UNIVERSIDADE FEDERAL DE OURO PRETO
ESCOLA DE MINAS
COLEGIADO DO CURSO DE ENGENHARIA DE CONTROLE E AUTOMAÇÃO - CECAU

MARLON MARTINS CUNHA

PLANEJAMENTO DE CAMINHO DE ROBÔS FUTEBOLISTAS UTILIZANDO BUSCA EM GRAFO

MONOGRAFIA DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Ouro Preto, 2019

MARLON MARTINS CUNHA

PLANEJAMENTO DE CAMINHO DE ROBÔS FUTEBOLISTAS UTILIZANDO BUSCA EM GRAFO

Monografia apresentada ao Curso de Engenharia de Controle e Automação da Universidade Federal de Ouro Preto como parte dos requisitos para a obtenção do Grau de Engenheiro de Controle e Automação.

Orientador: Prof. Elias José de Rezende Freitas, Me.

Coorientador: Prof. Agnaldo José da Rocha Reis, Dr.

Ouro Preto
Escola de Minas – UFOP
2019

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

C972p Cunha, Marlon Martins .
Planejamento de caminho de robôs futebolistas utilizando busca em grafo. [manuscrito] / Marlon Martins Cunha. - 2019.
81 f.: il.: color., tab..

Orientador: Prof. Me. Elias José de Rezende Freitas.
Coorientador: Prof. Dr. Agnaldo José da Rocha Reis.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Escola de Minas. Graduação em Engenharia de Controle e Automação .

1. Robôs móveis. 2. Planejamento de caminho. 3. Algoritmos . 4. Grafos. I. Freitas, Elias José de Rezende. II. Reis, Agnaldo José da Rocha. III. Universidade Federal de Ouro Preto. IV. Título.

CDU 621.865.8

Bibliotecário(a) Responsável: Angela Maria Raimundo - SIAPE: 1.644.803

A comissão avaliadora constituída pelos professores Elias José de Rezende Freitas, Agnaldo José da Rocha Reis, Tamires Martins Rezende e Luiz Olmes Carvalho atesta que a monografia intitulada "Planejamento de Caminho de Robôs Futebolistas Utilizando Busca em Grafo" foi defendida e aprovada em 18 de dezembro de 2019.

Prof. Me. Elias José de Rezende Freitas - Orientador

Prof. Dr. Agnaldo José da Rocha Reis – Coorientador

Profa. Me. Tamires Martins Rezende – Professora Convidada

Prof. Dr. Luiz Olmes Carvalho – Professor Convidado

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado força de vontade para não desistir. Em segundo lugar, agradeço aos meus pais, irmãos, namorada e amigos que sempre estiveram ao meu lado. Agradeço em especial aos meus amigos Hamilton e Paulo que contribuíram na realização deste trabalho, assim como ao meu orientador Elias, que nunca desistiu de mim apesar das dificuldades. Por fim, agradeço aos integrantes da equipe Rodetas, aos professores e aos projetos PETMAT e NEI, que contribuíram diretamente em minha formação acadêmica.

"Never give up. Today is hard, tomorrow will be worse, but the day after tomorrow will be sunshine"(Jack Ma)

RESUMO

A robótica móvel tem se mostrado uma área promissora, com saltos notáveis em pesquisas nas últimas décadas. Essa inovação na área, está levando a criação de robôs que ultrapassam as características fundamentais de um robô como locomoção e percepção do ambiente, avançando em direção a robôs que cada vez mais inteligentes que possuam autonomia de realizar tarefas sem a intervenção humana. Uma das etapas que precisa ser cumprida para que isso aconteça é o planejamento de caminhos, em que dado uma posição inicial e uma posição final, é calculado o melhor caminho entre esses pontos. Para tanto, o robô precisa conhecer informações sobre o ambiente em que se encontra. Desta forma, neste trabalho é proposto estudo do planejamento de caminho utilizando algoritmos de busca em grafos, para um mapa do ambiente previamente conhecido contendo obstáculos estáticos. Os resultados obtidos se mostraram satisfatórios, sendo possível planejar o melhor caminho livre de obstáculos para o robô no ambiente, com um baixo tempo de execução do algoritmo de busca.

Palavras-chaves: Robôs móveis. Planejamento de caminho. Algoritmos de busca. Grafos.

ABSTRACT

Mobile robotics has proven to be a promising area, with notable progress in research in recent decades. This innovation in the area is leading to the creation of robots that surpass the fundamental characteristics of a robot such as locomotion and environmental perception, advancing towards increasingly intelligent robots that have the autonomy to perform tasks without human intervention. One of the steps that need to be taken to make this happen is path planning, where given a start position and an end position, the path between these points is calculated. To do so, the robot needs to know information about its environment. Thus, this work proposes a study of path planning using graph search algorithms for a previously known environment map containing static obstacles. The results obtained are satisfactory, being possible the best obstacle-free path for the robot in the environment, with a low execution time of the search algorithm.

Key-words: Mobile robots. Path planning. Search algorithms. Graphs.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Por anos os robôs têm feito parte da imaginação de autores e produtores de ficção científica, estando presentes em variadas obras, desde clássicos como o livro Eu robô, de Isaac Asimov, até produções atuais como a série *sci-fi* Westworld (2016) da HBO. Tamanho fascínio pela robótica ocasionou uma vasta gama de pesquisas e estudos na área. Com o intuito inicial de auxiliar o ser humano em atividades repetitivas ou de difícil execução, ou até mesmo substituí-los em tarefas com elevado grau de risco, diferentes tipos de robôs foram desenvolvidos para os mais variados setores. No setor industrial, por exemplo, os manipuladores robóticos são amplamente utilizados, principalmente na indústria automobilística, devido a sua precisão, repetibilidade e velocidade de produção. Porém, não só no ramo industrial os robôs vêm ganhando espaço, na área da medicina, os avanços tecnológicos permitiram a criação de sistemas robóticos capazes de assistir aos cirurgiões em procedimentos cirúrgicos, possibilitando maior precisão ao procedimento e incisões menores, se comparadas às cirurgias convencionais (POFFO et al., 2013).

Nesse vasto ambiente de possibilidades, uma área promissora é a robótica móvel. Dentro desse ramo, inúmeras aplicações práticas foram desenvolvidas para simplificar e tornar eficiente a vida do ser humano. Robôs que realizam tarefas domésticas, como cortar grama ou aspirar o pó, que transportam cargas e até robôs que executam o monitoramento aéreo são alguns exemplos de aplicações (WOLF et al., 2009). Alguns exemplos de robôs móveis podem ser visualizados na Figura 1.

(a) Automower - Huskvarna (b) Roomba - iRobot

Figura 1 – (a) Automower: robô cortador de grama; (b) Roomba: robô aspirador.

Fonte – (a) Husqvarna (2019), (b) iRobot (2019).

Tais robôs são classificados mediante ao seu grau de autonomia, podendo ser definidos como: (a) veículos teleoperados: precisam de informações externas para operar; (b) semi-autônomos: são controlados remotamente, porém, possuem um determinado grau de independência, principalmente em tarefas encarregadas da segurança da navegação, como para evitar o choque com obstáculos; (c) autônomos: operam de maneira totalmente autônoma, sem necessidade de intervenção humana (FREITAS; PASSOS; PEREIRA, 2017). Uma das aplicações mais famosas foi o *rover*[1] *Sojourner*, como mostrado na Figura 2, que foi enviado ao planeta Marte em 1997 com o intuito de realizar uma exploração espacial. O *Sojourner*, assim como os seus sucessores *Spirit* e *Opportunity*, apresentava certa limitação de autonomia, pois necessitava de comandos à distância dos seres humanos, de forma a realizar a missão (JUNG et al., 2005).

Figura 2 – *Sojourner*, o primeiro *rover* em Marte.

Fonte – NASA (2019).

O processo de construção de um robô autônomo, desde o protótipo até uma aplicação real, envolve diversas etapas. Tais etapas começam nas escolhas dos sensores e atuadores e se estendem ao desenvolvimento das técnicas de desvio de obstáculos, localização, mapeamento do ambiente e planejamento de caminhos, constituindo uma estratégia de navegação. A etapa de planejamento de caminhos, em particular, é responsável por receber as informações do ambiente e, com base nessa informação definir um caminho que leve o robô de uma posição inicial a uma posição objetivo (WOLF et al., 2009).

Essa etapa se mostra extremamente importante no ramo da robótica móvel, pois, possibilita a concepção de caminhos seguros e eficientes que consideram os obstáculos presentes no ambiente. Haja vista a importância do planejamento de caminhos, este trabalho apresenta um estudo de algoritmos de busca em grafos para a obtenção de caminhos livres de obstáculos para

[1] *Rover* é o nome dado a um veículo robótico utilizado para explorar a superfície da lua ou de um planeta (PLACE, 2019)

um robô móvel diferencial. Isto é alcançado utilizando o método de decomposição aproximada para transformar o ambiente em um grafo que será utilizado pelos algoritmos de busca, especificamente Busca em Largura, Busca em Profundidade, Busca pela Melhor Escolha, algoritmo de Dijkstra e algoritmo A* (lido como “A estrela”).

O propósito deste trabalho é resumido na Figura 3. O robô futebolista[2] (quadrado branco com detalhes pretos) está posicionado em um campo (área em cinza) que contém obstáculos estáticos (objetos em amarelo) e o alvo (bola da cor laranja). Esse campo é dividido em células iguais que formam o grafo do ambiente. Caso exista a presença de obstáculos na célula, ela é considerada como obstruída. Com esse grafo, os algoritmos de busca implementados calculam o caminho que leva o robô da posição inicial ao alvo.

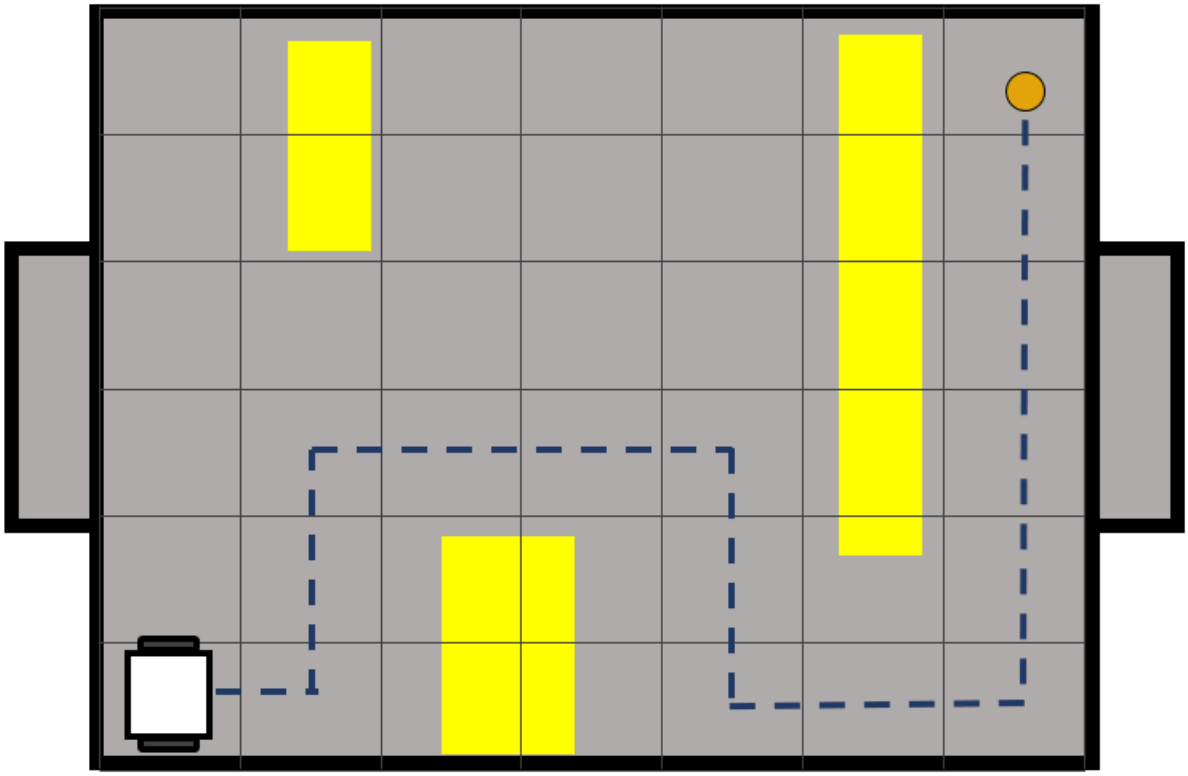


Figura 3 – Exemplo de planejamento de caminho proposto neste trabalho.

1.1 Justificativas e Relevância

O nível de autonomia em um sistema robótico varia de acordo com o modelo de controle empregado. Tais sistemas podem variar desde robôs mais simples como seguidores de linha até sistemas mais complexos projetados com o intuito de tornar o robô mais robusto, confiável e independente de intervenções humanas. Para que isso aconteça, devem ser implementadas técnicas que preservem a integridade física dos seres humanos que compartilham o ambiente

[2] Um robô futebolista é um robô desenvolvido para competições semelhantes ao futebol humano, porém com regras próprias. Neste trabalho esse robô é um robô autônomo diferencial e deve seguir as regras da competição da IEEE *Very Small Size Soccer*.

com robô, assim como técnicas que previnam danos ao ambiente e ao próprio robô (JUNG et al., 2005).

Nesse quesito, o planejamento e a navegação robótica são pontos extremamente importantes no projeto dos robôs autônomos. Desta forma, este trabalho propõe um estudo e a implementação de algoritmos de busca em grafos com o objetivo de planejar o caminho mais eficiente de uma posição-origem até uma posição-alvo para um robô futebolista, tendo como referência um mapa conhecido do ambiente com obstáculos.

1.2 Objetivos

O presente trabalho tem como objetivo geral estudar e implementar alguns dos principais algoritmos de busca em grafos e, com base neles, planejar o caminho para um robô futebolista, sendo o mapa do ambiente conhecido previamente. São objetivos específicos deste trabalho:

- Desenvolver um sistema de visão computacional capaz de detectar o robô e os obstáculos, gerando um mapa prévio;
- Implementar diferentes planejadores de caminho, baseados em algoritmos de busca em grafos;
- Analisar os resultados dos caminhos fornecidos pelos planejadores.

1.3 Organização e estrutura

O presente trabalho está dividido em 5 capítulos, adotando a seguinte estrutura: o Capítulo 1 apresenta a introdução do trabalho, a justificativa para a elaboração e os objetivos gerais e específicos.

O Capítulo 2 apresenta a fundamentação teórica abordando os conceitos necessários para um pleno entendimento do trabalho. São tratados neste capítulo temas como visão computacional, representação do mapa do ambiente, teoria dos grafos e estratégias de busca com e sem informação.

No Capítulo 3 são abordados os aspectos construtivos do robô móvel, as etapas de desenvolvimento da visão computacional e os passos utilizados para a implementação dos algoritmos de busca em grafos, assim como a integração entre o sistema de visão e o sistema de planejamento de caminho.

No Capítulo 4 são apresentados os resultados obtidos com o sistema de visão computacional e com os algoritmos de busca utilizados para planejar o caminho para o robô.

Por fim, o Capítulo 5 apresenta as considerações finais sobre o trabalho realizado e cita as melhorias que podem ser implementadas em trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

No problema de planejamento de caminho, o robô é considerado como um ponto no espaço de configurações e todas as dinâmicas e incertezas são desconsideradas. Dessa forma, o objetivo do planejamento é produzir uma ordem de configurações que leve o robô de uma configuração inicial a uma configuração de destino (MOLINA, 2014). Para que isso ocorra, o algoritmo precisa considerar o mapa de ocupação do ambiente e as posições do início e do fim do trajeto.

Sendo assim, para o desenvolvimento do trabalho torna-se necessário a criação de um sistema que seja capaz de obter o mapa do ambiente, destacando as informações pertinentes ao problema, como os obstáculos e as posições do robô e alvo. Para a criação do mapa e obtenção dessas informações são utilizadas técnicas de visão computacional que serão abordadas na Seção 2.1. Com as informações obtidas pela visão computacional, torna-se necessário a utilização de alguma técnica que possibilite a representação do ambiente. Para isso, foi utilizado a técnica de decomposição do ambiente em células, criando o mapa do ambiente, que irá representar o espaço de configurações do robô (Seção 2.2). O mapa do ambiente é representado por uma estrutura abstrata de dados, chamada grafo. Conceitos e características dessa estrutura são apresentados na Seção 2.3. Com o grafo criado, é necessário planejar o caminho que leva de um ponto inicial no mapa até um ponto objetivo. Esse caminho é calculado por meio dos algoritmos de busca apresentados nas Secões 2.4 e 2.5. Por fim, a Seção 2.6 apresenta as tecnologias necessárias para o desenvolvimento do trabalho.

2.1 Visão Computacional

A visão computacional pode ser compreendida como um conjunto de técnicas e métodos, que possibilitam a um sistema computacional simular a visão humana, recebendo como entrada uma imagem e fornecendo como saída a interpretação dessa imagem, sendo capaz de extrair informações da mesma de modo a alcançar um objetivo particular (MARENGONI; STRINGHINI, 2009).

Com o auxílio de uma câmera são capturadas as imagens do ambiente utilizado no trabalho. Entretanto, a maneira como o ser humano vê uma imagem se difere da forma que um computador enxerga. Para o ser humano, o processo se inicia com os olhos e o cérebro fica encarregado de realizar a interpretação da mesma. Em contrapartida, o que um computador vê é uma matriz de números que varia de dimensão de acordo com a resolução da imagem (BRADSKI; KAEHLER, 2008). A Figura 4 ilustra esse processo. Enquanto o ser humano classifica esta figura facilmente como um carro, o que o computador recebe são matrizes numéricas, o que ajuda a explicar o motivo de tarefas de visão computacional não serem triviais.

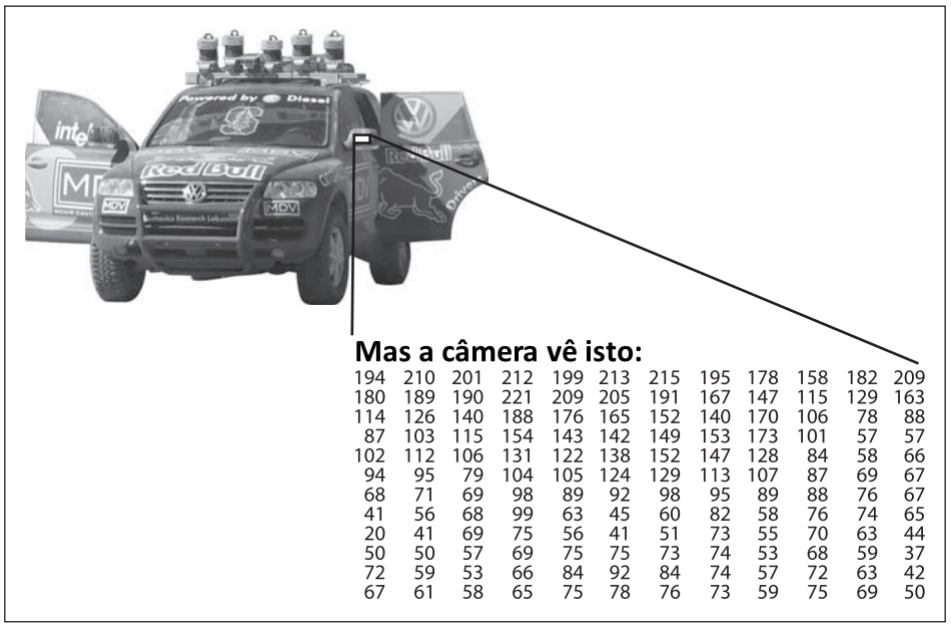


Figura 4 – Diferença de visualização de uma imagem entre o computador e um ser humano.

Fonte – Bradski e Kaehler (2008).

Além disso, após o processo de aquisição das imagens, muitas vezes é necessária uma etapa de pré-processamento das mesmas, ajustando o tamanho ou formato das imagens e aplicando filtros com o intuito de remover ruídos referentes ao processo de aquisição. Tais ruídos dificultam o processo de interpretação das informações obtidas e podem ser ocasionados pela iluminação do ambiente, posição relativa entre a câmera e o objeto, entre outras fontes (MARENGONI; STRINGHINI, 2009).

2.2 Mapa do ambiente

Com o auxílio da visão computacional são obtidas as imagens processadas que serão utilizadas na criação do mapa do ambiente. Esse mapa representará o espaço de configurações do robô, podendo ser construído com as seguintes estratégias: (i) Mapas de rotas; (ii) Decomposição em células; e (iii) Campos potenciais (LATOMBE, 1991). Dentre estas estratégias, a técnica de decomposição em células foi a escolhida para representar o ambiente, pois, com este método a construção do grafo utilizado nos algoritmos de busca é simplificada, sendo também um dos métodos mais estudados para planejamento de movimento (LATOMBE, 1991).

O método de decomposição em células consiste fundamentalmente em decompor o espaço livre do robô em divisões simples, denominadas células, de forma que seja possível gerar um caminho entre duas configurações em uma célula. Este método pode ser classificado em: (i) método de decomposição exata; e (ii) decomposição aproximada.

No método de decomposição exata, o contorno dos obstáculos presentes no espaço de configuração são representados por meio de equações matemáticas, definindo assim a proxi-

midade das células e a relação de vizinhança, de forma que um caminho entre duas células adjacentes não contenha obstáculos (MOLINA, 2014). Tal técnica apresenta como vantagem a capacidade de representar o espaço de configurações completamente, com a garantia de encontrar um caminho livre sempre que existir, caso as técnicas de busca e computação numéricas apropriadas forem utilizadas. Por outro lado, a representação matemática aumenta a complexidade do método, dificultando a sua implementação (LATOMBE, 1991). A Figura 5 apresenta um exemplo da aplicação do método de decomposição, em que são destacados os obstáculos no ambiente, a configuração inicial (q_{init}) e final (q_{goal}) do robô e o gráfico de conectividade obtido que demonstra o caminho do início até o objetivo.

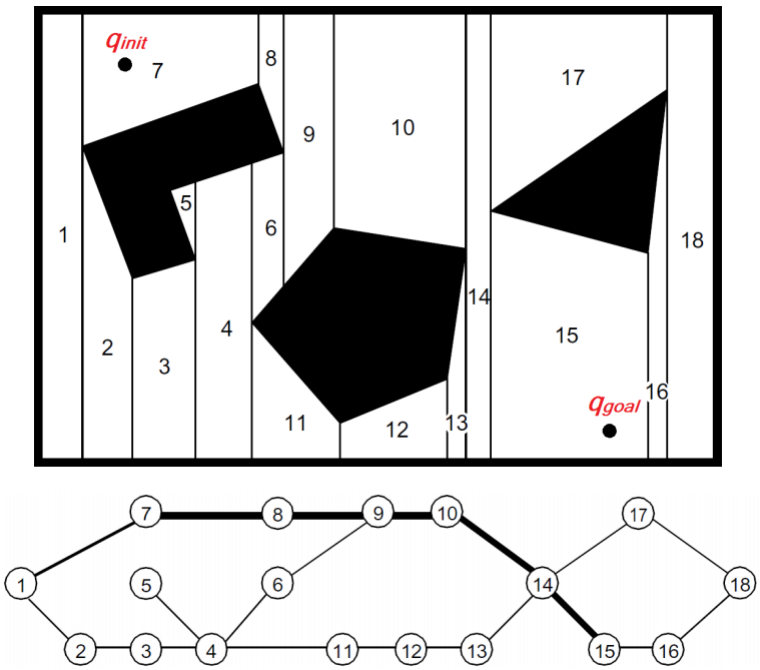


Figura 5 – Aplicação da técnica de decomposição exata e o grafo derivado.

Fonte – Molina (2014).

O método de decomposição aproximada, no que lhe concerne, divide o ambiente em regiões de formatos predefinidos, como quadrados ou retângulos, desconsiderando o contorno dos obstáculos presentes no mapa. Dentro desse método, uma das abordagens mais utilizadas são as grades (*grid*) regulares. Nessa técnica, cada célula, resultante da divisão do ambiente, é analisada a procura de obstáculos, sendo a célula classificada como ocupada se existir parte de um obstáculo em seu interior, e livre, caso contrário (LATOMBE, 1991).

A grande vantagem dessa metodologia é a baixa complexidade de representação do ambiente, fornecendo muitas vezes uma aproximação satisfatória do mapa. Por outro lado, os métodos de aproximação são incapazes de representar a totalidade do ambiente e podem não ser completos, como em casos onde existem passagens estreitas que desaparecem devido a especificação da célula utilizada (MOLINA, 2014), como poder ser visualizado na Figura 6. Esta figura exibe o mapa do ambiente e sua correspondente aproximação por grade regular. Como

pode ser observado, a passagem entre o pentágono e o triângulo desaparece na representação aproximada.

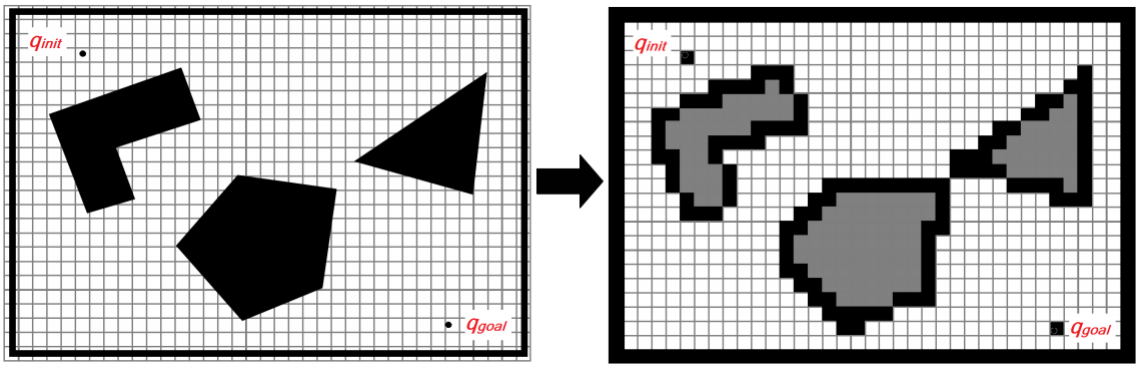


Figura 6 – Decomposição aproximada de um ambiente utilizando grade regular.

Fonte – Molina (2014).

Devido à simplicidade do método de decomposição aproximada, essa técnica foi utilizada neste trabalho para representar o ambiente. As desvantagens da mesma foram minimizadas corrigindo as dimensões da célula escolhida, ajustando assim a precisão da aproximação.

Após a etapa de decomposição é criado um grafo não dirigido, intitulado grafo de conectividade, que representa a adjacência entre as células, em que os nós desse grafo representam as regiões navegáveis do espaço e se duas células são adjacentes, essa conexão é representada como uma aresta (LATOMBE, 1991).

Para tornar claro o entendimento sobre grafos é apresentado na Seção 2.3 um pouco da história, conceitos, notações e representações acerca desse tema.

2.3 Teoria dos Grafos

O princípio da teoria dos grafos é, geralmente, associado com o Problema das Pontes Königsberg (PAOLETTI, 2011). A antiga cidade de Königsberg, Prússia, que atualmente é denominada Kaliningrad, foi construída ao redor do rio Pregel. No meio do rio existiam duas grandes ilhas que estavam ligadas uma à outra e às margens através de 7 pontes (CARDOSO, 2011), como mostrado na Figura 7.

Os habitantes de Königsberg gostavam de passear pela cidade nos fins de semanas, desfrutando da beleza do lugar. Durante esses passeios, foi inventado um jogo, em que o objetivo era criar uma forma de se deslocar pela cidade, cruzando todas as sete pontes apenas uma vez. Tal problema chegou ao conhecimento do matemático suíço Leonhard Euler (1707 – 1783), que ao primeiro contato o classificou como trivial. Entretanto, o problema o deixou intrigado, pois, não era possível resolvê-lo, utilizando conceitos de geometria, álgebra ou contagem. Para solucionar o enigma, Euler utilizou o conceito de grafos, modelando o problema como um multigrafo e

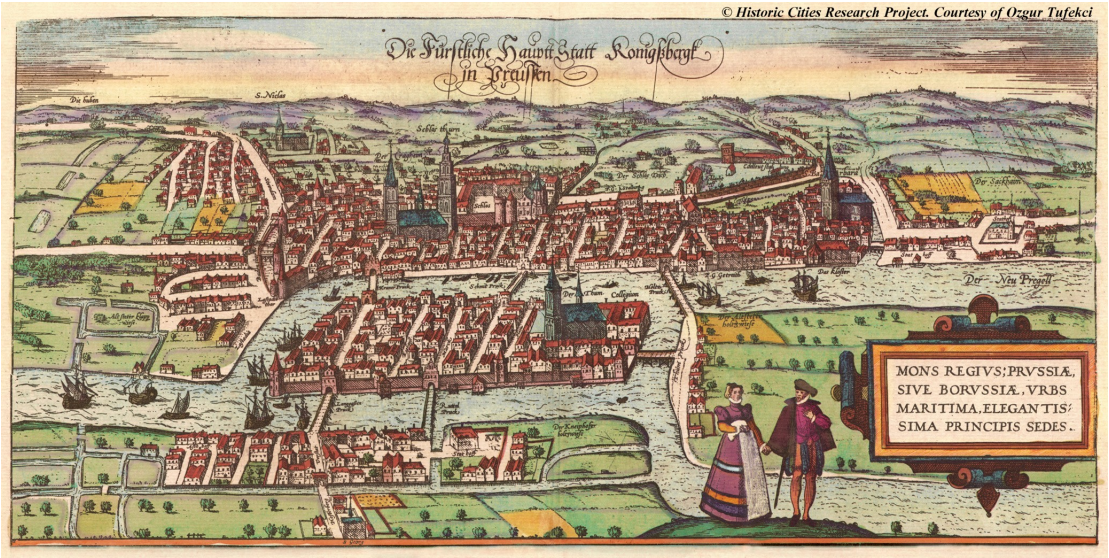
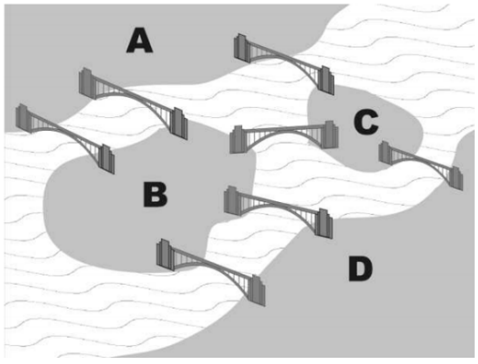


Figura 7 – A cidade de Königsberg em 1554.

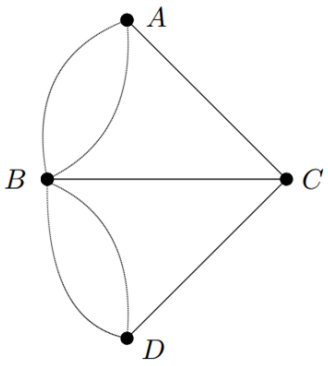
Fonte – História e Modernidade (2011).

descobrindo a inexistência de um trajeto que solucionasse o enigma das pontes (PAOLETTI, 2011).

A Figura 8a mostra uma representação mais simples da cidade de Königsberg, enquanto a Figura 8b mostra sua respectiva reprodução em grafos. Em 1736, Euler publicou um artigo em que descrevia um método geral para a resolução de problemas da mesma natureza, o qual teve grande importância para a teoria dos grafos e matemática, em geral (COSTA, 2011).



(a) Representação das pontes de Königsberg.



(b) Grafo correspondente as pontes.

Figura 8 – Representação das pontes e o respectivo grafo.

Fonte – Cardoso (2011).

2.3.1 Notação e noções fundamentais

Um grafo é uma estrutura abstrata de dados, muito utilizada na solução de problemas computacionais. Por definição, um grafo G é um par de conjuntos (V, E), de forma que $V = V(G) = \{v1, ..., vn\}$ representa o conjunto dos vértices e $E = E(G)$ representa o conjunto

das arestas, em que cada uma corresponde a um subconjunto de $V(G)$ de cardinalidade 2 (CARDOSO, 2011).

Um grafo simples *G* é composto por um par (V, E) onde *V* é um conjunto não vazio e *E* um conjunto de pares distintos não ordenados de elementos distintos de *V*. Já um grafo finito *G* é composto por um par (V, E) onde *V* é um conjunto finito não vazio e *E* uma coleção de pares não ordenados de elementos, não necessariamente distintos de *V* (COSTA, 2011).

Considerando-se dois vértices *v* e *w*, pode-se dizer que *v* e *w* são vizinhos ou adjacentes se estes são ligados por uma aresta *v,w*. Um caminho entre os vértices *v* e *w* é composto por uma sequência de vértices ou arestas que une esses vértices (COSTA, 2011),(FEOFILOFF; KOHAYAKAWA; WAKABAYASHI, 2011).

Os grafos podem ser representados graficamente com figuras planas compostas de linhas e pontos, em que as linhas representam as arestas (ou arcos) e os pontos representam os vértices, como apresentado na Figura 9, sendo os vértices representados pelos círculos azuis e as arestas pelas setas verdes. Outro tipo de representação inclui a adição de pesos às arestas, criando o conceito de grafo ponderado. A ponderação permite adicionar informações extras ao grafo por meio da inserção de valores em cada aresta. Essas informações podem ser referentes ao custo, capacidade, distância, entre os vértices (CARDOSO, 2009). Como exemplo, cada vértice de um grafo pode representar uma cidade e cada aresta simboliza uma estrada que liga tais cidades. Pode-se utilizar o peso para representar a distância entre as cidades, como indicado na Figura 10.

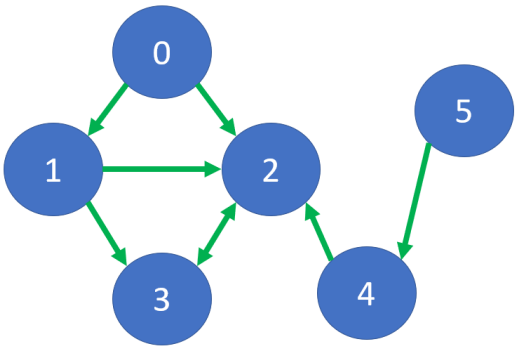


Figura 9 – Exemplo de grafo.

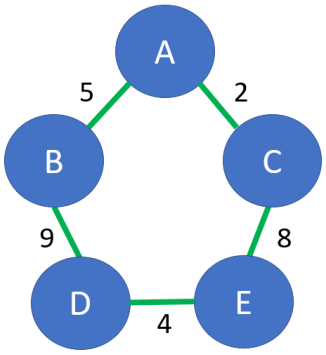


Figura 10 – Exemplo de grafo ponderado.

Na teoria dos grafos existem dois tipos de grafos muito comuns: grafos não-dirigidos e grafos dirigidos. Em um grafo dirigido, cada arco é composto por um par ordenado (v_1, v_2), ou seja, para um arco (v, w), o primeiro vértice *v* é a ponta inicial e o segundo vértice *w* é a ponta final do arco, como na Figura 11. Já em um grafo não-dirigido, cada aresta é composto por um conjunto $\{v_1, v_2\}$, ou seja, para uma aresta $\{v, w\}$, existe no grafo outra aresta $\{w, v\}$. De forma simplificada, pode-se dizer que não existe uma direção de trajeto entre os vértices (FEOFILOFF, 2017), como na Figura 12.

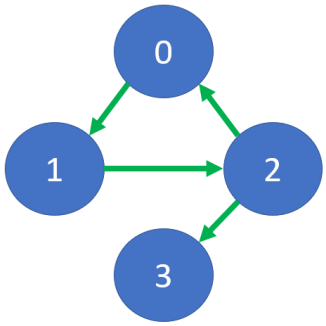


Figura 11 – Grafo dirigido.

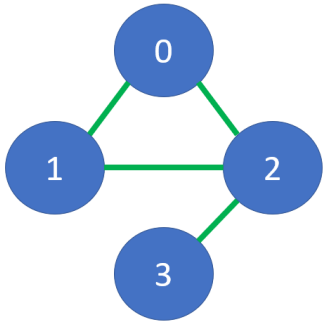


Figura 12 – Grafo não dirigido.

2.3.2 Representação de grafos computacionalmente

Existem diversas maneiras de representar grafos, cada uma apresentando suas qualidades e defeitos, que podem ser escolhidas dependendo do tipo de aplicação. Os critérios utilizados para avaliar a qualidade da representação consideram o tempo de execução e a quantidade de memória utilizada (CORMEN; BALKCOM, 2019). Serão apresentadas três formas de se representar um grafo:

- Lista de arestas

  Uma das formas mais simples de se representar um grafo é utilizando uma lista de *E* arestas. Desta forma, uma aresta é retratada como um arranjo de dois vértices sobre os quais tais arestas são incidentes. Caso as arestas contenham pesos, esse peso é simplesmente adicionado ao arranjo como um terceiro elemento (CORMEN; BALKCOM, 2019). Para a Figura 11, a lista de arestas, representada como um vetor, seria:

$$lista\_de\_arestas = [[0,1],[1,2],[2,0],[2,3]]$$

Dessa forma, o espaço necessário para armazenar tal lista, utilizando notação assintótica, será $O(E)$. Assim, percebe-se a simplicidade da lista de arestas. Entretanto, devido à estrutura ser basicamente um vetor, para se descobrir um componente dentro dela é necessário realizar uma iteração neste vetor. Como exemplo, para saber se os vértices 2 e 3 são conectados por uma aresta, é necessário verificar na lista de arestas a existência de um par $[2,3]$ ou $[3,2]$. Para a lista acima, isso seria verificado na última posição do vetor. Entretanto, se esse grafo fosse muito maior, a busca por uma aresta se tornaria mais custosa. De fato, como as arestas não estão organizadas em uma ordem definida, no pior dos casos seria necessária uma busca linear por todo o vetor para descobrir se existe tal aresta. Isso acarreta um tempo linear de $O(E)$, sendo *E* as arestas do grafo (CORMEN; BALKCOM, 2019).

- Matriz de adjacência

  Outra forma de representação computacional de grafos é por meio da matriz de adjacência. Ela é uma matriz de dimensão $|V| \times |V|$, onde *V* retrata os vértices do grafo, em que para cada aresta (a_{ij}), onde *i* representa a linha e *j* representa a coluna, tem-se:

$$a_{ij} = \begin{cases} 1, & \text{se existe aresta entre } v_i \text{ e } v_j \\ 0, & \text{caso contrário} \end{cases}$$

A Figura 13 exibe a matriz de adjacências para os grafos das Figuras 11 e 12, respectivamente.

	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	1	0	0	1
3	0	0	0	0

(a) Matriz de adjacência da figura 11.

	0	1	2	3
0	0	1	1	0
1	1	0	1	0
2	1	1	0	1
3	0	0	1	0

(b) Matriz de adjacência da figura 12.

Figura 13 – Representação em matrizes de adjacência.

Com essa matriz, para se descobrir a existência de uma aresta entre dois vértices, basta realizar-se uma busca na linha *i* e na coluna *j*. Como exemplo, para descobrir se o vértice 3 está conectado ao vértice 2, procura-se a interseção da linha 3 da matriz com a coluna 2 e obtém-se que para a matriz da Figura 13a não existe tal aresta, enquanto para a Figura

13b esta aresta existe. Analisando as duas matrizes da Figura 13 e recordando-se que a Figura 11 representa o grafo dirigido e a Figura 12 representa o grafo não dirigido, percebe-se uma característica dessa representação que é o fato de que para um grafo não direcionado a matriz é simétrica, enquanto para um grafo direcionado isso não precisa acontecer (CORMEN; BALKCOM, 2019).

Dessa forma, a matriz de adjacência apresenta como pontos positivos a facilidade de busca, inserção e remoção de arestas em um tempo constante $O(1)$. Entretanto, ela consome memória de forma indiscriminada. Indiferente se o grafo é esparso (apresenta poucas arestas) ou denso (contém muitas arestas), ela irá ocupar $O(V^2)$ de espaço de memória. No caso do grafo esparso, essa matriz apresentará vários 0's, o que demonstra um uso de memória não eficiente (CORMEN; BALKCOM, 2019).

- Lista de adjacência

  A lista de adjacência é uma forma de representação que mescla as características das listas de arestas com as matrizes de adjacência (CORMEN; BALKCOM, 2019). Basicamente, é um vetor com listas aninhadas, de forma que cada posição do vetor representa um vértice e para cada vértice *i* existe uma lista de vértices aos quais ele é ligado. Como exemplo, a Figura 14 apresenta a lista de adjacência para a Figura 9. Para verificar se o vértice 5 é adjacente ao vértice 4, basta pesquisar na lista de adjacência referente ao vértice 5 e verificar se existe o vértice 4 na lista, o que nesse exemplo é verdadeiro.

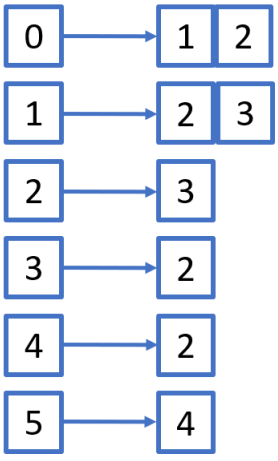


Figura 14 – Lista de adjacência para o grafo da Figura 9.

O processo de consultar a lista de adjacência de cada vértice é realizado em um tempo constante. Para descobrir se uma aresta (i, j) pertence a um determinado grafo é realizada uma busca em tempo constante pela lista de adjacência *i* e depois uma procura por *j* nesta lista. Caso o vértice *j* se encontre na última posição da lista, o que seria o pior caso, essa busca irá levar um tempo de $O(d)$, onde *d* representa o grau do vértice (número de arestas incidentes no vértice). O grau do vértice i pode ter o valor 0, caso não tenha arestas

incidentes, ou $|V| - 1$, quando *i* for adjacente a todos os outros vértices (CORMEN; BALKCOM, 2019).

Para um grafo não direcionado tem-se que cada aresta (i, j) irá aparecer duas vezes na lista de adjacência, uma vez na lista *i* e outra na lista *j*. Como o número de arestas é representado como $|E|$, o espaço de memória ocupado pela lista será de $2|E|$. Já para o grafo direcionado, como o número de arestas é $|E|$ elementos, o espaço de memória ocupado será $|E|$.

Considerando as características principais de cada representação e analisando as necessidades deste trabalho, optou-se por utilizar a lista de adjacência para representar os grafos por se tratar de uma maneira simples e eficiente.

2.3.3 Aplicações

Os conceitos de teoria dos grafos são amplamente empregados não só no campo da computação, mas também em outras áreas. Em ciência da computação, suas ideias são utilizadas em campos como redes, segmentação e captura de imagens, mineração de dados, pesquisa operacional, entre outras. Além desse ramo, os conceitos podem ser utilizados em sociologia para explorar mecanismos de difusão e avaliar o prestígio de atores, em química sendo utilizado no estudo de moléculas e átomos e até em biologia (SHIRINIVAS; VETRIVEL; ELANGO, 2010).

Com o grafo de conectividade criado, a próxima etapa consiste em se realizar uma busca no grafo com o intuito de descobrir um caminho que leva o robô de uma configuração inicial a uma configuração final. Para realizar tal tarefa, são utilizadas as estratégias de busca informada e não informada, presentes na Seção 2.5 e 2.4, respectivamente.

2.4 Estratégias de busca sem informação

O processo de examinar as arestas e vértices de um grafo é chamado busca ou percurso em grafos. Normalmente, durante este processo, o objetivo será encontrar um caminho que leve de um nó inicial a um nó desejado no grafo, gerando então uma solução. Para avaliar o desempenho dessa busca, podem ser considerados alguns critérios como a completeza e a otimização da algoritmo. Uma busca será completa se, dado um grafo com um caminho existente de um nó *i* a um nó *n*, o algoritmo de busca encontrar essa solução. Por outro lado, a busca será ótima se a solução encontrada apresentar o menor custo de caminho entre todas as outras possíveis soluções (NORVIG; RUSSELL, 2014).

O termo "sem informação"se refere ao fato do algoritmo não receber nenhuma informação prévia sobre os estados, como, por exemplo, a distância entre o nó objetivo e o nó atual. Dessa forma, a busca precisa gerar sucessores e fazer a distinção entre o nó objetivo e o não objetivo (NORVIG; RUSSELL, 2014). Para realizar esse processo existem dois métodos fundamentais:

Busca em Largura, em inglês *Breadth First Search* (BFS) e Busca em Profundidade, em inglês *Depth First Search* (DFS). Esses métodos se distinguem pela escolha utilizada na hora de explorar os vértices e arestas.

2.4.1 Busca em Largura

A Busca em Largura (*Breadth First Search* - BFS) é uma estratégia que explora todos os vértices (ou nós) de um grafo, utilizando como critério de seleção o vértice visitado e não marcado que foi adicionado há mais tempo. Basicamente escolhe-se um vértice raiz e expande-se todos os vértices adjacentes a esse vértice. Em seguida, esses vértices sucessores são expandidos, depois os sucessores dos sucessores até que a busca esteja completa (NORVIG; RUSSELL, 2014), como exemplificado pela Figura 15, em que a cada etapa, o próximo nó a ser explorado é identificado com uma seta. A cor azul indica que o vértice não foi explorado, a cor preta indica que o vértice foi explorado e a cor vermelha é utilizada para destacar a aresta explorada.

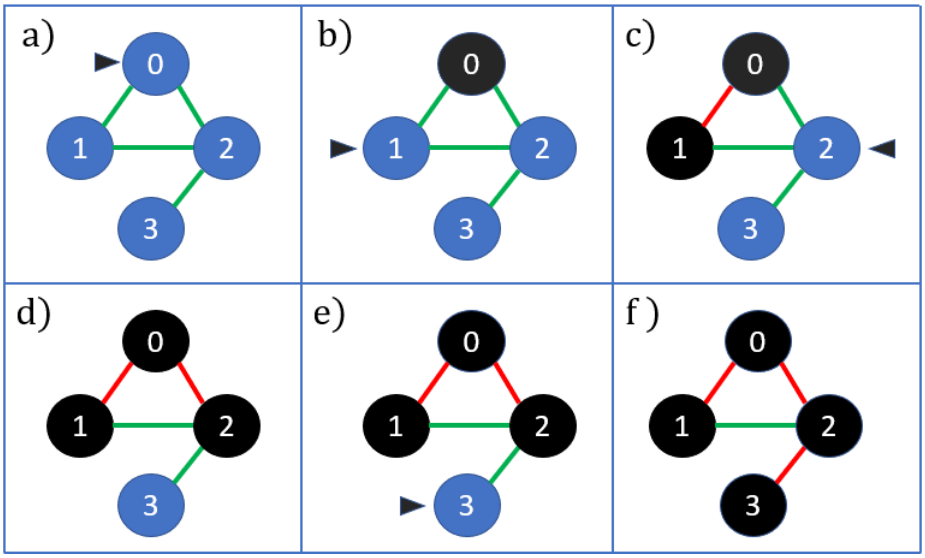


Figura 15 – Exemplo do funcionamento do algoritmo BFS.

A característica que distingue esse método do de busca em profundidade é a estrutura de dados utilizada para organizar a busca. No BFS é utilizada uma fila, enquanto no DFS é utilizada uma pilha (NORVIG; RUSSELL, 2014). Uma fila é uma estrutura com a política conhecida como FIFO (*First In, First Out*) ou em português, primeiro a entrar, primeiro a sair. Dessa forma, a inserção de elementos acontece na última posição da fila e a remoção de elementos acontece na primeira posição. O Algoritmo é apresentado em 1:

Algoritmo 1: Busca em largura

```
   Entrada: Grafo G = (V, E), vértice inicial v
1  início
2      Crie uma fila F vazia
3      Insira v na última posição da fila F
4      Marque v como visitado
5      enquanto F ≠ ∅ faça
6          v recebe o primeiro elemento de F
7          Remova o vértice v da fila
8          para todo vértice w vizinho de v faça
9              se w não foi marcado como visitado então
10                 Insira w na última posição de F
11                 Marque w como visitado
12             fim
13         fim
14     fim
15 fim
```

Para elucidar o funcionamento do algoritmo, observe a Figura 16. Neste exemplo, utiliza-se o vértice 0 como nó raiz. O procedimento de resolução será o seguinte:

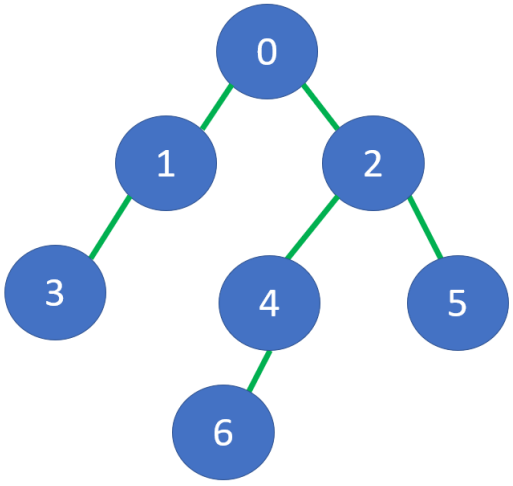


Figura 16 – Grafo não dirigido.

- Cria-se uma fila vazia. (Etapa referente a linha 2 do Algoritmo 1).
- Adiciona-se o nó 0 na fila de vértices a ser explorado. (Etapa referente a linha 3 do Algoritmo 1).
- Visita-se o primeiro elemento da fila, no caso o nó 0 e marque-o como visitado, como mostrado na Figura 17 (b). (Etapa referente a linha 4 do Algoritmo 1).
- Verifica-se a existência de vizinhos do vértice 0. Ele apresenta dois vizinhos, os vértices 1 e 2. Esses vértices ainda não foram explorados, então são adicionados à fila. (Etapa referente as linha 7, 8 e 9 do Algoritmo 1).
- Retira-se o vértice 0 da fila.

- A fila agora apresenta os vértices $[1, 2]$.
- Repete-se o processo novamente: visita-se o primeiro elemento da fila, no caso 1, como mostrado na Figura 17 (c). (Etapa referente a linha 6 do Algoritmo 1).
- Verificam-se os vizinhos, que para o vértice 1 são 0 e 3. Entretanto, o vértice 0 já foi visitado, então adiciona-se somente o vértice 3 na fila.
- Retira-se o vértice 1 da fila.
- A fila agora contém os elementos $[2, 3]$
- Visita-se o vértice 2 e marca-o como visitado, como pode ser visto na Figura 17 (d).
- Verificam-se os vizinhos de 2, no caso 0, 4 e 5. Como 0 já foi visitado, adicionam-se somente os vértices 4 e 5 na fila.
- Retira-se o vértice 2 da fila.
- A fila agora contém os nós $[3, 4, 5]$.
- Visita-se o vértice 3, como mostrado na Figura 17 (e). Ele só apresenta o vértice 1 como vizinho. Como 1 já foi visitado, nada acontece.
- Retira-se o vértice 3 da fila.
- A fila agora contém $[4, 5]$.
- Visita-se o nó 4, como pode ser visto na Figura 17 (f). Os vizinhos dele são 2 e 6. O vértice 2 já foi visitado, então adiciona-se somente o 6 a lista.
- Retira-se o nó 4 da fila.
- Agora a fila contém $[5, 6]$.
- Visita-se o vértice 5, como mostrado na Figura 17 (g). Seu único vizinho, o nó 2, já foi visitado. Dessa forma, não se adiciona nenhum elemento na fila.
- Retira-se o nó 5 da fila.
- A fila contém somente o elemento $[6]$
- Visita-se o elemento 6, como pode ser visto na Figura 17 (h) O mesmo só tem o elemento 4 como vizinho, porém, este elemento já foi visitado. Então nenhuma alteração é realizada.
- Retira-se o nó 6 da fila.
- A fila agora está vazia. Dessa forma, a busca já foi realizada no grafo.

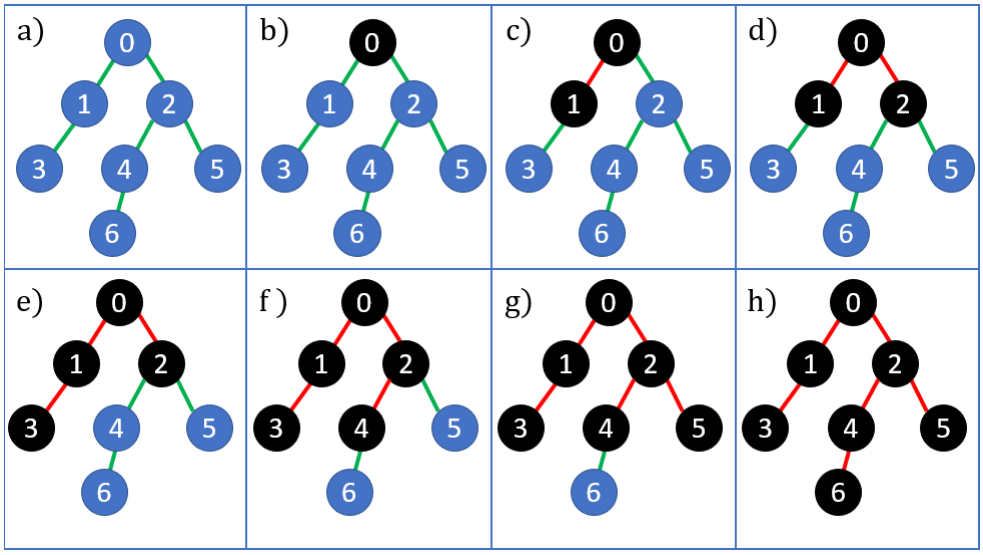


Figura 17 – Etapas da expansão da busca em largura.

A busca resultante desse grafo seria: $[0, 1, 2, 3, 4, 5, 6]$. O processo de expansão da busca BFS para esse exemplo é exposto na Figura 17.

A busca em largura fornece uma solução completa, ou seja, tem a garantia de encontrar uma solução quando esta existir, se o nó objetivo estiver a uma profundidade finita (NORVIG; RUSSELL, 2014). Além disso, para grafos não ponderados ela fornece o menor caminho entre qualquer nó e o nó raiz (BEAMER; ASANOVIĆ; PATTERSON, 2013). Para analisar a complexidade de tempo e de memória são utilizados os seguintes termos: *b*, representa o número máximo de vizinhos de qualquer nó, também chamado fator de ramificação e *d*, o número de passos no decorrer do caminho da raiz até o estado objetivo mais próximo (NORVIG; RUSSELL, 2014).

Como visto anteriormente, a busca em largura atua expandindo os nós sucessores até encontrar o nó objetivo. Cada uma dessas expansões é considerada um nível. Dessa forma, saindo do vértice raiz são gerados *b* vértices no primeiro nível, e para cada um dos vértices serão gerados outros *b* vértices, totalizando b^2 vértices no segundo nível. Essa expansão contínua até que todos os nós sejam visitados ou até encontrar o nó objetivo. Sendo assim, para o pior dos casos, em que o último nó se encontra na profundidade *d*, o número total de nós gerados seria $O(b^d)$. Já em relação à memória, devido à necessidade de armazenar os nós expandidos no conjunto explorado, a complexidade de espaço será $O(b^d)$ (NORVIG; RUSSELL, 2014). Essa necessidade de espaço se caracteriza como uma das desvantagens do uso da busca em largura (KORF, 1985).

2.4.2 Busca em Profundidade

A Busca em Profundidade (*Depth First Search* - DFS), diferentemente da busca em largura, percorre todos os vértices do grafo utilizando como critério de seleção o vértice visitado e não marcado que foi adicionado há menos tempo. Dessa forma, o algoritmo sempre expande o

nó mais profundo até que não existam sucessores (NORVIG; RUSSELL, 2014). Basicamente escolhe-se um vértice raiz, verifica-se os vizinhos e expande-se o último vizinho adicionado. Esse processo se repete até que não exista um sucessor. Quando isso acontece, executa-se um retrocesso e o próximo vizinho é escolhido para a expansão. A Figura 18 exemplifica o funcionamento do DFS. Em cada etapa, o próximo nó a ser explorado é identificado com uma seta. A cor azul indica que o vértice não foi explorado, a cor preta indica que o vértice foi explorado, a seta indica o nó em expansão e a cor vermelha a aresta explorada.

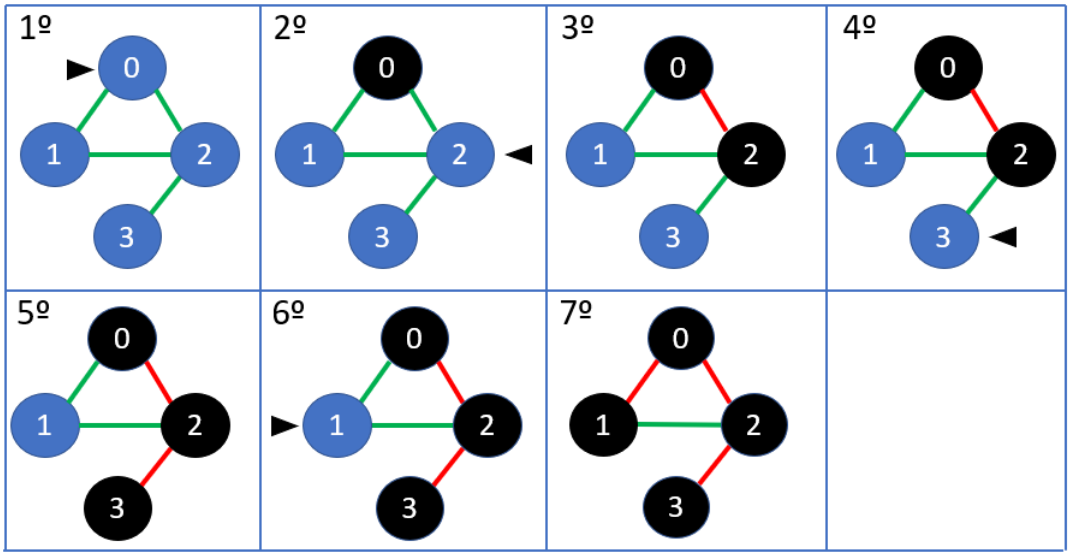


Figura 18 – Busca em profundidade.

A característica principal desse método é a utilização da estrutura de dados do tipo pilha (NORVIG; RUSSELL, 2014). Uma pilha é uma estrutura com a política de LIFO (*Last In, First Out*) ou em português, primeiro a entrar, último a sair. Pode-se comparar essa estrutura a uma pilha de pratos sujos, em que a inserção de elementos acontece no topo da pilha e a remoção também acontece no topo da pilha.

O Algoritmo é apresentado em 2:

Para explicar o funcionamento do algoritmo, novamente utiliza-se a Figura 16, com o vértice 0 como nó raiz. O procedimento de resolução será o seguinte:

- Cria-se uma pilha vazia. (Etapa referente a linha 2 do Algoritmo 2).
- Adiciona-se o nó 0 na pilha de vértices a ser explorado. (Etapa referente a linha 3 do Algoritmo 2).
- Visita-se o elemento do topo da pilha, no caso o nó 0 e marque-o como visitado, conforme mostrado na Figura 19 (b). (Etapa referente a linha 4 do Algoritmo 2).

Algoritmo 2: Busca em profundidade

```
   Entrada: Grafo G = (V, E), vértice inicial v
1  início
2      Crie uma pilha P vazia
3      Insira v no topo da pilha P
4      Marque v como visitado
5      enquanto P ≠ ∅ faça
6          v recebe o elemento do topo de P
7          Remova o vértice v da fila
8          para todo vértice w vizinho de v faça
9              se w não foi marcado como visitado então
10                 Insira w no topo de P
11                 Marque w como visitado
12             fim
13         fim
14     fim
15 fim
```

- Verifica-se a existência de vizinhos do vértice 0. Ele apresenta dois vizinhos, os vértices 1 e 2. Esses vértices ainda não foram explorados, então são adicionados à pilha. (Etapa referente as linha 7, 8 e 9 do Algoritmo 2).
- Retira-se o vértice 0 da pilha.
- A pilha agora apresenta os vértices $[1, 2]$.
- Repete-se o processo novamente: visita-se o elemento do topo da pilha, no caso 2, como mostrado na Figura 19 (c).
- Verificam-se os vizinhos, que para o vértice 2 são 0, 4 e 5. No entanto, o vértice 0 já foi visitado, então adicionam-se somente os vértices 4 e 5 na pilha.
- Retira-se o vértice 2 da pilha.
- A pilha agora contém os elementos $[1, 4, 5]$.
- Visita-se o vértice 5 e marque-o como visitado, conforme mostrado na Figura 19 (d).
- Verificam-se os vizinhos de 5, no caso o vértice 2. Como 2 já foi visitado e não existem mais vizinhos, realiza-se o retrocesso.
- Retira-se o vértice 5 da pilha.
- A pilha agora contém os nós $[1, 4]$.
- Visita-se o vértice 4, como mostrado na Figura 19 (e). Ele apresenta os vértice 2 e 6 como vizinhos. Como 2 já foi visitado, adiciona-se somente o vértice 6 na pilha.
- Retira-se o vértice 4 da pilha.

- A fila agora contém $[1, 6]$.
- Visita-se o nó 6, como mostrado na Figura 19 (f). O único vizinho é 4. O vértice 4 já foi visitado e não existem mais vizinhos, então realiza-se o retrocesso.
- Retira-se o nó 6 da pilha.
- Agora a pilha contém somente o elemento $[1]$.
- Visita-se o vértice 1, como pode ser visto na Figura 19 (g). Os seus vizinhos são 0 e 3. Como o vértice 0 já foi visitado, adiciona-se somente o 3 na pilha.
- Retira-se o nó 1 da pilha.
- A fila contém somente o elemento $[3]$.
- Visita-se o elemento 3, como mostrado na Figura 19 (h). O mesmo só tem o elemento 1 como vizinho, porém, este elemento já foi visitado. Então nenhuma alteração é realizada.
- A pilha agora está vazia. Dessa forma, a busca já foi realizada no grafo.

A busca resultante desse grafo seria: $[0, 2, 5, 4, 6, 1, 3]$. O processo de expansão da busca DFS para esse exemplo é exposto na Figura 19.

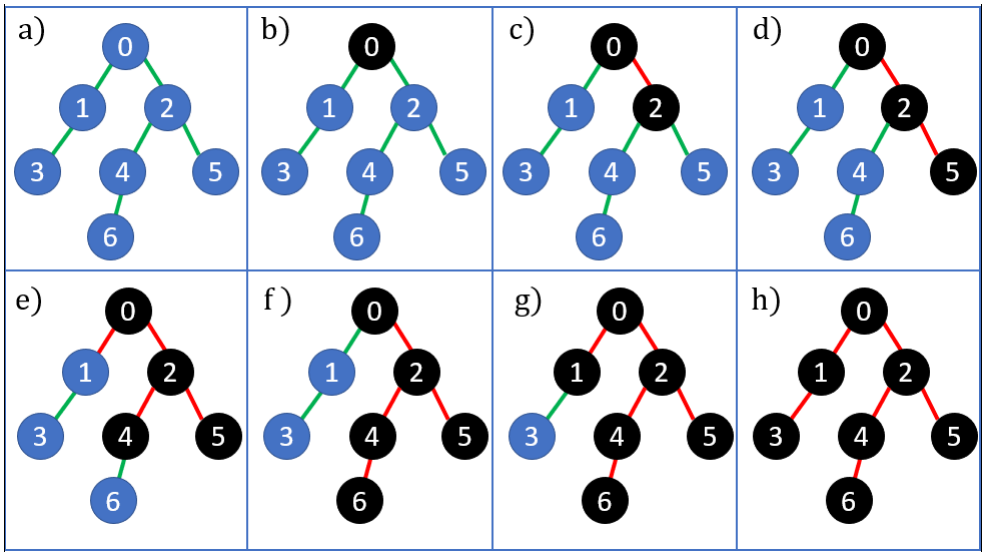


Figura 19 – Etapas da expansão da busca em profundidade.

Com relação ao desempenho, a versão do DFS em grafos, que evita caminhos redundantes e estados repetidos, é completa, pois, irá expandir cada nó (NORVIG; RUSSELL, 2014). Além disso, diferentemente da busca em largura, não fornece o menor caminho entre dois nós no grafo, só informa se existe um caminho entre os dois nós, sem garantia de ser o menor. Devido à forma em que se expande, o algoritmo contorna a questão da limitação de memória, pois, só armazena o caminho do nó inicial ao nó atual na execução do algoritmo. Uma vez que um nó é expandido

ele pode ser removido da memória. Sendo assim , a complexidade de memória é avaliada como $O(d)$, onde d representa o número de passos no decorrer do caminho, do estado inicial até o estado objetivo mais próximo (KORF, 1985).

2.5 Estratégias de busca informada

O conceito de busca informada significa que o método apresenta algum tipo de conhecimento do problema em questão, além das definições iniciais, como a estimativa de distância entre os vértices ou o custo de travessia. Dessa forma, é possível obter soluções de maneira mais eficiente se comparado as buscas sem informação (NORVIG; RUSSELL, 2014).

2.5.1 Busca pela melhor escolha

A busca pela melhor escolha é um algoritmo que percorre um grafo que utiliza como critério de seleção para a expansão de um nó uma função conhecida como **função de avaliação** $f(n)$. Essa função de avaliação é interpretada como uma estimativa do custo, dessa forma, para se expandir o nó é analisado aquele que detém o menor valor (menor custo) na função de avaliação (NORVIG; RUSSELL, 2014). Para as estratégias de busca pela melhor escolha, a forma como f é escolhida determina o tipo de algoritmo. Para grande parte desses algoritmos, um componente essencial em f é a função heurística. A **função heurística** é um dos meios de transmitir conhecimento adicional ao algoritmo de busca e será denotada por $h(n)$:

$h(n) =$ estimativa do custo do caminho com o menor custo do nó n até um nó objetivo

2.5.1.1 Busca gulosa pela melhor escolha

A busca gulosa pela melhor escolha, em inglês *Greedy Best First Search* (GBFS) é um algoritmo que procura expandir o nó que se encontra mais próximo do objetivo. Para isso, utiliza apenas o termo referente a função heurística em sua função de avaliação, $f(n) = h(n)$. A ideia por trás do algoritmo é que a expansão para o nó mais próximo do objetivo possa gerar uma solução mais rápida (ASAI; FUKUNAGA, 2017).

Para exemplificar o funcionamento da busca gulosa, será utilizado a Figura 20, na qual é apresentado um grafo com os valores calculados de uma função heurística. A função heurística utilizada foi a estimativa da distância em linha reta de cada nó até o nó objetivo escolhido, que no caso é o I. Como exemplo, tem-se que a distância do nó A ao nó I é $h(A) = 24$. Além disso, para o nó *I*, que é o nó objetivo, $h(I) = 0$.

Como visto anteriormente, o critério de seleção para o próximo vértice a ser explorado, considera o vértice que apresenta o menor valor da função de avaliação. O algoritmo funcionará da seguinte forma:

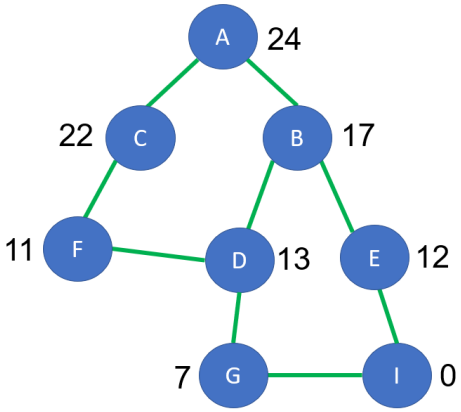


Figura 20 – Grafo com valores da função heurística.

- Iniciando no nó A, o próximo nó com o menor valor de heurística será o nó B com $h(B) = 17$, conforme mostrado na Figura 21 (b).
- A partir do nó B, o próximo nó com o menor valor será o nó E com $h(E) = 12$, como mostrado na Figura 21 (c).
- Por fim, o último nó com p menor valor de heurística será o nó I com $h(I) = 0$, conforme mostrado na Figura 21 (d).

Tais etapas são visualizadas na Figura 21, que exibe os nós não expandidos com a cor azul e os nós expandidos com a cor preta.

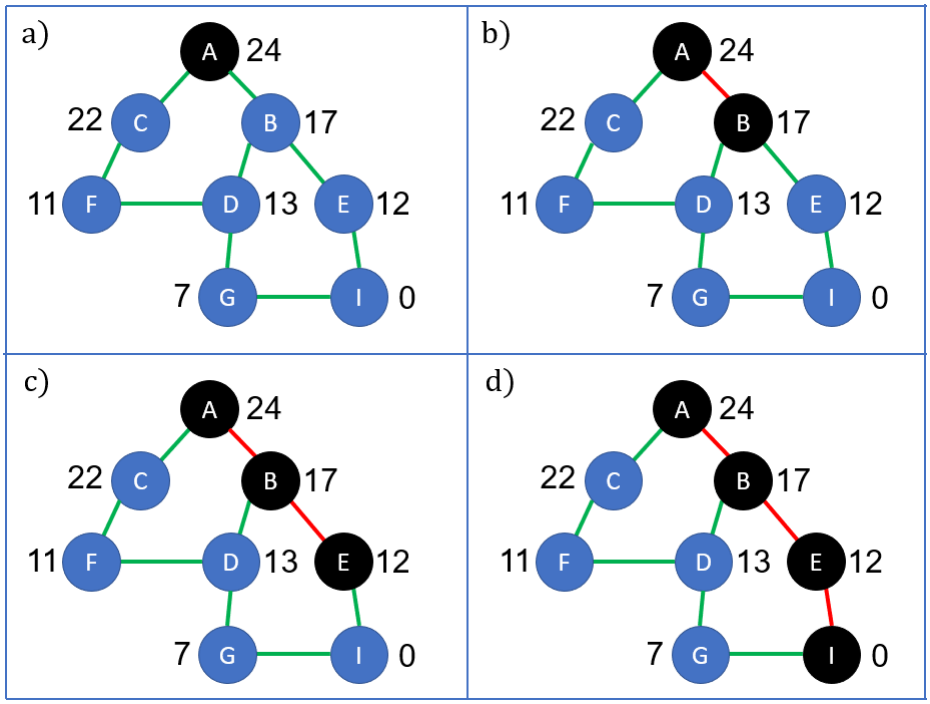


Figura 21 – Etapas de expansão da busca gulosa de melhor escolha.

Analisando a resposta obtida no exemplo, percebe-se que a busca gulosa sempre tenta chegar o mais próximo possível do objetivo a cada passo, considerando o menor valor da função

de avaliação, por esse motivo recebe o apelido de “busca gulosa”. Como a busca só expande o nó que está no caminho da solução, o custo de busca é mínimo (NORVIG; RUSSELL, 2014). Entretanto, a busca não apresenta resultado ótimo. A Figura 22 apresenta o mesmo grafo da Figura 20, mas com os valores de custo entre vértices. O caminho obtido anteriormente com o GBFS indicou o caminho $[A, B, E, I]$ como menor caminho. Calculando-se o custo deste trajeto, obtém-se o valor de 30 unidades. Porém, se o caminho $[A, B, D, G, I]$ for utilizado, o custo é de 27 unidades. Isso demonstra que a busca gulosa de melhor escolha não é ótima.

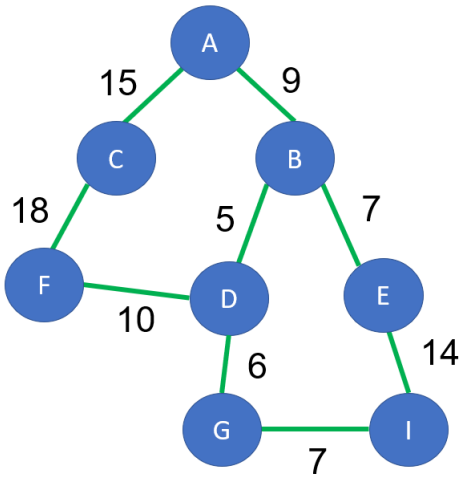


Figura 22 – Grafo com custos de trajeto entre vértices.

De acordo com Asai e Fukunaga (2017), o GFBS se mostra um algoritmo útil em encontrar uma solução satisfatória rapidamente e tem sido a base para planejadores independentes de caminho. Com relação ao desempenho, a versão da busca em grafos é completa quando aplicada em espaços finitos, porém, o mesmo não se repete em espaços infinitos. Com relação à complexidade de espaço e tempo, no pior dos casos esse valor é de $O(b^d)$, sendo d a profundidade máxima do espaço de busca (NORVIG; RUSSELL, 2014).

2.5.1.2 Algoritmo de Dijkstra

O algoritmo de Dijkstra foi criado por Edsger Wybe Dijkstra, de onde se origina o nome, e publicado em 1959 (FELNER, 2011). A principal ideia do algoritmo é classificar os vértices de um grafo durante a exploração, considerando como critério a distância do vértice de origem ao vértice explorado e armazenar isso em uma estrutura do tipo fila de prioridade. Depois disso, os vértices que são mais próximos da origem são removidos e a lista de vizinhos é atualizada (FELNER, 2011). Este algoritmo é utilizado para descobrir o menor caminho entre um nó origem e todos os outros vértices no grafo, caso os pesos das arestas sejam não negativos (XU et al., 2007), (GOLDBERG; TARJAN, 1996). O Algoritmo é apresentado em 3:

Algoritmo 3: Algoritmo de Dijkstra

```
   Entrada: Grafo G = (V, E), vértice inicial v
 1 início
 2     Defina a distância do vértice inicial para o vértice inicial como 0
 3     Defina a distância do vértice inicial para todos os outros vértices como ∞
 4     enquanto existirem vértices não visitados faça
 5         Visite o vértice não visitado com a menor distância conhecida do vértice inicial
 6         Defina esse vértice como vértice atual
 7         para todo vizinho não visitado do vértice atual faça
 8             Calcule a distância do vértice inicial
 9             se a distância calculada desse vértice é menor do que a distância conhecida então
10                 Atualize a menor distância para esse vértice
11                 Atualize o vértice anterior com o vértice atual
12             fim
13             Adicione o vértice atual a lista de vértices visitados
14         fim
15     fim
16 fim
```

O funcionamento básico do algoritmo consiste das seguintes etapas (NOTO; SATO, 2000), (JOSHI, 2017):

1. Defina o nó que será visitado baseado no menor custo/distância. (Etapa referente as linhas 5 e 6 do Algoritmo 3).

2. Uma vez definido o nó que será visitado, verifique os vizinhos desse nó. (Etapa referente a linha 7 do Algoritmo 3).

3. Calcule o custo para cada nó vizinho, realizando a soma dos custos das arestas do nó origem até o nó em questão. (Etapa referente a linha 8 do Algoritmo 3).

4. Se o custo para um nó for menor que um certo custo conhecido, atualize os valores das distâncias. (Etapa referente a linha 9, 10 e 11 do Algoritmo 3).

5. Escolha o próximo nó baseado no menor custo. (Etapa referente a linha 5 do Algoritmo 3).

6. Repita as etapas até que a busca seja finalizada.

Para exemplificar o funcionamento do algoritmo de Dijkstra, será utilizado como nó inicial o vértice 1 e como objetivo o vértice 5, conforme o grafo da Figura 23.

De início, considera-se a distância entre o nó inicial e os outros nós como infinito, pois, de fato não se sabe se esses outros vértices podem ser alcançados. Como a busca se inicia no nó 1, a distância para esse vértice será definida como 0. Utiliza-se uma tabela para manter as informações da busca, tais como a situação do vértice, a menor distância entre o vértice inicial e os outros vértices, assim como a coluna vértice anterior que ajuda a manter um histórico de onde cada vértice é oriundo. A situação inicial da busca é assim como a tabela podem ser

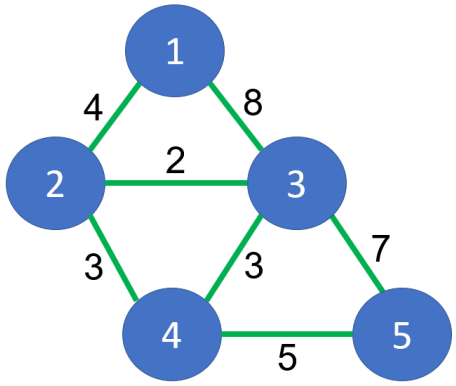
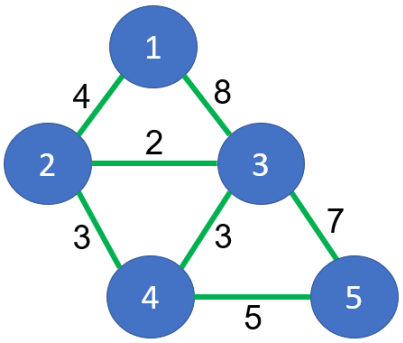


Figura 23 – Grafo ponderado não dirigido.

visualizadas na Figura 24. Outro passo importante é o acompanhamento dos vértices visitados e não visitados. Para isso são utilizados dois vetores: *visitados* e *não_visitados*. Como nenhum dos nós foi visitado, os dois vetores criados apresentam a seguinte forma:

$$\text{visitados} = [\ ] \text{ e não\_visitados} = [1, 2, 3, 4, 5]$$



Vértice	Situação	Menor distância de 1	Vértice anterior
1	-	0	-
2	-	∞	-
3	-	∞	-
4	-	∞	-
5	-	∞	-

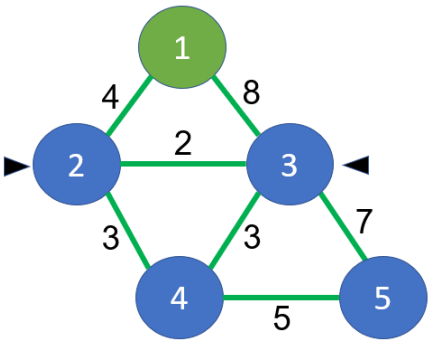
Figura 24 – Etapa inicial da busca.

Seguindo as etapas descritas anteriormente, deve-se iniciar a busca no vértice com a menor distância. De acordo com a tabela da Figura 24, todos os vértices apresentam distância infinita, exceto pelo vértice 1. Por isso, a busca inicia-se nele. Os vizinhos do nó 1 são os nós 2 e 3, assim calcula-se a distância para esses dois nós, que será a soma das distâncias de 1 ao respectivo nó:

$$dist_{1\to 2} = 0 + 4 = 4$$

$$dist_{1\to 3} = 0 + 8 = 8$$

Agora compara-se a distância calculada com a distância atual exibida na Figura 24. Como as distâncias para 2 e 3 estão marcadas como infinito, a tabela será atualizada com os novos valores. Como os vizinhos de 1 já foram analisados, ele será marcado como visitado. O resultado dessa etapa é mostrado na Figura 25.



Vértice	Situação	Menor distância de 1	Vértice anterior
1	atual	0	-
2	-	4	1
3	-	8	1
4	-	∞	-
5	-	∞	-

Figura 25 – Etapa 2 da busca.

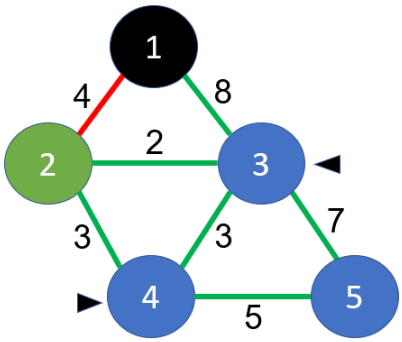
A situação atual dos vetores é: visitados = $[1]$ e não_visitados = $[2, 3, 4, 5]$

As etapas se repetem. Procura-se o vértice não visitado com o menor custo, que nesse caso será o vértice 2, com o valor de 4 unidades. O próximo passo é verificar os vizinhos de 2 que são 3 e 4. Após isso, calcula-se a distância para os dois vértices:

$$dist_{1\to3} = 4 + 2 = 6$$

$$dist_{1\to4} = 4 + 3 = 7$$

A distância para 3 passando pelo nó 2 tem o valor de 6, sendo menor que a distância apresentada anteriormente na tabela. Dessa forma, essa o valor será atualizado, a mesma coisa acontece para o nó 4. O resultado pode ser visualizado na Figura 26.



Vértice	Situação	Menor distância de 1	Vértice anterior
1	visitado	0	-
2	atual	4	1
3	-	6	2
4	-	7	2
5	-	∞	-

Figura 26 – Etapa 3 da busca.

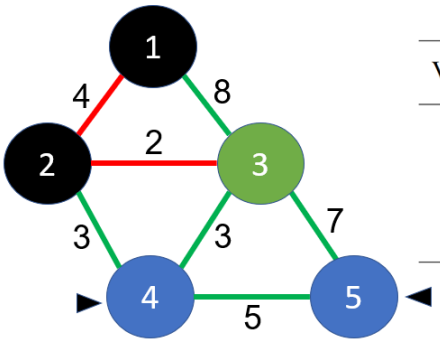
A situação atual dos vetores é: visitados = $[1, 2]$ e não_visitados = $[3, 4, 5]$

Continua-se a busca escolhendo o vértice não visitado com o menor valor de distância. Para essa etapa, o nó escolhido será o 3. Os vizinhos do nó 3, não visitados, são 4 e 5. Então, calcula-se a distância para esses nós:

$$dist_{1\to4} = 6 + 3 = 9$$

$$dist_{1\to4} = 6 + 7 = 13$$

A distância atual na tabela para o nó 4 apresenta um valor inferior ao valor calculado, dessa forma não será atualizada. Já para o nó 5, o valor será atualizado, pois, até agora o valor apresentado era infinito. A Figura 27 exibe o resultado desta etapa.



Vértice	Situação	Menor distância de 1	Vértice anterior
1	visitado	0	-
2	visitado	4	1
3	atual	6	2
4	-	7	2
5	-	13	3

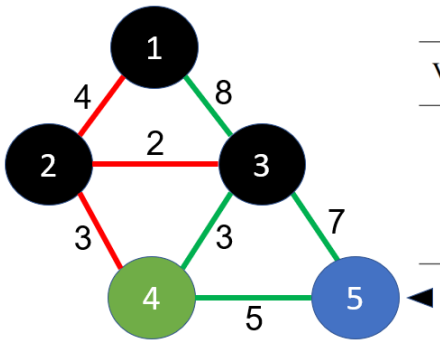
Figura 27 – Etapa 4 da busca.

A situação dos vetores é: visitados = [1,2,3] e não_visitados = [4,5]

Para a próxima etapa do processo, o nó não visitado com o menor valor será o 4. O único vizinho ainda não visitado é o nó 5. O cálculo da distância para ele será:

$$dist_{1\to5} = 7 + 5 = 12$$

O valor calculado é inferior ao valor atual presente na tabela, dessa forma a ela será atualizada com o novo valor, como pode ser visualizado na Figura 28.



Vértice	Situação	Menor distância de 1	Vértice anterior
1	visitado	0	-
2	visitado	4	1
3	visitado	6	2
4	atual	7	2
5	-	12	4

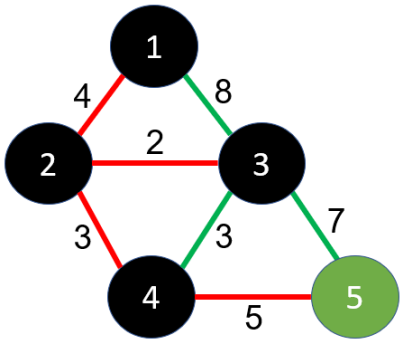
Figura 28 – Etapa 5 da busca.

Os vetores agora são: visitados = [1,2,3,4] e não_visitados = [5]

Por último, visita-se o vértice 5. Como este não apresenta mais nenhum vizinho não visitado, a busca está completa, conforme a Figura 29.

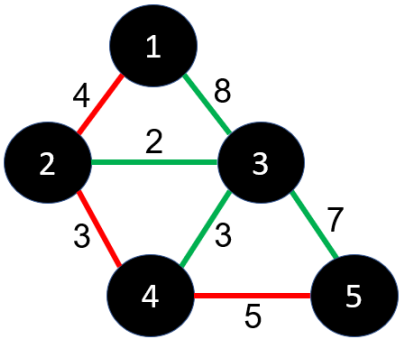
O resultado obtido pelo algoritmo de Dijkstra é que o menor caminho do nó 1 ao nó 5 apresenta um custo de 12 unidades e é composto pelos vértices: $[1, 2, 4, 5]$, conforme exibido na Figura 30. Esse caminho é destacado pelas arestas de cor vermelha.

Conforme abordado anteriormente, uma característica importante do algoritmo de Dijkstra é que ele não só fornece o menor caminho de um nó inicial a um nó objetivo, como também



Vértice	Situação	Menor distância de 1	Vértice anterior
1	visitado	0	-
2	visitado	4	1
3	visitado	6	2
4	visitado	7	2
5	atual	12	4

Figura 29 – Etapa 6 da busca.



Vértice	Situação	Menor distância de 1	Vértice anterior
1	visitado	0	-
2	visitado	4	1
3	visitado	6	2
4	visitado	7	2
5	visitado	12	4

Figura 30 – Caminho obtido pelo algoritmo.

fornece o menor caminho desse mesmo nó inicial a qualquer outro nó no grafo, caso esse nó seja alcançável (MEHLHORN; SANDERS, 2008). Por exemplo, se o nó objetivo fosse o nó 3, o menor caminho teria 6 unidades e seria composto de $[1, 2, 3]$.

Em relação ao desempenho, o algoritmo de Dijkstra fornece o caminho ótimo. Entretanto, devido à forma que o método funciona, existe uma baixa eficiência e um longo tempo de busca, quando a distância para o objetivo é grande, sendo inadequado para problemas de tempo real (NOTO; SATO, 2000).

A complexidade do algoritmo, considerando a implementação original de Dijkstra era de $O(n^2)$, devido a sua implementação utilizar uma estrutura de dados do tipo *heap*[1]. Uma implementação moderna, utilizando um *Fibonacci heap* (um tipo de estrutura de dados para filas de prioridade) roda em tempo de $O(m + n \log n)$, em que *m* é o número de arestas e *n* o de vértices (XU et al., 2007), (MEHLHORN; ORLIN; TARJAN, 1987). Devido a complexidade de se utilizar a *Fibonacci heap*, optou-se pela utilização de um *heap*.

2.5.1.3 Busca A*

Outro algoritmo que compõem a classe dos algoritmos de melhor escolha é a busca A*. Este algoritmo é amplamente utilizado em planejamento de caminhos e busca em grafos (DUCHOŇ et al., 2014). O A* foi desenvolvido em 1968 por três pesquisadores, mais especifica-

[1] Um *heap* é uma estrutura de dados que pode ser visualizada como uma árvore binária quase completa (CORMEN et al., 2001).

mente Bertram Raphael, Nils Nilsson e Peter Hart, do *Stanford Research International* (HART; NILSSON; RAPHAEL, 1968). O método desenvolvido por esses pesquisadores classifica os nós mediante uma combinação de $g(n)$, que é o custo do caminho do nó inicial até o nó *n*, e $h(n)$, que é uma função heurística que estima o custo do caminho de menor custo do nó *n* ao nó objetivo (NORVIG; RUSSELL, 2014), de forma que a função de avaliação $f(n) = g(n) + h(n)$ seja minimizada.

Como visto anteriormente, o GBFS encontra uma solução satisfatória rapidamente, pois, expande a busca em soluções promissoras baseadas na função heurística, entretanto, não garante o menor caminho. Por outro lado, o algoritmo de Dijkstra fornece o menor caminho, mas devido à forma que se expande acaba explorando nós não promissores. A busca A* une as melhores características dos dois algoritmos citados e, dependendo da função heurística $h(n)$ utilizada, a busca será ótima e completa (NORVIG; RUSSELL, 2014). Assim como no algoritmo de Dijkstra, o A* utiliza uma fila de prioridade para armazenar os nós candidatos a expansão (ZENG; CHURCH, 2009).

O Algoritmo é apresentado em 4. Para a implementação do algoritmo são utilizadas duas estruturas de dados, um conjunto que contém os vértices abertos *O* definida como uma fila de prioridade e um conjunto que contém os vértices processados *C* (CHOSET et al., 2005).

Algoritmo 4: Algoritmo A*

Entrada: Grafo $G = (V, E)$, vértice inicial v
Saída: Um caminho entre o nó inicial e o objetivo

início
  Insira o vértice inicial em *O*
  Defina *C* como vazia
  enquanto $O \neq \emptyset$ **faça**
    Pegue o vértice de *O* com o menor valor de $f(n)$
    Defina esse vértice como v_{atual}
    Remova v_{atual} de *O* e adicione em *C*
    se v_{atual} = *objetivo* **então**
      Termine a busca
    fim
    para todo *vizinho de* $v_{atual} \notin C$ **faça**
      se *vizinho* $\notin$ *O* **então**
        Adicione o vizinho a *O*
      senão
        se $g(v_{atual}) + custo(v_{atual}, vizinho) < g(vizinho)$ **então**
          Atualize o vértice pai do vizinho para v_{atual}
          $g(vizinho) = g(v_{atual}) + custo(v_{atual}, vizinho)$
        fim
      fim
    fim
  fim
fim

O ponto crucial para determinar se a busca A* é ótima será a heurística $h(n)$ adotada. O primeiro requisito é que $h(n)$ seja uma **heurística admissível**, o que significa que ela jamais superestima o custo de alcançar o objetivo. Um exemplo de heurística admissível é a distância

em linha reta entre dois pontos, pois, a menor distância entre dois pontos é uma reta (NORVIG; RUSSELL, 2014). Quanto mais próximo o $h(n)$ estimado estiver da distância real, melhor será o processamento.

O segundo requisito para a otimalidade é a chamada **consistência**. Uma heurística $h(n)$ é considerada consistente se, para todo nó *n* e para todo sucessor n' de n originado de certa ação a, a estimativa de custo para atingir o objetivo de n não exceder a soma do custo do passo de alcançar n' e o custo estimado para alcançar o objetivo de n' (NORVIG; RUSSELL, 2014):

$$h(n) \leq c(n, a, n') + h(n')$$

Alguns exemplos convencionais de heurísticas utilizadas são as de Manhattan, a Euclideana e a de Chebyshev (DUCHOŇ et al., 2014). As equações utilizadas para o cálculo são apresentadas a seguir (SINGH; YADAV; RANA, 2013). Para o cálculo das distâncias, são considerados dois pontos: $A = (x_a, y_a)$ e $B = (x_b, y_b)$, que representam os vértices do grafo no plano cartesiano.

A distância Euclidiana (Equação 2.1) calcula a raiz quadrada da diferença entre as coordenadas de um par de objetos da forma:

$$Dist_{AB} = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2} \tag{2.1}$$

A distância de Manhattan (Equação 2.2) calcula a diferença absoluta entre as coordenadas de um par de objetos da seguinte forma:

$$Dist_{AB} = |x_b - x_a| + |y_b - y_a| \tag{2.2}$$

A distância de Chebyshev (Equação 2.3) é calculada como a magnitude absoluta da diferença entre as coordenadas de um par de objetos:

$$Dist_{AB} = \max(|x_b - x_a|, |y_b - y_a|) \tag{2.3}$$

Foram implementadas essas três heurísticas, porém, a heurística que apresentou os melhores resultados foi a euclideana, sendo esta utilizada no restante do trabalho.

A seguir é apresentado um exemplo com intuito de elucidar o funcionamento do algoritmo. Considere a Figura 31,que apresenta um grafo não direcionado e ponderado. Os números da cor preta representam a heurística $h(n)$, calculada pela distância em linha reta entre o nó *n* e o nó objetivo. Os números em azul representam os pesos da aresta. Para esse exemplo, o nó inicial será o vértice A e o nó objetivo o vértice G.

Para organizar as etapas da busca A* serão utilizadas tabelas para armazenar informações como a situação de cada vértice, a menor distância para o nó raiz ($g(n)$), o valor calculado da heurística ($h(n)$), a distância total ($F(n)$) e o vértice anterior.

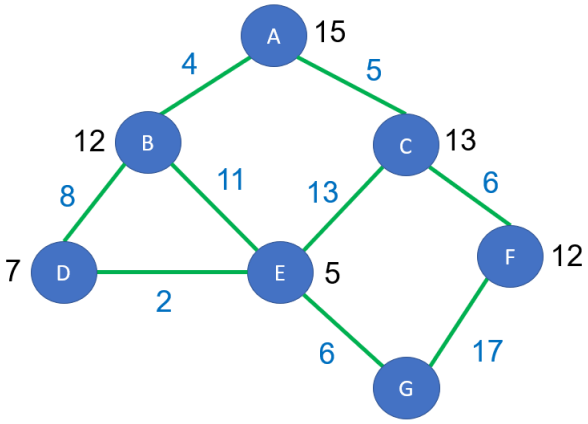


Figura 31 – Grafo com pesos e valores de $h(n)$.

O primeiro passo a ser tomado é atualizar o vértice A como o vértice atual e definir a distância de A para cada vértice como infinito, pois, ainda não se sabe se o vértice é alcançável, como observado na na Tabela 1 e Figura 32.

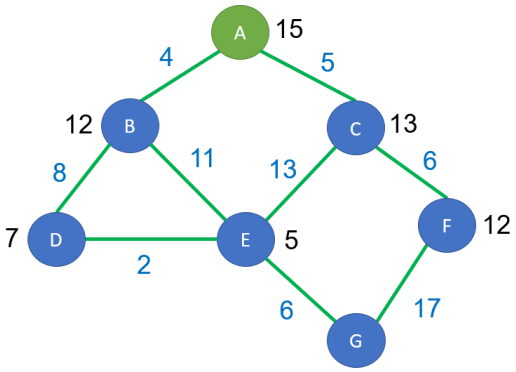


Figura 32 – Etapa inicial do A*.

Tabela 1 – Etapa inicial do A*.

Vértice	Situação	Menor distância de A (g(n))	Heurística Distância para G (h(n))	Distância Total (f(n))	Vértice anterior
A	Atual	0	15	15	-
B	-	∞	12	-	-
C	-	∞	13	-	-
D	-	∞	7	-	-
E	-	∞	5	-	-
F	-	∞	12	-	-
G	-	∞	0	-	-

No segundo plano, checa-se todos os nós adjacentes ao nó A e atualiza-se a menor distância de A. Com isso, soma-se as duas parcelas $h(n)$ e $g(n)$ e obtém-se o novo $f(n)$, como indicado na Figura 33 e na Tabela 2.

Após essa etapa, o nó A pode ser definido como visitado. O próximo passo é escolher o nó com a menor distância total $f(n)$, que será o nó B. Com isso, checa-se todos os vizinhos não visitados de B e calcula-se a nova distância para A, referente a cada vizinho, da seguinte forma:

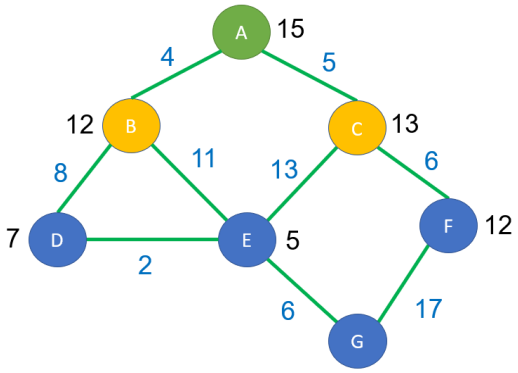


Figura 33 – Etapa 2 do A*.

Tabela 2 – Etapa 2 do A*.

Vértice	Situação	Menor distância de A (g(n))	Heurística Distância para G (h(n))	Distância Total (f(n))	Vértice anterior
A	Atual	0	15	15	-
B	-	4	12	16	A
C	-	5	13	18	A
D	-	∞	7	-	-
E	-	∞	5	-	-
F	-	∞	12	-	-
G	-	∞	0	-	-

$$Dist_{A \to D} = Dist_{A \to B} + Dist_{B \to D} = 4 + 8 = 12$$

$$Dist_{A \to E} = Dist_{A \to B} + Dist_{B \to E} = 4 + 11 = 15$$

Com esse valor calculado, verifica-se na Tabela 2 a distância atual que esses dois vértices tem para A, se o valor calculado for menor, a tabela será atualizada, como apresentado na Figura 34 e na Tabela 3.

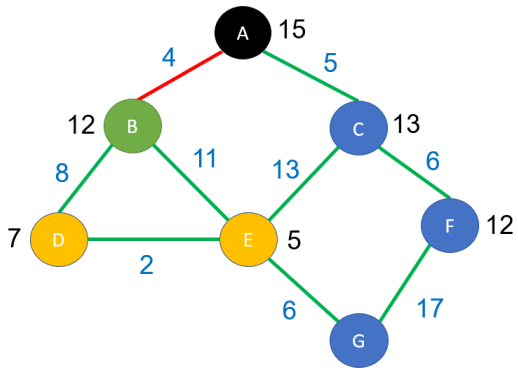


Figura 34 – Etapa 3 do A*.

O próximo nó a ser expandido será o C, pois, apresenta a menor $f(n)$. Calculam-se as novas distâncias para os vizinhos não visitados de C:

$$Dist_{A \to E} = 5 + 13 = 18$$

Tabela 3 – Etapa 3 do A*.

Vértice	Situação	Menor distância de A (g(n))	Heurística Distância para G (h(n))	Distância Total (f(n))	Vértice anterior
A	Visitado	0	15	15	-
B	Atual	4	12	16	A
C	-	5	13	18	A
D	-	12	7	19	B
E	-	15	5	20	B
F	-	∞	12	-	-
G	-	∞	0	-	-

$$Dist_{A \to F} = 5 + 6 = 11$$

Comparam-se os valores obtidos com os valores presentes na Tabela 3. Para o nó E, a nova distância obtida é maior que o valor encontrado anteriormente, dessa forma mantém-se o mesmo valor. Já para o nó F a distância para o nó A será atualizada.

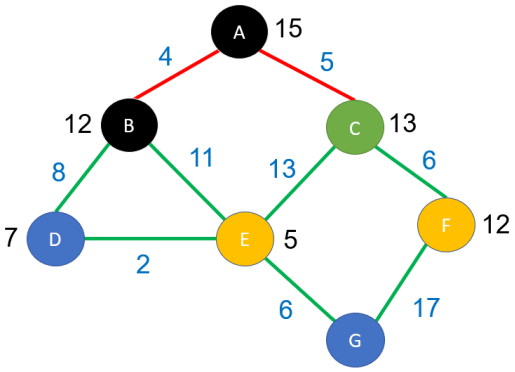


Figura 35 – Etapa 4 do A*.

Tabela 4 – Etapa 4 do A*.

Vértice	Situação	Menor distância de A (g(n))	Heurística Distância para G (h(n))	Distância Total (f(n))	Vértice anterior
A	Visitado	0	15	15	-
B	Visitado	4	12	16	A
C	Atual	5	13	18	A
D	-	12	7	19	B
E	-	15	5	20	B
F	-	11	12	23	C
G	-	∞	0	-	-

Na próxima etapa verifica-se novamente o vértice não visitado com menor *f(n)*, no caso o D. Verifica-se quais vértices são adjacentes a ele e calcula-se a distância de A até cada vértice. Para o D, o único vértice adjacente e não visitado é o E. A distância será:

$$Dist_{A \to E} = 12 + 2 = 14$$

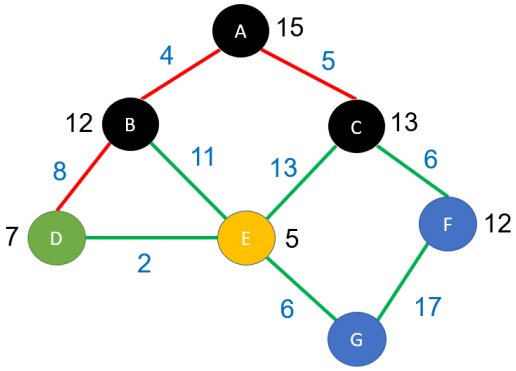


Figura 36 – Etapa 5 do A*.

Tabela 5 – Etapa 5 do A*.

Vértice	Situação	Menor distância de A (g(n))	Heurística Distância para G (h(n))	Distância Total (f(n))	Vértice anterior
A	Visitado	0	15	15	-
B	Visitado	4	12	16	A
C	Visitado	5	13	18	A
D	Atual	12	7	19	B
E	-	14	5	19	D
F	-	11	12	23	C
G	-	∞	0	-	-

como ela é menor do que a distância atual, a tabela será atualizada com o novo valor.

O próximo nó escolhido é o E. O único vizinho não visitado de E é o G. Calcula-se, então, a distância de A para G:

$$Dist_{A \to G} = 14 + 6 = 20$$

Como o valor é menor do que o contido na Tabela 5, atualiza-se a tabela.

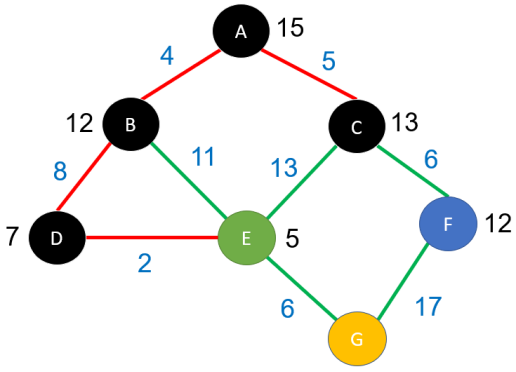


Figura 37 – Etapa 6 do A*.

Tabela 6 – Etapa 6 do A*.

Vértice	Situação	Menor distância de A (g(n))	Heurística Distância para G (h(n))	Distância Total (f(n))	Vértice anterior
A	Visitado	0	15	15	-
B	Visitado	4	12	16	A
C	Visitado	5	13	18	A
D	Visitado	12	7	19	B
E	Atual	14	5	19	D
F	-	11	12	23	C
G	-	20	0	20	E

O próximo nó a ser escolhido é o nó G, porém, esse já é o nó objetivo. Então, o menor caminho de A a G foi encontrado. Para descobrir qual o caminho, inicia-se a busca pelo vértice G na Tabela 7 e procura-se pelo vértice antecessor a ele, que no caso é o E. Essa etapa é repetida até chegar ao nó inicial A, obtendo como menor caminho [A,B,D,E,G].

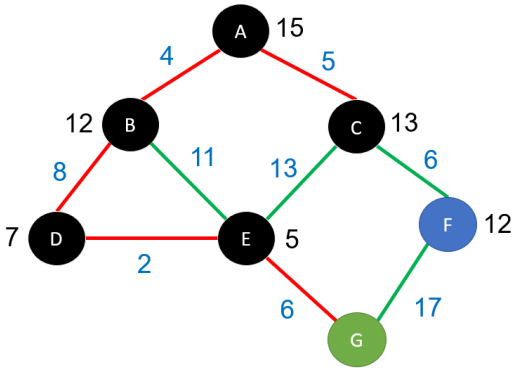


Figura 38 – Etapa 7 do A*.

Tabela 7 – Etapa 7 do A*.

Vértice	Situação	Menor distância de A (g(n))	Heurística Distância para G (h(n))	Distância Total (f(n))	Vértice anterior
A	Visitado	0	15	15	-
B	Visitado	4	12	16	A
C	Visitado	5	13	18	A
D	Visitado	12	7	19	B
E	Visitado	14	5	19	D
F	-	11	12	23	C
G	Atual	20	0	20	E

Com relação ao desempenho do algoritmo, tem-se que a busca em grafos do A* é ótima, caso a heurística $h(n)$ seja consistente (HART; NILSSON; RAPHAEL, 1968). Além disso, contanto que exista uma quantidade finita de nós, a busca é completa. Outro ponto importante é que o A* é otimamente eficiente, pois, não existe garantia que qualquer outro algoritmo expanda menos nós do que ele. Isso se deve ao fato de que qualquer algoritmo que não expanda todos os nós com $f(n) < C^*$, sendo C^* o custo do caminho da solução ótima, tem o risco de perder a solução ótima (NORVIG; RUSSELL, 2014).

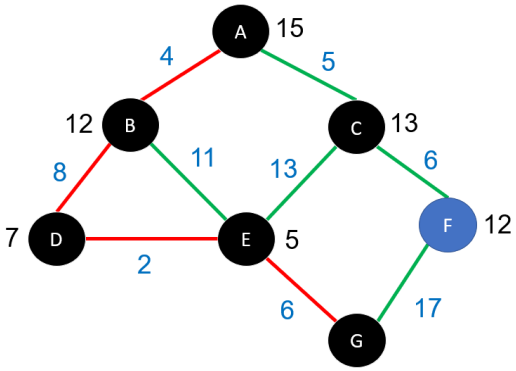


Figura 39 – Etapa 8 do A*.

Além disso, a complexidade dessa busca depende do tipo de heurística utilizada. No pior dos casos essa complexidade é exponencial (NORVIG; RUSSELL, 2014). Em outras implementações, a complexidade pode ser $O(k \log_k v)$, em que v representa o número de nós e k representa as arestas (DUCHOŇ et al., 2013).

2.6 Tecnologias utilizadas

No desenvolvimento desse trabalho utilizou-se como base a linguagem de programação Python, sendo utilizado na implementação dos algoritmos de busca e na visão computacional. O Python é um *software* de código livre criado por Guido Van Rossum em 1990 e se trata de uma linguagem interpretada, de alto nível e com suporte a orientação a objetos. Devido à clareza de sua sintaxe, aliada com os diversos módulos e *frameworks* disponíveis, possibilita um desenvolvimento mais rápido e efetivo (BORGES, 2014).

Outro componente importante no desenvolvimento da visão computacional foi o OpenCV. O OpenCV é uma biblioteca *open source* para visão computacional e aprendizado de máquina, que fornece mais de 2500 algoritmos otimizados que podem ser usados para detecção e reconhecimento de faces, identificação e rastreamento de objetos, rastrear movimentos de câmeras, entre outras aplicações (OPENCV, 2019a). Além disso, possui interfaces para linguagens como C++, Python, Java e Matlab.

3 METODOLOGIA

Neste capítulo são abordadas as principais ferramentas utilizadas na produção do trabalho. São discutidos os aspectos construtivos do robô móvel (Seção 3.1), onde são apresentadas em subseções as especificações do robô utilizado para o desenvolvimento desse trabalho. O desenvolvimento da visão computacional está presente na Seção 3.2, onde é especificado como se deu o tratamento das imagens obtidas através da câmera. Por fim, na Seção 3.3, o planejador de caminhos, que é composto pelos algoritmos de busca implementados para o obtenção do caminho.

3.1 Robô Móvel

Um robô móvel autônomo tem a capacidade de analisar informações do ambiente e com base nelas se locomover no espaço, tomando decisões e executando tarefas, sem a intervenção humana (GONÇALVES, 2007). O robô utilizado neste trabalho é propriedade da Equipe Rodetas, competidora na categoria de futebol de robôs IEEE Very Small Size Soccer (VSSS). Essa categoria propõe uma competição de futebol de robôs autônomos, estando presente em diversas competições, como a Competição Latino-Americana de Robótica (LARC) e a Robocore Winter Challenge.

Cada equipe competidora é responsável por desenvolver a mecânica, a eletrônica e a programação de cada um dos três robôs móveis. Durante uma partida desta categoria, os robôs se comportam autonomamente, isto é, devem buscar os gols sem o controle humano, apenas utilizando as estratégias previamente programadas e armazenadas em memória. De acordo com as regras (IEEE, 2008), uma câmera é posicionada acima do campo, e se comporta como o sensor de posição dos robôs, repassando a localização dos elementos do jogo para um computador, que envia o comando da estratégia ao robô através de uma conexão sem fio, para que o mesmo execute o movimento por meio do ajuste da velocidade de rotação dos motores de corrente contínua conectados às rodas.

Nos subtópicos subsequentes são apresentadas de forma mais detalhada as características físicas do robô.

3.1.1 Estrutura Mecânica

A estrutura mecânica do robô foi desenvolvida de acordo com as limitações estabelecidas na regra (IEEE, 2008). Dessa forma, o robô tem o formato delimitado por um cubo com dimensões máximas de 7,5x7,5x7,5cm. A estrutura foi desenvolvida utilizando o *software* SolidWorks ® e confeccionada por um sistema de impressão 3D em ABS (Acrilonitrila Butadieno Estireno). O resultado pode ser observado na Figura 40.

Figura 40 – Modelo projetado e impressão 3D do robô.

Fonte – (ANDRADE et al., 2018).

3.1.2 Dispositivos Eletrônicos

A seguir são expostos os componentes eletrônicos que compõem o sistema de *hardware* do robô:

1. *Bateria LiPoly*: responsável pela alimentação dos circuitos eletrônicos, essa bateria de polímeros de lítio fornece uma tensão nominal de 7,4V, capacidade de 1000mAh e regime de descarga de 25C).

2. *Conversor CC-CC*: o conversor MP1584 é utilizado para manter uma tensão constante de 6V nos motores. Internamente, esse componente é dotado de um circuito *buck step-down*, para a conversão CC-CC.

3. *Motor CC*: Para a movimentação do robô são utilizados dois motores CC da Pololu com caixa de redução. A tensão nominal do motor é de 6V e rotação nominal de 400rpm.

4. *Driver Ponte H*: o driver TB6612FNG possibilita o acionamento dos dois motores de corrente contínua simultaneamente, bem como o controle do sentido de giro do motor.

5. *Processador*: A plataforma de desenvolvimento Arduino Nano, baseada no microcontrolador Atmega328p, foi utilizada para realizar o processamento do sistema embarcado. Nesta plataforma é executado o algoritmo que possibilita o controle de velocidade das rodas do robô.

6. *Comunicação*: A comunicação entre o computador e o sistema embarcado é realizada com um módulo Xbee que trabalha na faixa de frequência de 2,4 GHz.

3.1.3 Placa de circuito impressa

A placa de circuito impresso (PCI) é responsável por reunir todos os componentes eletrônicos mencionados anteriormente de forma compacta, garantindo a integração desses

dispositivos. Ela foi desenvolvida com a ferramenta EAGLE ® e confeccionada pela combinação dos métodos de transferência por luz UV, corrosão por percloreto de ferro e a utilização da fresa CNC. O *layout* produzido no programa, bem como o resultado após a produção da PCI podem ser observados na Figura 41.

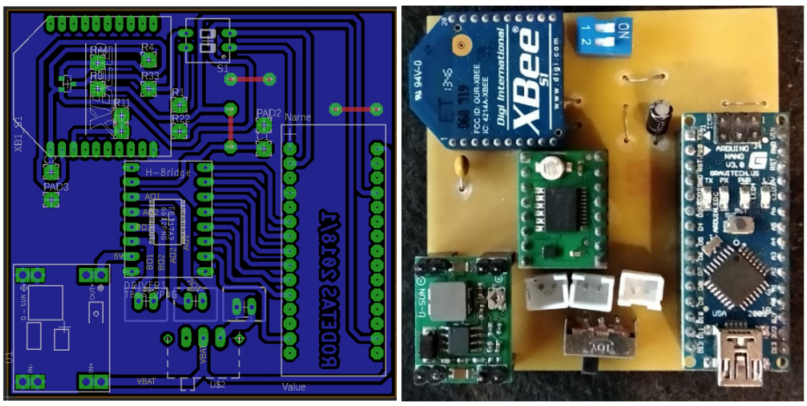


Figura 41 – *Layout* da PCI e placa pronta.

Fonte – (ANDRADE et al., 2018).

3.2 Implementação do sistema de visão computacional

O primeiro passo na criação do sistema de visão é a aquisição das imagens. Para obtenção dessas imagens pelo computador é utilizada uma câmera modelo ELP-USBFHD01M-MFV(5-50mm), conforme Figura 42, capaz de fornecer imagens de 640x480 *pixels* a uma velocidade de 120 quadros por segundo (fps).

Figura 42 – Câmera utilizada.

A câmera é posicionada acima do centro do campo em um suporte, de modo a fornecer uma visão superior do mesmo. A Figura 43 ilustra o posicionamento da câmera nesse suporte, e a sua posição em relação ao campo.

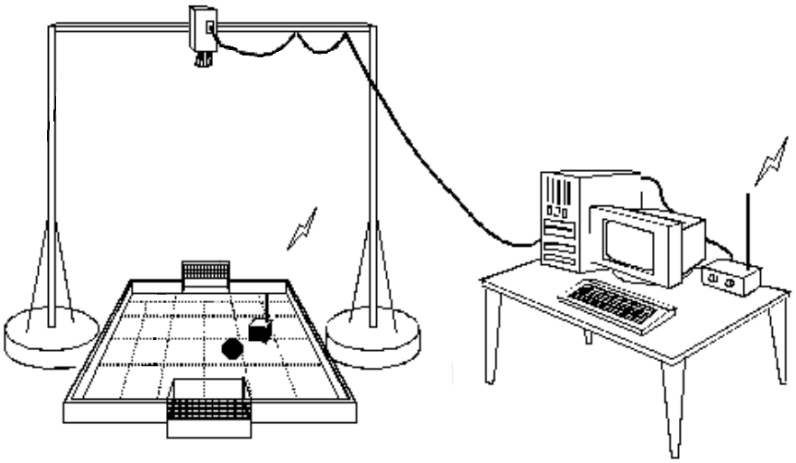


Figura 43 – Esquema de posicionamento da câmera sobre o campo.

Fonte – Figura adaptada (IEEE, 2008).

O campo é uma estrutura de madeira onde as partidas de futebol de robô acontecem. Suas dimensões são de 150cm de largura e 130cm de comprimento, como pode ser visualizado na Figura 44.

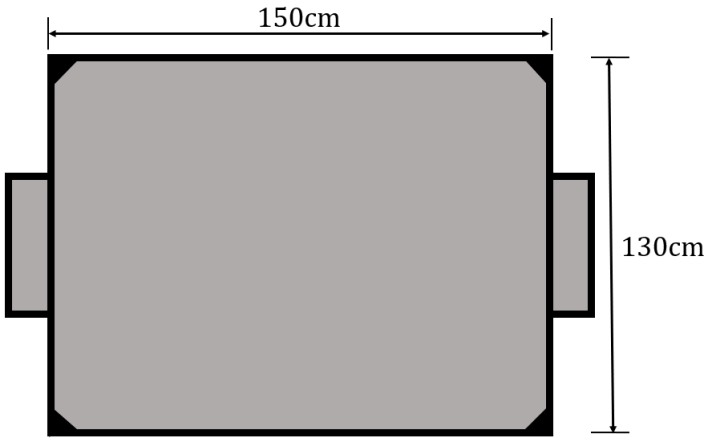


Figura 44 – Dimensões do campo utilizado.

3.2.1 Calibração da câmera

Em sistemas de visão computacional, ao se trabalhar com câmeras, muitas vezes é possível se notar distorções nas imagens capturadas. Essas distorções prejudicam o desenvolvimento do projeto, pois podem alterar a posição e tamanho relativo de um objeto, dependendo do local que este objeto se encontra na imagem obtida pela câmera.

Tais distorções são causadas por parâmetros intrínsecos da câmera, como a posição e inclinação do sensor de captura de imagem e o foco das lentes. Os principais tipos de distorções são: (i) distorção radial que produz um efeito de curva nas linhas retas de uma imagem; e (ii) distorção tangencial, que produz um efeito de proximidade em certas regiões da imagem (OPENCV, 2017).

Esses efeitos podem ser observados na Figura 45. Nesta figura, as bordas do campo apresentam um formato mais "curvado", que não condiz com a realidade, como pode ser analisado na comparação com as retas vermelhas desenhadas na figura. Além disso, o centro da imagem parece estar mais próximo do que as bordas.

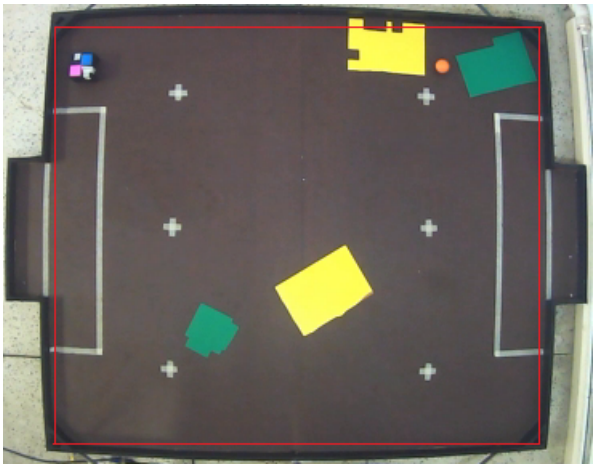


Figura 45 – Imagem com distorções.

Tais distorções podem ser contornadas utilizando funções fornecidas pelo OpenCV. Para o processo de calibração devem ser tiradas fotos de um tabuleiro xadrez em diferentes ângulos e posições, como pode ser visualizado na Figura 46. Segundo OpenCV (2017) devem ser capturadas no mínimo dez imagens. O próximo passo é fornecer essas imagens a um algoritmo que irá calcular os parâmetros K, que é uma matriz 3x3 da câmera, e D, que representa os coeficientes de distorção da imagem. Por fim, esses parâmetros são utilizados pelo OpenCV para a calibração da imagem (JIANG, 2017), (OPENCV, 2017).

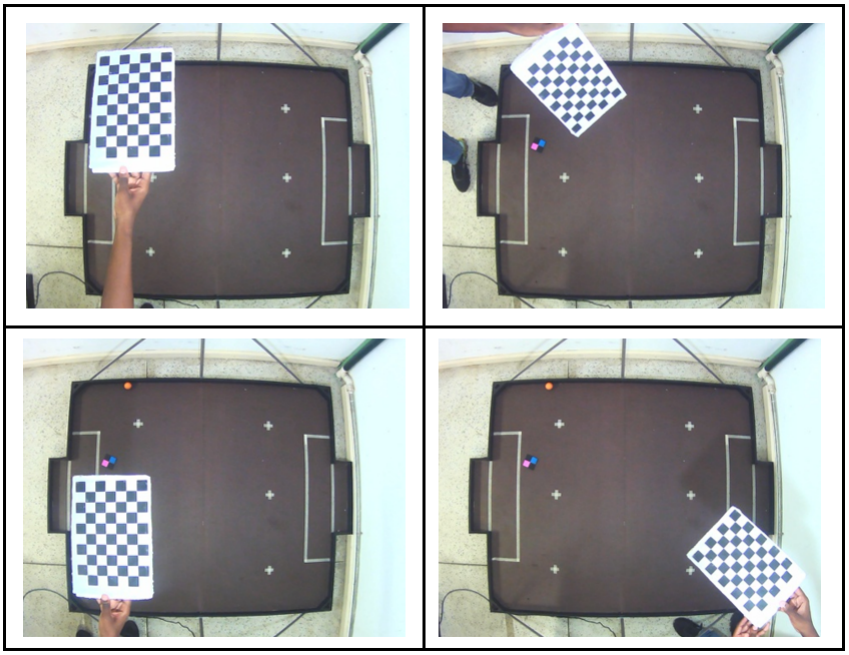


Figura 46 – Imagens do padrão xadrez tiradas com a câmera.

O resultado do processo de calibração da imagem pode ser observado na Figura 47. Ao contrário da Figura 45, as bordas do campo se encontram alinhadas com as retas vermelhas desenhadas na imagem, com a remoção quase completa do efeito de curva na mesma. Além disso, o efeito de proximidade em certas áreas da figura foi suavizado.

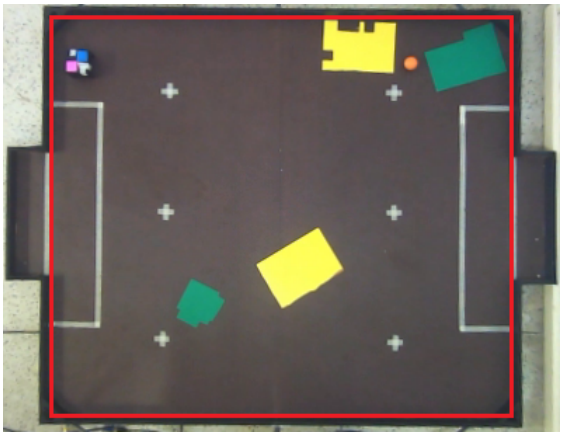


Figura 47 – Imagem com distorções corrigidas.

3.2.2 Remoção de ruídos

Após o processo de aquisição das imagens, elas passam por um pré-processamento com o intuito de remover ruídos e corrigir as distorções geométricas.

Os filtros são ferramentas capazes de reduzir o distúrbio nas imagens, possibilitando uma melhor interpretação das informações contidas nela. Para facilitar a utilização desses filtros, o OpenCV possui já implementados diversos filtros, dentre os quais foram utilizados no trabalho: (i) filtros de *blur*; (ii) filtro *inrange*; (iii) filtros de dilatação e erosão.

Os filtros de *blur* realizam operações de suavização na imagem, o que é realizado geralmente com o intuito de reduzir o ruído, detalhes irrelevantes ou a resolução da mesma. Com esse tipo de filtro, a imagem passa por uma transformação em que fica embaçada, perdendo alguns detalhes. Em compensação, isso facilita na detecção de formas ou cores. A Figura 48 exibe a aplicação de um filtro de *blur*.

O filtro *inRange* é utilizado para a detecção de algum parâmetro, por exemplo, uma cor, dentro da imagem. Para isso, ele percorre a imagem a procura de pixels daquela cor, utilizando os intervalos de valores mínimo e máximos determinados, desprezando os outros pixels que não se encontram no intervalo. O resultado é uma imagem binarizada em que os pixels presentes no intervalo tornam-se brancos e os pixels desconsiderados tornam-se pretos. Como exemplo, observe a Figura 49. Nesta figura, utiliza-se o filtro *inRange* para filtrar a cor azul. O resultado, como pode ser observado, é uma imagem em preto e branco, em que os pixels brancos são referentes ao azul. Pode-se notar ainda, a existência de alguns ruídos após o processo de filtragem.

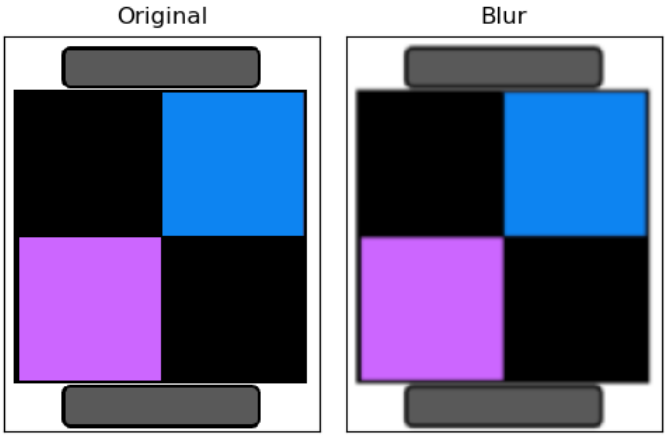


Figura 48 – Aplicação do filtro de *blur*, especificamente o filtro *GaussianBlur*.

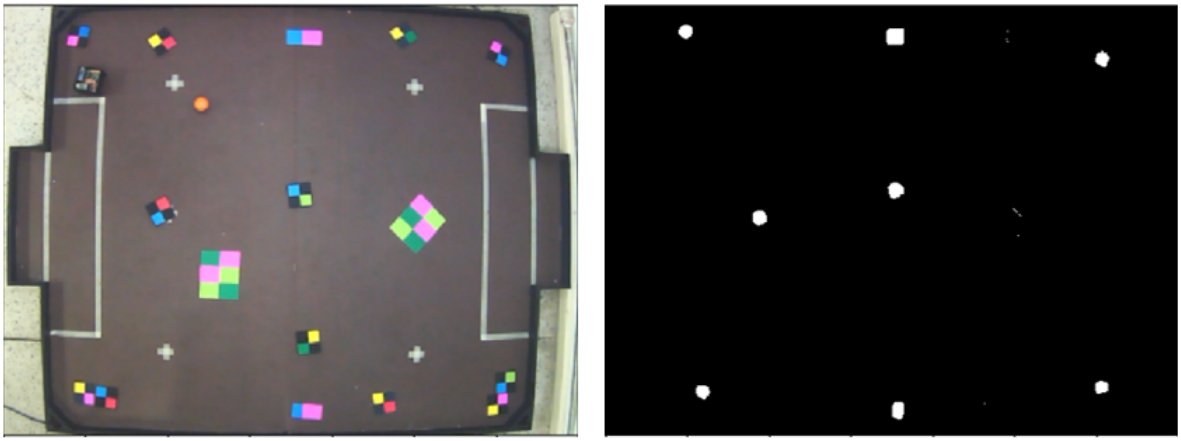


Figura 49 – Aplicação do filtro de *inRange* para detecção da cor azul.

Os filtros de erosão e dilatação fazem parte do conceito de transformações morfológicas. As transformações morfológicas são simples operações que alteram a forma de uma imagem. Elas podem ser utilizadas para a remoção de ruídos, união de elementos diferentes em uma imagem, isolamento de elementos individuais, entre outras aplicações (BRADSKI; KAEHLER, 2008). O filtro de erosão, como o próprio nome indica, tem a função de reduzir uma região, corroendo limites do objeto em primeiro plano. No que lhe concerne, o filtro de dilatação realiza o oposto, expandindo uma região. Normalmente, para a remoção de ruídos, o filtro de erosão é seguido do de dilatação. A Figura 50 exibe o resultado das transformações de dilatação e erosão em uma imagem.

(a) Imagem original. (b) Filtro de erosão. (c) Filtro de dilatação.

Figura 50 – Operações de erosão e dilatação em uma imagem.

Fonte – OpenCV (2019b).

3.2.3 Sistema de cores

No trabalho proposto, tanto o robô, o alvo e os obstáculos apresentam cores de identificação e é por meio delas que é possível fazer o reconhecimento de cada objeto. A Figura 51 exibe o robô com o seu padrão de identificação, possibilitando por meio de simples cálculos encontrar a posição e orientação do mesmo no ambiente.

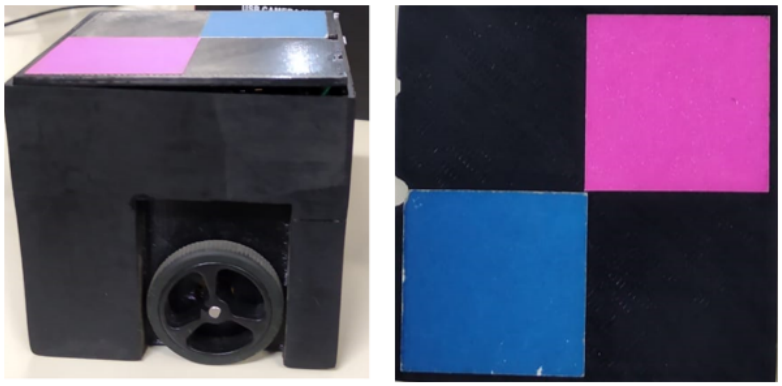


Figura 51 – Padrão de identificação do robô.

Desta forma, é necessário utilizar algum tipo de sistema de cor para a representação do problema. Existem diversos sistemas de cores disponíveis, dentre os quais pode-se citar o RGB (*Red, Green and Blue*), HSV (*Hue, Saturation and Value*), HSL (*Hue, Saturation and Lightness*), entre outros.

O sistema RBG, nome que deriva das iniciais das cores *Red* (vermelho), *Blue* (azul) e *Green* (verde), é um sistema de cores aditivas em que a combinação dessas cores forma o espectro cromático. Esse sistema é utilizado em objetos emissores de luz, como televisores e câmeras digitais, e é o espaço de cores similar ao da visão humana (SIMÕES; COSTA, 2001). Uma desvantagem desse padrão de cores é a dificuldade para se separar cores muito similares e a influencia da variação da iluminação. O espaço de cores RGB pode ser visualizado como um cubo de faces *Red*, *Green* e *Blue*, como pode ser observado na Figura 52.

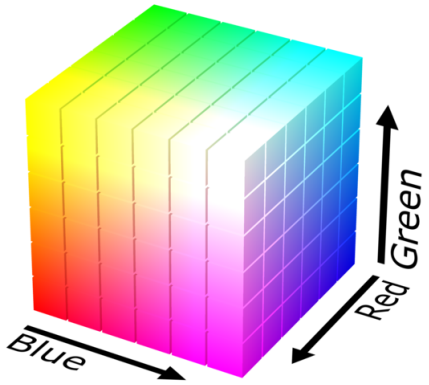


Figura 52 – Representação do cubo RGB.

Fonte – Commons (2018b).

O sistema de cores HSV é formado pela combinação dos componentes *Hue* (tonalidade) que é o valor do comprimento de onda prevalecente, *Saturation* (saturação) que representa o nível de pureza da cor e *Value* (valor) que é associada a iluminação (SIMÕES; COSTA, 2001). A Figura 53 apresenta uma representação desse sistema, na qual a tonalidade é indicada como um ângulo, a saturação é expressa como o raio desse ângulo, e a iluminação é o eixo perpendicular aos outros dois componentes. A vantagem do espectro em HSV é que ele apresenta uma maior robustez em relação à variação da luminosidade.

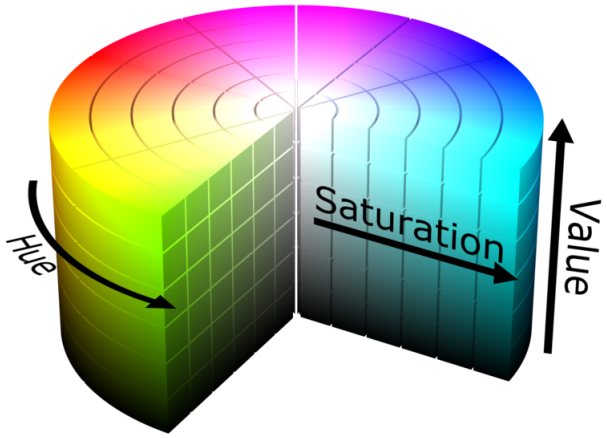


Figura 53 – Representação sistema de cores HSV.

Fonte – Commons (2018a).

O sistema HSV foi escolhido para a identificação das cores no presente trabalho devido aos bons resultados obtidos pela equipe Rodetas utilizando este sistema.

3.2.4 Algoritmo de visão

O algoritmo de visão tem como função principal analisar o cenário e identificar o robô móvel, os obstáculos e o alvo que é representado por uma bola laranja. De forma simplificada, ele pode ser dividido em duas etapas: (i) Obtenção da imagem; e (ii) Reconhecimento do ambiente.

A etapa de obtenção da imagem é responsável por capturar uma imagem do campo e realizar a calibração da mesma de modo a diminuir os efeitos dos distúrbios mencionados na Seção 3.2.1. Por fim, é recortada a região de interesse o que acelera as etapas de processamento posteriores. O pseudocódigo é apresentado em 5.

Algoritmo 5: Obtenção da imagem

Entrada: Imagens enviadas pela câmera
Saída: Uma imagem modificada

```
1  início
2      Recebe uma imagem da câmera
3      Aplica os coeficientes de correção da lente
4      Recorta a região de interesse
5      Salva a imagem em formato png
6  fim
```

Já a etapa de reconhecimento é encarregada de executar todo o processamento de imagem, aplicando os filtros e transformações morfológicas mencionadas na Seção 3.2.2 com o intuito de reconhecer os obstáculos, robô, bola e bordas do campo, que em conjunto representam o chamado mapa do ambiente. Após o reconhecimento de todos os objetos, o ambiente é decomposto em células de tamanho previamente determinado e é obtido o espaço de configuração do robô livre do robô. O pseudocódigo dessa etapa é apresentado em 6.

Algoritmo 6: Reconhecimento do ambiente

Entrada: Imagem obtida pelo Algoritmo 5
Saída: Informações referentes a posição dos obstáculos, robô e alvo presentes no campo

```
1  início
2      Define a dimensão das células utilizadas
3      Recebe a imagem da etapa de obtenção
4      Calcula a dimensão da imagem em pixels
5      Divide a dimensão da imagem pela dimensão das células, obtendo o número de células utilizadas para a
       representação do ambiente
6      Converte a imagem do sistema RGB para o sistema HSV
7      Aplica o filtro para suavização da imagem
8      Aplica o filtro inRange para detectar as cores utilizadas pelo robô, obstáculos, alvo e bordas do campo
9      Aplica os filtros de erosão e dilatação para a redução dos ruídos
10     Encontra os contornos de cada objeto
11     Utiliza os contornos da bola e calcula o centro desse objeto
12     Utiliza o contornos dos quadrados de identificação do robô para calcular o centro de cada quadrado
13     Calcula o centro do robô e a sua orientação
14     Cria uma máscara que contém os obstáculos e as bordas do campo
15     Decompõe a máscara em células aproximadas
16     Para cada célula, verifica na máscara se existe obstáculo nesse local
17     se existe obstáculo na célula então
18         Define a célula como ocupada
19     fim
20     Salva em um arquivo o número de células utilizadas, os obstáculos, a posição do robô e a posição da bola
21 fim
```

3.2.5 Mapa do ambiente

A criação do mapa do ambiente é uma tarefa presente no Algoritmo 6. Recebendo como entrada a imagem, denominada máscara, que combina todos os obstáculos presentes no campo em conjunto com as bordas do mesmo, o algoritmo vasculha toda a máscara, analisando se existe a presença de obstáculos no interior de cada célula. Caso exista determinada porcentagem de obstáculo na célula, a célula é classificada como ocupada. Este processo é o responsável por classificar as regiões navegáveis e não navegáveis do espaço de configuração.

Como visto na Seção 2.2, o tamanho da célula influencia na qualidade da aproximação do mapa. Dessa forma, considerando as dimensões físicas do robô, assim como o comprimento e largura em *pixels* da imagem, o tamanho da célula foi definido de forma que o algoritmo de busca fornecesse um caminho preciso, o que acontece com a diminuição do tamanho de cada célula, aliado a um tempo de processamento adequado, pois, quanto mais células (nós no grafo), maior o custo computacional para se encontrar o caminho. Dessa forma, após diversos testes, a dimensão da célula ideal encontrada foi de 10x10 *pixels*.

A Figura 54 representa uma imagem do ambiente contendo o robô, os obstáculos em amarelo e o alvo, simbolizado pela círculo laranja. Essa imagem é dividida em células de tamanhos iguais, como mostrado na Figura 55 e as células classificadas como ocupadas recebem destaque com o contorno em preto, como pode ser visto na Figura 56.

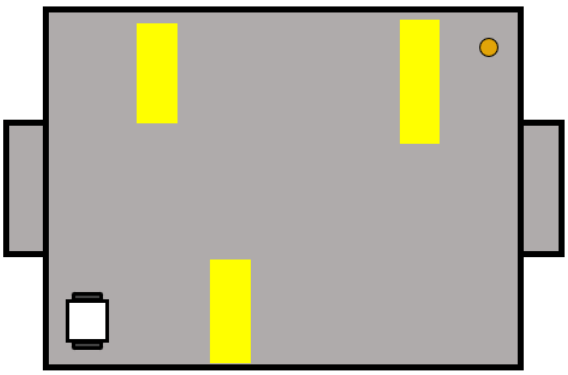


Figura 54 – Imagem do ambiente inicial.

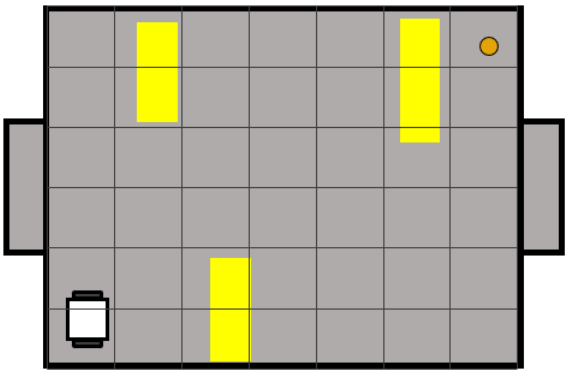


Figura 55 – Ambiente dividido em células.

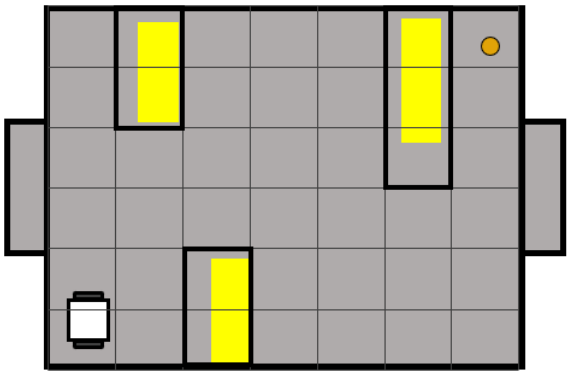


Figura 56 – Destaque das células ocupadas.

Durante o processo de decomposição do ambiente em células aproximadas, a posição dos objetos no campo deixa de ser representada em *pixels* e passa a ser representada em células, como mostrado na Figura 57. Desta forma, a posição do robô será dada pelas coordenadas (0,5), e a bola por (6,0), por exemplo.

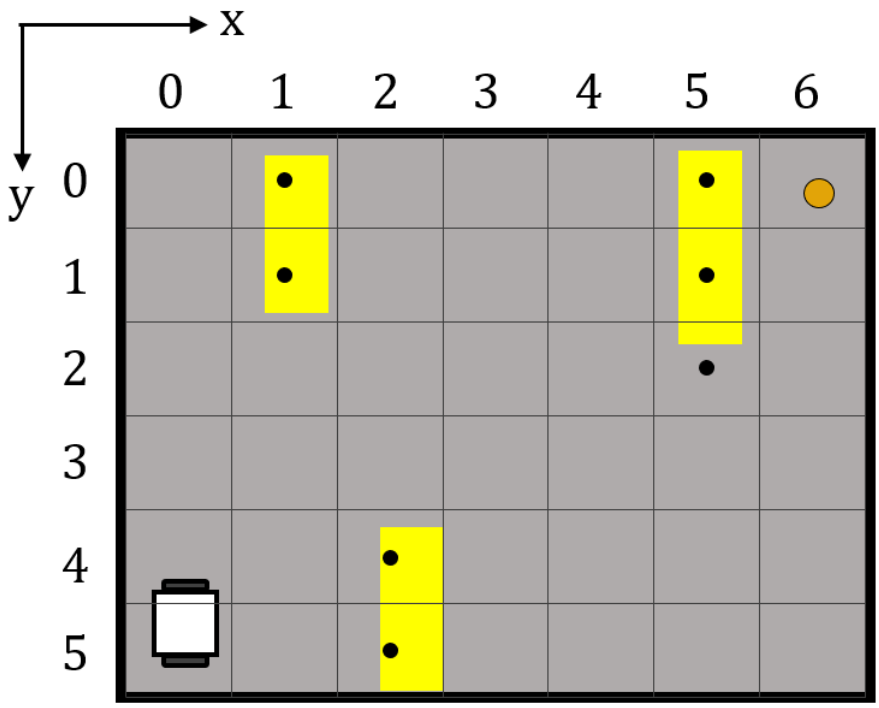


Figura 57 – Sistema de coordenadas modificado.

Além disso, para cada célula classificada como ocupada, é criado um vértice com as coordenadas dessa célula. Esses vértices são representados como pontos pretos no centro de cada célula ocupada na Figura 57. Dessa forma, para a figura em questão os vértices do grafo classificados como obstáculos são: (1,0), (1,1), (2,4), (2,5), (5,0), (5,1) e (5,2).

3.3 Algoritmos de busca

Os algoritmos de busca funcionam como o cérebro do robô. Eles são responsáveis por obter o caminho livre de obstáculos entre o robô e o alvo, com base no grafo de conectividade do ambiente. Os algoritmos implementados foram: (i) busca em largura; (ii) busca em profundidade; (iii) busca gulosa; (iv) algoritmo de Dijkstra; e (v) A*. Cada um deles possui as suas particularidades com relação as estruturas de dados utilizadas e a forma em que realizam a expansão dos vértices. No entanto, ambos compartilham da mesma estrutura na etapa de criação dos grafos, conforme será apresentado na Seção 3.3.1.

Antes de realizar a etapa de busca, é necessário se obter as configurações iniciais do mapa ambiente. Neste estágio, o algoritmo principal lê as informações fornecidas pela visão computacional e utiliza esses dados para definir o problema em questão. O pseudocódigo é exibido em 7.

Algoritmo 7: Algoritmo principal

Entrada: Arquivo contendo as informações do ambiente
Saída: Caminhos obtidos pelos algoritmos de busca

```
1  início
2      Lê as informações salvas pelo algoritmo de visão
3      Com base nessas informações define as configurações iniciais do problema
4      Define o vértice inicial e o vértice objetivo
5      Define as dimensões dos grafos que serão criados
6      Cria a configuração base dos grafos com e sem peso
7      Define os obstáculos dos grafos
8      Chama os algoritmos de busca passando o grafo, o início da busca e o alvo
9      Exibe resultado de cada busca na tela
10     Salva o resultado de cada busca em um arquivo
11 fim
```

3.3.1 Criação dos grafos

O grafo de conectividade que representa as regiões navegáveis do espaço de configuração do robô é criado de acordo com o processo de expansão de cada busca. Sua configuração inicial é composta pelos vértices que são obstáculos e por sua dimensão que é correspondente ao número de células em que o mapa do ambiente foi dividido.

Durante o processo de busca, quando existe a expansão dos vizinhos de cada vértice é chamada uma função denominada confere_vizinhos que é responsável por criar os vizinhos válidos do vértice em questão. O pseudocódigo desta função é apresentado em 8.

Algoritmo 8: Confere vizinhos

Entrada: Vértice atual da busca
Saída: Lista de vizinhos válidos
início
    Define todos os vizinhos possíveis do vértice analisado, baseado no tipo de movimentação entre os vértices. Os movimentos podem variar entre 4 e 8 direções
    Confere se os vizinhos estão dentro das dimensões do grafo
    Confere se os vizinhos não são obstáculos
    Retorna a lista de vizinhos válidos
fim

A adjacência entre os vértices pode ser encontrada de duas formas: (i) movimento em quatro direções, em que para cada vértice são considerados os vértices não ocupados ao norte, leste, sul e oeste; (ii) movimento em oito direções, em que além das quatro direções citadas anteriormente, são considerados os vértices das diagonais. As Figuras 58 e 59 exibem a representação da movimento em quatro e oito direções, respectivamente.

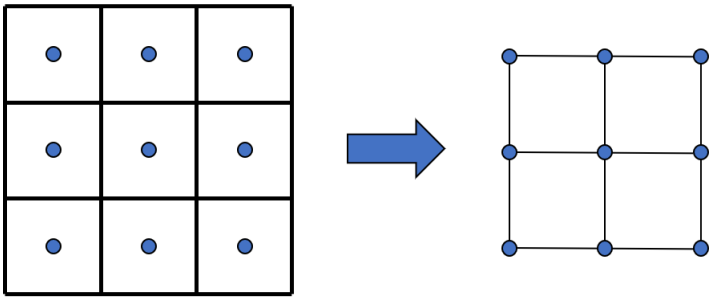


Figura 58 – Representação do grafo utilizando o movimento em 4 direções.

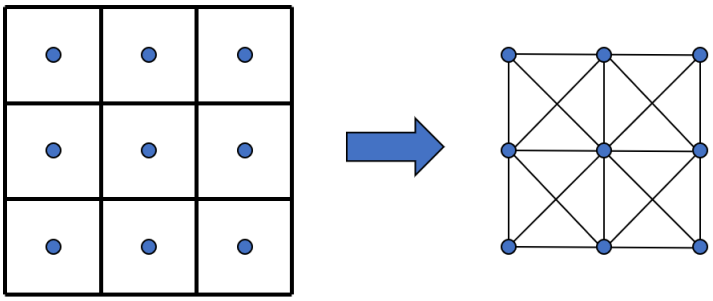
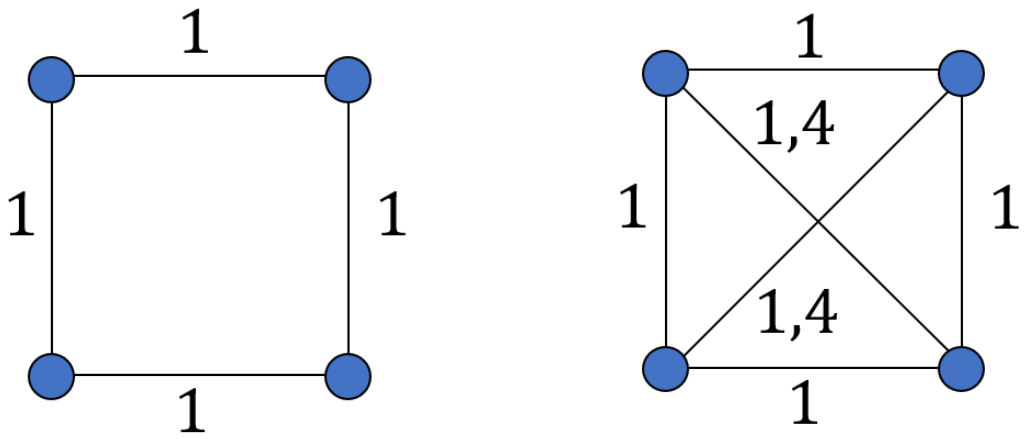


Figura 59 – Representação do grafo utilizando o movimento em 8 direções.

O tipo de movimento utilizado irá variar a forma como os pesos das arestas são atribuídos. Para o movimento em 4 direções, o custo de movimento horizontal ou vertical será definido como 1, conforme a Figura 60a. Já para o movimento em 8 direções, o custo de movimento horizontal e vertical também será 1, no entanto, para se deslocar diagonalmente, o custo será definido como a $\sqrt{2}$. Isso se deve ao fato de que dado um quadrado de dimensões 1x1, a diagonal desse quadrado terá a distância real de $\sqrt{2}$. Para efeitos de simplificação, o valor utilizado será 1.4, como pode ser observado na Figura 60b.



(a) Custo do movimento em 4 direções.

(b) Custo do movimento em 8 direções.

Figura 60 – Representação do custo para os movimentos em 4 e 8 direções.

3.3.2 Estrutura de dados utilizadas

Conforme mencionado nas Seções 2.5 e 2.4, uma característica importante nos algoritmos de busca é a estrutura de dados utilizada para armazenar os vértices resultantes do processo de expansão. Essa estrutura vai variar de acordo com o tipo de busca, sendo a fila utilizada para a busca em largura, a pilha utilizada para a busca em profundidade e a fila de prioridade utilizada na busca gulosa, algoritmo de Dijkstra e A*.

A fila e a pilha foram implementadas utilizando uma estrutura do tipo *deque* presente no módulo *collections* do Python. Essa estrutura permite a inserção e remoção de elementos em qualquer uma de suas extremidades com desempenho aproximado de *O(1)*. Já a fila de prioridade foi implementada utilizando a estrutura *heap* do módulo *heapq* que se trata de um algoritmo de fila de prioridades, que sempre retorna o elemento com o menor peso.

4 RESULTADOS

Nesta capítulo serão apresentados os resultados e análises de cada etapa do trabalho, desde a obtenção da imagem até a criação do caminho que leva o robô da posição final até a posição objetivo.

4.1 Resultados da visão computacional

O processo de execução da visão computacional se inicia no Algoritmo 5. Esse algoritmo vai utilizar os parâmetros K e D obtidos durante o processo de calibração da câmera e irá corrigir as distorções da imagem. Após a calibração, a imagem é recortada de acordo com região de interesse. O resultado da execução é mostrado na Figura 61.

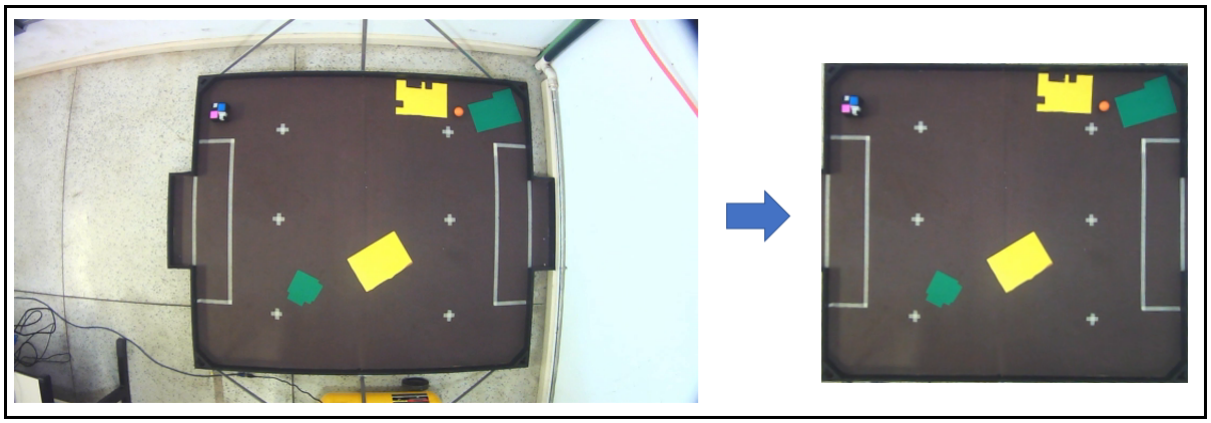


Figura 61 – Imagem obtida pela câmera e imagem tratada.

Após a obtenção da imagem calibrada e recortada, é utilizado o Algoritmo 6, que realizará o reconhecimento dos objetos no campo e criará o mapa do ambiente. Para demonstrar o funcionamento dos algoritmos implementados, será utilizado como base o cenário da Figura 62.

O processo de obtenção de informações a partir da imagem pode ser resumido nas seguintes etapas:

- Conversão da imagem do sistema RGB para o sistema HSV, como mostrado na Figura 63:
- Aplicação do filtro gaussiano para suavização da imagem, conforme a Figura 64 :
- Aplicação dos filtros *inRange*, erosão e dilatação, como mostrado na Figura 65.
- Obtenção do contorno dos obstáculos, robô e alvo, conforme a Figura 66.
- Decomposição do ambiente em células, como mostrado na Figura 67.
- Classificação das células com obstáculos como ocupadas, como mostrado na Figura 68.

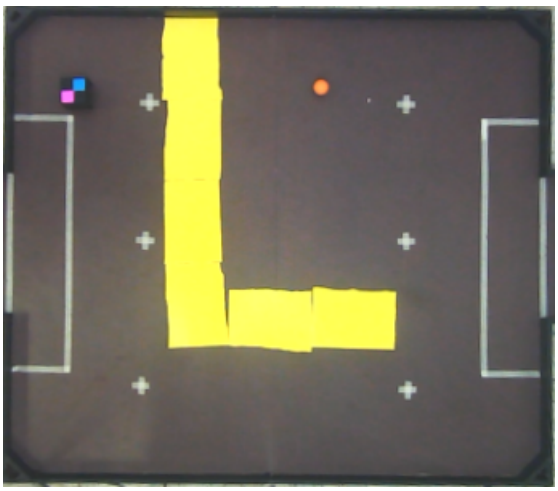


Figura 62 – Cenário contendo o robô, os obstáculos e o alvo.

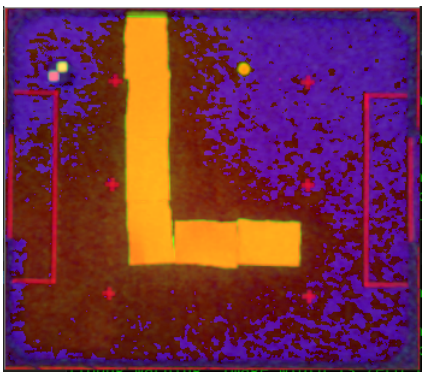


Figura 63 – Imagem convertida de RGB para HSV.

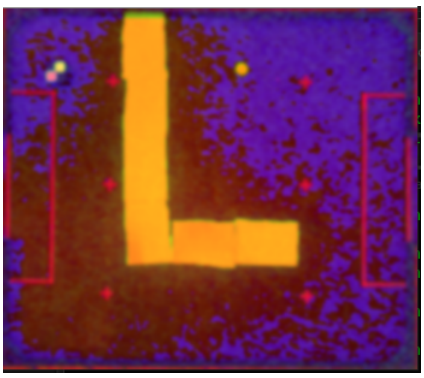


Figura 64 – Imagem após a aplicação do filtro gaussiano.

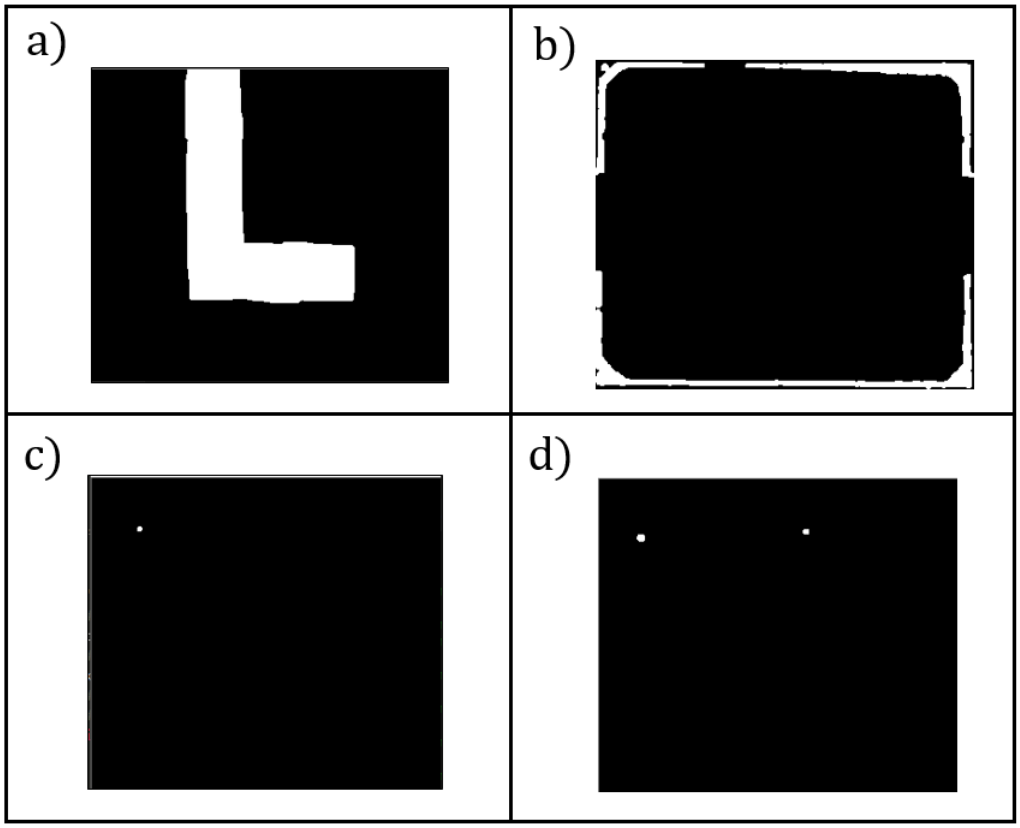


Figura 65 – Imagem após a aplicação dos filtros *inRange*, erosão e dilatação. Na figura são representados os filtros para: (a) obstáculos, (b) bordas do campo, (c) cor azul e (d) cor rosa e bola.

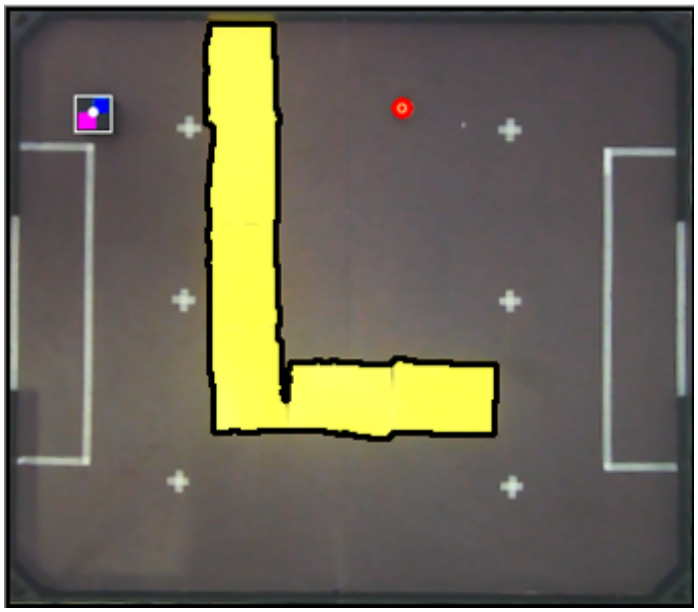


Figura 66 – Contorno dos obstáculos, bola e robô.

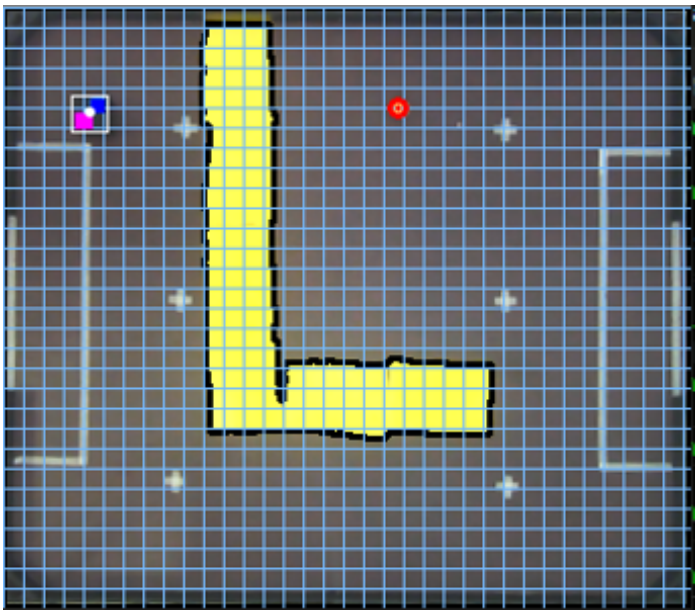


Figura 67 – Divisão do ambiente em células iguais.

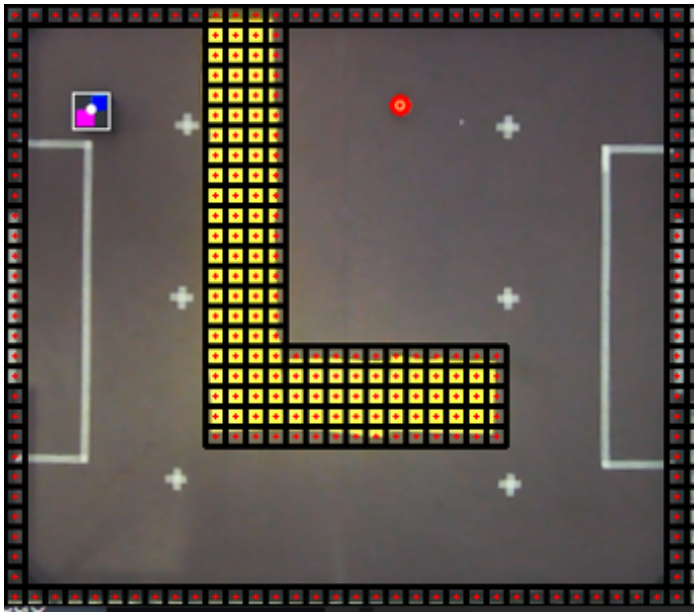


Figura 68 – Destaque das células que foram consideradas ocupadas pela algoritmo.

De modo a garantir a integridade física do robô e do cenário, foi implementado um fator de segurança que expande as regiões não navegáveis virtualmente, de forma que o algoritmo reconheça regiões próximas aos obstáculos reais também como obstáculos. Esse fator é alterado mediante uma modificação nos parâmetros do filtro de erosão. O resultado dessa alteração pode ser observado na Figura 69.

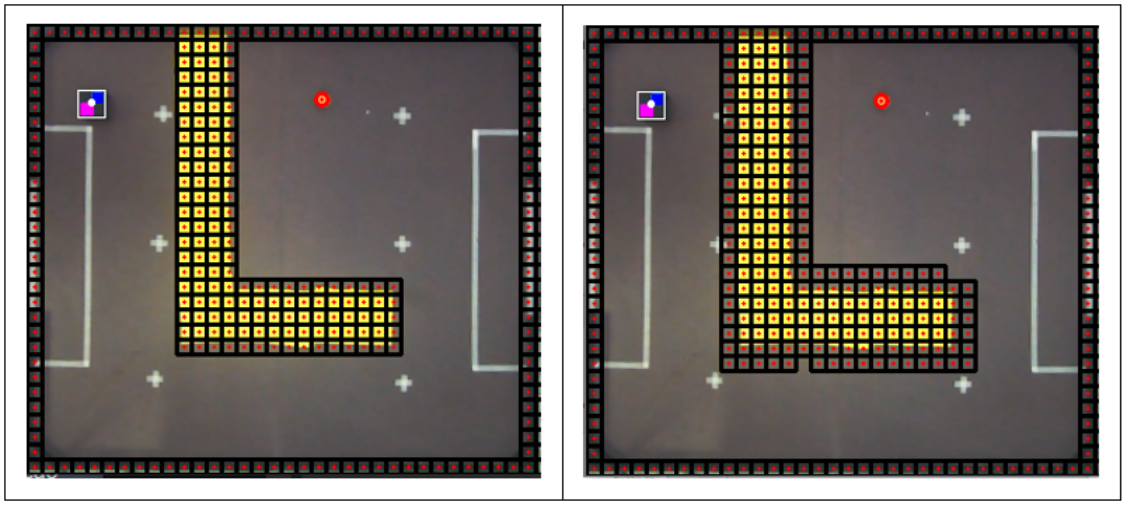


Figura 69 – Imagem de antes e depois da alteração do fator de segurança.

4.2 Resultados dos algoritmos de busca

Os algoritmos de busca irão calcular o caminho que leva o robô da posição inicial até a bola. Para que isso aconteça, primeiro é necessário conhecer a localização dos obstáculos, do robô e do alvo. Dessa forma, a primeira etapa é utilizar o Algoritmo 7 que realizará as configurações iniciais necessárias para o funcionamento dos algoritmos de busca. Esse algoritmo irá repassar às estratégias de busca essas informações e assim o trajeto será calculado.

O resultado do planejamento de caminho obtido pelos algoritmos A* e busca em largura, pode ser visualizado na Figura 70. Já o resultado obtido pelos algoritmos de Dijkstra e busca gulosa, pode ser visualizado na Figura 71. Por fim, o resultado obtido pela busca em profundidade é mostrado na Figura 72.

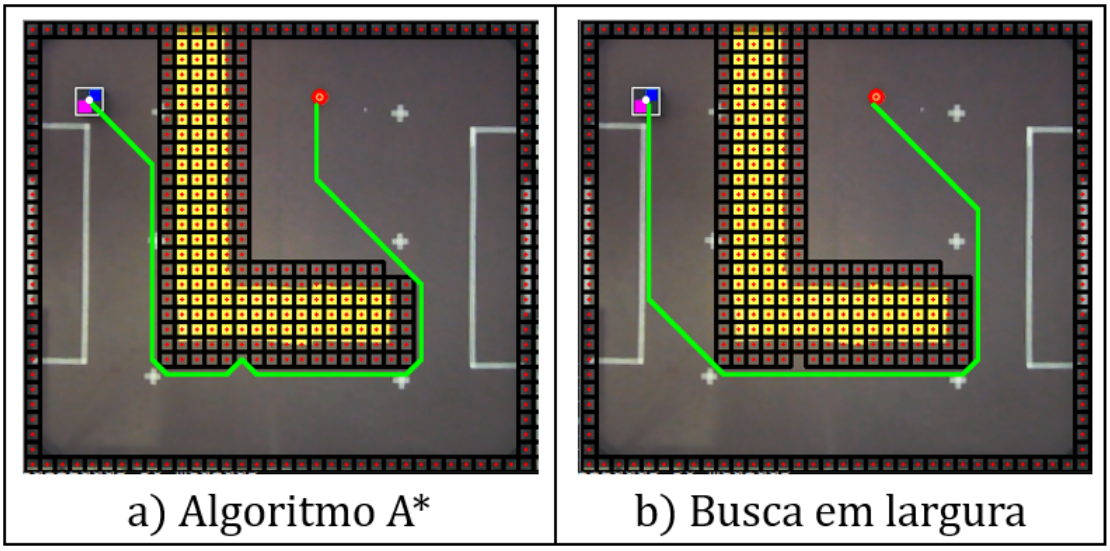


Figura 70 – Caminho obtido pelo Algoritmo A* (a) e busca em largura (b).

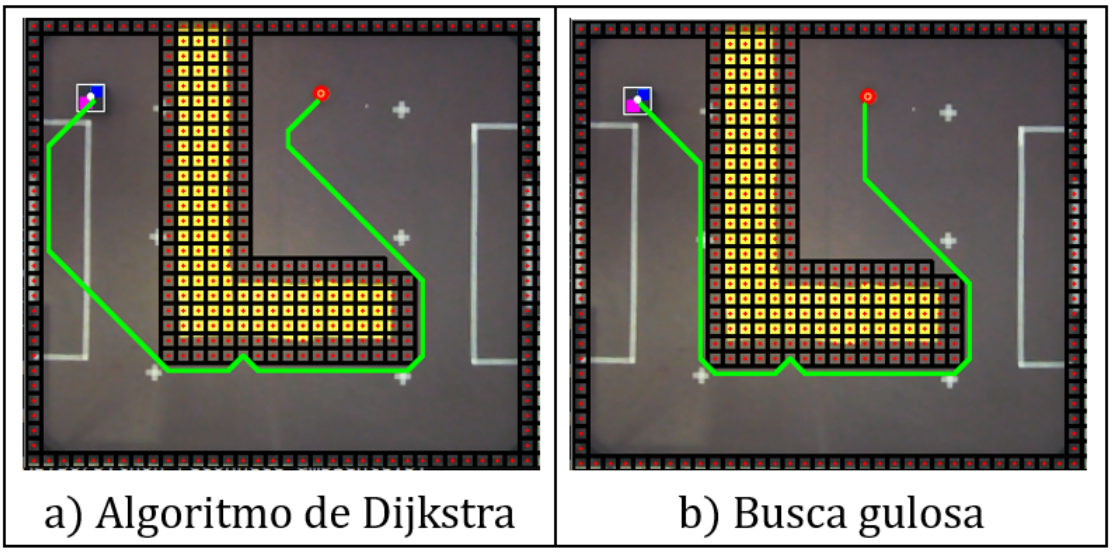


Figura 71 – Caminho obtido pelo algoritmo de Dijkstra (a) e busca gulosa (b).

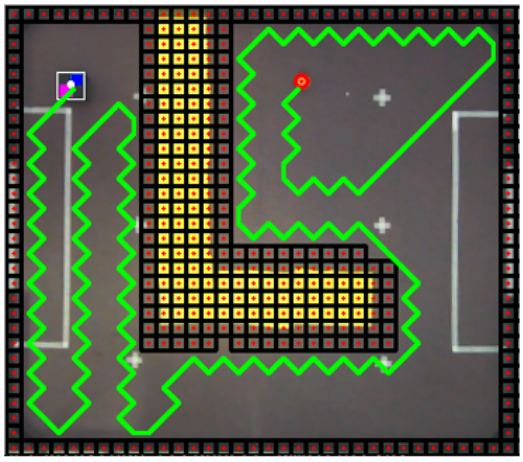


Figura 72 – Caminho obtido pelo busca em profundidade.

Para analisar o desempenho dos algoritmos no cenário da Figura 62, são dispostos na Tabela 8 os tempos de execução de cada algoritmo e o tamanho do caminho obtido. Para a criação das tabelas, os algoritmos de busca foram executados 50 vezes para cada ambiente e o tempo de execução apresentado foi a média desses valores encontrados.

Tabela 8 – Comparação dos resultados obtidos para o cenário da Figura 62.

Estratégia de busca	Tempo de execução (segundos)	Tamanho do caminho (passos)
Busca em largura	0.05194	54
Busca em profundidade	0.01566	168
Busca gulosa	0.03359	54
Algoritmo de Dijkstra	0.08152	54
Algoritmo A*	0.04576	54

Por meio de uma análise visual dos resultados obtidos em cada algoritmo, percebe-se que o algoritmo que apresentou o pior desempenho foi a busca em profundidade, como mostrado na Figura 72. Comparando esses resultados com a Tabela 8, é possível notar que mesmo obtendo o maior caminho dentre todos os métodos, o DFS ainda foi o algoritmo que encontrou o objetivo mais rápido. Os outros métodos obtiveram o caminho com o mesmo número de passos, tendo o A* encontrado o mesmo caminho que a busca gulosa, mas com o tempo de execução superior.

A Figura 73 representa um novo cenário em que os obstáculos presentes no campo foram alterados e o alvo foi trocado de lugar.

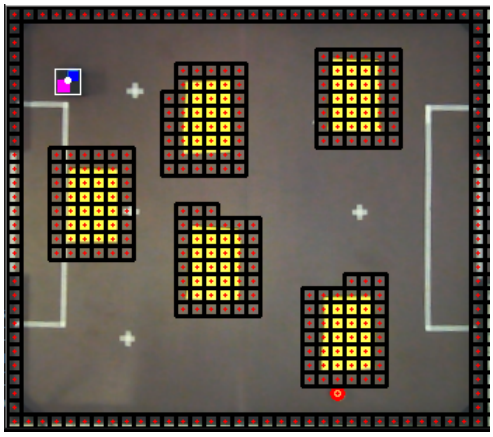


Figura 73 – Cenário com novos obstáculos e novo alvo.

O resultado do planejamento de caminho obtido pelos algoritmos A* e busca em largura, para esse novo cenário, pode ser visualizado na Figura 74. Já o resultado obtido pelos algoritmos de Dijkstra e busca gulosa, pode ser visualizado na Figura 75. Por fim, o resultado obtido pela busca em profundidade é mostrado na Figura 76.

Para analisar o desempenho dos algoritmos no cenário da Figura 73, são dispostos na Tabela 9 os tempos de execução de cada algoritmo e o tamanho do caminho obtido.

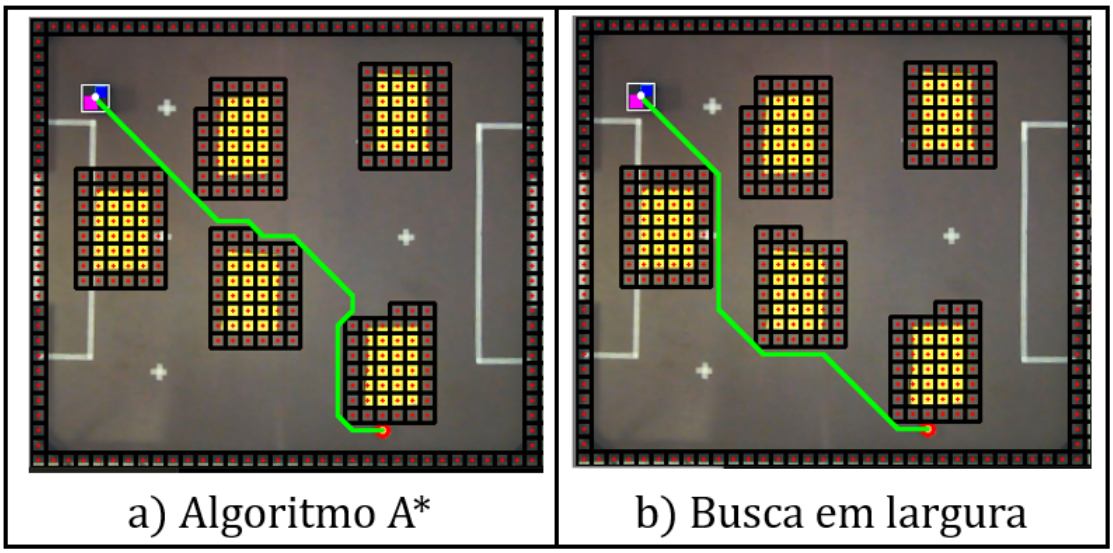


Figura 74 – Caminho obtido pelo Algoritmo A* (a) e busca em largura (b).

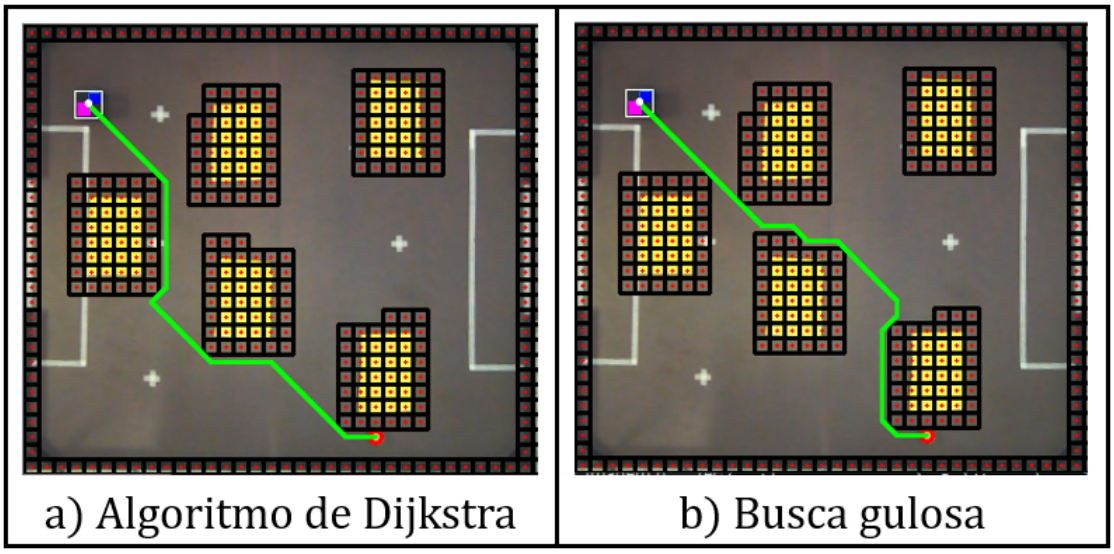


Figura 75 – Caminho obtido pelo algoritmo de Dijkstra (a) e busca em gulosa (b).

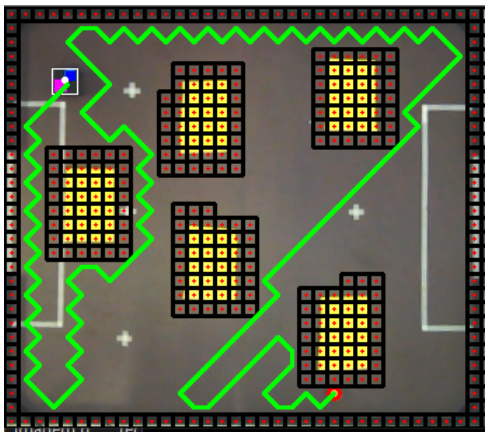


Figura 76 – Caminho obtido pelo busca em profundidade.

Tabela 9 – Comparação dos resultados obtidos para o cenário da Figura 73.

Estratégia de busca	Tempo de execução (segundos)	Tamanho do caminho (passos)
Busca em largura	0.07057	30
Busca em profundidade	0.02207	124
Busca gulosa	0.00526	30
Algoritmo de Dijkstra	0.11001	30
Algoritmo A*	0.01440	30

Analisando visualmente os caminhos obtidos pelos algoritmos de busca, mais uma vez notou-se a discrepância entre a resposta fornecida pelo DFS se comparada a dos outros algoritmos. Enquanto eles planejam um trajeto semelhante a uma diagonal entre o início do percurso e o final, a busca em profundidade realiza uma expansão desordenada pelo ambiente. O caminho planejado pelo método de Dijkstra foi semelhante mesmo ao planejado pela busca em largura, mas com um tempo de execução de aproximadamente 1,5 vezes maior. Novamente, o caminho obtido pelo A* e pela busca gulosa foram os mesmos, porém, o tempo de execução do A* é aproximadamente 2,5 vezes maior.

O planejamento de caminho para os cenários das Figuras 62 e 73 demonstrou uma superioridade da busca gulosa em relação ao algoritmo A*, apresentando o mesmo caminho, porém, com um tempo de execução menor. Entretanto, esse resultado não pode ser generalizado, visto que a busca gulosa não é ótima (Seção 2.5.1.1) e o A*, com uma heurística admissível, é ótimo (Seção 2.5.1.3).

Para demonstrar um exemplo dessa situação, foi utilizado um algoritmo que cria obstáculos aleatórios que são repassados aos algoritmos de busca para o planejamento do caminho. Na Figura 77 é apresentado o caminho calculado pela busca gulosa, enquanto na Figura 78 é apresentado o caminho referente ao A*. Em ambas as figuras, os obstáculos são representados por "#"e o caminho é representado pela letra "o", marcado em verde para facilitar a visualização.

Tabela 10 – Comparação dos resultados entre o A* e a busca gulosa.

Estratégia de busca	Tempo de execução (segundos)	Tamanho do caminho (passos)
Busca gulosa	0.00547	39
Algoritmo A*	0.00902	38

Analisando a Tabela 10 é possível perceber que o tempo de execução da busca gulosa ainda continua menor do que o do A*. No entanto, ao se analisar o tamanho do caminho obtido, percebe-se que o A* planejou um caminho com um número menor de passos, se comparado a busca gulosa. Isso demonstra que a busca gulosa, apesar de obter soluções rápidas, nem sem sempre irá fornecer o caminho ótimo.

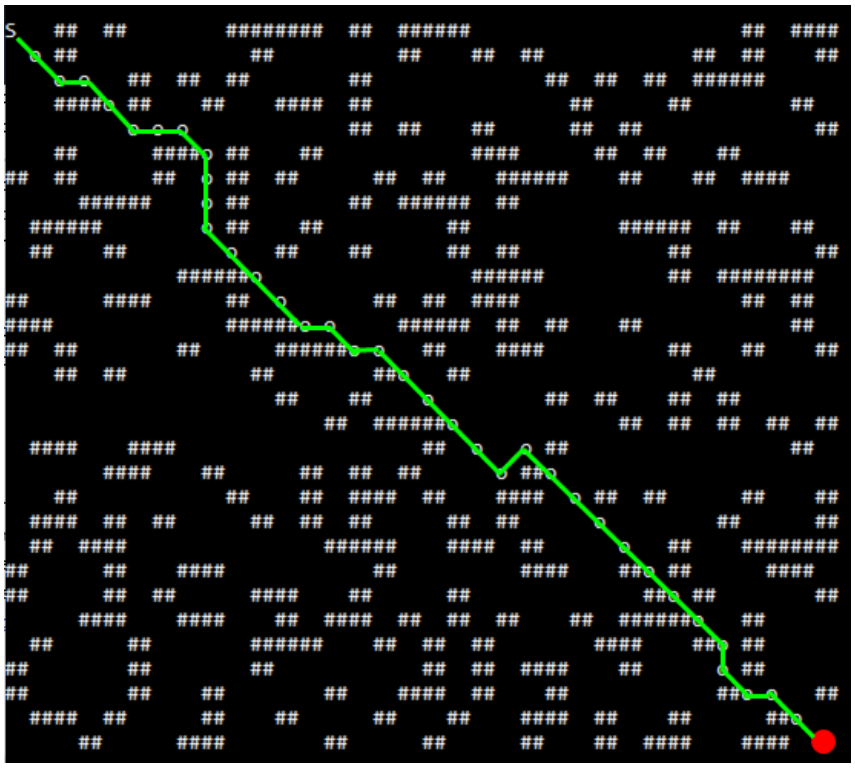


Figura 77 – Caminho obtido pela busca gulosa.

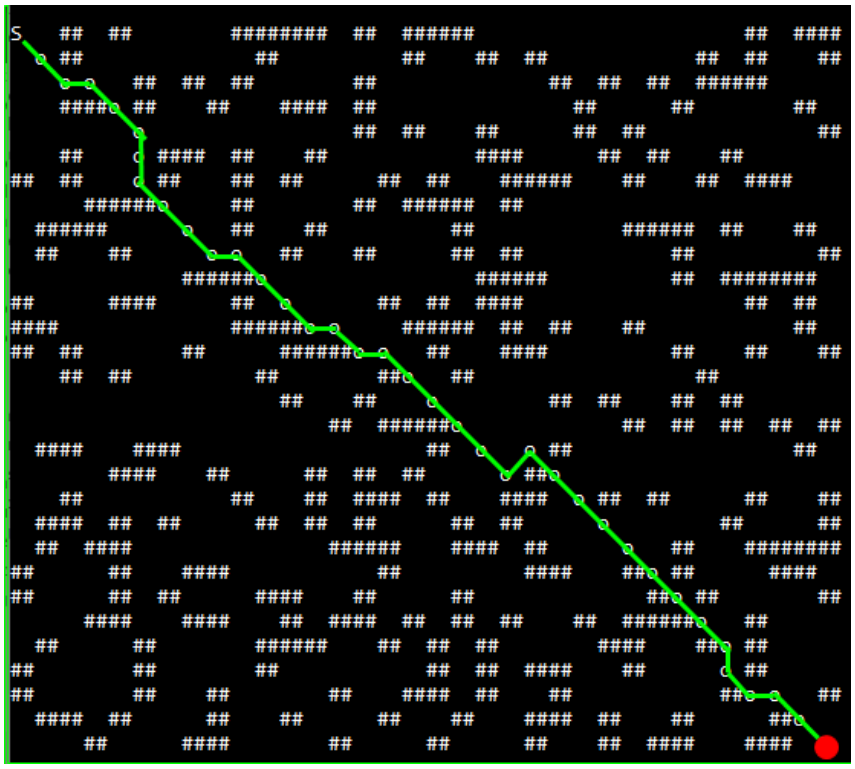


Figura 78 – Caminho obtido pelo A*.

5 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

No presente trabalho foram implementados algoritmos de busca em grafos para o planejamento de caminhos para um robô móvel diferencial. Utilizando as imagens fornecidas por uma câmera, um sistema de visão computacional reconhece obstáculos estáticos presentes no campo, assim como a posição inicial do robô e o alvo do trajeto. Por meio do método de decomposição de células, o ambiente é dividido em regiões regulares e em conjunto com as informações referentes aos objetos presentes no campo, é criado o grafo de conectividade. Esse grafo é utilizado pelos algoritmos de busca, especificamente busca em largura, busca em profundidade, busca gulosa de melhor escolha, algoritmo de Dijkstra e algoritmo A*, para calcular o caminho livre de obstáculos que leve o robô da posição inicial até a posição objetivo.

Dada à importância do planejamento de caminhos no ramo da robótica móvel, esse trabalho atendeu aos objetivos propostos, criando um sistema de visão computacional capaz de reconhecer os objetos presentes no cenário em diversos testes e desenvolvendo um planejador de caminhos eficiente que calcula o melhor caminho para o robô, dando a ele a capacidade de planejar os seus movimentos.

Este trabalho indica ainda a possibilidade de utilizar o algoritmo A* como um planejador para robôs futebolistas, já que o mesmo tende a encontrar o caminho ótimo até a objetivo, em um tempo reduzido, principalmente, em situações em que a região livre para se locomover é pequena, quando comparado com os outros algoritmos. É interessante notar que a busca gulosa também poderia ser utilizada em um ambiente com regiões mais livres de obstáculos, encontrando o menor caminho em menor tempo, como visto nos primeiros testes realizados. Porém não é garantido que o mesmo forneça o caminho ótimo, já que não modifica a conexão entre os nós do caminho encontrado, durante o processo de busca.

Como sugestões de trabalhos futuros, propõe-se a implementação de um controle de posição para o robô, de modo que o mesmo possa seguir o caminho planejado previamente e um controle embarcado de velocidade.

Também propõe-se que sejam realizados testes em ambientes dinâmicos reais, de maneira que o planejador por busca possa ser responsável por alterar dinamicamente o caminho planejado, conforme sugerido em (FREITAS; PASSOS; PEREIRA, 2017) para um planejador probabilístico.

REFERÊNCIAS

ANDRADE, W. et al. Team description paper equipe rodetas robô clube universidade federal de ouro preto. In: . [S.l.: s.n.], 2018. Citado 2 vezes nas páginas 53 e 54.

ASAI, M.; FUKUNAGA, A. Exploration among and within plateaus in greedy best-first search. In: *Twenty-Seventh International Conference on Automated Planning and Scheduling*. [S.l.: s.n.], 2017. Citado 2 vezes nas páginas 36 e 38.

BEAMER, S.; ASANOVIĆ, K.; PATTERSON, D. Direction-optimizing breadth-first search. *Scientific Programming*, 2013. Hindawi, v. 21, n. 3-4, p. 137–148, 2013. Citado na página 32.

BORGES, L. E. *Python para desenvolvedores: aborda Python 3.3*. [S.l.]: Novatec Editora, 2014. Citado na página 51.

BRADSKI, G.; KAEHLER, A. *Learning OpenCV: Computer vision with the OpenCV library*. [S.l.]: "O'Reilly Media, Inc.", 2008. Citado 3 vezes nas páginas 19, 20 e 58.

CARDOSO, A. M. D. S. *Teoria dos grafos: uma reflexão sobre a sua abordagem no ensino não universitário*. Dissertação (Mestrado) — Universidade Portucalense, 2009. Citado na página 24.

CARDOSO, D. M. Teoria dos grafos e aplicações. 2011. 2011. Citado 3 vezes nas páginas 22, 23 e 24.

CHOSET, H. M. et al. *Principles of robot motion: theory, algorithms, and implementation*. [S.l.]: MIT press, 2005. Citado na página 44.

COMMONS, W. *File:HSV color solid cylinder.png*. 2018. Disponível em: <https://commons-.wikimedia.org/w/index.php?title=File:HSV_color_solid_cylinder.png>. Acesso em: 10 nov. 2019. Citado na página 60.

COMMONS, W. *File:RGB color solid cube.png*. 2018. Disponível em: <https://commons-.wikimedia.org/w/index.php?title=File:RGB_color_solid_cube.png>. Acesso em: 10 nov. 2019. Citado na página 60.

CORMEN, T.; BALKCOM, D. *Representando grafos*. 2019. Disponível em: <https:-//pt.khanacademy.org/computing/computer-science/algorithms/graph-representation/a-/representing-graphs>. Acesso em: 07 out. 2019. Citado 4 vezes nas páginas 25, 26, 27 e 28.

CORMEN, T. H. et al. Introduction to algorithms second edition. *The Knuth-Morris-Pratt Algorithm, year*, 2001. 2001. Citado na página 43.

COSTA, P. P. da. Teoria dos grafos e suas aplicações. 2011. Universidade Estadual Paulista (UNESP), 2011. Citado 2 vezes nas páginas 23 e 24.

DUCHOŇ, F. et al. Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, 2014. Elsevier, v. 96, p. 59–69, 2014. Citado 2 vezes nas páginas 43 e 45.

DUCHOŇ, F. et al. Optimal navigation for mobile robot in known environment. In: TRANS TECH PUBL. *Applied Mechanics and Materials*. [S.l.], 2013. v. 282, p. 33–38. Citado na página 51.

FELNER, A. Position paper: Dijkstra's algorithm versus uniform cost search or a case against dijkstra's algorithm. In: *Fourth annual symposium on combinatorial search*. [S.l.: s.n.], 2011. Citado na página 38.

FEOFILOFF, P. *Algoritmos para Grafos*. 2017. Disponível em: <https://www.ime.usp.br-/~pf/algoritmos_para_grafos/aulas/graphs.html>. Acesso em: 07 out. 2019. Citado na página 25.

FEOFILOFF, P.; KOHAYAKAWA, Y.; WAKABAYASHI, Y. Uma introdução sucinta à teoria dos grafos. 2011. 2011. Citado na página 24.

FREITAS, E. J. R.; PASSOS, H. A.; PEREIRA, G. A. S. Desvio de obstáculos por robôs semiautônomos usando planejamento de caminhos. *XIII Simpósio Brasileiro de Automação Inteligente*, 2017. Porto Alegre, p. 1043–1048, 2017. Citado 2 vezes nas páginas 16 e 77.

GOLDBERG, A. V.; TARJAN, R. E. Expected performance of dijkstra's shortest path algorithm. *NEC Research Institute Report*, 1996. Citeseer, 1996. Citado na página 38.

GONÇALVES, P. C. Protótipo de um robô móvel de baixo custo para uso educacional. *Programa de Pós-Graduação da CAPES. Universidade Estadual de Maringá*, 2007. 2007. Citado na página 52.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 1968. IEEE, v. 4, n. 2, p. 100–107, 1968. Citado 2 vezes nas páginas 44 e 50.

HISTÓRIA E MODERNIDADE. *Königsberg em desenhos e gravuras antigas. Parte 2*. 2011. Disponível em: <https://visualhistory.livejournal.com/39249.html>. Acesso em: 07 out. 2019. Citado na página 23.

HUSQVARNA. *Automower Husqvarna*. 2019. Disponível em: <https://www.husqvarna.com/br-/produtos/automower/>. Acesso em: 18 nov. 2019. Citado na página 15.

IEEE. *Rules for the IEE Very Small Competition*. 2008. Disponível em: <http://www.cbrobotica-.org/wp-content/uploads/2014/03/VerySmall2008_en.pdf>. Acesso em: 15 nov. 2019. Citado 2 vezes nas páginas 52 e 55.

IROBOT. *Roomba série 900*. 2019. Disponível em: <https://www.irobot.com.br/roomba/900-series>. Acesso em: 18 nov. 2019. Citado na página 15.

JIANG, K. *Calibrate fisheye lens using OpenCV — part 1*. 2017. Disponível em: <https:/-/medium.com/@kennethjiang/calibrate-fisheye-lens-using-opencv-333b05afa0b0>. Acesso em: 20 nov. 2019. Citado na página 56.

JOSHI, V. *Finding The Shortest Path, With A Little Help From Dijkstra*. 2017. Disponível em: <https://medium.com/basecs/finding-the-shortest-path-with-a-little-help-from-dijkstra-613149fbdc8e>. Acesso em: 07 out. 2019. Citado na página 39.

JUNG, C. R. et al. Computação embarcada: Projeto e implementação de veículos autônomos inteligentes. *Anais do CSBC*, 2005. v. 5, p. 1358–1406, 2005. Citado 2 vezes nas páginas 16 e 18.

KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 1985. Elsevier, v. 27, n. 1, p. 97–109, 1985. Citado 2 vezes nas páginas 32 e 36.

LATOMBE, J.-C. *Robot motion planning*. New York: Kluwer Academic Publisher (KAP), 1991. Citado 3 vezes nas páginas 20, 21 e 22.

MARENGONI, M.; STRINGHINI, S. Tutorial: Introdução à visão computacional usando opencv. *Revista de Informática Teórica e Aplicada*, 2009. v. 16, n. 1, p. 125–160, 2009. Citado 2 vezes nas páginas 19 e 20.

MEHLHORN, K.; ORLIN, J.; TARJAN, R. *Faster algorithms for the shortest path problem*. [S.l.], 1987. Citado na página 43.

MEHLHORN, K.; SANDERS, P. *Algorithms and data structures: The basic toolbox*. [S.l.]: Springer Science & Business Media, 2008. Citado na página 43.

MOLINA, L. *Planejamento de movimento para robôs móveis baseado na condição de horizonte continuado*. Tese (Doutorado) — Universidade Federal de Campina Grande, 2014. Citado 3 vezes nas páginas 19, 21 e 22.

NASA. *NASA's First Rover on the Red Planet*. 2019. Disponível em: <https://www.nasa.gov-/image-feature/nasas-first-rover-on-the-red-planet>. Acesso em: 18 nov. 2019. Citado na página 16.

NORVIG, P.; RUSSELL, S. *Inteligência Artificial: Tradução da 3a Edição*. [S.l.]: Elsevier Brasil, 2014. Citado 11 vezes nas páginas 28, 29, 32, 33, 35, 36, 38, 44, 45, 50 e 51.

NOTO, M.; SATO, H. A method for the shortest path search by extended dijkstra algorithm. In: IEEE. *Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0*. [S.l.], 2000. v. 3, p. 2316–2320. Citado 2 vezes nas páginas 39 e 43.

OPENCV. *Camera Calibration*. 2017. Disponível em: <https://opencv-python-tutroals-.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html>. Acesso em: 20 nov. 2019. Citado 2 vezes nas páginas 55 e 56.

OPENCV. *About OpenCV*. 2019. Disponível em: <https://opencv.org/about/>. Acesso em: 10 nov. 2019. Citado na página 51.

OPENCV. *Morphological Transformations*. 2019. Disponível em: <https://docs.opencv.org-/master/d9/d61/tutorial_py_morphological_ops.html>. Acesso em: 10 nov. 2019. Citado na página 59.

PAOLETTI, T. *Leonard Euler's Solution to the Konigsberg Bridge Problem*. 2011. Disponível em: <https://www.maa.org/press/periodicals/convergence/leonard-eulers-solution-to-the-konigsberg-bridge-problem>. Acesso em: 07 out. 2019. Citado 2 vezes nas páginas 22 e 23.

PLACE, N. S. *The Mars Rovers*. 2019. Disponível em: <https://spaceplace.nasa.gov/mars-rovers/en/>. Acesso em: 25 nov. 2019. Citado na página 16.

POFFO, R. et al. Cirurgia robótica em cardiologia: um procedimento seguro e efetivo. *Einstein (São Paulo)*, 2013. v. 11, n. 3, p. 296–302, 2013. Citado na página 15.

SHIRINIVAS, S.; VETRIVEL, S.; ELANGO, N. Applications of graph theory in computer science an overview. *International journal of engineering science and technology*, 2010. v. 2, n. 9, p. 4610–4621, 2010. Citado na página 28.

SIMÕES, A.; COSTA, A. Classificação de cores por redes neurais artificiais: um estudo do uso de diferentes sistemas de representação de cores no futebol de robôs móveis autônomos. In: *XXI Congresso da Sociedade Brasileira de Computação*. [S.l.: s.n.], 2001. v. 1, p. 182. Citado 2 vezes nas páginas 59 e 60.

SINGH, A.; YADAV, A.; RANA, A. K-means with three different distance metrics. *International Journal of Computer Applications*, 2013. Citeseer, v. 67, n. 10, 2013. Citado na página 45.

WOLF, D. F. et al. Intelligent robotics: From simulation to real world applications, sbc-jai 2009-congresso da sbc–sociedade brasileira de computação (brasil). *SBC Jornada de Atualização em Informática*, 2009. p. 279–330, 2009. Citado 2 vezes nas páginas 15 e 16.

XU, M. et al. An improved dijkstra's shortest path algorithm for sparse network. *Applied Mathematics and Computation*, 2007. Elsevier, v. 185, n. 1, p. 247–254, 2007. Citado 2 vezes nas páginas 38 e 43.

ZENG, W.; CHURCH, R. L. Finding shortest paths on real road networks: the case for a. *International journal of geographical information science*, 2009. Taylor & Francis, v. 23, n. 4, p. 531–543, 2009. Citado na página 44.