas coerentes aos comandos de controle. A forma como os movimentos, acelerações e interações físicas são simulados permite observar o desempenho dos planejadores em condições próximas às de jogo (MONAJJEMI; KOOCHAKZADEH; GHIDARY, 2012). Além disso, assim como em um ambiente físico, os experimentos podem apresentar variabilidade natural nos resultados, uma vez que o próprio simulador incorpora modelos de cinemática e dinâmica dos robôs, influenciando a execução das trajetórias em função de acelerações, velocidades e interações físicas. Dessa forma, essa correspondência garante que os resultados obtidos sejam consistentes e úteis para análise comparativa dos algoritmos. Por esse motivo, o simulador é adotado como base confiável para a realização dos casos de testes apresentados neste trabalho.

3.2 Implementação dos algoritmos

Esta seção apresenta o desenvolvimento das etapas de implementação adotadas para cada algoritmo de planejamento. Embora o VG seja classicamente definido como uma estrutura estática, construída a partir de posições fixas de obstáculos, sua aplicação neste estudo também foi estendida para cenários dinâmicos. O VG é utilizado como uma representação do espaço livre navegável, no qual os vértices correspondem aos vértices dos obstáculos e as arestas representam conexões de visibilidade que não interceptam regiões ocupadas, ou seja, caminhos válidos para deslocamento do robô. Dessa forma, o planejamento não ocorre sobre os obstáculos, mas sim sobre as conexões livres entre eles, garantindo trajetórias livres de colisão.

Nos testes em que há movimentação de obstáculos, o grafo de visibilidade é reconstruído a cada atualização relevante do ambiente, de modo que as conexões representem fielmente o espaço livre disponível para o robô. Essa abordagem garante que os planejadores operem sempre sobre uma estrutura coerente com o estado atual da simulação, evitando inconsistências entre o mapa e a realidade dinâmica do jogo.

Portanto, o VG foi adotado como base comum para todos os algoritmos avaliados por fornecer uma representação geométrica precisa do ambiente, permitindo a geração de caminhos próximos do ótimo em termos de distância e garantindo uma base consistente para comparação entre diferentes estratégias de busca. A partir dessa estrutura atualizada, é possível comparar o desempenho de algoritmos que não são originalmente projetados para lidar com ambientes mutáveis, como o *Dijkstra* e o A*, em contraste com os planejadores incrementais, como o LPA* e o *D* Lite,* que conseguem atualizar apenas os trechos afetados.

3.2.1 VG + *Dijkstra*

A implementação do algoritmo de *Dijkstra* neste trabalho atua sobre o grafo geométrico previamente construído pelo VG. O pré-processamento seleciona e aproxima os obstáculos relevantes, converte essas geometrias para o formato poligonal triangular e obtém o conjunto de vértices e arestas que compõem a estrutura de visibilidade. A escolha por representar os obstáculos como polígonos triangulares foi adotada com o objetivo de reduzir o custo computacional na etapa de construção do grafo, uma vez que, no VG, o número de arestas pode crescer de forma tipicamente quadrática em relação ao número de vértices. Assim, a utilização de polígonos mais elaborados implicaria em um aumento significativo na quantidade de arestas geradas e, consequentemente, no tempo necessário para construção e processamento do grafo. O grafo resultante funciona como representação do espaço livre, sobre o qual o algoritmo de *Dijkstra* determina a rota entre os vértices definidos como origem e destino.

Como foi citado no capítulo 2, esse método de planejamento de rotas é atualmente utilizado pela equipe *Red Dragons*, da UFSCar, e a escolha da equipe por esse algoritmo clássico se justifica pelo fato de ser um método eficaz de busca em grafos ponderados, capaz de garantir a obtenção do caminho de menor custo sem a necessidade de heurísticas.

Dessa forma, ele servirá como referência de comparação para as demais abordagens, oferecendo uma base sólida para avaliar ganhos de eficiência de algoritmos mais sofisticados.

O pseudocódigo 1 representa de forma simplificada o código implementado para testes e o código completo consta no apêndice A.

Algorithm 1 : *Dijkstra* sobre grafo gerado pelo Visibility Graph

```
1: for all v ∈ V do
2:     dist[v] ← ∞, antecessor[v] ← ∅
3: end for
4: dist[origem] ← 0; fila ← {(0, origem)}
5: while fila ≠ ∅ do
6:     (d, u) ← extrair_min(fila)
7:     if d ≠ dist[u] then
8:         continue
9:     end if
10:    if u = destino then
11:        break
12:    end if
13:    for all (v, w) ∈ adj[u] do
14:        if dist[u] + w < dist[v] then
15:            dist[v] ← dist[u] + w; antecessor[v] ← u
16:            inserir(fila, (dist[v], v))
17:        end if
18:    end for
19: end while
20: reconstruir_caminho(origem, destino)
```

Ao iniciar o algoritmo são preparadas duas estruturas por vértice. O vetor $dist[v]$ guarda a melhor estimativa conhecida do custo da origem até v e $antecessor[v]$ armazena o predecessor usado para reconstruir a rota. Todas as entradas de $dist$ são inicializadas em $+\infty$ exceto a origem que recebe zero. A fila de prioridade contém tuplas do tipo (chave, v) e permite extrair sempre o vértice com a menor estimativa atual, impondo a ordem crescente de custos que garante a correção do método.

No laço principal extrai se da fila a tupla (d, u) de menor chave e valida se essa tupla representa o estado atual comparando d com $dist[u]$. Essa verificação descarta entradas antigas geradas por reinserções e evita reprocessamento desnecessário. Se u for o destino a execução pode interromper mais cedo. Para cada aresta (u, v) com peso w calcula se a estimativa candidata $alt = dist[u] + w$. Quando $alt < dist[v]$ realiza se a atualização de custo atribuindo $dist[v] \leftarrow alt$, definindo $antecessor[v] \leftarrow u$ e inserindo na fila a nova tupla $(dist[v], v)$.

Ao término, se $dist[destino] = +\infty$ não existe caminho viável, caso contrário reconstrói se a rota seguindo os ponteiros de *antecessor* do destino até a origem e invertendo a sequência. O apêndice A apresenta a implementação completa do algoritmo de *Dijkstra* aplicada neste estudo.

3.2.2 VG + A*

A implementação do algoritmo A* neste trabalho utiliza o mesmo grafo geométrico citado anteriormente, obtido pelo VG, que representa as conexões válidas entre vértices livres do ambiente. O papel do grafo permanece o de estruturar o espaço de navegação, enquanto a busca é guiada pelo A*, que combina custo acumulado e heurística para escolher os caminhos mais promissores.

A principal diferença em relação ao *Dijkstra* está no uso da função heurística, que antecipa o custo até o destino e orienta a expansão da busca. Essa estratégia permite reduzir o número de nós processados sem comprometer a assertividade, desde que a heurística seja admissível. Assim, o A* torna-se mais eficiente em comparação com o *Dijkstra*, preservando a mesma garantia de solução ótima.

O pseudocódigo 2 representa de forma simplificada a implementação adotada para os testes e o código completo consta no apêndice B.

Algorithm 2 : A* sobre grafo gerado pelo Visibility Graph

```
1: for all v ∈ V do
2:     dist[v] ← ∞, antecessor[v] ← ∅
3: end for
4: dist[origem] ← 0;  fila ← {(h(origem), origem)}
5: while fila ≠ ∅ do
6:     (f, u) ← extrair_min(fila)
7:     if u = destino then
8:         break
9:     end if
10:    for all (v, w) ∈ adj[u] do
11:        g_cand ← dist[u] + w
12:        if g_cand < dist[v] then
13:            dist[v] ← g_cand;  antecessor[v] ← u
14:            inserir(fila, (dist[v] + h(v), v))
15:        end if
16:    end for
17: end while
18: reconstruir_caminho(origem, destino)
```

A execução inicializa pela criação das duas estruturas básicas: o vetor $dist[v]$ armazena o custo acumulado conhecido desde a origem até v e $antecessor[v]$ guarda o nó predecessor para reconstrução da rota. Todas as entradas de $dist$ começam em $+\infty$ exceto a origem que recebe zero. A fila de prioridade é inicializada com a tupla (h(origem), origem) e, em geral, armazena pares (f, v) em que $f = g(v) + h(v)$ representa a função de avaliação que orienta a expansão.

No laço principal extrai-se da fila a tupla (f, u) de menor valor e, se u for o destino, a execução pode interromper-se. Para cada aresta (u, v) com peso w calcula-se a custo

candidato $g_{\text{cand}} = dist[u] + w$. Quando $g_{\text{cand}} < dist[v]$ atualiza-se $dist[v] \leftarrow g_{\text{cand}}$, grava-se $antecessor[v] \leftarrow u$ e insere-se na fila a nova tupla $(dist[v] + h(v), v)$. Essa reinserção substitui na prática uma operação de atualização direta da chave e exige que o código trate entradas obsoletas ao extraí-las.

Ao final da execução, se o valor em $dist[destino]$ permanecer em $+\infty$, significa que não existe caminho viável ligando origem e destino. Caso contrário, a rota é reconstruída a partir dos ponteiros de *antecessor*, partindo do destino e retrocedendo até a origem. A utilização da heurística h garante que a expansão privilegie os nós mais promissores, reduzindo o número de vértices processados em comparação com o *Dijkstra* e permitindo encontrar o caminho ótimo de maneira mais eficiente quando h é admissível. O apêndice B apresenta a implementação completa do algoritmo A* aplicada neste estudo.

3.2.3 VG + LPA*

Assim como nos algoritmos implementados anteriormente, o LPA* também opera sobre os grafos de visibilidade gerado pelo VG. A diferença principal em relação ao A* está na forma como o algoritmo lida com mudanças no ambiente, em vez de recalcular toda a busca, ele incorpora estruturas incrementais que permitem reaproveitar cálculos prévios e atualizar apenas os trechos afetados pelas modificações. Essa característica o torna mais eficiente em cenários dinâmicos ou parcialmente conhecidos, onde a topologia do grafo pode sofrer alterações locais durante a navegação.

O funcionamento baseia-se em manter, para cada vértice, informações que permitem identificar rapidamente inconsistências e direcionar o processamento apenas aos nós relevantes. Com isso, o LPA* consegue reduzir o esforço computacional quando comparado ao A*, processando apenas as regiões impactadas em vez de expandir novamente todo o espaço de busca.

O pseudocódigo 3 representa de forma simplificada a implementação adotada neste trabalho e o código completo consta no apêndice C.

Algorithm 3 : LPA* sobre grafo gerado pelo Visibility Graph

```
1: construir_grafo_vg(obstaculos)
2: for all v ∈ V do
3:     g[v] ← ∞, rhs[v] ← ∞
4: end for
5: rhs[goal] ← 0;  inserir(U, (k(goal), goal))
6: function UPDATE_VERTEX(u)
7:     if u ≠ goal then
8:         rhs[u] ← min_{s∈adj[u]}(c(u, s) + g[s])
9:     end if
10:    if g[u] ≠ rhs[u] then
11:        inserir(U, (k(u), u))
12:    end if
13: end function
14: function COMPUTE_SHORTEST_PATH
15:    while não convergido do
16:        (k, u) ← extrair_menor(U)
17:        if g[u] > rhs[u] then
18:            g[u] ← rhs[u]
19:        else
20:            g[u] ← ∞
21:        end if
22:        for all p ∈ pred[u] do
23:            update_vertex(p)
24:        end for
25:    end while
26: end function
27: function OBTER_CAMINHO(start,goal)
28:    seguir vizinhos que minimizem c(u, v) + g[v] até goal
29:    return caminho
30: end function
```

A formulação do LPA* introduzida no pseudocódigo 3 acrescenta estruturas incrementais sobre a lógica do A*. Cada vértice mantém dois valores: $g[v]$, que armazena o custo conhecido até o nó, e $rhs[v]$, que funciona como uma estimativa "um passo à frente". Inicialmente todos os valores são definidos como $+\infty$, exceto o destino que recebe $rhs = 0$. A fila de prioridade U é organizada pelas chaves $k(v)$, calculadas a partir de g, rhs e da heurística, e armazena apenas os nós inconsistentes, ou seja, aqueles em que $g \neq rhs$.

A rotina *UpdateVertex* é responsável por recalcular o valor de rhs de um vértice com base nos custos dos vizinhos. Se após a atualização, houver inconsistência, o nó é inserido novamente na fila com sua chave recalculada. Dessa forma, a fila funciona como mecanismo de foco, processando apenas os vértices que podem impactar a solução, em contraste com o A*, que pode expandir áreas maiores mesmo em mudanças localizadas.

A função *ComputeShortestPath* executa o núcleo do algoritmo. Em cada iteração é extraído o nó de menor chave da fila, e seu estado é comparado, ou seja, quando

$g[u] > rhs[u]$, o valor de g é atualizado para rhs; caso contrário, $g[u]$ retorna a $+\infty$. Em seguida os predecessores de u são recalculados por meio de chamadas a *UpdateVertex*, propagando de forma seletiva as mudanças necessárias. Esse mecanismo incremental permite que pequenas alterações na estrutura do grafo resultem em poucos nós reprocessados, diferentemente do A* que exigiria uma nova execução completa.

Por fim, a função *ObterCaminho* reconstrói a rota a partir do nó inicial, escolhendo a cada passo o vizinho que minimiza o custo da transição somado ao valor de g armazenado. Esse processo continua até alcançar o destino. A evolução em relação ao A* está justamente na capacidade de reutilizar cálculos prévios, enquanto o A* precisaria resolver o problema do zero a cada modificação, o LPA* aproveita os resultados anteriores e corrige apenas o que foi afetado, tornando-se mais eficiente e adequado em ambientes dinâmicos ou parcialmente conhecidos. O apêndice C apresenta a implementação completa do algoritmo LPA* aplicada neste estudo.

3.2.4 VG + *D* Lite*

Por fim, O algoritmo *D* Lite* também foi implementado sobre os mesmos grafos de visibilidade utilizados nas demais abordagens, preservando a mesma estrutura de vértices e arestas previamente construídos. A diferença central em relação ao algoritmo anterior está no fato de que o *D* Lite* não executa uma nova busca completa sempre que ocorre uma mudança no ambiente. Em vez disso, ele reaproveita cálculos anteriores e realiza apenas as correções necessárias, o que o torna mais eficiente em cenários dinâmicos. Essa característica reduz significativamente o esforço computacional em situações nas quais apenas pequenas partes do grafo são alteradas.

Esse funcionamento incremental é garantido pelo uso de uma fila de prioridade que organiza os nós que precisam ser reprocessados, evitando expandir toda a região do grafo a cada atualização. Assim, quando o robô avança no ambiente ou quando ocorre a detecção de um obstáculo inesperado, somente os nós realmente impactados pela alteração são ajustados. Isso permite que o *D* Lite* mantenha a consistência da solução sem a necessidade de reiniciar o planejamento, o que é um avanço em relação ao LPA* e aos algoritmos clássicos de busca apresentados.

O pseudocódigo 4 representa de forma simplificada a implementação adotada neste trabalho e o código completo consta no apêndice D.

Algorithm 4 : *D* Lite* sobre grafo gerado pelo Visibility Graph

```
1: construir_grafo_vg(obstaculos)
2: for all v ∈ V do
3:     g[v] ← ∞, rhs[v] ← ∞
4: end for
5: rhs[goal] ← 0;  inserir(U, (k(goal), goal))
6: function UPDATE_VERTEX(u)
7:     if u ≠ goal then
8:         rhs[u] ← min_{s∈adj[u]}(c(u, s) + g[s])
9:     end if
10:    if g[u] ≠ rhs[u] then
11:        inserir(U, (k(u), u))
12:    end if
13: end function
14: function COMPUTE_SHORTEST_PATH
15:    while U.top < k(start) ou rhs[start] ≠ g[start] do
16:        (k, u) ← extrair_menor(U)
17:        if g[u] > rhs[u] then
18:            g[u] ← rhs[u]
19:        else
20:            g[u] ← ∞
21:        end if
22:        for all p ∈ pred[u] do
23:            update_vertex(p)
24:        end for
25:    end while
26: end function
27: function OBTER_CAMINHO(start,goal)
28:    seguir vizinhos que minimizem c(u, v) + g[v] até goal
29:    return caminho
30: end function
```

A formulação do *D* Lite* apresentada no pseudocódigo 4 mantém a mesma base incremental do LPA*, mas introduz mecanismos que o tornam mais adequado para navegação em tempo real. Cada vértice possui os valores $g[v]$ e $rhs[v]$, com inicialização em $+\infty$, exceto o destino que recebe $rhs = 0$. A fila de prioridade U armazena os nós inconsistentes, isto é, aqueles em que $g \neq rhs$, organizados pelas chaves $k(v)$ que combinam custo acumulado, heurística e um deslocamento k_m atualizado a cada movimento do robô.

A rotina *UpdateVertex* funciona de forma semelhante à do LPA*, recalculando o valor de rhs com base nos sucessores e, caso o nó permaneça inconsistente, reinsere-o na fila com a chave ajustada. Assim, a fila continua atuando como filtro, garantindo que apenas os nós relevantes sejam reprocessados. O diferencial do *D* Lite* está no parâmetro k_m, que ajusta dinamicamente a prioridade dos vértices sem reiniciar toda a busca, preservando a coerência do planejamento mesmo quando a origem se desloca.

A função *ComputeShortestPath* realiza o processamento central. Enquanto a condição

de parada não for satisfeita, o nó de menor chave é extraído da fila e comparado, ou seja, se $g[u] > rhs[u]$, então $g[u]$ recebe o valor de $rhs[u]$, caso contrário, $g[u]$ retorna a $+\infty$. Em ambos os casos os predecessores de u são atualizados por chamadas a *UpdateVertex*, propagando apenas as mudanças necessárias. Esse comportamento permite que pequenas modificações no grafo resultem em reprocessamentos localizados, mantendo a consistência de forma incremental.

Por fim, a função *ObterCaminho* reconstrói a trajetória escolhendo, a cada passo, o vizinho que minimiza $c(u, v) + g[v]$ até o objetivo. Em suma, a evolução em relação ao LPA* está no fato de que o *D* Lite* planeja com menos frequência, ou seja, quando o mapa sofre apenas alterações pontuais, ele evita reexecutar a busca completa e reutiliza cálculos anteriores, propagando correções mínimas. Essa característica reduz o esforço computacional e assim que ele se torna especialmente eficiente em ambientes dinâmicos ou parcialmente conhecidos, como no caso da navegação de robôs móveis em campo. O apêndice D apresenta a implementação completa do algoritmo D* Lite aplicada neste estudo.

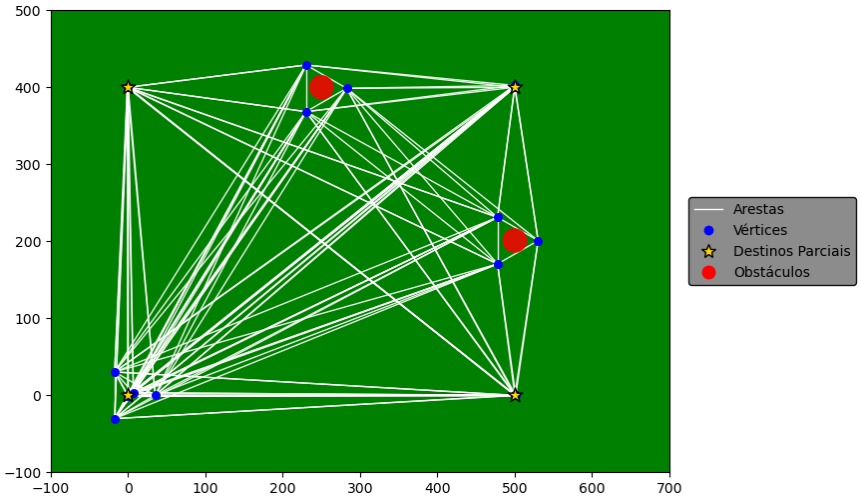
3.3 Cenários de teste

Inicialmente, foram definidos quatro cenários de teste usados para avaliar os algoritmos planejadores de rotas estudados. Os testes foram definidos de forma a explorar aspectos distintos do problema de planejamento em ambientes com obstáculos estáticos e dinâmicos e para permitir a comparação adequada entre as abordagens implementadas. Os cenários variam em complexidade e no tipo de desafio imposto ao planejador.

- **Teste da trajetória retangular**

O primeiro caso de teste consiste em uma trajetória retangular a ser percorrida pelo robô com dois robôs obstáculos estáticos posicionados no meio do caminho. O objetivo deste teste é avaliar o comportamento básico dos planejadores em uma situação simples porém representativa em que o caminho objetivo é intuitivo mas exige desvio de obstáculos estáticos. A topologia do cenário permite observar se o planejador gera um desvio eficiente, se preserva a suavidade da trajetória e qual o custo em comprimento e em tempo de planejamento para encontrar a solução. Além disso o caso favorece a comparação entre a trajetória planejada e a trajetória executada em simulação para verificar estabilidade do seguimento e ocorrência de correções de rota. Na Figura 9 é possível observar o grafo de visibilidade gerado pelo VG para o primeiro cenário de teste.

Figura 9 – Grafo de visibilidade - trajetória retangular

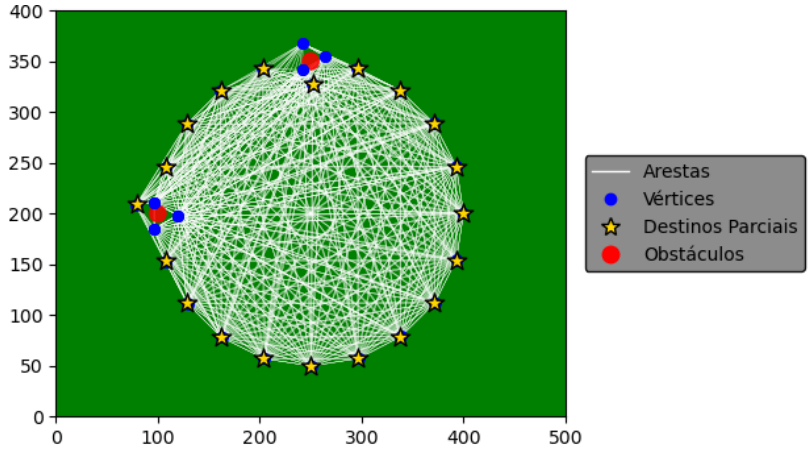


Fonte: Autoria própria

- **Teste da trajetória Circular**

O segundo caso de teste consiste em uma trajetória circular construída de forma aproximada por um polígono com inúmeros lados, com dois obstáculos estáticos ocupando trechos do percurso circular. Para que o robô percorra a trajetória, a estratégia adotada troca o alvo continuamente entre vértices consecutivos do polígono. Assim que o robô alcança ou fica bem próximo do vértice alvo, este é substituído pelo próximo vértice ao longo do polígono de modo a reproduzir de forma aproximada um seguimento contínuo do contorno circular sem exigir que o planejador produza curvas analíticas. Essa é uma forma de avaliar como os planejadores estudados reagem a atualização frequente dos *checkpoints* em espaço e tempo e se o algoritmo é capaz de manter a trajetória sem gerar oscilações excessivas ou correções bruscas de rota ao mesmo tempo em que o robô precisa desviar de obstáculos que intersectam o percurso. Na Figura 10 é possível observar o grafo de visibilidade gerado pelo VG para o segundo cenário de teste.

Figura 10 – Grafo de visibilidade - trajetória circular

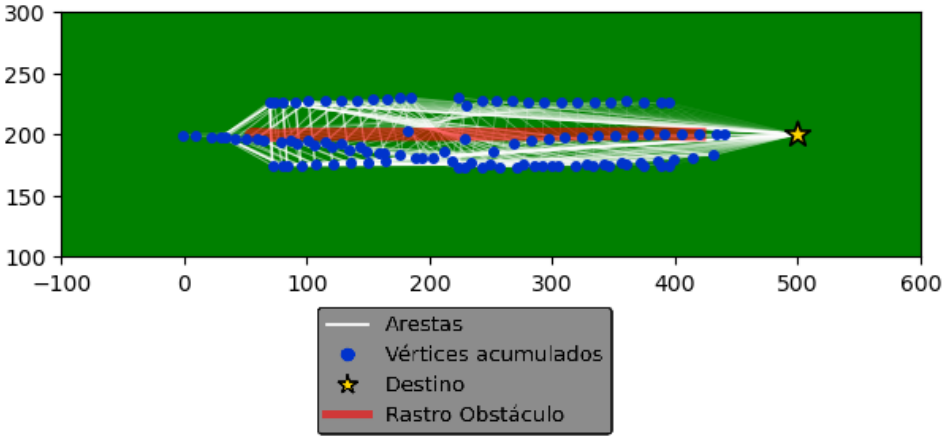


Fonte: Autoria própria

- **Teste da reta com obstáculo móvel**

Este cenário reproduz uma situação real de jogo em que o robô atacante avança em linha reta em direção ao campo de ataque enquanto o zagueiro adversário se desloca em sentido contrário para interceptar sua rota. O atacante segue um traçado retilíneo e o adversário é modelado como um obstáculo móvel que se aproxima da interseção das trajetórias. O teste expõe os planejadores à necessidade de ajustar a rota no momento certo para evitar a colisão com o zagueiro sem interromper o avanço do atacante. A configuração exige planejamento com origem em movimento e adaptações à medida que o obstáculo se aproxima, revelando limitações de algoritmos que não atualizam o plano de forma incremental. Na prática a situação evidencia comportamentos distintos entre os algoritmos e dentre as limitações, destaca-se a dificuldade do *Dijkstra* e do A* em evitar colisões com obstáculos dinâmicos e a maior frequência de replanejamentos em abordagens reativas, como é o caso do LPA*. Na Figura 11 é possível observar o grafo de visibilidade gerado pelo VG para o terceiro cenário de teste. Nota-se que para este caso está sendo apresentado o acumulado dos grafos gerados para cada uma das vezes que os algoritmos que possuem buscas incrementais forçaram a atualização do grafo gerado pelo VG. Vale ressaltar que como a quantidade de nós e arestas gerados é muito elevado e não seria possível visualizar todo o conjunto em uma única imagem, está sendo representado apenas uma amostra de 20% de todos os traços gerados, para isso foi uma capturada uma amostragem das arestas e vértices do grafo a cada 5 passos.

Figura 11 – Grafo de visibilidade - Reta com obstáculo móvel



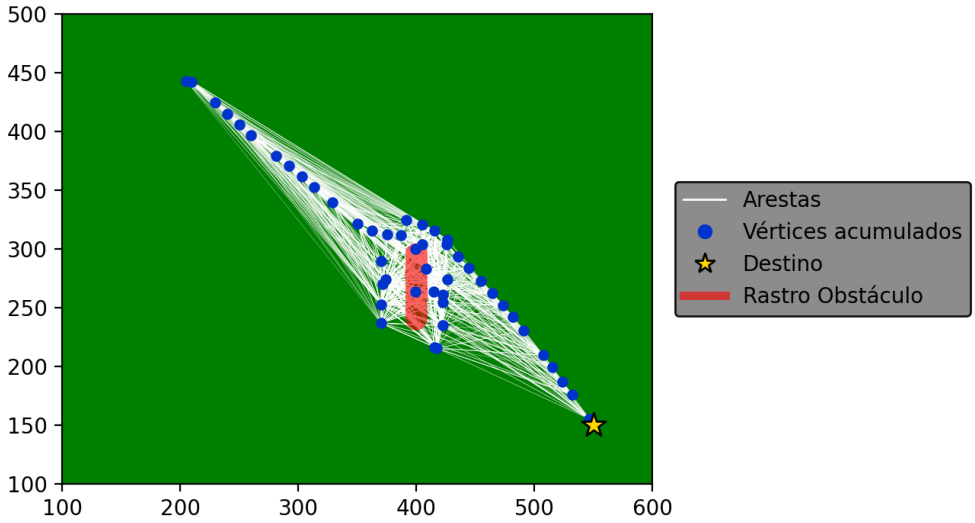
Fonte: Autoria própria

- **Teste do chute com obstáculo móvel em zigue-zague**

Por fim, o último caso de teste reproduz outra situação real de jogo em que o atacante se desloca em direção à bola para executar um chute ao gol enquanto um zagueiro se comportando como um obstáculo móvel percorre um padrão em zigue-zague no meio do trajeto do atacante. Ou seja, o robô atacante mantém um movimento direcionado à bola

e o obstáculo realiza variações laterais que aumentam a imprevisibilidade do encontro. O objetivo do teste é avaliar se os planejadores são capazes de conservar o avanço do atacante e ao mesmo tempo ajustar a rota diante de um obstáculo com movimento imprevisível, desviando do obstáculo e mantendo o caminho sem provocar manobras bruscas que comprometam a execução do chute. Assim como no teste da reta com obstáculo móvel, a configuração exige planejamento com origem em movimento e adaptações contínuas ao espaço livre, mas adiciona o desafio da oscilação lateral rápida do obstáculo. Na Figura 12 é possível observar o grafo de visibilidade gerado pelo VG para o último cenário de teste. Assim como na Figura 11, está sendo apresentado o acumulado dos grafos gerados para cada uma das vezes que os algoritmos que possuem buscas incrementais forçaram a atualização do grafo gerado pelo VG e é exibido apenas uma amostra de 20% de todos os traços gerados para tornar a imagem representativa e com uma visualização clara.

Figura 12 – Grafo de visibilidade - Chute com obstáculo móvel



Fonte: Autoria própria

Vale ressaltar que os testes foram executados mantendo tudo constante, exceto o algoritmo planejador de rotas. O simulador grSim foi utilizado para todos os cenários e o mesmo grafo gerado pelo VG serviu de base para todos os planejadores. A infraestrutura de execução reutilizou a mesma rotina de percurso e a mesma função de seguimento de trajetória, de modo que o comportamento do controlador não influenciasse as comparações. Todas as configurações do robô no simulador permaneceram idênticas entre execuções, incluindo limites máximos de velocidade linear e angular e limites de aceleração, passo de simulação e frequência de atualização do mapa e do planejador. Para garantir isonomia entre os algoritmos cada cenário foi repetido em múltiplas execuções e os dados brutos foram registrados de forma padronizada. Assim foi possível garantir que as diferenças observadas fossem devidas exclusivamente ao método de planejamento e não a variações na configuração do simulador ou do robô.

Os critérios de avaliação adotados nos cenários de teste foram definidos para captar diferentes dimensões do desempenho dos planejadores, desde a qualidade geométrica das soluções até a eficiência temporal e a capacidade de operar em ambientes dinâmicos. A seguir será descrito cada critério com a sua respectiva justificativa de utilização.

Um dos critérios de teste foi o tamanho do caminho que quantifica a extensão do trajeto gerado pelo planejador e percorrido pelo robô. Caminhos mais curtos indicam maior eficiência geométrica e tendem a reduzir energia e tempo de deslocamento quando as velocidades são semelhantes. Além disso comparar comprimentos ajuda a identificar comportamentos indesejados como voltas desnecessárias ou caminhos muito tortuosos gerados por aproximações do espaço ou heurísticas inadequadas. Essa métrica fornece um referencial direto para comparar a qualidade das soluções entre os algoritmos.

Outra métrica usada para comparação foi o tempo para cálculo do *Path Planning*, que mede o intervalo entre a solicitação de planejamento e o retorno de um caminho válido. Esse indicador é decisivo para entender a aplicabilidade do método em cenários com restrição temporal, como é o caso do futebol de robôs, pois um algoritmo rápido pode ser preferível a um algoritmo ligeiramente melhor caso o tempo de resposta for inadequado.

Complementando essas medidas, avaliou-se o tempo para execução da trajetória, que corresponde ao tempo que o robô leva para seguir o caminho planejado até o objetivo sob as mesmas condições de controle, velocidade e aceleração. Esse critério permite observar que percursos com muitas manobras ou replanejamentos tendem a aumentar o tempo de execução mesmo quando o comprimento é parecido. Vale ressaltar que na prática, décimos de segundo podem fazer a diferença para garantir que o robô chegue até o destino, por exemplo a bola, antes dos robôs adversários e tenha vantagem durante o jogo.

Além das métricas temporais e geométricas, foi realizada uma comparação visual entre o caminho planejado e o caminho executado por meio de visualizações gráficas. A avaliação visual permite identificar de forma mais clara qual foi o caminho planejado pelo algoritmo e além disso, observar se o caminho é algo factível para ser percorrido pelo robô. Como se trata de um simulador, assume-se um controlador de baixo nível ideal, portanto os resultados não possuem influência dessa variável.

Também foi avaliada a capacidade de desviar dos obstáculos, ou seja a habilidade do planejador em gerar trajetos livres de colisão e com margens de segurança adequadas em presença de obstáculos estáticos e móveis. Esse critério observa se as manobras são coerentes com um comportamento seguro em campo, por exemplo se o desvio é eficiente sem comprometer o avanço do robô. Em cenários dinâmicos essa métrica ganha importância porque a utilidade prática de um planejador depende da sua capacidade de antecipar ou reagir a mudanças no espaço livre.

Para os planejadores incrementais LPA* e *D* Lite* incluíram se métricas adicionais que registram o número de replanejamentos e o tempo total gasto com o cálculo das trajetórias ao longo da execução. Essas medidas mostram o custo da reatividade do algoritmo, pois

um alto número de replanejamentos pode indicar sensibilidade a pequenas variações e levar a sobrecarga computacional. O tempo acumulado de planejamento evidencia o impacto dessa reatividade durante a missão e permite comparar soluções que possuem um fator de replanejamento muito sensível com outras que reagem menos frequentemente, mas com ajustes mais eficazes.

Complementarmente aos quatro casos de teste, foram executadas algumas simulações de partidas completas utilizando a mesma estratégia de jogo para ambos times e fixa para todos os jogos, variando apenas o algoritmo de planejamento de rotas empregado em cada partida. A proposta desse ensaio final é observar o comportamento dos planejadores em uma situação contínua, dinâmica e repleta de interações simultâneas entre múltiplos robôs, ambiente condizente com a realidade da liga. Para essa comparação foram escolhidos dois algoritmos: o *Dijkstra*, que representa o modelo atualmente utilizado pela equipe *Red Dragons*, e o *D* Lite,* que apresentou o melhor desempenho entre os métodos avaliados nos cenários anteriores.

Para cada partida foi construída uma nuvem de pontos correspondente às colisões detectadas ao longo do jogo, possibilitando visualizar de forma direta a diferença de comportamento entre os planejadores quando submetidos a um ambiente com movimentações simultâneas e imprevisíveis, disputas pela bola, reposicionamentos frequentes e incertezas naturais das partidas da SSL. Esse teste funciona como a etapa mais abrangente do estudo, pois integra todos os elementos da dinâmica de jogo e permite uma avaliação chave das consequências práticas do uso de cada algoritmo. Os resultados obtidos são apresentados e discutidos detalhadamente na Seção 4.

4 Resultados

Este capítulo apresenta os resultados obtidos a partir dos testes definidos no estudo, organizados de forma a destacar o comportamento dos algoritmos em diferentes situações de navegação. Cada cenário foi executado com os quatro planejadores avaliados e gerou registros gráficos e métricas que permitem observar o desempenho de cada método. A disposição dos resultados busca facilitar a comparação visual e numérica entre os algoritmos e evidenciar as diferenças que surgem quando o ambiente exige respostas a obstáculos dinâmicos.

4.1 Resultados por cenário de teste

Em cada cenário de teste são apresentados dois conjuntos de gráficos que possibilitam a visualização do comportamento dos algoritmos. O primeiro mostrará a trajetória planejada no espaço plano, com vista superior, destacando o caminho escolhido pelo algoritmo, a real trajetória executada e a relação com os obstáculos presentes no ambiente. O segundo apresentará a evolução da posição do robô e dos obstáculos ao longo do tempo, permitindo observar a execução da trajetória e eventuais colisões causadas pelas características de cada planejador. A interpretação conjunta desses gráficos facilita a identificação de diferenças entre os algoritmos e torna mais clara a forma como cada um deles reage às condições impostas em cada teste.

- **Teste da trajetória retangular**

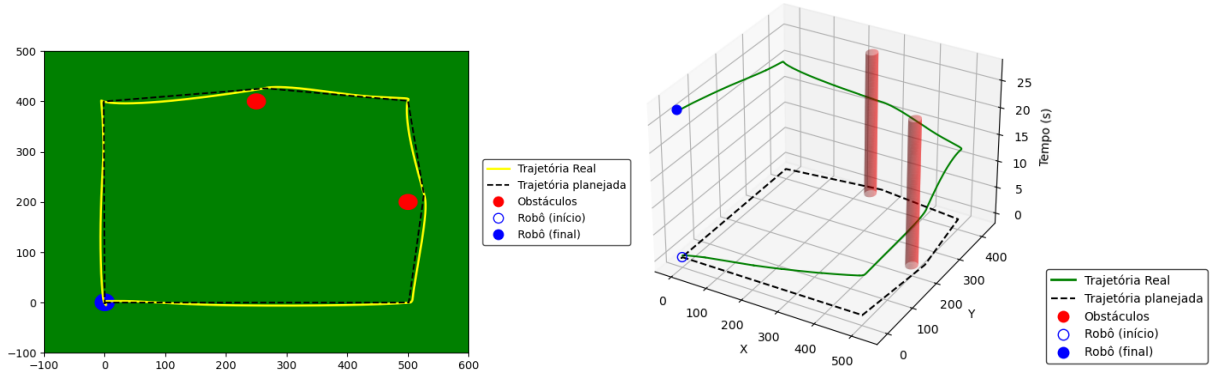
No teste da trajetória retangular, os quatro algoritmos apresentaram trajetos semelhantes nas visualizações bidimensionais apresentadas nas Figuras 13a, 14a, 15a e 16a. Como os obstáculos do ambiente são estáticos, todos os algoritmos foram capazes de gerar uma trajetória desviando de forma eficiente e sem necessidade de ajustes ao longo do percurso. Apesar disso, é possível notar algumas diferenças na suavidade da trajetória, sendo que as curvas planejadas pelo *Dijkstra* e pelo A* são mais agudas em relação aos outros algoritmos.

As curvas de trajetória ao longo do tempo nas Figuras 13b, 14b, 15b e 16b também mostram comportamentos praticamente equivalentes. Nesse tipo de cenário estático, a estrutura do grafo domina o comportamento do planejador, o que explica a proximidade entre as soluções.

Apesar dessa semelhança visual, existem diferenças mensuráveis, como o tamanho final da trajetória e o tempo de cálculo do planejamento para cada algoritmo. Esses pontos

não são visíveis diretamente nas figuras, mas serão discutidos nas análises quantitativas apresentadas posteriormente.

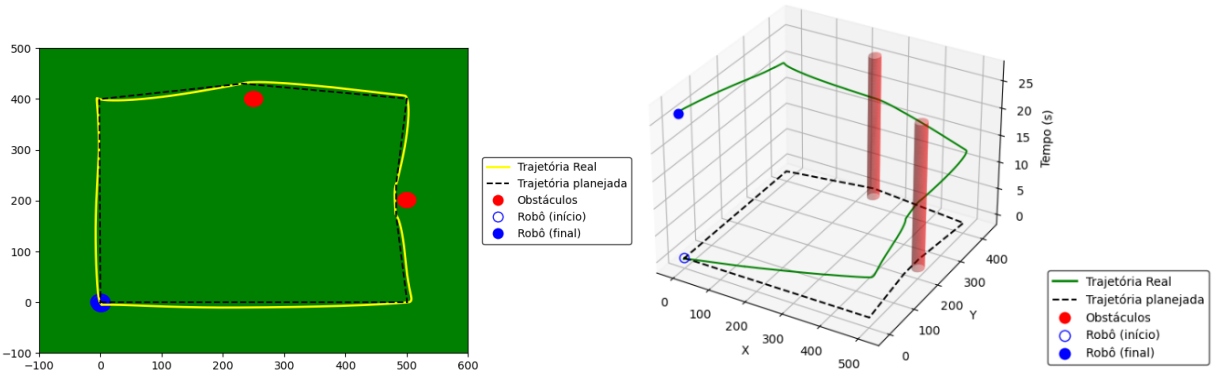
Figura 13 – Teste de trajetória retangular - *Dijkstra*



(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

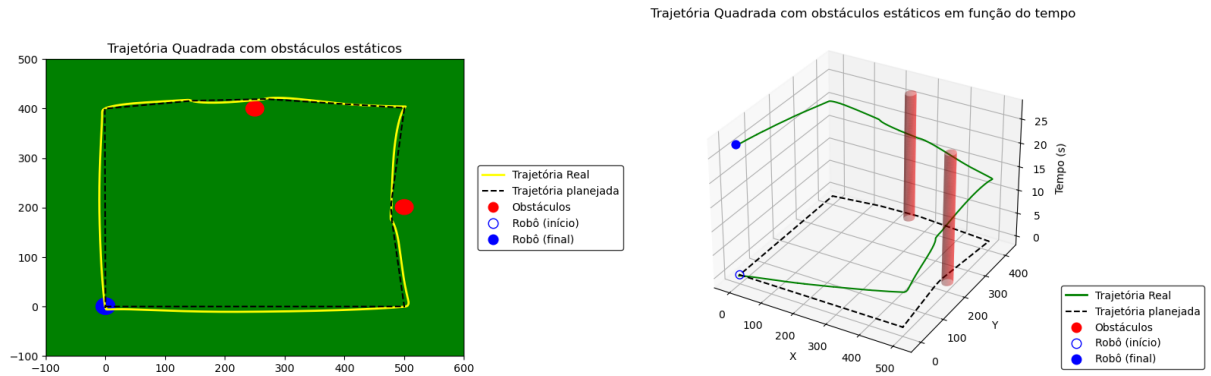
Figura 14 – Teste de trajetória retangular - A*



(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

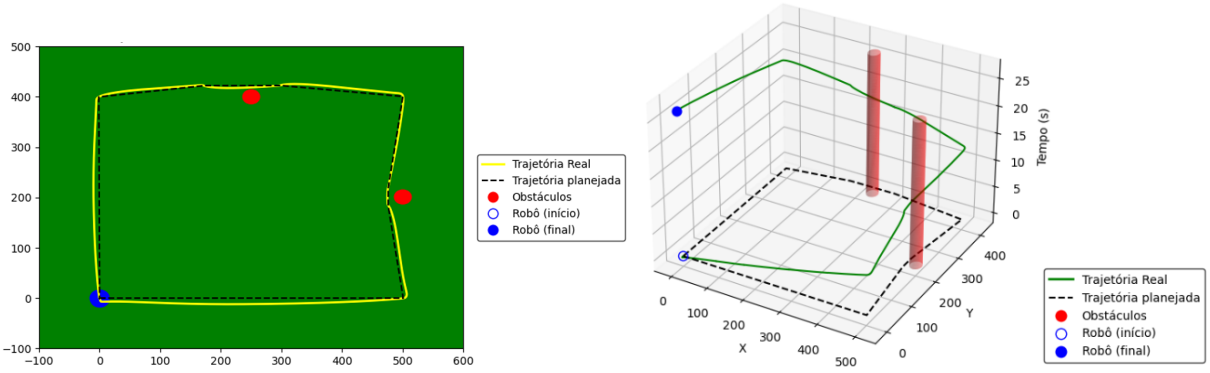
Figura 15 – Teste de trajetória retangular - LPA*



(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

Figura 16 – Teste de trajetória retangular - *D* Lite*



(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

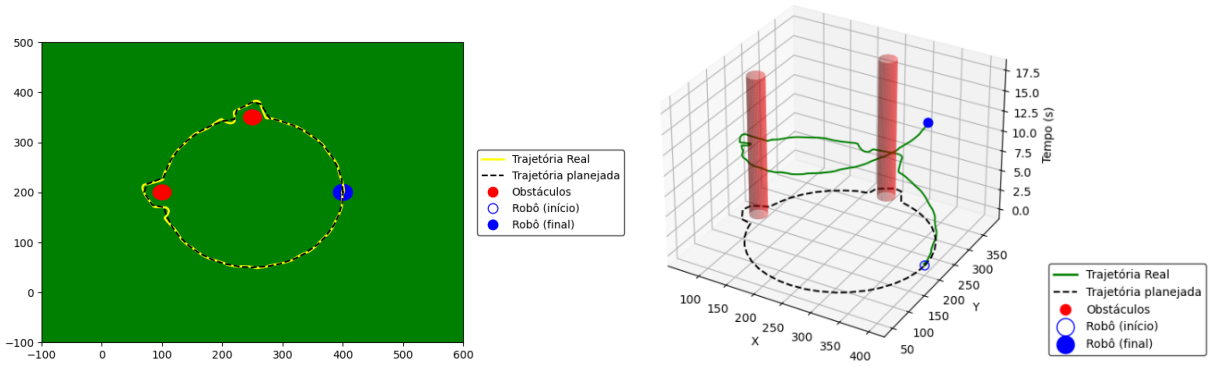
- **Teste da trajetória Circular**

No teste da trajetória circular, os quatro algoritmos apresentaram trajetos semelhantes de maneira geral, como mostrado nas Figuras 17a, 18a, 19a e 20a. Porém, algumas diferenças de comportamento entre os planejadores ficam mais evidentes devido ao formato curvo da trajetória e à mudança contínua de destino para a execução do círculo. Dentre os pontos a serem destacados, nota-se que o A* gerou um desvio mais agudo ao contornar os obstáculos, produzindo uma curva menos suave. O *Dijkstra* apresentou dificuldade para restabelecer a rota ideal após o desvio, o que resultou em um caminho ligeiramente mais longo. E o LPA* exibiu um comportamento levemente oscilatório logo após contornar os obstáculos.

Nas curvas de trajetória ao longo do tempo, ilustradas nas Figuras 17b, 18b, 19b e 20b, essas diferenças também podem ser percebidas em detalhes, embora o comportamento global ainda permaneça parecido entre os algoritmos. Assim como no teste retangular, o ambiente estático faz com que a estrutura do grafo seja o principal fator determinante do formato final do trajeto, o que explica a proximidade geral entre as soluções encontradas.

Apesar da similaridade visual, métricas como o comprimento final da trajetória e o tempo de cálculo do planejamento diferem entre os algoritmos e serão analisadas nos resultados quantitativos apresentados posteriormente.

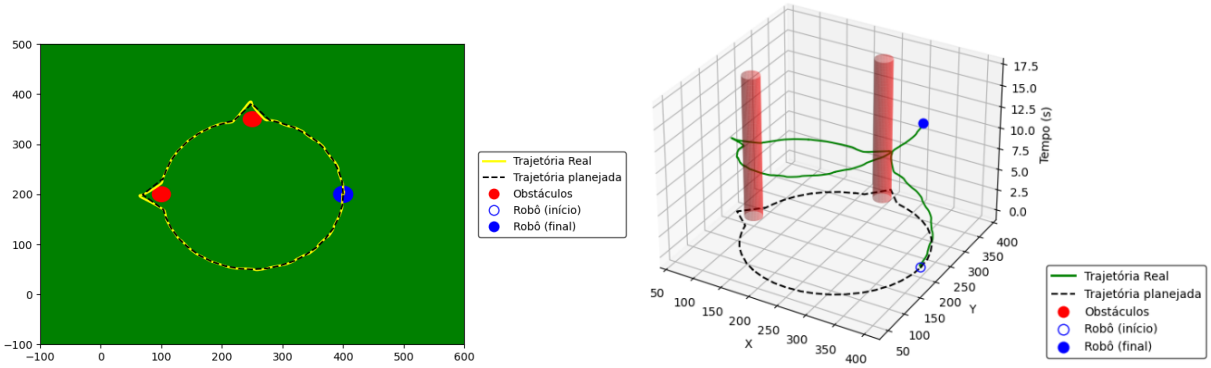
Figura 17 – Teste de trajetória circular - Dijkstra



(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

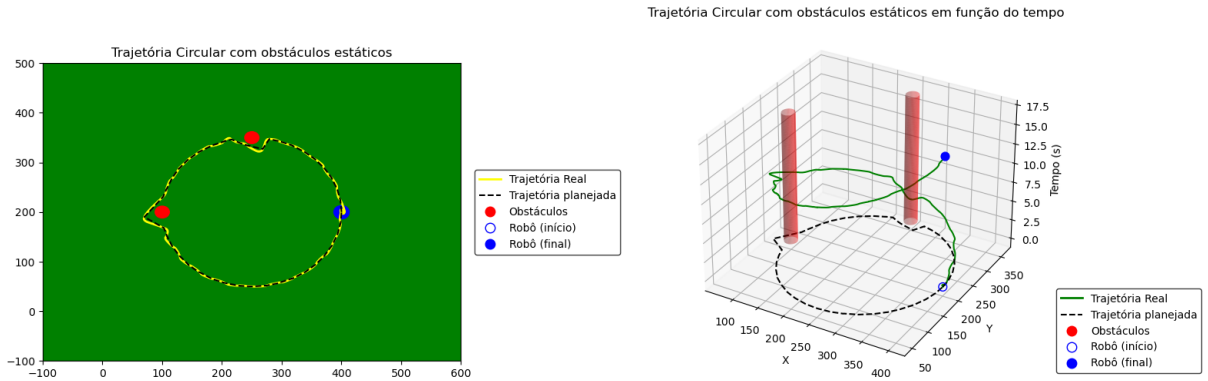
Figura 18 – Teste de trajetória circular - A*



(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

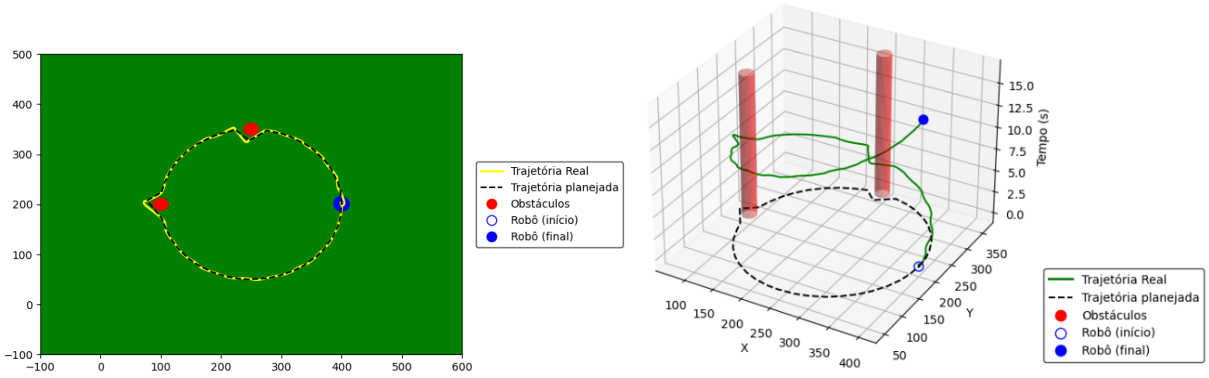
Figura 19 – Teste de trajetória circular - LPA*



(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

Figura 20 – Teste de trajetória circular - *D* Lite*



(a) Visualização 2D. (b) Trajetória x Tempo.

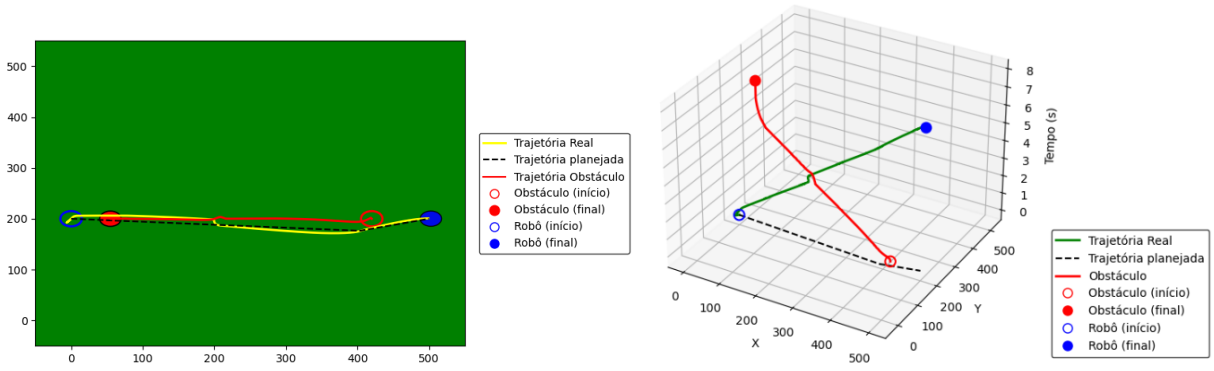
Fonte: Autoria própria.

- **Teste da reta com obstáculo móvel**

Este cenário apresenta uma mudança importante em relação aos anteriores, pois o obstáculo passa a se mover durante a trajetória, modificando continuamente o espaço livre e exigindo que o planejador reaja às novas condições do ambiente. Esse teste evidencia claramente como cada algoritmo lida com atualizações na configuração do ambiente, revelando suas limitações e pontos fortes no contexto de situações dinâmicas e mais próximas das encontradas em partidas reais da SSL.

Para o caso do teste realizado com o *Dijkstra*, o algoritmo em sua essência está configurado para calcular o caminho apenas no instante inicial, considerando a posição original do obstáculo. Essa característica pode ser observada na visualização superior da Figura 21a, em que o robô segue a rota planejada sem levar em consideração o deslocamento posterior do obstáculo. Quando o obstáculo se desloca e passa a bloquear o trajeto, o robô não possui mecanismos para replanejar sua rota ou avaliar um novo caminho seguro, resultando em uma colisão e causando uma mudança brusca na trajetória executada. Isso fica evidente também na Figura 21b, que mostra o robô permanecendo por um período de tempo na mesma posição pois o obstáculo estava obstruindo o caminho. Esse aspecto ilustra a limitação do método quando utilizado em ambientes que se modificam ao longo do tempo.

Figura 21 – Teste de trajetória reta com obstáculo móvel - *Dijkstra*

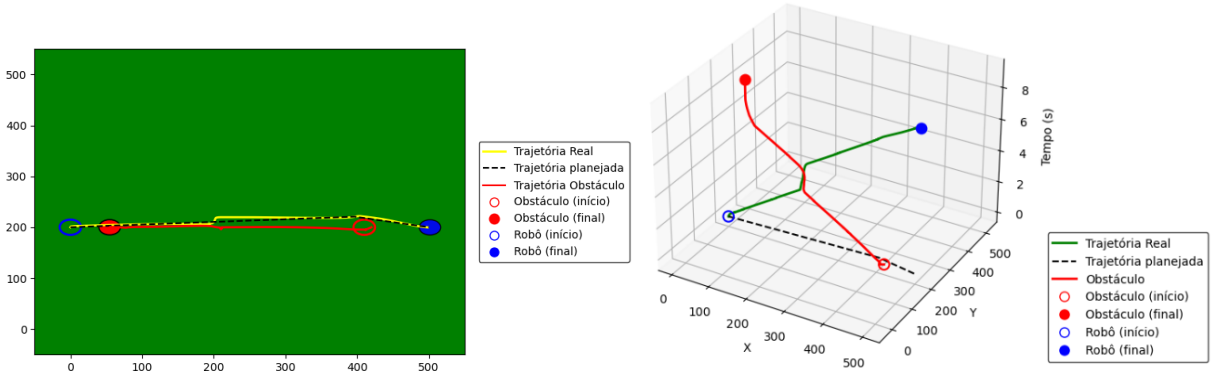


(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

Neste caso, o algoritmo A*, sem atualizações, apresenta um comportamento muito semelhante. A trajetória é planejada apenas uma vez, considerando o obstáculo em sua posição inicial, como é possível observar na Figura 22a. Quando o obstáculo se desloca, o caminho previamente planejado se torna inviável e o robô não encontra alternativa para desviar, resultando novamente em uma colisão. A Figura 22b apresenta claramente o momento em que o robô encontra o obstáculo e permanece travado no mesmo ponto até conseguir se livrar do obstáculo, reafirmando dificuldades semelhantes ao *Dijkstra* diante de ambientes dinâmicos.

Figura 22 – Teste de trajetória reta com obstáculo móvel - A*



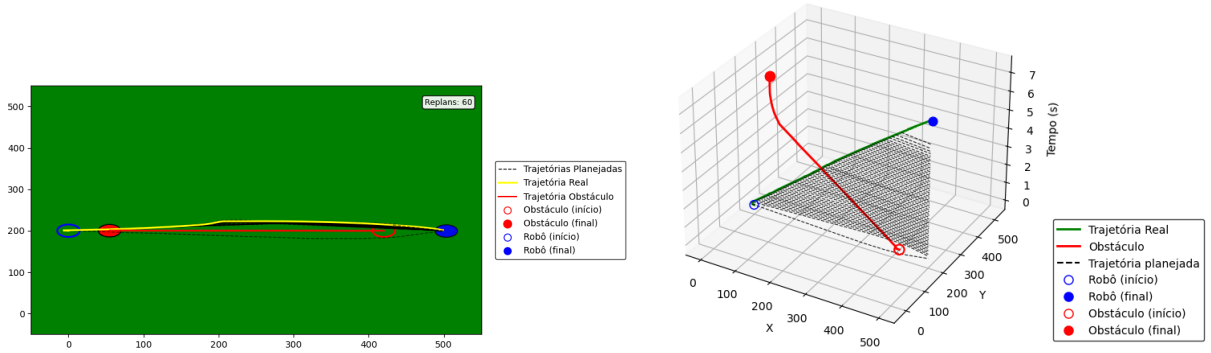
(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

O LPA* apresenta um comportamento distinto, já que atualiza o caminho sempre que o grafo sofre alterações através de uma atualização incremental. Conforme o obstáculo se movimenta, o algoritmo detecta a mudança e replaneja apenas os trechos necessários para encontrar uma nova rota viável, como visto na Figura 23a. Esse processo resulta em múltiplos pequenos ajustes na trajetória ao longo do tempo, gerando um comportamento visualmente caracterizado por correções sucessivas. A Figura 23b deixa claro esse padrão, mostrando uma grande quantidade de trajetórias planejadas ao longo do tempo, o que reflete a quantidade maior de reavaliações realizadas a cada nova mudança no ambiente.

Porém destaca-se que apesar dessa característica, o algoritmo foi capaz de proporcionar o sucesso do robô para desviar do obstáculo que estava vindo em sua direção de forma eficiente em todas as repetições do teste.

Figura 23 – Teste de trajetória reta com obstáculo móvel - LPA*

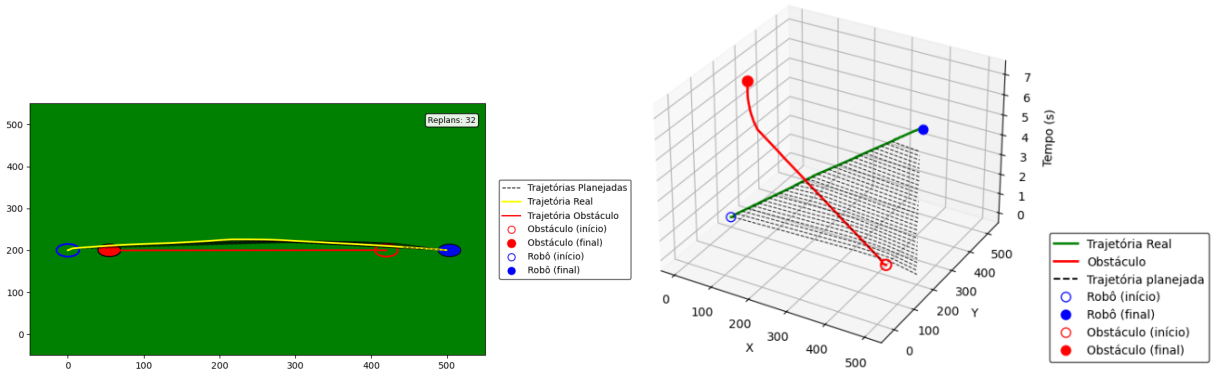


(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

O D* *Lite* apresentou uma característica geral de trajetória bastante semelhante à do LPA*, ajustando o caminho de forma incremental conforme o obstáculo se movia. Assim como no caso anterior, o algoritmo foi capaz de reposicionar a trajetória sempre que necessário para manter o desvio do obstáculo com sucesso, como pode ser observado na Figura 24a. Entretanto, a principal diferença está na quantidade de reavaliações realizadas durante o percurso. O D* *Lite* é menos sensível a variações no ambiente e precisa recalcular o caminho menos vezes, como fica evidente na quantidade de trajetórias geradas ao longo do tempo apresentada na Figura 24b em comparação com a 23b. Esse comportamento reduz o esforço computacional e torna o processo mais eficiente, mantendo a adaptação ao cenário dinâmico sem gerar oscilações frequentes no trajeto.

Figura 24 – Teste de trajetória reta com obstáculo móvel - *D* Lite*



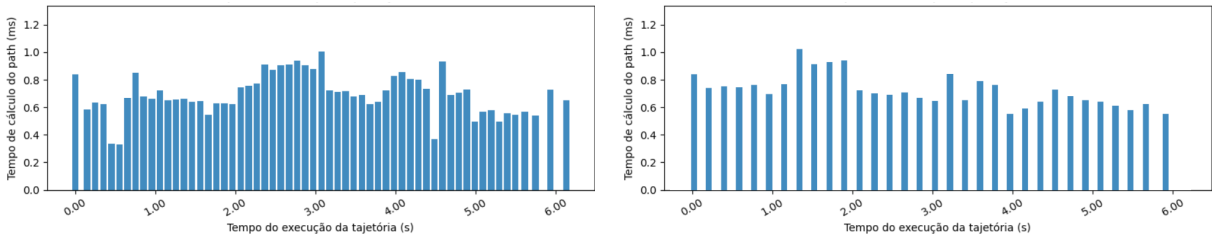
(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

O espectro dos tempos de planejamento apresentados na Figura 25a e na Figura 25b reforça a diferença de comportamento entre o LPA* e o D* *Lite* durante o cenário com

obstáculo móvel. No LPA*, o tempo total gasto com planejamento ao longo da trajetória é maior, resultado da maior quantidade de atualizações realizadas a cada mudança detectada no ambiente. Já o D* *Lite* consegue se adaptar ao movimento do obstáculo demandando menos atualizações e menos processamento acumulado. Esses resultados mostram que, embora ambos os algoritmos tenham conseguido ajustar suas rotas e evitar a colisão, o D* *Lite* alcançou esse desempenho com menor custo computacional ao longo da execução.

Figura 25 – Comparação de tempo de *Path Planning* na trajetória reta com obstáculo móvel



(a) LPA* (b) *D* Lite*

Fonte: Autoria própria.

De forma geral, este teste destaca a diferença fundamental entre algoritmos que atualizam a trajetória ao longo da execução e aqueles que apenas calculam a trajetória uma única vez. Enquanto *Dijkstra* e A* estão definidos de forma dependente das condições iniciais do ambiente e acabam colidindo diante da mudança do obstáculo, o LPA* e o D* *Lite* se adaptam ao movimento, preservando a navegabilidade até o objetivo final. As diferenças quantitativas entre os algoritmos, considerando métricas como número de replanejamentos, tempo total de planejamento e qualidade da trajetória serão apresentadas posteriormente.

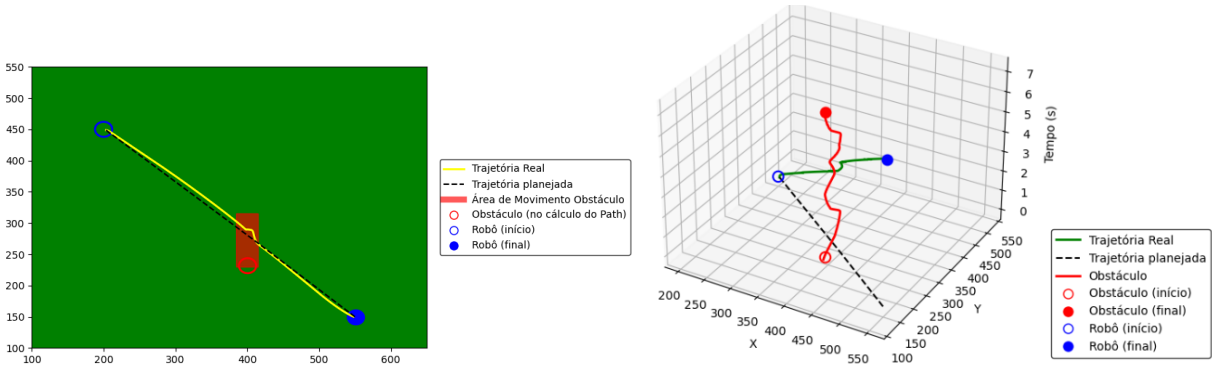
- **Teste do chute com obstáculo móvel em zigue-zague**

Este cenário mantém o objetivo de avaliar o comportamento dos algoritmos diante de obstáculos dinâmicos, porém agora o robô precisa se deslocar até uma posição de chute específica enquanto o obstáculo executa um movimento em zigue-zague. O teste cria uma situação mais próxima de uma jogada real, em que o defensor muda rapidamente de direção e força o planejador a reagir em pouco tempo. Como esperado, os resultados gerais são semelhantes ao experimento anterior, mas a variação mais brusca no movimento do obstáculo tornou alguns padrões mais evidentes e facilitou observar a eficiência de cada método em se adaptar à mudança no ambiente.

No teste realizado com *Dijkstra*, como o algoritmo calcula a trajetória apenas no início, o caminho planejado não reflete o deslocamento posterior do obstáculo. Assim, quando o robô encontra o obstáculo no novo posicionamento, ocorre uma colisão e uma mudança abrupta na trajetória executada fica visível na visualização superior da Figura

26a. A colisão também fica notável na Figura 26b e confirma a limitação do método em ambientes dinâmicos.

Figura 26 – Teste do chute com obstáculo móvel - *Dijkstra*

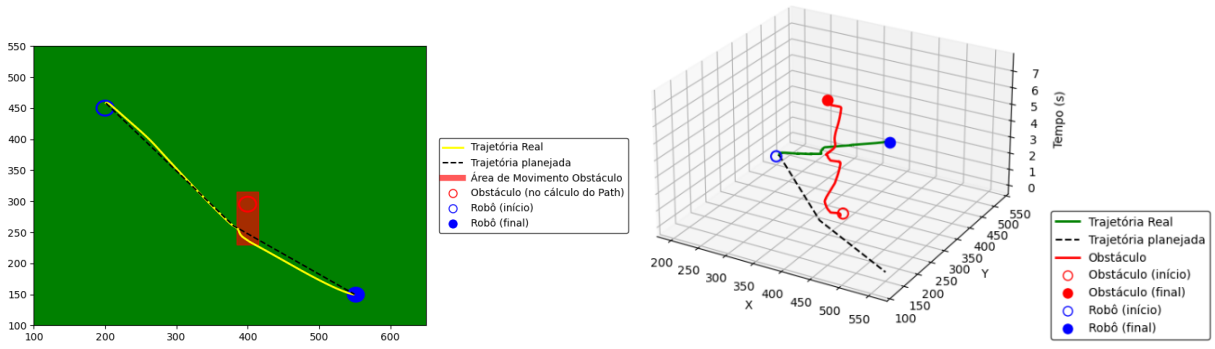


(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

O A* apresentou comportamento muito semelhante ao de *Dijkstra*, já que também não realiza replanejamento ao longo da execução. A trajetória planejada leva em conta apenas a posição original do obstáculo e, quando este se desloca para bloquear o caminho, o robô não consegue reagir adequadamente, como pode ser observado na Figura 27a. Assim como no caso anterior, a mudança brusca de direção indica o momento em que a colisão ocorre. Inclusive, na Figura 27b é possível notar que no momento em que o robô ficou em colisão, a posição se manteve inalterada por determinado período de tempo.

Figura 27 – Teste do chute com obstáculo móvel - A*

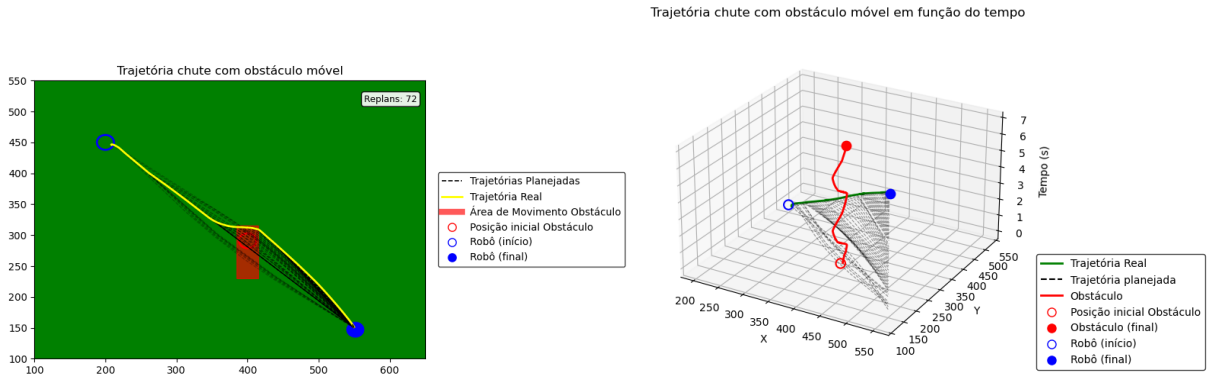


(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

No caso do LPA*, o algoritmo atualiza o caminho sempre que detecta mudanças no grafo, realizando pequenos replanejamentos incrementais. À medida que o obstáculo executa o movimento em zigue-zague, o LPA* adapta a trajetória progressivamente, como mostrado nas Figuras 28a e 28b. Isso resulta em uma rota composta por várias pequenas correções, mas que mantém distância segura do obstáculo e garante que o robô alcance o objetivo sem colisões.

Figura 28 – Teste do chute com obstáculo móvel - LPA*

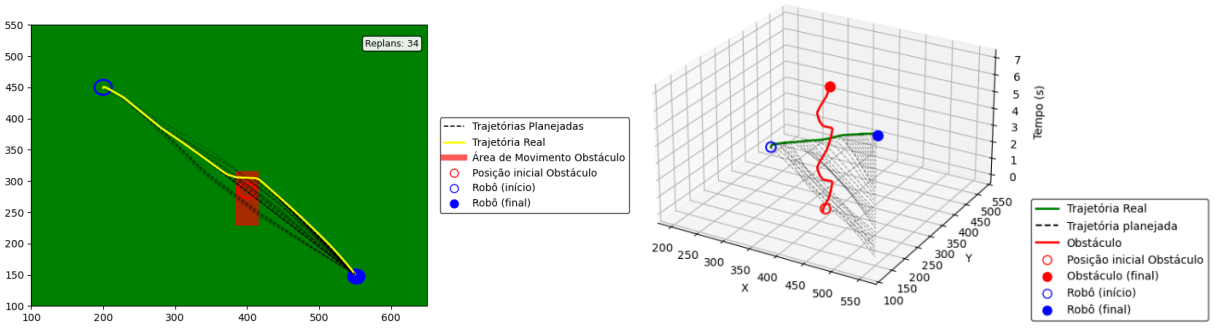


(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

Por fim, o *D* Lite* apresentou um comportamento muito próximo ao LPA*, mas com menos atualizações ao longo do percurso. A Figura 29a mostra que o algoritmo também ajusta o caminho conforme o obstáculo se move, porém de maneira mais econômica em termos computacionais quando comparado ao LPA*, já que só reavalia os trechos necessários. Isso fica mais evidente ao comparar a quantidade de trajetórias geradas na Figura 29b em relação a 28b. Assim, o robô desvia do obstáculo com sucesso, mas realizando menos correções ao longo do deslocamento.

Figura 29 – Teste do chute com obstáculo móvel - *D* Lite*

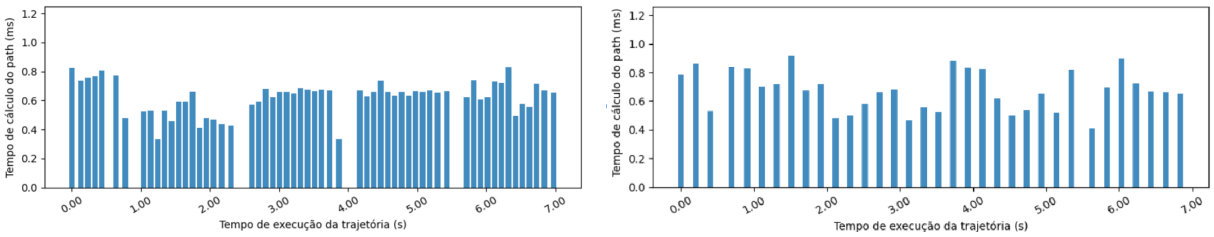


(a) Visualização 2D. (b) Trajetória x Tempo.

Fonte: Autoria própria.

A comparação dos espectros do tempo de planejamento apresentados nas Figuras 30a e 30b reforça a maior frequência de replanejamento do LPA* durante o deslocamento do obstáculo. Entretanto, para este caso é interessante destacar que a frequência de replanejamentos diminui nos momentos em que o robô alcança as extremidades do movimento de zigue-zague, já que as mudanças de posição do obstáculo são irrelevantes nesses pontos. Já o *D* Lite,* assim como no outro cenário dinâmico, mantém um número reduzido de atualizações ao longo da execução, ajustando a trajetória apenas quando necessário e apresentando menor custo computacional acumulado.

Figura 30 – Comparação de tempo de *Path Planning* no chute com obstáculo móvel



(a) LPA* (b) *D* Lite*

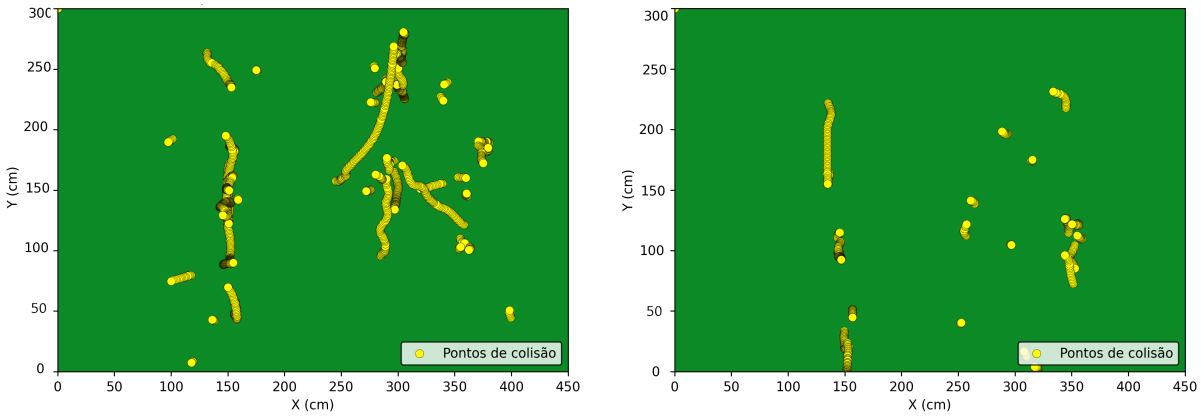
Fonte: Autoria própria.

De modo geral, os resultados reforçam a diferença entre algoritmos que replanejam sua rota durante a execução e aqueles que mantêm o caminho calculado apenas no início. *Dijkstra* e A* não são eficientes para lidar com o movimento em zigue-zague do obstáculo e acabam colidindo, enquanto LPA* e *D* Lite* se adaptam ao ambiente dinâmico e concluem o trajeto com sucesso, cada um com estratégias e frequências de atualização distintas. As análises quantitativas serão apresentadas posteriormente.

- **Partida de SSL**

Após a execução de partidas de dois minutos utilizando a mesma estratégia, porém com planejadores de rotas diferentes, é possível analisar a nuvem de pontos de colisão apresentada nas Figuras 31a e 31b, evidenciando o impacto direto da escolha do planejador de rotas no comportamento global da equipe durante uma partida. Como o simulador grSim não fornece diretamente a informação de colisão entre robôs, foi adotado um critério geométrico para sua identificação, considerando a ocorrência de colisão quando dois robôs ocupam posições com distância inferior a $1mm$ entre si. No jogo executado com VG + *Dijkstra*, observa-se uma concentração significativamente maior de colisões distribuídas ao longo do campo, formando trajetórias contínuas de contatos sucessivos entre robôs. Esse padrão ocorre principalmente em situações de disputa de espaço, em que dois ou mais robôs se deslocam lado a lado tentando alcançar a mesma região. Como o *Dijkstra* não atualiza seu planejamento diante da movimentação dos oponentes, o robô frequentemente insiste em seguir sua rota original, mesmo quando há outro robô ocupando aquela área, o que resulta em sequências prolongadas de colisões.

Figura 31 – Nuvem de pontos de colisões durante um jogo



(a) Jogo com VG + *Dijkstra*

(b) Jogo com VG + *D* Lite*

Fonte: Autoria própria.

Como é evidenciado na Figura 31b, no caso do jogo com VG + *D* Lite*, a nuvem de pontos revela uma quantidade bem menor de colisões e uma distribuição mais pontual. Embora ainda existam conflitos naturais do jogo, especialmente nos momentos de disputa pela bola, o D* *Lite* tende a ajustar sua trajetória quando identifica que um oponente bloqueia a rota prevista, reduzindo a insistência em caminhos inviáveis e evitando contatos repetitivos. As cadeias contínuas de colisões vistas no primeiro caso aparecem de forma muito mais limitada, indicando que o algoritmo consegue reposicionar o robô de maneira eficiente para contornar situações congestionadas. Portanto, os resultados mostram que a capacidade de adaptação do D* *Lite* reduz significativamente a duração e a frequência das colisões mais prolongadas, o que contribui para partidas mais limpas e com uma maior fluidez de movimentação

4.2 Análise dos resultados

As análises quantitativas apresentadas nesta seção complementam as discussões feitas anteriormente sobre o comportamento visual dos algoritmos em cada cenário de teste. Para garantir maior confiabilidade estatística, cada experimento foi repetido 5 vezes para cada cenário considerado, sendo os resultados apresentados correspondentes aos valores médios obtidos. Os valores consolidados para todos os cenários estão apresentados na Tabela 1, enquanto a Tabela 3 traz métricas adicionais, específicas para os algoritmos incrementais.

Tabela 1 – Resumo de resultados obtidos por cenário de teste

Métrica	Algoritmo	Trajetória retangular	Trajetória circular	Reta com obstáculo móvel	Chute com obstáculo móvel	Variação
Tempo cálculo Path Planning (ms)	Dijkstra	2,35	2,33	1,61	1,30	+169%
	A*	0,98	1,04	0,41	0,40	
	LPA*	1,22	1,28	0,67	0,66	**+35%**
	D* *Lite*	1,24	1,27	0,72	0,71	+39%
Tempo execução trajetória (s)	Dijkstra	26,80	17,49	7,74	7,93	+5,26%
	A*	26,65	16,72	7,25	8,25	+3,34%
	LPA*	26,78	16,64	7,32	7,06	+1,46%
	D* *Lite*	26,59	16,25	7,20	6,93	
Tamanho da trajetória (cm)	Dijkstra	1471,43	862,63	425,93	377,25	+2,89%
	A*	1465,66	835,67	421,57	371,97	+1,50%
	LPA*	1467,44	833,26	411,09	377,22	+1,31%
	D* *Lite*	1461,55	813,09	402,94	371,59	
Desvio de obstáculos	Dijkstra	Sim	Sim	Não	Não	
	A*	Sim	Sim	Não	Não	
	LPA*	Sim	Sim	Sim	Sim	
	D* *Lite*	Sim	Sim	Sim	Sim	

Para fins estatísticos, segue abaixo a Tabela 2, que apresenta o desvio padrão de cada um dos valores obtidos na tabela 1.

Tabela 2 – Desvio padrão dos resultados por cenário de teste

Métrica	Algoritmo	Trajetória retangular	Trajetória circular	Reta com obstáculo móvel	Chute com obstáculo móvel
Tempo cálculo Path Planning (ms)	Dijkstra	0,12	0,02	0,11	0,09
	A*	0,09	0,02	0,10	0,05
	LPA*	0,19	0,02	0,03	0,01
	D* *Lite*	0,17	0,04	0,06	0,03
Tempo execução trajetória (s)	Dijkstra	0,09	0,16	0,14	0,05
	A*	0,04	0,27	0,22	0,12
	LPA*	0,08	0,23	0,04	0,07
	D* *Lite*	0,09	0,20	0,03	0,14
Tamanho da trajetória (cm)	Dijkstra	1,18	6,54	8,63	8,55
	A*	3,67	4,49	6,99	6,36
	LPA*	3,94	3,79	1,33	2,31
	D* *Lite*	2,01	5,75	0,82	1,89

Os valores de desvio padrão obtidos indicam que os resultados apresentam baixa variabilidade entre as execuções, mantendo um comportamento consistente ao longo dos diferentes cenários analisados. De modo geral, a dispersão observada é reduzida, o que reforça a repetibilidade dos experimentos e a confiabilidade dos dados utilizados na comparação entre os algoritmos.

A partir da Tabela 1, observando o tempo de cálculo do *Path Planning*, nota-se que A* apresentou os menores valores em todos os cenários. Isso acontece porque o algoritmo trabalha sobre o mesmo grafo estático que *Dijkstra*, mas com uma heurística admissível que direciona a busca e diminui o número de nós expandidos. Em cenários estáticos isso se traduz não apenas em trajetórias equivalentes às de *Dijkstra*, mas em uma redução significativa no tempo de cálculo. Os algoritmos incrementais analisados, LPA* e *D* Lite,* ficaram em uma faixa intermediária, pois calculam o caminho de forma semelhante

ao A* na primeira execução, com custo ligeiramente maior devido à estrutura adicional de armazenamento necessária para permitir atualizações posteriores.

Quando analisado o tempo total de execução da trajetória, o comportamento se mantém relativamente próximo entre os algoritmos nos cenários estáticos, porém com ligeira vantagem para o caminho executado pelo *D* Lite*. Nos cenários com obstáculos móveis, o algoritmo apresenta de forma mais acentuada os melhores tempos de execução da trajetória, refletindo o fato de conseguir se adaptar ao movimento do obstáculo mantendo rota viável, sem interrupções e com poucas revisões no caminho. Vale ressaltar que no caso de *Dijkstra* e A*, houveram inclusive testes em que o robô ficou um período de tempo em colisão até conseguir prosseguir, o que reflete diretamente em tempos maiores de execução.

O tamanho final das trajetórias também indica vantagem do *D* Lite,* que obtém sistematicamente o menor comprimento de rota nos quatro cenários. Isso acontece porque, embora o planejador utilize a mesma base de grafo, o mecanismo incremental de atualização permite que o algoritmo ajuste a solução para rotas mais curtas conforme o obstáculo se movimenta, evitando desvios exagerados ou trajetórias não uniformes.

Nota-se que última coluna da tabela apresenta a variação percentual em relação ao melhor resultado obtido em cada métrica, servindo como uma forma direta de comparação entre os algoritmos, permitindo uma visualização de maneira mais imediata do quanto cada método ficou acima da melhor solução alcançada em cada caso.

Por fim, no critério mais importante para cenários dinâmicos, que é a capacidade de desviar do obstáculo em movimento, observa-se uma divisão clara, já que *Dijkstra* e A* apresentam sucesso apenas nos cenários estáticos, enquanto falham nos testes com movimentos de obstáculo, exatamente devido ao fato de não haver atualização da rota durante a execução. Já LPA* e D* *Lite* foram capazes de contornar o obstáculo com sucesso nos quatro testes, destacando sua adequação para ambientes que mudam ao longo do tempo.

A Tabela 3 apresenta métricas focadas em detalhar as diferenças de comportamento dos dois algoritmos incrementais. O número de replanejamentos necessários para completar os testes foi significativamente menor para o *D* Lite*, quase o dobro de diferença em ambos os cenários dinâmicos. Com menos replanejamentos, o custo acumulado de computação também foi reduzido, o que explica os menores tempos totais de planejamento, mesmo garantindo rotas mais eficientes durante a execução. Já o LPA* conseguiu completar os testes com segurança, mas realizou mais reavaliações sucessivas do caminho, refletindo em maior custo de processamento ao longo da trajetória.

Tabela 3 – Resultados complementares para os algoritmos LPA* e *D* Lite*

Métrica	**Algoritmo**	**Reta com obstáculo móvel**	**Chute com obstáculo móvel**
Número médio de Replans	LPA*	59	62
	D* *Lite*	32	34
Tempo total de Path Planning (ms)	LPA*	40,23	40,33
	D* *Lite*	23,17	24,03

Do ponto de vista geral, os resultados quantitativos observados nos quatro cenários de teste confirmam as interpretações visuais apresentadas anteriormente e trazem evidências imperceptíveis ao olho humano que reforçam os conceitos teóricos estudados. O consolidado dos testes mostram um panorama claro sobre o comportamento de cada abordagem e evidenciam diferenças que nem sempre aparecem apenas pela observação visual. O desempenho superior dos algoritmos incrementais em ambientes dinâmicos, a consistência do *D* Lite* em obter as melhores trajetórias e os ganhos de eficiência no tempo de cálculo formam um conjunto sólido de conclusões que valida os experimentos realizados. Essa combinação de resultados oferece uma síntese clara e confiável sobre o potencial e as limitações de cada algoritmo.

As partidas que foram executadas com os diferentes algoritmos reforçam esse panorama, já que reproduzem uma situação real de jogo com os múltiplos agentes disputando espaço e interagindo continuamente. A diferença observada no número e no padrão das colisões entre os dois planejadores testados confirma na prática o que foi identificado nos testes controlados.

5 Conclusões

A conclusão geral deste trabalho garante que os objetivos propostos foram alcançados. A implementação e comparação dos algoritmos de planejamento de rotas sobre o mesmo grafo gerado pelo VG permitiram analisar o comportamento de cada método em cenários que representam situações reais da SSL. Os resultados, reunindo tanto a análise visual quanto as métricas quantitativas, mostram de forma clara como cada abordagem reage aos diferentes tipos de trajetória e aos desafios impostos por obstáculos móveis. Esse conjunto forma um panorama consistente sobre o potencial e as limitações de cada algoritmo dentro do contexto estudado e gera uma base de consulta para a *Red Dragons* e outras equipes da SSL.

Um desafio encontrado durante o desenvolvimento foi a necessidade de adaptar cada planejador à estrutura do VG e à estratégia já utilizada pela equipe. Foi necessário estruturar o código de maneira modular para que os planejadores funcionassem no modelo *plug and play*, permitindo a troca imediata do algoritmo sem interferir nas demais camadas do sistema. Esse esforço fundamental para isolar a variável de teste, garantindo que as diferenças observadas nos resultados fossem decorrentes exclusivamente da lógica de planejamento e não de outras variáveis do sistema.

Os resultados mostraram diferenças marcantes entre os algoritmos, especialmente nos cenários dinâmicos. O *Dijkstra* e o A* tiveram bom desempenho nos testes estáticos, mas não conseguiram reagir à movimentação dos obstáculos, o que resultou em colisões e trajetórias interrompidas. Por outro lado, os algoritmos incrementais mostraram uma vantagem clara. O LPA* foi eficiente para ajustar a rota sempre que o ambiente mudava, garantindo a chegada ao objetivo, embora com muitas revisões sucessivas do caminho, enquanto o *D* Lite* se destacou como a abordagem mais equilibrada entre qualidade da trajetória, adaptação ao cenário e eficiência computacional, apresentando os melhores resultados gerais. Esses comportamentos foram reforçados também nas partidas completas, cujas nuvens de colisões mostraram, na prática, a diferença entre utilizar um planejador estático e um planejador incremental no ambiente de jogo.

Do ponto de vista aplicado, o estudo fornece um conjunto de informações que pode apoiar decisões internas da equipe *Red Dragons* ou de outras equipes da liga, sobre a evolução do seu sistema de planejamento de rotas. Já no aspecto científico, o trabalho contribui ao apresentar uma comparação detalhada entre algoritmos clássicos operando sobre a mesma estrutura de grafo, algo que pode ser útil também para outras aplicações de robótica móvel.

Como continuidade natural deste estudo, alguns caminhos se mostram especialmente relevantes. O primeiro é aplicar os planejadores implementados para testes em robôs

físicos, validando na prática como os resultados se comportam fora do ambiente simulado. Além disso, uma evolução importante consiste na incorporação de técnicas de estimação de estado, como o *Extended Kalman Filter* (EKF), permitindo a previsão da posição dos robôs e reduzindo a necessidade de reconstrução frequente do grafo a cada atualização do ambiente.

Outro ponto relevante é a realização de um estudo comparativo entre diferentes formas de representação do ambiente, mantendo fixo um algoritmo de planejamento e avaliando o impacto de abordagens como VG, grades ocupacionais ou métodos baseados em amostragem. Essa análise complementa diretamente o presente trabalho, no qual foi adotada a estratégia inversa, fixando a representação do grafo e variando os algoritmos de busca. Esses caminhos podem ampliar ainda mais o alcance dos resultados e aprofundar a compreensão sobre o papel da modelagem do ambiente e da predição no desempenho do planejamento de rotas.

Em suma, o trabalho consolida uma visão clara sobre as vantagens e limites das abordagens estudadas, fornecendo uma base prática e acessível para futuras decisões dentro das equipes de SSL e para aplicações semelhantes em outros contextos de robótica móvel.

Referências Bibliográficas

AL-MUTIB, K. et al. D* lite based real-time multi-agent path planning in dynamic environments. In: *2011 Third International Conference on Computational Intelligence, Modelling & Simulation*. [S.l.: s.n.], 2011. p. 170–174.

AZIZI, H.; SULAIMANY, S. A review of visibility graph analysis. *IEEE Access*, IEEE, v. 12, p. 93517–93530, 2024.

CHOUDHARY, A. Sampling-based path planning algorithms: A survey. *arXiv preprint arXiv:2304.14839*, 2023.

CHU, L. et al. Intelligent vehicle path planning based on optimized A* algorithm. *Sensors*, MDPI, v. 24, n. 10, p. 3149, 2024.

COSTA, L. da S.; TONIDANDEL, F. Comparison and analysis of the DVG+A* and rapidly-exploring random trees path-planners for the RoboCup-Small Size League. In: *2019 Latin American Robotics Symposium (LARS), 2019 Brazilian Symposium on Robotics (SBR) and 2019 Workshop on Robotics in Education (WRE)*. [S.l.: s.n.], 2019. p. 1–6.

DHULKEFL, E.; DURDU, A.; TERZIOĞLU, H. Dijkstra algorithm using UAV path planning. *Konya Journal of Engineering Sciences*, Konya Technical University, v. 8, p. 92–105, 2020.

DUMITRUF, P. et al. 2020 team description paper: UBC Thunderbots. In: UNIVERSITY OF BRITISH COLUMBIA. *RoboCup 2020 Team Description Papers*. Vancouver, Canada, 2020.

D'AMATO, E. et al. A visibility graph approach for path planning and real-time collision avoidance on maritime unmanned systems. In: *2021 International Workshop on Metrology for the Sea; Learning to Measure Sea Health Parameters (MetroSea)*. [S.l.: s.n.], 2021. p. 400–405.

FADZLI, S. A. et al. Robotic indoor path planning using Dijkstra's algorithm with multi-layer dictionaries. In: *2015 2nd International Conference on Information Science and Security (ICISS)*. [S.l.: s.n.], 2015. p. 1–4.

KALISCH, M.; PANFIL, W. Autonomous control of the Small Size League robots group. In: IEEE. *2015 10th International Workshop on Robot Motion and Control (RoMoCo)*. [S.l.], 2015. p. 7–14.

KALUDER, H.; BREZAK, M.; PETROVIC, I. A visibility graph based method for path planning in dynamic environments. In: *2011 Proceedings of the 34th International Convention MIPRO*. [S.l.: s.n.], 2011. p. 717–721.

KATONA, K.; NEAMAH, H. A.; KORONDI, P. Obstacle avoidance and path planning methods for autonomous navigation of mobile robot. *Sensors*, MDPI, v. 24, n. 11, p. 3573, 2024.

KOENIG, S.; LIKHACHEV, M. D* lite. In: *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI)*. Edmonton, Alberta, Canada: AAAI Press, 2002. p. 476–483.

LEE, W.; CHOI, G.-H.; KIM, T. wan. Visibility graph-based path-planning algorithm with quadtree representation. *Applied Ocean Research*, v. 117, p. 102887, 2021. ISSN 0141-1187. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0141118721003588>.

LI, D.; NIU, K. Dijkstra's algorithm in AGV. In: *2014 9th IEEE Conference on Industrial Electronics and Applications*. [S.l.: s.n.], 2014. p. 1867–1871.

LIM, J.; SALZMAN, O.; TSIOTRAS, P. Class-ordered LPA*: An incremental-search algorithm for weighted colored graphs. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. [S.l.: s.n.], 2021. p. 6907–6913.

LIU, W. et al. A novel graph-based motion planner of multi-mobile robot systems with formation and obstacle constraints. *IEEE Transactions on Robotics*, v. 40, p. 714–728, 2024.

LU, Y. et al. Incremental multi-scale search algorithm for dynamic path planning with low worst-case complexity. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, v. 41, n. 6, p. 1556–1570, 2011.

MARANHÃO, A. et al. Itandroids Small Size League team description paper for RoboCup 2020. *RoboCup*, 2020.

MAXIMO, M. et al. Path planning based on visibility graph with circular obstacles for RoboCup Small Size League. In: . [S.l.: s.n.], 2025. p. 176–181.

MONAJJEMI, V.; KOOCHAKZADEH, A.; GHIDARY, S. S. grSim – Robocup Small Size Robot Soccer Simulator. In: RÖFER, T. et al. (Ed.). *RoboCup 2011: Robot Soccer World Cup XV*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 450–460.

NILAVAR, A. C. et al. Implementation of a navigational path planning algorithm for an autonomous mobile robot. In: *2021 IEEE India Council International Subsections Conference (INDISCON)*. [S.l.: s.n.], 2021. p. 1–5.

OKUMUŞ, F.; KOCAMAZ, A. F. Comparing path planning algorithms for multiple mobile robots. In: *2018 International Conference on Artificial Intelligence and Data Processing (IDAP)*. [S.l.: s.n.], 2018. p. 1–4.

REYES, N. H.; BARCZAK, A. L.; SUSNIAK, T. Autonomous navigation in partially known confounding maze-like terrains using D*Lite with poisoned reverse. In: *2018 World Symposium on Digital Intelligence for Systems and Machines (DISA)*. [S.l.: s.n.], 2018. p. 67–76.

ROBOCUP. *RoboCup Small Size League Rules and Regulations*. 2025. <https://ssl.robocup.org/>. Acesso em: 01 nov. 2025.

SÁNCHEZ-IBÁÑEZ, J. R.; PULGAR, C. J. Pérez-del; GARCÍA-CEREZO, A. Path planning for autonomous mobile robots: A review. *Sensors*, MDPI, v. 21, n. 23, p. 7898, 2021.

SILVA, A. et al. Roboime: On the road to RoboCup 2023. *RoboCup*, 2023.

TRINH, L. et al. Turtlerabbit 2024 SSL team description paper. *RoboCup*, 2024.

WAGA, A. et al. A survey on autonomous navigation for mobile robots: From traditional techniques to deep learning and large language models. *Journal of King Saud University Computer and Information Sciences*, Springer, v. 37, n. 7, p. 198, 2025.

YAO, J. Path planning algorithm of indoor mobile robot based on ROS system. In: *2023 IEEE International Conference on Image Processing and Computer Applications (ICIPCA)*. [S.l.: s.n.], 2023. p. 523–529.

YU, J. et al. Improved D*Lite algorithm path planning in complex environment. In: *2020 Chinese Automation Congress (CAC)*. [S.l.: s.n.], 2020. p. 2226–2230.

ZICKLER, S. et al. Ssl-vision: The shared vision system for the RoboCup Small Size League. In: *Robot Soccer World Cup*. [S.l.]: Springer, 2009. p. 425–436.

A Implementação do Algoritmo Dijkstra

```python
import numpy as np
import time

from commons.math import angle_between, rotate_vector, ortogonal_projection
from entities.Obstacle import Obstacle

IS_PYTHON_MODULE = False

if IS_PYTHON_MODULE:
    import pyvisgraph as vg
else:
    import path.cppvisgraph.build.cppyvisgraph as vg

class VisibilityGraphPlanner:
    """
    Planejador de rotas baseado no algoritmo Visibility Graph com busca de menor caminho utilizando Dijkstra.
    """

    def __init__(self) -> None:
        self.obstacle_map = vg.VisGraph()
        self.origin = vg.Point(0.0, 0.0)
        self.target = vg.Point(0.0, 0.0)

        self.vg_obstacles = []
        self.logger_obstacle = False

    # ------------------------------------------------------------------
    # Definição de origem e alvo
    # ------------------------------------------------------------------

    def set_origin(self, coordinates: np.ndarray) -> None:
        """
        Define o ponto inicial do planejamento.
        """
        self.origin = vg.Point(float(coordinates[0]), float(coordinates[1]))

    def set_target(self, coordinates: np.ndarray) -> None:
```

```
39          """
40          Define o ponto final do planejamento.
41          """
42          self.target = vg.Point(float(coordinates[0]), float(coordinates
    [1]))
43
44      # ------------------------------------------------------------------
45      # Modelagem dos obstáculos
46      # ------------------------------------------------------------------
47
48      def robot_triangle_obstacle(self, obstacle: Obstacle, robot) -> list
    :
49          """
50          Gera um polígono triangular que circunscreve o obstáculo,
    orientado em relação à posição do robô.
51          Esse polígono é utilizado como obstáculo no grafo de
    visibilidade.
52          """
53          obst_coords = obstacle.get_coordinates()
54          obst_coords = np.array([obst_coords.X, obst_coords.Y])
55
56          robot_coords = robot.get_coordinates()
57          robot_coords = np.array([robot_coords.X, robot_coords.Y])
58
59          r = obstacle.radius
60
61          # Triângulo padrão centrado na origem
62          p1 = np.array([ r, -np.sqrt(3) * r])
63          p2 = np.array([ r,  np.sqrt(3) * r])
64          p3 = np.array([-2 * r, 0.0])
65
66          # Ângulo entre obstáculo e robô
67          ref_vector = np.array([1.0, 0.0])
68          theta = angle_between(robot_coords - obst_coords, ref_vector)
69
70          # Rotação do triângulo
71          p1 = rotate_vector(p1, theta)
72          p2 = rotate_vector(p2, theta)
73          p3 = rotate_vector(p3, theta)
74
75          # Translação para a posição do obstáculo
76          p1 += obst_coords
77          p2 += obst_coords
78          p3 += obst_coords
79
80          return [p1, p2, p3]
81
```

```
82      def convert_to_vgPoly(self, points: list) -> list:
83          """
84          Converte uma lista de pontos numpy em pontos do Visibility Graph
    .
85          """
86          return [vg.Point(float(p[0]), float(p[1])) for p in points]
87
88      # ------------------------------------------------------------------
89      # Construção do grafo de visibilidade
90      # ------------------------------------------------------------------
91
92      def update_obstacle_map(self) -> None:
93          """
94          Constrói o grafo de visibilidade a partir da lista de obstáculos
95          previamente definida.
96          """
97          self.obstacle_map = vg.VisGraph()
98
99          if len(self.vg_obstacles) == 0:
100             return
101
102         if IS_PYTHON_MODULE:
103             self.obstacle_map.build(self.vg_obstacles, status=False,
    workers=1)
104         else:
105             self.obstacle_map.build(self.vg_obstacles)
106
107     # ------------------------------------------------------------------
108     # Planejamento de caminho
109     # ------------------------------------------------------------------
110
111     def get_path(self) -> list:
112         """
113         Calcula o menor caminho entre origem e alvo utilizando Dijkstra
114         sobre o grafo de visibilidade.
115         Retorna uma lista de pontos do Visibility Graph.
116         """
117         return self.obstacle_map.shortest_path(self.origin, self.target)
118
119     # ------------------------------------------------------------------
120     # Funções auxiliares de alto nível
121     # ------------------------------------------------------------------
122
123     def update_target_with_obstacles(self, robot, field, x_target,
    y_target, cont_target):
124         """
125         Atualiza o alvo considerando obstáculos estáticos e retorna
```

```
126         o próximo ponto do caminho planejado.
127         """
128         current_position = np.array([
129             robot.get_coordinates().X,
130             robot.get_coordinates().Y
131         ])
132 
133         current_target = np.array([
134             x_target[cont_target],
135             y_target[cont_target]
136         ])
137 
138         self.set_origin(current_position)
139         self.set_target(current_target)
140 
141         vg_obstacles = []
142         obstacles = robot.map_obstacle.get_map_obstacle()
143 
144         for obstacle in obstacles:
145             if not self.ignore_obstacle(robot, field.ball, obstacle):
146                 triangle = self.robot_triangle_obstacle(obstacle, robot)
147                 vg_triangle = self.convert_to_vgPoly(triangle)
148                 vg_obstacles.append(vg_triangle)
149 
150         self.vg_obstacles = vg_obstacles
151 
152         t_start = time.time()
153         self.update_obstacle_map()
154         t_end = time.time()
155 
156         if self.logger_obstacle:
157             print("Tempo de construção do grafo de visibilidade:",
158                   (t_end - t_start) * 1000, "ms")
159 
160         path = self.get_path()
161 
162         if path:
163             next_point = path[1] if len(path) > 1 else path[0]
164             return np.array([next_point.x, next_point.y])
165 
166         return current_target
167 
168     def ignore_obstacle(self, robot, ball, obstacle) -> bool:
169         """
170         Indica se um obstáculo deve ser desconsiderado com base na posiç
    ão relativa entre robô, bola e obstáculo.
171         """
```

```
172         p_robot = np.array([robot.get_coordinates().X, robot.
    get_coordinates().Y])
173         p_ball = np.array([ball.get_coordinates().X, ball.
    get_coordinates().Y])
174         p_obstacle = np.array([
175             obstacle.get_coordinates().X,
176             obstacle.get_coordinates().Y
177         ])
178 
179         _, t = ortogonal_projection(p_robot, p_ball, p_obstacle)
180 
181         return t > 1
```

B Implementação do Algoritmo A*

```python
import math
import heapq
import numpy as np
import time

from commons.math import angle_between, rotate_vector, ortogonal_projection
from entities.Obstacle import Obstacle


class VisibilityGraphAStarPlanner:
    """
    Planejador de rotas baseado no Visibility Graph utilizando o algoritmo A*.

    O grafo de visibilidade é construído considerando obstáculos, e o caminho é calculado por meio de busca heurística (A*), utilizando
    distância euclidiana como heurística admissível.
    """

    def __init__(self) -> None:
        self.origin = None
        self.target = None

        self.nodes = []
        self.node_index = {}
        self.adjacency = {}

        self.vg_obstacles = []
        self.logger_obstacle = False

    # ------------------------------------------------------------------
    # Definição de origem e alvo
    # ------------------------------------------------------------------

    def set_origin(self, coordinates: np.ndarray) -> None:
        """
        Define o ponto inicial do planejamento.
        """
        self.origin = (float(coordinates[0]), float(coordinates[1]))

    def set_target(self, coordinates: np.ndarray) -> None:
```

```
40          """
41          Define o ponto final do planejamento.
42          """
43          self.target = (float(coordinates[0]), float(coordinates[1]))
44
45      # ---------------------------------------------------------------
46      # Modelagem dos obstáculos
47      # ---------------------------------------------------------------
48
49      def robot_triangle_obstacle(self, obstacle: Obstacle, robot) -> list
    :
50          """
51          Gera um polígono triangular que circunscreve o obstáculo,
52          orientado em relação à posição do robô.
53          """
54          obst_coords = obstacle.get_coordinates()
55          obst_coords = np.array([obst_coords.X, obst_coords.Y])
56
57          robot_coords = robot.get_coordinates()
58          robot_coords = np.array([robot_coords.X, robot_coords.Y])
59
60          r = obstacle.radius
61
62          p1 = np.array([ r, -np.sqrt(3) * r])
63          p2 = np.array([ r,  np.sqrt(3) * r])
64          p3 = np.array([-2 * r, 0.0])
65
66          ref_vector = np.array([1.0, 0.0])
67          theta = angle_between(robot_coords - obst_coords, ref_vector)
68
69          p1 = rotate_vector(p1, theta)
70          p2 = rotate_vector(p2, theta)
71          p3 = rotate_vector(p3, theta)
72
73          p1 += obst_coords
74          p2 += obst_coords
75          p3 += obst_coords
76
77          return [p1, p2, p3]
78
79      def convert_to_vgPoly(self, points: list) -> list:
80          """
81          Converte uma lista de pontos numpy em tuplas (x, y).
82          """
83          return [(float(p[0]), float(p[1])) for p in points]
84
85      # ---------------------------------------------------------------
```

```
86      # Construção do grafo de visibilidade
87      # ------------------------------------------------------------------
88
89      def update_obstacle_map(self) -> None:
90          """
91          Constrói os nós e a lista de adjacência do grafo de visibilidade
        .
92          """
93          polygons = [self.convert_to_vgPoly(poly) for poly in self.
        vg_obstacles]
94
95          vertices = []
96          for poly in polygons:
97              for v in poly:
98                  if v not in vertices:
99                      vertices.append(v)
100
101         if self.origin not in vertices:
102             vertices.append(self.origin)
103         if self.target not in vertices:
104             vertices.append(self.target)
105
106         self.nodes = vertices
107         self.node_index = {v: i for i, v in enumerate(vertices)}
108         self.adjacency = {i: [] for i in range(len(vertices))}
109
110         edges = []
111         for poly in polygons:
112             n = len(poly)
113             for i in range(n):
114                 edges.append((poly[i], poly[(i + 1) % n]))
115
116         def segments_intersect(a, b, c, d):
117             def orient(p, q, r):
118                 return (q[0] - p[0]) * (r[1] - p[1]) - (q[1] - p[1]) * (
        r[0] - p[0])
119
120             o1 = orient(a, b, c)
121             o2 = orient(a, b, d)
122             o3 = orient(c, d, a)
123             o4 = orient(c, d, b)
124
125             return o1 * o2 < 0 and o3 * o4 < 0
126
127         for i in range(len(vertices)):
128             for j in range(i + 1, len(vertices)):
129                 a = vertices[i]
```

```
130                 b = vertices[j]
131
132                 visible = True
133                 for c, d in edges:
134                     if a in (c, d) or b in (c, d):
135                         continue
136                     if segments_intersect(a, b, c, d):
137                         visible = False
138                         break
139
140                 if visible:
141                     cost = math.hypot(a[0] - b[0], a[1] - b[1])
142                     self.adjacency[i].append((j, cost))
143                     self.adjacency[j].append((i, cost))
144
145     # ------------------------------------------------------------------
146     # Planejamento de caminho com A*
147     # ------------------------------------------------------------------
148
149     def get_path(self) -> list:
150         """
151         Calcula o caminho utilizando o algoritmo A* sobre o grafo de
    visibilidade.
152         """
153         start = self.node_index[self.origin]
154         goal = self.node_index[self.target]
155
156         def heuristic(i):
157             x, y = self.nodes[i]
158             gx, gy = self.nodes[goal]
159             return math.hypot(x - gx, y - gy)
160
161         open_set = []
162         heapq.heappush(open_set, (heuristic(start), start))
163
164         g_cost = {start: 0.0}
165         parent = {}
166         closed = set()
167
168         while open_set:
169             _, current = heapq.heappop(open_set)
170
171             if current == goal:
172                 break
173
174             if current in closed:
175                 continue
```

```
176
177             closed.add(current)
178
179             for neighbor, cost in self.adjacency[current]:
180                 tentative = g_cost[current] + cost
181
182                 if tentative < g_cost.get(neighbor, float("inf")):
183                     g_cost[neighbor] = tentative
184                     parent[neighbor] = current
185                     f = tentative + heuristic(neighbor)
186                     heapq.heappush(open_set, (f, neighbor))
187
188         if goal not in parent and goal != start:
189             return []
190
191         path = []
192         node = goal
193         while True:
194             path.append(self.nodes[node])
195             if node == start:
196                 break
197             node = parent[node]
198
199         path.reverse()
200         return path
201
202     # ------------------------------------------------------------------
203     # Interface de alto nível
204     # ------------------------------------------------------------------
205
206     def update_target_with_obstacles(self, robot, field, x_target,
    y_target, cont_target):
207         """
208         Atualiza o alvo considerando obstáculos estáticos e retorna
209         o próximo ponto do caminho planejado.
210         """
211         current_position = np.array([
212             robot.get_coordinates().X,
213             robot.get_coordinates().Y
214         ])
215
216         current_target = np.array([
217             x_target[cont_target],
218             y_target[cont_target]
219         ])
220
221         self.set_origin(current_position)
```

```
222         self.set_target(current_target)
223 
224         vg_obstacles = []
225         obstacles = robot.map_obstacle.get_map_obstacle()
226 
227         for obstacle in obstacles:
228             if not self.ignore_obstacle(robot, field.ball, obstacle):
229                 triangle = self.robot_triangle_obstacle(obstacle, robot)
230                 vg_obstacles.append(triangle)
231 
232         self.vg_obstacles = vg_obstacles
233 
234         t_start = time.time()
235         self.update_obstacle_map()
236         t_end = time.time()
237 
238         if self.logger_obstacle:
239             print("Tempo de construção do grafo:",
240                   (t_end - t_start) * 1000, "ms")
241 
242         path = self.get_path()
243 
244         if path:
245             return np.array(path[1] if len(path) > 1 else path[0])
246 
247         return current_target
248 
249     def ignore_obstacle(self, robot, ball, obstacle) -> bool:
250         """
251         Indica se um obstáculo pode ser ignorado com base na projeção
252         da posição da bola em relação ao robô.
253         """
254         p_robot = np.array([robot.get_coordinates().X, robot.
    get_coordinates().Y])
255         p_ball = np.array([ball.get_coordinates().X, ball.
    get_coordinates().Y])
256         p_obstacle = np.array([
257             obstacle.get_coordinates().X,
258             obstacle.get_coordinates().Y
259         ])
260 
261         _, t = ortogonal_projection(p_robot, p_ball, p_obstacle)
262         return t > 1
```

C Implementação do Algoritmo LPA*

```python
import math
import heapq
import time
import numpy as np
from collections import defaultdict

from commons.math import angle_between, rotate_vector, ortogonal_projection
from entities.Obstacle import Obstacle


class VisibilityGraphLPAStarPlanner:
    """
    Planejador de rotas baseado em Visibility Graph utilizando o algoritmo LPA* (Lifelong Planning A*).

    Diferentemente do A*, este algoritmo mantém informações intermediárias de custo (g e rhs), permitindo atualizações incrementais quando o grafo sofre modificações.
    """

    def __init__(self, debug=False) -> None:
        self.origin = None
        self.target = None

        self.nodes = []
        self.node_index = {}
        self.adjacency = defaultdict(list)
        self.predecessors = defaultdict(list)

        self.vg_obstacles = []

        # Estruturas centrais do LPA*
        self.g = {}
        self.rhs = {}
        self.open_list = []
        self.open_entry = {}

        self.start_idx = None
        self.goal_idx = None

```

```
38          self.debug = debug
39          self.tempo_total_path = 0.0
40
41      # ------------------------------------------------------------------
42      # Definição de origem e alvo
43      # ------------------------------------------------------------------
44
45      def set_origin(self, coordinates: np.ndarray) -> None:
46          self.origin = (float(coordinates[0]), float(coordinates[1]))
47          self.start_idx = self.node_index.get(self.origin, None)
48
49      def set_target(self, coordinates: np.ndarray) -> None:
50          self.target = (float(coordinates[0]), float(coordinates[1]))
51          self.goal_idx = self.node_index.get(self.target, None)
52
53      # ------------------------------------------------------------------
54      # Heurística e chave do LPA*
55      # ------------------------------------------------------------------
56
57      def heuristic(self, u_idx: int, v_idx: int) -> float:
58          ux, uy = self.nodes[u_idx]
59          vx, vy = self.nodes[v_idx]
60          return math.hypot(ux - vx, uy - vy)
61
62      def calculate_key(self, u_idx: int) -> tuple:
63          g_u = self.g.get(u_idx, float("inf"))
64          rhs_u = self.rhs.get(u_idx, float("inf"))
65          min_val = min(g_u, rhs_u)
66
67          if self.start_idx is None:
68              h = 0.0
69          else:
70              h = self.heuristic(self.start_idx, u_idx)
71
72          return (min_val + h, min_val)
73
74      # ------------------------------------------------------------------
75      # Fila de prioridade
76      # ------------------------------------------------------------------
77
78      def push_open(self, u_idx: int) -> None:
79          key = self.calculate_key(u_idx)
80          heapq.heappush(self.open_list, (key[0], key[1], u_idx))
81          self.open_entry[u_idx] = key
82
83      def pop_open(self):
84          while self.open_list:
```

```
            k0, k1, u = heapq.heappop(self.open_list)
            current = self.open_entry.get(u)
            if current is None:
                continue
            if (k0, k1) != current:
                continue
            del self.open_entry[u]
            return u
        return None

    def top_key(self) -> tuple:
        while self.open_list:
            k0, k1, u = self.open_list[0]
            current = self.open_entry.get(u)
            if current is None or current != (k0, k1):
                heapq.heappop(self.open_list)
                continue
            return (k0, k1)
        return (float("inf"), float("inf"))

    # ------------------------------------------------------------------
    # Modelagem dos obstáculos
    # ------------------------------------------------------------------

    def robot_triangle_obstacle(self, obstacle: Obstacle, robot) -> list
    :
        obst_coords = obstacle.get_coordinates()
        obst_coords = np.array([obst_coords.X, obst_coords.Y])

        robot_coords = robot.get_coordinates()
        robot_coords = np.array([robot_coords.X, robot_coords.Y])

        r = obstacle.radius

        p1 = np.array([ r, -np.sqrt(3) * r])
        p2 = np.array([ r,  np.sqrt(3) * r])
        p3 = np.array([-2 * r, 0.0])

        ref_vector = np.array([1.0, 0.0])
        theta = angle_between(robot_coords - obst_coords, ref_vector)

        p1 = rotate_vector(p1, theta)
        p2 = rotate_vector(p2, theta)
        p3 = rotate_vector(p3, theta)

        p1 += obst_coords
        p2 += obst_coords
```

```
131         p3 += obst_coords
132 
133         return [p1, p2, p3]
134 
135     def convert_to_vgPoly(self, points: list) -> list:
136         return [(float(p[0]), float(p[1])) for p in points]
137 
138     # ------------------------------------------------------------------
139     # Construção do grafo de visibilidade
140     # ------------------------------------------------------------------
141 
142     def update_obstacle_map(self) -> None:
143         t0 = time.time()
144 
145         polygons = [self.convert_to_vgPoly(poly) for poly in self.
    vg_obstacles]
146 
147         vertices = set()
148         for poly in polygons:
149             for v in poly:
150                 vertices.add(v)
151 
152         if self.origin is not None:
153             vertices.add(self.origin)
154         if self.target is not None:
155             vertices.add(self.target)
156 
157         self.nodes = list(vertices)
158         self.node_index = {v: i for i, v in enumerate(self.nodes)}
159 
160         self.adjacency.clear()
161         self.predecessors.clear()
162 
163         edges = []
164         for poly in polygons:
165             n = len(poly)
166             for i in range(n):
167                 edges.append((poly[i], poly[(i + 1) % n]))
168 
169         def segments_intersect(a, b, c, d):
170             def orient(p, q, r):
171                 return (q[0] - p[0]) * (r[1] - p[1]) - (q[1] - p[1]) * (
    r[0] - p[0])
172 
173             o1 = orient(a, b, c)
174             o2 = orient(a, b, d)
175             o3 = orient(c, d, a)
```

```
176             o4 = orient(c, d, b)
177             return o1 * o2 < 0 and o3 * o4 < 0
178 
179         for i in range(len(self.nodes)):
180             for j in range(i + 1, len(self.nodes)):
181                 a = self.nodes[i]
182                 b = self.nodes[j]
183 
184                 visible = True
185                 for c, d in edges:
186                     if a in (c, d) or b in (c, d):
187                         continue
188                     if segments_intersect(a, b, c, d):
189                         visible = False
190                         break
191 
192                 if visible:
193                     cost = math.hypot(a[0] - b[0], a[1] - b[1])
194                     self.adjacency[i].append((j, cost))
195                     self.adjacency[j].append((i, cost))
196                     self.predecessors[j].append((i, cost))
197                     self.predecessors[i].append((j, cost))
198 
199         self.start_idx = self.node_index.get(self.origin, None)
200         self.goal_idx = self.node_index.get(self.target, None)
201 
202         self.g = {i: float("inf") for i in range(len(self.nodes))}
203         self.rhs = {i: float("inf") for i in range(len(self.nodes))}
204 
205         self.open_list.clear()
206         self.open_entry.clear()
207 
208         if self.goal_idx is not None:
209             self.rhs[self.goal_idx] = 0.0
210             self.push_open(self.goal_idx)
211 
212         self.tempo_total_path += time.time() - t0
213 
214     # ------------------------------------------------------------------
215     # Atualização incremental do LPA*
216     # ------------------------------------------------------------------
217 
218     def update_vertex(self, u: int) -> None:
219         if u != self.goal_idx:
220             self.rhs[u] = min(
221                 cost + self.g[s] for s, cost in self.adjacency.get(u,
    [])
```

```
222             )
223
224         if u in self.open_entry:
225             del self.open_entry[u]
226
227         if self.g[u] != self.rhs[u]:
228             self.push_open(u)
229
230     def compute_shortest_path(self) -> None:
231         while True:
232             top = self.top_key()
233             if self.start_idx is None:
234                 break
235
236             start_key = self.calculate_key(self.start_idx)
237             if not (
238                 top < start_key or
239                 self.rhs[self.start_idx] != self.g[self.start_idx]
240             ):
241                 break
242
243             u = self.pop_open()
244             if u is None:
245                 break
246
247             if self.g[u] > self.rhs[u]:
248                 self.g[u] = self.rhs[u]
249                 for p, _ in self.predecessors.get(u, []):
250                     self.update_vertex(p)
251             else:
252                 self.g[u] = float("inf")
253                 self.update_vertex(u)
254                 for p, _ in self.predecessors.get(u, []):
255                     self.update_vertex(p)
256
257     # ------------------------------------------------------------------
258     # Extração do caminho
259     # ------------------------------------------------------------------
260
261     def get_path(self) -> list:
262         t0 = time.time()
263
264         if self.origin is None or self.target is None:
265             return []
266
267         if self.start_idx is None or self.goal_idx is None:
268             self.update_obstacle_map()
```

```
269
270         self.compute_shortest_path()
271
272         if self.g[self.start_idx] == float("inf"):
273             return []
274
275         path = [self.start_idx]
276         current = self.start_idx
277
278         while current != self.goal_idx:
279             next_node = min(
280                 self.adjacency[current],
281                 key=lambda s: s[1] + self.g[s[0]]
282             )[0]
283             path.append(next_node)
284             current = next_node
285
286         self.tempo_total_path += time.time() - t0
287         return [self.nodes[i] for i in path]
288
289     # ------------------------------------------------------------------
290     # Interface de alto nível
291     # ------------------------------------------------------------------
292
293     def update_target_with_obstacles(self, robot, field, x_target,
    y_target, cont_target):
294         current_position = np.array([
295             robot.get_coordinates().X,
296             robot.get_coordinates().Y
297         ])
298
299         current_target = np.array([
300             x_target[cont_target],
301             y_target[cont_target]
302         ])
303
304         self.set_origin(current_position)
305         self.set_target(current_target)
306
307         vg_obstacles = []
308         obstacles = robot.map_obstacle.get_map_obstacle()
309
310         for obstacle in obstacles:
311             if not self.ignore_obstacle(robot, field.ball, obstacle):
312                 triangle = self.robot_triangle_obstacle(obstacle, robot)
313                 vg_obstacles.append(triangle)
314
```

```
315         self.vg_obstacles = vg_obstacles
316         self.update_obstacle_map()
317 
318         path = self.get_path()
319         if path:
320             return np.array(path[1] if len(path) > 1 else path[0])
321 
322         return current_target
323 
324     def ignore_obstacle(self, robot, ball, obstacle) -> bool:
325         p_robot = np.array([robot.get_coordinates().X, robot.
    get_coordinates().Y])
326         p_ball = np.array([ball.get_coordinates().X, ball.
    get_coordinates().Y])
327         p_obstacle = np.array([
328             obstacle.get_coordinates().X,
329             obstacle.get_coordinates().Y
330         ])
331 
332         _, t = ortogonal_projection(p_robot, p_ball, p_obstacle)
333         return t > 1
```

D Implementação do Algoritmo D* Lite

```python
import math
import heapq
import time
import numpy as np
from collections import defaultdict

from commons.math import angle_between, rotate_vector, ortogonal_projection
from entities.Obstacle import Obstacle


class VisibilityGraphDStarLitePlanner:
    """
    Planejador de rotas baseado em Visibility Graph utilizando o algoritmo D* Lite. Permite o reaproveitamento das informações de custo quando a posição inicial do robô é alterada, mantendo o grafo de visibilidade fixo entre atualizações consecutivas.
    """

    def __init__(self, debug=False) -> None:
        self.origin = None
        self.target = None

        self.nodes = []
        self.node_index = {}
        self.adjacency = defaultdict(list)
        self.predecessors = defaultdict(list)

        self.vg_obstacles = []

        # Estruturas centrais do D* Lite
        self.g = {}
        self.rhs = {}
        self.open_list = []
        self.open_entry = {}

        self.start_idx = None
        self.goal_idx = None

        self.km = 0.0
        self.last_origin = None
```

```
38
39          self.debug = debug
40          self.tempo_total_path = 0.0
41
42      # ---------------------------------------------------------------------
43      # Definição de origem e alvo
44      # ---------------------------------------------------------------------
45
46      def set_origin(self, coordinates: np.ndarray) -> None:
47          new_origin = (float(coordinates[0]), float(coordinates[1]))
48
49          if self.origin is not None:
50              dx = new_origin[0] - self.origin[0]
51              dy = new_origin[1] - self.origin[1]
52              self.km += math.hypot(dx, dy)
53
54          self.origin = new_origin
55          self.start_idx = self.node_index.get(self.origin, None)
56          self.last_origin = new_origin
57
58      def set_target(self, coordinates: np.ndarray) -> None:
59          self.target = (float(coordinates[0]), float(coordinates[1]))
60          self.goal_idx = self.node_index.get(self.target, None)
61
62      # ---------------------------------------------------------------------
63      # Heurística e chave do D* Lite
64      # ---------------------------------------------------------------------
65
66      def heuristic(self, u_idx: int, v_idx: int) -> float:
67          ux, uy = self.nodes[u_idx]
68          vx, vy = self.nodes[v_idx]
69          return math.hypot(ux - vx, uy - vy)
70
71      def calculate_key(self, u_idx: int) -> tuple:
72          g_u = self.g.get(u_idx, float("inf"))
73          rhs_u = self.rhs.get(u_idx, float("inf"))
74          m = min(g_u, rhs_u)
75
76          if self.start_idx is None:
77              h = 0.0
78          else:
79              h = self.heuristic(self.start_idx, u_idx)
80
81          return (m + h + self.km, m)
82
83      # ---------------------------------------------------------------------
84      # Fila de prioridade
```

```
85      # ------------------------------------------------------------------
86
87      def push_open(self, u_idx: int) -> None:
88          key = self.calculate_key(u_idx)
89          heapq.heappush(self.open_list, (key[0], key[1], u_idx))
90          self.open_entry[u_idx] = key
91
92      def pop_open(self):
93          while self.open_list:
94              k0, k1, u = heapq.heappop(self.open_list)
95              if self.open_entry.get(u) != (k0, k1):
96                  continue
97              del self.open_entry[u]
98              return u
99          return None
100
101     def top_key(self) -> tuple:
102         while self.open_list:
103             k0, k1, u = self.open_list[0]
104             if self.open_entry.get(u) != (k0, k1):
105                 heapq.heappop(self.open_list)
106                 continue
107             return (k0, k1)
108         return (float("inf"), float("inf"))
109
110     # ------------------------------------------------------------------
111     # Modelagem dos obstáculos
112     # ------------------------------------------------------------------
113
114     def robot_triangle_obstacle(self, obstacle: Obstacle, robot) -> list
        :
115         obst_coords = obstacle.get_coordinates()
116         obst_coords = np.array([obst_coords.X, obst_coords.Y])
117
118         robot_coords = robot.get_coordinates()
119         robot_coords = np.array([robot_coords.X, robot_coords.Y])
120
121         r = obstacle.radius
122
123         p1 = np.array([ r, -np.sqrt(3) * r])
124         p2 = np.array([ r,  np.sqrt(3) * r])
125         p3 = np.array([-2 * r, 0.0])
126
127         ref_vector = np.array([1.0, 0.0])
128         theta = angle_between(robot_coords - obst_coords, ref_vector)
129
130         p1 = rotate_vector(p1, theta)
```

```
131         p2 = rotate_vector(p2, theta)
132         p3 = rotate_vector(p3, theta)
133 
134         p1 += obst_coords
135         p2 += obst_coords
136         p3 += obst_coords
137 
138         return [p1, p2, p3]
139 
140     def convert_to_vgPoly(self, points: list) -> list:
141         return [(float(p[0]), float(p[1])) for p in points]
142 
143     # ------------------------------------------------------------------
144     # Construção do grafo de visibilidade
145     # ------------------------------------------------------------------
146 
147     def update_obstacle_map(self) -> None:
148         t0 = time.time()
149 
150         polygons = [self.convert_to_vgPoly(poly) for poly in self.
    vg_obstacles]
151 
152         vertices = set()
153         for poly in polygons:
154             for v in poly:
155                 vertices.add(v)
156 
157         if self.origin is not None:
158             vertices.add(self.origin)
159         if self.target is not None:
160             vertices.add(self.target)
161 
162         self.nodes = list(vertices)
163         self.node_index = {v: i for i, v in enumerate(self.nodes)}
164 
165         self.adjacency.clear()
166         self.predecessors.clear()
167 
168         edges = []
169         for poly in polygons:
170             n = len(poly)
171             for i in range(n):
172                 edges.append((poly[i], poly[(i + 1) % n]))
173 
174         def segments_intersect(a, b, c, d):
175             def orient(p, q, r):
```

```
176                 return (q[0] - p[0]) * (r[1] - p[1]) - (q[1] - p[1]) * (
    r[0] - p[0])
177             o1 = orient(a, b, c)
178             o2 = orient(a, b, d)
179             o3 = orient(c, d, a)
180             o4 = orient(c, d, b)
181             return o1 * o2 < 0 and o3 * o4 < 0
182 
183         for i in range(len(self.nodes)):
184             for j in range(i + 1, len(self.nodes)):
185                 a = self.nodes[i]
186                 b = self.nodes[j]
187                 visible = True
188                 for c, d in edges:
189                     if a in (c, d) or b in (c, d):
190                         continue
191                     if segments_intersect(a, b, c, d):
192                         visible = False
193                         break
194                 if visible:
195                     cost = math.hypot(a[0] - b[0], a[1] - b[1])
196                     self.adjacency[i].append((j, cost))
197                     self.adjacency[j].append((i, cost))
198                     self.predecessors[j].append((i, cost))
199                     self.predecessors[i].append((j, cost))
200 
201         self.start_idx = self.node_index.get(self.origin, None)
202         self.goal_idx = self.node_index.get(self.target, None)
203 
204         self.g = {i: float("inf") for i in range(len(self.nodes))}
205         self.rhs = {i: float("inf") for i in range(len(self.nodes))}
206 
207         self.open_list.clear()
208         self.open_entry.clear()
209         self.km = 0.0
210 
211         if self.goal_idx is not None:
212             self.rhs[self.goal_idx] = 0.0
213             self.push_open(self.goal_idx)
214 
215         self.tempo_total_path += time.time() - t0
216 
217     # ------------------------------------------------------------------
218     # Núcleo do D* Lite
219     # ------------------------------------------------------------------
220 
221     def update_vertex(self, u: int) -> None:
```

```
222         if u != self.goal_idx:
223             self.rhs[u] = min(
224                 cost + self.g[s] for s, cost in self.adjacency.get(u,
    [])
225             )
226 
227         if u in self.open_entry:
228             del self.open_entry[u]
229 
230         if self.g[u] != self.rhs[u]:
231             self.push_open(u)
232 
233     def compute_shortest_path(self) -> None:
234         while True:
235             top = self.top_key()
236             if self.start_idx is None:
237                 break
238 
239             start_key = self.calculate_key(self.start_idx)
240             if not (
241                 top < start_key or
242                 self.rhs[self.start_idx] != self.g[self.start_idx]
243             ):
244                 break
245 
246             u = self.pop_open()
247             if u is None:
248                 break
249 
250             if self.g[u] > self.rhs[u]:
251                 self.g[u] = self.rhs[u]
252                 for p, _ in self.predecessors.get(u, []):
253                     self.update_vertex(p)
254             else:
255                 self.g[u] = float("inf")
256                 self.update_vertex(u)
257                 for p, _ in self.predecessors.get(u, []):
258                     self.update_vertex(p)
259 
260     # ------------------------------------------------------------------
261     # Extração do caminho
262     # ------------------------------------------------------------------
263 
264     def get_path(self) -> list:
265         if self.start_idx is None or self.goal_idx is None:
266             return []
267 
```

```
268         self.compute_shortest_path()
269 
270         if self.g[self.start_idx] == float("inf"):
271             return []
272 
273         path = [self.start_idx]
274         current = self.start_idx
275 
276         while current != self.goal_idx:
277             next_node = min(
278                 self.adjacency[current],
279                 key=lambda s: s[1] + self.g[s[0]]
280             )[0]
281             path.append(next_node)
282             current = next_node
283 
284         return [self.nodes[i] for i in path]
285 
286     # ------------------------------------------------------------------
287     # Interface de alto nível
288     # ------------------------------------------------------------------
289 
290     def update_target_with_obstacles(self, robot, field, x_target,
    y_target, cont_target):
291         current_position = np.array([
292             robot.get_coordinates().X,
293             robot.get_coordinates().Y
294         ])
295 
296         current_target = np.array([
297             x_target[cont_target],
298             y_target[cont_target]
299         ])
300 
301         self.set_origin(current_position)
302         self.set_target(current_target)
303 
304         vg_obstacles = []
305         obstacles = robot.map_obstacle.get_map_obstacle()
306 
307         for obstacle in obstacles:
308             if not self.ignore_obstacle(robot, field.ball, obstacle):
309                 triangle = self.robot_triangle_obstacle(obstacle, robot)
310                 vg_obstacles.append(triangle)
311 
312         self.vg_obstacles = vg_obstacles
313         self.update_obstacle_map()
```

```
314
315         path = self.get_path()
316         if path:
317             return np.array(path[1] if len(path) > 1 else path[0])
318
319         return current_target
320
321     def ignore_obstacle(self, robot, ball, obstacle) -> bool:
322         p_robot = np.array([robot.get_coordinates().X, robot.
    get_coordinates().Y])
323         p_ball = np.array([ball.get_coordinates().X, ball.
    get_coordinates().Y])
324         p_obstacle = np.array([
325             obstacle.get_coordinates().X,
326             obstacle.get_coordinates().Y
327         ])
328
329         _, t = ortogonal_projection(p_robot, p_ball, p_obstacle)
330         return t > 1
```