

UNIVERSIDADE FEDERAL DE ITAJUBÁ

**PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA**



**APLICAÇÃO *PATHFINDING* E ÁRVORE DE DECISÃO PARA
NAVEGAÇÃO DE SISTEMA ROBÓTICO AUTÔNOMO**

JOSÉ EDUARDO SILVA GOMES

ITAJUBÁ, ABRIL DE 2010

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA



JOSÉ EDUARDO SILVA GOMES

APLICAÇÃO *PATHFINDING* E ÁRVORE DE DECISÃO PARA
NAVEGAÇÃO DE SISTEMA ROBÓTICO AUTÔNOMO

Dissertação submetida ao Programa de Pós-Graduação em Engenharia Elétrica como parte dos requisitos para obtenção do Título de Mestre em Ciências em Engenharia Elétrica.

Área de Concentração: Automação e Sistemas Elétricos Industriais.

ORIENTADOR: PROF. DR. LEONARDO DE MELLO HONÓRIO
CO-ORIENTADOR: PROF. DR. CARLOS HENRIQUE V. DE MORAES

ABRIL DE 2010
ITAJUBÁ - MG

Ficha catalográfica elaborada pela Biblioteca Mauá –
Bibliotecária Margareth Ribeiro- CRB_6/1700

G633a

Gomes, José Eduardo Silva

Aplicação Pathfinding e árvore de decisão para navegação de
sistema robótico autônomo / José Eduardo Silva Gomes. -- Itajubá,
(MG) : [s.n.], 2010.

120 p. : il.

Orientador: Prof. Dr. Leonardo de Mello Honório.

Coorientador: Prof. Dr. Carlos Henrique V. de Moraes.

Dissertação (Mestrado) – Universidade Federal de Itajubá.

1. Inteligência artificial. 2. Robôs móveis autônomos. 3. Algoritmos de busca. 4. Pathfinding A*. I. Honório, Leonardo de Mello, orient. II. Moraes, Carlos Henrique V. de, coorient. III. Universidade Federal de Itajubá. IV. Título.

DEDICATÓRIA

Dedico este trabalho ao meu saudoso pai,
Rui Gomes da Silva, que me serviu de exemplo
pelo entusiasmo, dedicação e responsabilidade
com que conduziu a família e a sua profissão. E,
apesar de sua pouca instrução, muito me ensinou.

À minha querida mãe, Maria Aparecida, que
sempre me apoiou e aconselhou.

À minha esposa Juliana por seu extremo
companheirismo e meus filhos Paula, Júlia e Rui,
pelo carinho e paciência que tiveram durante as
ausências ao longo deste período.

AGRADECIMENTOS

Gostaria de agradecer a todos que, direta ou indiretamente, contribuíram para a realização deste trabalho e também colaboraram na conclusão do curso de Mestrado em Engenharia Elétrica. Abaixo cito aqueles que se destacaram como grandes colaboradores e a quem serei eternamente grato. Aproveito para me desculpar por ter, eventualmente, negligenciado a nossa amizade no afã de cumprir as minhas tarefas acadêmicas com esmero.

Agradeço principalmente a Deus, pela saúde e perseverança que me foi conferida para a conclusão desta etapa da minha vida.

À minha querida esposa Juliana, que sempre me faz superar os desafios da vida com coragem e pensamento positivo, pela dedicação e compreensão.

Aos meus filhos Paula, Júlia e Rui pela compreensão e pelo carinho.

À minha família, em especial meus pais, Maria Aparecida e Rui (*in memoriam*) pelos conselhos e estímulo.

A EPTV, pela oportunidade e confiança. Em especial ao Sr. Clovis Couto pelo apoio, compreensão e incentivo.

A Universidade Federal de Itajubá e professores, pela oportunidade, estrutura disponibilizada e por compartilharem o conhecimento com dedicação e paciência. Especialmente ao Professor Carlos Henrique Valério de Moraes pelos ensinamentos e incentivo.

Aos companheiros de trabalho do CRTi pela contribuição e apoio.

Sou particularmente grato ao meu orientador, Professor Leonardo de Mello Honório pela confiança e orientação.

*"Aprender é a única coisa de que a mente nunca se cansa,
nunca tem medo e nunca se arrepende."*

Leonardo Da Vinci

RESUMO

Na robótica atual é crescente o desenvolvimento de pesquisas e aplicações focadas na construção de robôs móveis autônomos. Um robô autônomo e inteligente deve ser capaz de se movimentar e de tomar decisões sem a necessidade da intervenção humana. A programação para um sistema robótico autônomo não é trivial, ela não deve ser elaborada através da programação tradicional e exige soluções de Inteligência Artificial (IA).

O objetivo deste trabalho é implementar um sistema que realize o mapeamento de um ambiente desconhecido, o planejamento do caminho a ser seguido e tome as decisões necessárias para que um robô possa se movimentar por este ambiente de forma eficiente e segura. As estratégias propostas têm como base o algoritmo de busca de caminho A^* (pronuncia-se A estrela). Algoritmos de busca de caminho (*pathfinding*) são técnicas de IA que determinam o melhor trajeto entre dois pontos, sem que haja colisão com obstáculos. Para validar as estratégias propostas, foi desenvolvida uma ferramenta de simulação que representa a navegação do robô virtual em um ambiente desconhecido.

Palavras-chave: Inteligência Artificial, Robôs Móveis Autônomos, Mapeamento, Navegação, Algoritmos de Busca, *Pathfinding* A^* .

ABSTRACT

In the current robotics is increasing the development of research and applications focused on building autonomous mobile robots. An intelligent and autonomous robot must be able to move and make decisions without human intervention. The programming for an autonomous robotic system is not trivial, it should not be elaborated through the traditional programming and demands solutions of Artificial Intelligence (AI).

The objective of this work is to implement a system that performs the mapping of an unknown environment, the path planning to be followed and take the necessary decisions so that a robot can move in this environment efficiently and safely. The proposed strategies have as their base the A* (pronounced A star) path search algorithm. Algorithms for path search (pathfinding) are AI techniques that determine the best path between two points, without having collision with obstacles. To validate the proposed strategies, a simulation tool was developed to represent the navigation of the virtual robot in an unknown environment.

Keywords: Artificial Intelligence, Autonomous Mobile Robots, Mapping, Navigation, Search Algorithms, A* Pathfinding.

LISTA DE FIGURAS

Figura 2.1 – Linha de montagem automobilística (TAYLOR, 2009)	9
Figura 2.2 – Robô de exploração (NASA, 2007)	10
Figura 2.3 – Representação métrica do ambiente.....	13
Figura 2.4 – Representação topológica do ambiente	13
Figura 2.5 – Busca em Largura	17
Figura 2.6 – Busca em Profundidade	19
Figura 2.7 – Interface Gráfica.....	25
Figura 2.8 – Cálculo dos valores de $h(n)$ e $g(n)$	27
Figura 2.9 – Execução do algoritmo A^*	28
Figura 2.10 – Execução do algoritmo de busca Gulosa	28
Figura 2.11 – Execução do algoritmo de Dijkstra	29
Figura 2.12 – Algoritmo A^* com os valores de $f(n)$ para cada nó expandido	30
Figura 2.13 – Algoritmo de busca Gulosa com os valores de $h(n)$ para cada nó expandido.....	30
Figura 2.14 – Algoritmo de Dijkstra com os valores de $g(n)$ para cada nó expandido.....	31
Figura 2.15 – Heap binário mínimo (LESTER, 2003)	34
Figura 2.16 – Heap em árvore e respectiva representação em vetor (LESTER, 2003)	34
Figura 2.17 – Adicionado um novo item no heap (LESTER, 2003)	35
Figura 2.18 – Retirando um item do heap (LESTER, 2003).....	36
Figura 3.1 – Interface Gráfica.....	39
Figura 3.2 – Imagem parcial da Área de Exploração	40
Figura 3.3 – Área de Visualização.....	41
Figura 3.4 – Indicação de Direção de Visualização.....	41
Figura 3.5 – Controles de Formação do Cenário	42
Figura 3.6 – Controles de Busca do Objetivo	43

Figura 3.7 – O robô atravessa os cantos dos obstáculos.....	46
Figura 3.8 – O robô não atravessa os cantos dos obstáculos.....	47
Figura 3.9 – Recupera um cenário armazenado	52
Figura 3.10 – Armazena um cenário criado na Área de Exploração	53
Figura 3.11 – Melhor caminho entre o robô e a bola.....	55
Figura 3.12 – Botão antes de executar o Melhor Caminho	56
Figura 3.13 – Botão depois de executar o Melhor Caminho	56
Figura 3.14 – Custos F do pathfinding do Melhor Caminho	57
Figura 3.15 – Mapeamento do ambiente.....	58
Figura 3.16 – Botão antes de executar o Mapear Área.....	59
Figura 3.17 – Botão depois de executar o Mapear Área	59
Figura 3.18 – Parte do ambiente que será explorado pelo robô.....	60
Figura 3.19 – Início do mapeamento da Área de Exploração	60
Figura 3.20 – Dimensões da Área de Visualização.....	61
Figura 3.21 – Borda que mostra a área que o robô está visualizando	62
Figura 3.22 – Imagem inicial da Área de Visualização.....	62
Figura 3.23 – Visualização em cada direção.....	63
Figura 3.24 – Cálculo de um Ponto de Verificação	64
Figura 3.25 – A visão do robô atravessa os obstáculos	66
Figura 3.26 – A visão do robô não atravessa os obstáculos	67
Figura 3.27 – Pontos de busca em todo perímetro da Área de Visualização.....	70
Figura 3.28 – Pontos de busca seguindo os critérios descritos.....	70
Figura 3.29 – O robô tem a visão obstruída dentro do seu campo de visão	71
Figura 3.30 – Pontos de busca gerados dentro da Área de Visualização	71
Figura 3.31 – Primeira visualização	71
Figura 3.32 – Segunda visualização	71
Figura 3.33 – Pontos de busca gerados na Área de Visualização	72
Figura 3.34 – Pontos de busca inseridos na Área de Exploração	72
Figura 3.35 – Custos F até cada ponto de busca.....	73
Figura 3.36 – Robô em um novo ponto de visualização.....	75
Figura 3.37 – O robô encontra o seu objetivo	75
Figura 3.38 – O objetivo não foi encontrado	76
Figura 3.39 – Melhor caminho e o resultado do mapeamento do ambiente.....	77

Figura 4.1 – Interface Gráfica.....	79
Figura 4.2 – Posicionando o carro e o objetivo	80
Figura 4.3 – Melhor caminho entre o carro e o objetivo	81
Figura 4.4 – Visualização na Área de Exploração e na Área de Visualização	82
Figura 4.5 – Possíveis pontos de busca.....	82
Figura 4.6 – Deslocamento do veículo no ambiente	83
Figura 4.7 – Percorrendo o campus em busca do objetivo	84
Figura 4.8 – Visualizando e mapeando o ambiente	84
Figura 4.9 – O veículo autônomo chega ao seu objetivo	85
Figura 4.10 – O mapeamento do ambiente desconhecido e o melhor caminho simultaneamente	85

LISTA DE TABELAS

Tabela 2.1 – Estados da Área de Busca	26
Tabela 3.1 – Estados da Área de Visualização	66
Tabela 3.2 – Ponto de Busca	69
Tabela 3.3 – Estados dos caminhos percorridos pelo robô.....	73

LISTA DE ABREVIATURAS E SIGLAS

A*	- <i>Algoritmo de Busca A*.</i>
AI	- <i>Artificial Intelligence.</i>
C	- <i>Linguagem de Programação C.</i>
C++	- <i>Linguagem de Programação C++.</i>
C#	- <i>Linguagem de Programação Visual C Sharp.</i>
CRTi	- <i>Centro de Referência em Tecnologias da Informação.</i>
EPTV	- <i>Empresas Pioneiras de Televisão.</i>
GBytes	- <i>Giga Bytes.</i>
GDI+	- <i>Graphics Device Interface.</i>
GHz	- <i>Giga Hertz.</i>
JAVA	- <i>Linguagem de Programação JAVA.</i>
IA	- <i>Inteligência Artificial.</i>
IDE	- <i>Integrated Development Environment.</i>
MBytes	- <i>Mega Bytes.</i>
NASA	- <i>National Aeronautics and Space Administration.</i>
.NET	- <i>Microsoft .NET, plataforma para desenvolvimento de aplicações.</i>
RAM	- <i>Random Access Memory.</i>
SLAM	- <i>Simultaneous Localization and Mapping.</i>
UNIFEI	- <i>Universidade Federal de Itajubá.</i>

SUMÁRIO

DEDICATÓRIA	I
AGRADECIMENTOS	II
RESUMO	IV
ABSTRACT	V
LISTA DE FIGURAS	VI
LISTA DE TABELAS	IX
LISTA DE ABREVIATURAS E SIGLAS	X
SUMÁRIO	XI
1. INTRODUÇÃO	1
1.1. CONSIDERAÇÕES INICIAIS	1
1.2. OBJETIVO	2
1.3. O SIMULADOR	3
1.4. ORGANIZAÇÃO DA DISSERTAÇÃO	5
2. REFERENCIAL TEÓRICO	7
2.1. INTRODUÇÃO	7
2.2. ROBÔ MÓVEL AUTÔNOMO	8
2.3. MAPEAMENTO E NAVEGAÇÃO	11
2.3.1. MAPEAMENTO	12
2.3.2. NAVEGAÇÃO	14
2.4. ALGORITMOS DE BUSCA	15
2.4.1. BUSCA SEM INFORMAÇÃO	16
2.4.1.1. Busca em Largura	17
2.4.1.2. Busca de Custo Uniforme	18
2.4.1.3. Busca em Profundidade	18
2.4.1.4. Busca em Profundidade Limitada	20
2.4.1.5. Busca de Aprofundamento Iterativo	20
2.4.2. BUSCA COM INFORMAÇÃO	21
2.4.2.1. Busca Gulosa pela Melhor Escolha	22
2.4.2.2. Busca A*	22
2.4.3. COMPARAÇÃO ENTRE ALGORITMOS DE BUSCA DE CUSTO UNIFORME (DIJKSTRA), GULOSA PELA MELHOR ESCOLHA E A*	24

2.5. HEAP BINÁRIO EM APLICAÇÃO PATHFINDING	32
2.5.1. ADICIONAR UM NOVO ITEM NO <i>HEAP</i>	34
2.5.2. RETIRAR UM ITEM NO <i>HEAP</i>	36
3. IMPLEMENTAÇÃO DO SIMULADOR.....	38
3.1. INTRODUÇÃO	38
3.2. A INTERFACE GRÁFICA	39
3.2.1. ANIMAÇÃO DO SIMULADOR.....	39
3.2.1.1. Área de Exploração do Robô.....	40
3.2.1.2. Área de Visualização do Robô.....	40
3.2.1.3. Direção de Visualização do Robô.....	41
3.2.2. FORMAÇÃO DO CENÁRIO	41
3.2.2.1. Grupo de Seleção “Robô” / “Bola” / “Obstáculo” / “Apagar”.....	42
3.2.2.2. Botão “Limpar”.....	42
3.2.2.3. Botão “Abrir”	42
3.2.2.4. Botão “Salvar”	43
3.2.3. BUSCA DO OBJETIVO.....	43
3.2.3.1. Botão “Melhor Caminho”.....	43
3.2.3.2. Botão “Custos”	43
3.2.3.3. Botão “Mapear Área”	44
3.2.3.4. Botão “Visão Geral”	44
3.3. ALGORITMO DE <i>PATHFINDING A*</i>	44
3.3.1. MÉTODO CONSTRUTOR (<i>PATHFINDING</i>)	45
3.3.2. MÉTODO <i>FINDPATH</i>	45
3.3.3. MÉTODO <i>LOADWALKABILITY</i>	48
3.3.4. MÉTODOS DE LEITURA DE DADOS	48
3.3.4.1. Método <i>ReadPathLength</i>	48
3.3.4.2. Método <i>ReadPathX</i>	48
3.3.4.3. Método <i>ReadPathY</i>	48
3.3.4.4. Método <i>ReadFcost</i>	49
3.3.4.5. Método <i>ReadGcost</i>	49
3.3.4.6. Método <i>ReadHcost</i>	49
3.3.4.7. Método <i>ReadOpenX</i>	49
3.3.4.8. Método <i>ReadOpenY</i>	49
3.4. IMPLEMENTAÇÃO DO SIMULADOR GRÁFICO.....	50
3.4.1. FORMAÇÃO DO CENÁRIO	50
3.4.1.1. Inserção de Objetos na Área de Exploração.....	50
3.4.1.2. Limpar a Área de Exploração	51
3.4.1.3. Recuperar um Cenário	52
3.4.1.4. Salvar um Cenário.....	53
3.4.2. BUSCA DO OBJETIVO	54
3.4.2.1. Navegação pelo Melhor Caminho.....	55
3.4.2.2. Mostra os Custos F do Melhor Caminho.....	57
3.4.2.3. Mapeamento do Ambiente.....	57

3.4.2.3.1. Evento Click do botão “Mapear Área”	59
3.4.2.3.2. Função ProcuraAlvo	59
3.4.2.3.3. Função Visualizar	60
3.4.2.3.4. Função PontosBusca.....	69
3.4.2.3.5. Função ProximoPonto	72
3.4.2.4. Visão Geral.....	76
3.4.2.5. Atualiza a Área de Exploração.....	77
3.4.2.6. Atualiza a Área de Visualização	78
4. APLICAÇÃO E RESULTADOS.....	79
5. CONCLUSÃO.....	86
5.1. CONSIDERAÇÕES FINAIS	86
5.2. SUGESTÕES PARA TRABALHOS FUTUROS	88
APÊNDICE A.....	90
REFERÊNCIAS BIBLIOGRÁFICAS.....	117

1

INTRODUÇÃO

1.1. CONSIDERAÇÕES INICIAIS

Sistemas robóticos estão cada vez mais presentes na sociedade, frequentemente eles são encontrados em sistemas de automação industrial, comercial ou residencial. Inicialmente o maior emprego dos robôs foi na área industrial, sobretudo os robôs manipuladores, que normalmente eram fixos, não-autônomos, ou seja, necessitavam da assistência de um operador, e desprovidos de qualquer inteligência. O surgimento de novas aplicações exigiu a criação de robôs com capacidade de mobilidade e com mais autonomia. Entre estas novas necessidades pode-se citar a utilização de robôs móveis na execução de tarefas penosas e perigosas, ou em ambientes hostis ou inadequados ao ser humano, como, por exemplo, na exploração espacial e subaquática, remoção de bombas e minas, localização e resgate de pessoas, cirurgias, conflitos militares, patrulhamento em áreas de segurança, etc. Os robôs móveis autônomos também são utilizados em situações menos críticas, executando tarefas domésticas, como cozinhar, limpar a casa e regar plantas ou mesmo apenas como robôs de entretenimento ou de companhia.

Um robô autônomo e inteligente deve ser capaz de se movimentar e de tomar decisões sem a necessidade da intervenção humana. Para se conferir capacidade de autonomia a um robô móvel deve-se dotá-lo de dispositivos que façam o sensoramento do ambiente, atuadores que realizem a sua movimentação e um sistema computacional para controle e tomada de decisões. Entretanto a programação de sistemas robóticos autômatos tende a ser complexa e não deve ser

implementada através da programação tradicional, ela deve ser abordada por meio de técnicas de Inteligência Artificial (IA). Assim o desenvolvimento de robôs móveis autônomos, capazes de operar no mundo real, se tornou um desafio para a área de IA. Muitos pesquisadores têm concebido sistemas robóticos utilizando *algoritmos de busca*, que são técnicas de IA que pertencem a uma área chamada de Busca em Espaço de Estados. As estratégias propostas na presente dissertação têm como base o algoritmo de busca de caminho (*pathfinding*) A^* , que determina o melhor trajeto entre dois pontos, sem que haja colisão com obstáculos. A partir das informações obtidas pelos sensores do robô, que fazem o mapeamento do ambiente, o sistema desenvolvido determina a estratégia de exploração e emprega o algoritmo A^* nos processos de planejamento de trajetória e tomada de decisões.

1.2. OBJETIVO

O objetivo deste trabalho é desenvolver estratégias para a exploração de ambientes desconhecidos por sistemas robóticos móveis autônomos e inteligentes. Para desempenhar esta tarefa de forma eficiente e segura, o sistema implementado inicialmente realiza o mapeamento do ambiente, construindo uma representação interna através das informações recebidas dos sensores do robô. Este mapeamento contém as informações necessárias para a tomada de decisões no que se refere à estratégia de exploração e ao planejamento de trajetória. A estratégia de exploração define a nova posição para onde o robô deve se deslocar para continuar o mapeamento do ambiente e o planejamento de trajetória determina o caminho a ser percorrido pelo robô até atingir a posição objetivo, sem colidir com os obstáculos. As informações que são geradas permitem que o sistema execute a navegação autônoma pelo ambiente, através de comandos que são enviados para os atuadores do robô.

Pode-se dividir o sistema em quatro módulos: Mapeamento, Estratégia de Exploração, Planejamento de Trajetória e Navegação. O módulo de Mapeamento fornece os dados a respeito do ambiente, porém a qualidade das informações que são geradas pelos módulos de Estratégia de Exploração e de Planejamento de

Trajetória é que vão permitir que o módulo de Navegação faça com que o robô se locomova de forma autônoma e inteligente pelo ambiente.

Para testar e validar as estratégias propostas foi desenvolvida uma ferramenta que faz simulações computacionais, onde um robô virtual navega por ambientes que podem ser criados pelo usuário. Neste software é possível visualizar o comportamento do robô durante a exploração do ambiente. A implementação de um simulador facilita a depuração do sistema, pois possibilita a abstração de alguns aspectos relativos à constituição física do robô. A construção de um robô costuma ser dispendiosa e tomar muito tempo, enquanto um simulador permite se experimentar várias configurações, tanto nas características do robô como na criação dos ambientes, em curto espaço de tempo e sem colocar em risco a integridade física do robô ou dos objetos presentes no ambiente. As soluções validadas no simulador poderão ser, posteriormente, aplicadas a situações reais. Segundo Pegden *et al.* (1995), a simulação é um processo de projetar um modelo computacional de um sistema real e conduzir experimentos com este modelo com o propósito de entender seu comportamento e/ou avaliar estratégias para sua operação.

1.3. O SIMULADOR

Para se proceder a simulação deve-se, inicialmente, criar o ambiente virtual que o robô irá explorar. No simulador pode-se criar ambientes de diversos formatos como, por exemplo, quarteirões de uma cidade, labirintos, plantas de imóveis ou mesmo um ambiente construído aleatoriamente. Em seguida deve-se inserir o robô virtual na sua posição inicial. A busca no simulador pode ser feita de duas formas: por um objetivo específico, representado pela inserção de uma bola laranja na posição desejada, ou optar por mapear todo o ambiente, e neste caso não deve ser posicionada a bola. No simulador o usuário tem a possibilidade de salvar o ambiente criado em arquivo, caso queira reutilizá-lo posteriormente.

A princípio o robô móvel não tem qualquer informação sobre o ambiente a ser explorado. No simulador foi definido que o robô virtual faz o sensoriamento do ambiente desconhecido através de um sistema de visão computacional e para isto foi implementado um módulo que simula a percepção visual do robô. Assim como em um sistema de visão computacional real, no simulador também é estipulado um limite para o campo de visão do robô, ou seja, a distância máxima na qual a imagem adquirida é suficientemente boa para detecção de obstáculos, espaços livres e o objetivo no ambiente.

À medida que o robô visualiza a região ao seu redor o simulador realiza o mapeamento do ambiente. Com as informações deste mapa, o módulo de estratégia de exploração gera possíveis pontos para onde o robô pode se deslocar para continuar a busca do objetivo. Neste caso o algoritmo de *pathfinding* A^* é utilizado para a tomada de decisão sobre qual o próximo ponto de busca para onde o robô deve se locomover. Definido o ponto para onde o robô deve se deslocar, o módulo de planejamento de trajetória calcula, também baseado no mapa do ambiente construído até o momento e no algoritmo A^* , o melhor caminho entre o ponto onde se encontra o robô e o novo ponto de busca selecionado. Com estas informações o módulo de navegação envia comandos que possibilitam o robô virtual percorrer eficientemente o ambiente desconhecido. Estes procedimentos são repetidos até que o robô encontre o objetivo ou até que o ambiente esteja totalmente mapeado, ou seja, não exista mais nenhuma área desconhecida acessível.

Na interface gráfica do simulador, uma animação mostra o progresso do mapeamento e a navegação do robô pelo ambiente desconhecido. A imagem apresentada ilustra a área de exploração do robô, nele são exibidos os locais ainda desconhecidos, os espaços livres para navegação, as áreas ocupadas por obstáculos, os pontos para verificações futuras, as posições onde o robô realizou uma visualização do ambiente, os caminhos percorridos pelo robô e a chegada ao objetivo.

O simulador também permite que seja mostrado o melhor caminho entre o local onde o robô foi inserido no ambiente e o objetivo. Neste caso é representada a navegação do robô em um ambiente conhecido, ou seja, ele tem a informação prévia da sua localização, o mapa do ambiente e a posição do objetivo. Desta forma

o *software* permite a comparação entre os caminhos percorridos pelo robô durante o mapeamento do ambiente desconhecido e o melhor caminho entre o robô e o objetivo.

Este simulador foi implementado em ambiente Windows utilizando a linguagem de programação Visual C# (lê-se *C sharp*), da plataforma Microsoft Visual Studio 2008. O C# é uma linguagem totalmente orientada a objetos que une o poder e a flexibilidade das linguagens C e C++ com a produtividade de um IDE (*Integrated Development Environment* – ambiente de desenvolvimento integrado) bastante interativo, atributos que o qualifica para o desenvolvimento de programas computacionais que empregam algoritmos complexos e pesados como os de IA, mas que fazem uso intenso de recursos gráficos.

No capítulo 3 deste trabalho serão apresentados maiores detalhes a respeito da operação e da implementação do simulador.

1.4. ORGANIZAÇÃO DA DISSERTAÇÃO

A presente dissertação está estruturada em cinco capítulos e um apêndice. O capítulo 1 é composto por esta introdução ao trabalho.

O segundo capítulo apresenta o referencial teórico com os principais fundamentos que nortearam o desenvolvimento deste trabalho. Neste capítulo são abordados os conceitos básicos da Robótica Móvel Autônoma. Em seguida são contextualizados os problemas de Mapeamento e Navegação, essenciais para uma movimentação autônoma eficiente. Na seção seguinte são apresentados os Algoritmos de Busca, com ênfase à busca A* que foi o algoritmo empregado no sistema desenvolvido nesta dissertação. Nesta seção também foi realizado um estudo comparativo entre três algoritmos de busca utilizando uma ferramenta de *software* desenvolvida pelo autor. Finalmente é abordado a conceituação básica do método de ordenação *Binary Heap*, uma estrutura de dados fundamental para um bom desempenho do algoritmo de busca A*.

Na sequência, a terceira parte apresenta a especificação e a implementação de uma ferramenta computacional que simula as estratégias desenvolvidas para que um robô virtual realize o mapeamento e a navegação em ambientes previamente conhecidos ou desconhecidos. Nesse capítulo são descritos a interface gráfica e sua usabilidade, o desenvolvimento do algoritmo que executa a busca A*, as técnicas utilizadas na representação gráfica que mostra a exploração do ambiente pelo robô virtual e as rotinas de tomadas de decisões relativas à estratégia de exploração e ao planejamento de trajetória.

No quarto capítulo é demonstrado um exemplo de aplicação do sistema desenvolvido. Este programa realiza uma simulação que tem como cenário o campus da UNIFEI em Itajubá e a sua execução mostra a forma como um veículo autônomo mapeia e navega neste ambiente enquanto procura pelo objetivo.

O quinto capítulo expõe as conclusões finais da dissertação e são apresentadas propostas para trabalhos futuros.

Em seguida é apresentado o apêndice A, nele são destacados trechos relevantes da implementação do simulador que foram citadas no capítulo 3.

No final da dissertação encontram-se as referências bibliográficas utilizadas para o desenvolvimento do trabalho.

2

REFERENCIAL TEÓRICO

Em que vemos os principais conceitos teóricos necessários ao desenvolvimento deste trabalho.

2.1. INTRODUÇÃO

Este capítulo apresenta os principais fundamentos teóricos que nortearam o desenvolvimento deste trabalho. Inicialmente são apresentados os conceitos básicos da **robótica móvel autônoma**. Em seguida são abordados os problemas de **mapeamento e navegação**, essenciais para uma movimentação autônoma eficiente. Na seção seguinte são apresentados os **algoritmos de busca**, técnicas de Inteligência Artificial aplicadas a problemas complexos, com ênfase à busca A* que foi o algoritmo empregado nos processos de planejamento de trajetória e tomada de decisões do sistema implementado nesta dissertação. Tomada de decisões é o processo pelo qual são escolhidas algumas ou apenas uma entre muitas alternativas para as ações a serem realizadas (TOMADA, 2009).

Nesta seção também foram realizados testes comparativos de desempenho entre três algoritmos de busca, utilizando uma ferramenta de *software* desenvolvida pelo autor. Finalmente foi abordado o método de ordenação **binary heap**, uma estrutura de dados fundamental para um bom desempenho do algoritmo de busca A*.

O homem há tempos vem construindo máquinas com capacidade de movimento, muitas vezes similares aos seus, e com o sonho inerente de inserir inteligência nestes dispositivos. A maior parte das invenções humanas tem origem na imaginação e fantasia. Pandora, Golem e Frankenstein são considerados mais como sendo seres artificiais do que naturais (ULLRICH, 1987).

2.2. ROBÔ MÓVEL AUTÔNOMO

Ultimamente tem-se observado uma presença crescente de sistemas robóticos na sociedade, assim como o maior volume de pesquisa e desenvolvimento nesta área de conhecimento tem produzido robôs cada vez mais potentes e complexos. Segundo Nehmzow (2003, tradução minha) o robô é um equipamento multifuncional e reprogramável, projetado para movimentar materiais, peças, ferramentas ou dispositivos especializados através de movimentos variáveis e programados, para a execução de uma variedade de tarefas.

A robótica é uma área de pesquisa multidisciplinar que abrange conhecimentos em engenharia mecânica, engenharia eletrônica e computação, onde as técnicas de inteligência artificial (IA) tem sido extensivamente empregadas. De acordo com Taylor (2009, tradução minha),

Os sistemas robóticos continuam a evoluir, penetrando lentamente em muitas áreas de nossas vidas, desde a produção, a medicina e a exploração remota até o entretenimento, a segurança e a assistência pessoal. Desenvolvedores no Japão estão construindo atualmente robôs para ajudar aos idosos, enquanto a NASA desenvolve a próxima geração de exploradores do espaço, e os artistas estão explorando novos caminhos de entretenimento.

Os robôs podem ser classificados em duas categorias: robôs manipuladores e robôs móveis. Inicialmente houve um grande desenvolvimento dos robôs manipuladores, sobretudo para aplicações voltadas à área industrial, com o intuito de automatizar e aumentar a produção. Os robôs manipuladores operam em ambientes bem estruturados, altamente controlados, em um espaço de atuação limitado e não podem se mover, já que possuem uma base fixa. A figura 2.1 a seguir

mostra robôs manipuladores na sua principal aplicação, ou seja, em uma linha de montagem automobilística.

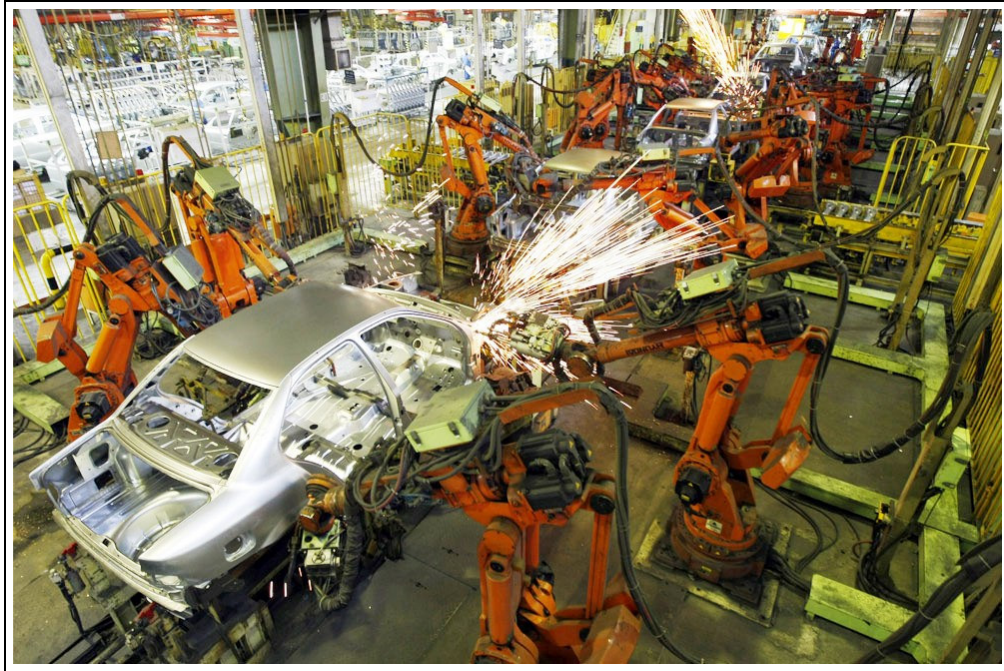


Figura 2.1 – Linha de montagem automobilística (TAYLOR, 2009).

Uma segunda classe de robôs são os robôs móveis, que têm a capacidade de se locomoverem no ambiente em que estão inseridos utilizando-se de seu sistema sensorial e de seu sistema atuador. De acordo com Nehmzow (2003, tradução minha),

Aplicações industriais de robôs móveis estão ganhando importância continuamente, em particular sob considerações de confiabilidade (execução ininterrupta e confiável de tarefas monótonas, como a vigilância), acessibilidade (inspeção de locais que são inacessíveis aos seres humanos, por exemplo, espaços apertados, ambientes perigosos ou locais remotos) ou custo (sistemas de transporte baseados em robôs móveis autônomos podem ser mais barato do que formas tradicionais de transporte).

De acordo com o tipo de comando os robôs móveis podem ser subdivididos em três categorias, como descrito em Bianchi (2007):

- Teleoperados: um operador define todos os movimentos que o robô deve executar.
- Semiautônomos: um operador indica o macro comando a ser executado e o robô o executa sozinho.

- Autônomos: o robô realiza suas tarefas sozinho, tomando suas próprias decisões.

A figura 2.2 abaixo mostra a concepção artística de um robô móvel, trata-se de um veículo de exploração da NASA utilizado na missão ao planeta Marte denominada *Mars Exploration Rover*, que em 2004 enviou dois veículos robóticos de controle remoto (*rovers*), os gêmeos *Spirit* e *Opportunity*.

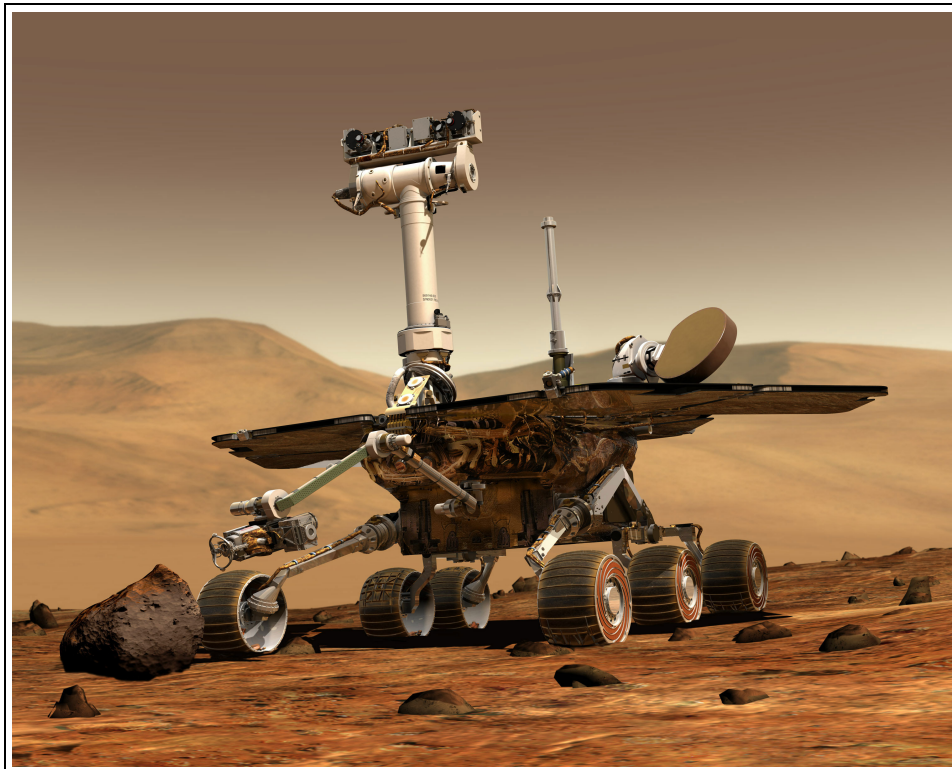


Figura 2.2 – Robô de exploração (NASA, 2007).

Em algumas situações deseja-se que os robôs tenham a capacidade de se locomover pelo ambiente sem a intervenção de operadores, ou seja, de forma autônoma. Citando Bianchi (2007), a autonomia é a habilidade de tomar suas próprias decisões e agir de acordo com elas. Para robôs, autonomia significa a habilidade de perceber e agir em uma dada situação apropriadamente.

Muitos robôs móveis são construídos para operar em ambientes previamente conhecidos, porém alguns sistemas robóticos devem ter a capacidade de desenvolver tarefas em locais pouco estruturados e desconhecidos, nos quais eles precisam tomar decisões de forma autônoma. Esta é uma das principais áreas de pesquisa na robótica atual, onde o robô deve fazer o reconhecimento do ambiente

em que está inserido, se autolocalizar, planejar a trajetória e executar uma movimentação até a posição objetivo, sem colidir com os possíveis obstáculos que forem detectados no caminho.

Um robô móvel autônomo tem a capacidade de movimentar-se no ambiente em que estiver inserido, perceber o ambiente através de seus sensores, adaptar-se às mudanças no ambiente, aprender a partir de experiências, construir representações internas do seu ambiente, que possam ser usadas no seu processo de tomada de decisão (BIANCHI, 2007 *apud* NEHMZOW, 2000) .

Para a tomada de decisões, agentes autônomos e inteligentes devem ser capazes de adquirir informações e interagir com o ambiente, segundo Russell (2003) um agente é tudo o que pode ser considerado capaz de perceber o seu ambiente por meio de sensores e de agir sobre este ambiente por intermédio de atuadores.

Para o robô móvel desenvolver uma movimentação autônoma eficiente ele deve ter as habilidades de exploração, mapeamento, autolocalização e de navegação, temas que serão apresentados na próxima seção deste referencial teórico.

2.3. MAPEAMENTO E NAVEGAÇÃO

Esta seção trata de duas importantes áreas de pesquisa da robótica móvel autônoma, os problemas de mapeamento e de navegação. Nem sempre está disponível, *a priori*, um mapa do ambiente, assim o robô autônomo precisa criar a sua representação interna do ambiente através de seu sistema de sensoramento. Esta representação é fundamental para que o robô possa se localizar e fornecer informações seguras que orientem o planejamento do caminho a ser seguido até a posição objetivo.

A exploração de um ambiente, feita para que o robô conheça o meio e realize uma posterior navegação, implica em que seja construído um mapa. Este mapa conterá informações de espaço livre para navegação, como pontos do espaço ocupados por obstáculos. Pode conter também informações de tomada de decisão do robô, como trajetórias a serem seguidas (DECCÓ, 2004)

O planejamento do caminho e o mapa provêm as informações necessárias para que o robô possa se movimentar pelo ambiente, ou seja, executar a navegação. Mapeamento e navegação eficientes são essenciais para que o robô móvel atinja um bom nível de autonomia.

2.3.1. MAPEAMENTO

Frequentemente um sistema robótico autônomo é inserido em ambiente sobre o qual não possui nenhuma informação e então é necessário que ele faça o reconhecimento do ambiente a sua volta e construa um mapa com a localização dos objetos contidos neste ambiente. Este procedimento é conhecido em robótica como mapeamento. Segundo Russell (2003), na literatura, o problema de mapeamento de robôs frequentemente é chamado localização e mapeamento simultâneos, abreviado como SLAM (*Simultaneous Localization and Mapping*). O robô não apenas deve construir um mapa, mas deve fazê-lo sem saber onde está. No mapeamento de um ambiente é de suma importância que o sistema robótico autônomo tenha a capacidade de **autolocalização**, que é a habilidade de determinar a sua posição dentro do mapa. De acordo com Cox (1991, tradução minha) localizar o robô no seu ambiente é o problema mais fundamental para se ter um robô móvel com capacidades de autonomia. Fox (1999, tradução minha) afirma que o conhecimento sobre a sua posição permite que um robô móvel realize suas tarefas de forma eficiente e confiável.

Basicamente existem duas técnicas para a representação do ambiente em um mapa, no formato métrico ou no formato topológico. Na representação métrica o ambiente é apresentado de maneira mais detalhada, porém tem um maior custo computacional. Conforme Pedrosa (2006) ele apresenta todas as características métricas de que é constituído, como posição de paredes, passagens, obstáculos, entre outros. A sua grande vantagem é manter uma boa relação com o mundo real. O formato métrico geralmente é implementado utilizando uma grade de ocupação que divide a área a ser mapeada em unidades menores, formando uma representação bidimensional com células que representam uma região do mundo real. Assim o ambiente pode ser representado por uma matriz onde aos elementos,

ou células, são atribuídos valores que indicam se a área correspondente está ocupada ou não. O tamanho da célula da grade é definido considerando aspectos como o custo computacional e a precisão que se deseja no mapeamento do ambiente. A figura 2.3 abaixo ilustra a representação métrica do ambiente.

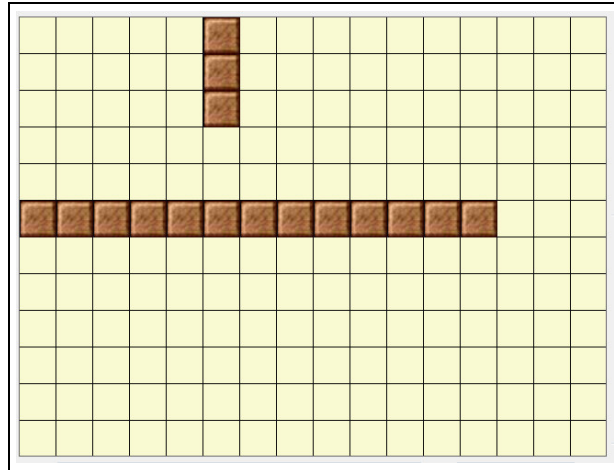


Figura 2.3 – Representação métrica do ambiente.

Na representação topológica os mapas contêm informações apenas a respeito dos locais e a relação entre eles. Esta técnica utiliza nós para representar lugares no ambiente e arcos que correspondem aos caminhos. Estes mapas não necessitam da localização precisa do robô e ocupam pouco espaço de armazenamento, mas em ambiente grandes são difíceis de serem construídos e mantidos. Segundo Pedrosa (2006) os mapas topológicos utilizam grafos para representar o ambiente de trabalho do robô. Normalmente, estes grafos descrevem os espaços livres para a execução de tarefas. A figura 2.4 abaixo mostra a representação topológica do ambiente.

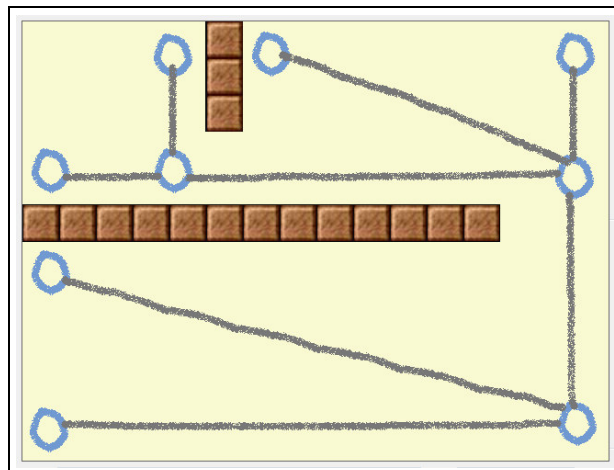


Figura 2.4 – Representação topológica do ambiente.

2.3.2. NAVEGAÇÃO

Durante o mapeamento o robô móvel constrói uma representação interna do ambiente e realiza sua autolocalização. Estas são as informações necessárias para que o mesmo possa planejar a navegação autônoma de sua posição atual até uma posição objetivo. Assim, a navegação em robótica é a capacidade do robô se deslocar para lugares baseado no conhecimento da sua localização e do ambiente ao seu redor. Segundo Siegwart (2004, tradução minha),

A navegação é uma das competências mais desafiadoras exigidas de um robô móvel. O sucesso na navegação exige o sucesso nos quatro blocos de construção da navegação: percepção, o robô deve interpretar seus sensores para extrair dados significativos; localização, o robô deve determinar a sua posição no ambiente; cognição, o robô deve decidir como agir para alcançar seus objetivos; e controle de movimento, o robô deve modular as saídas para seus motores para conseguir a trajetória desejada.

De posse de um mapa do ambiente, de sua localização e da posição objetivo o robô pode realizar o planejamento de trajetória (*path planning*), que define previamente o caminho a ser executado pelo robô até atingir o destino, sem colidir com os obstáculos, ou indica que o destino está inacessível. Frequentemente existem vários caminhos possíveis e os critérios para escolha podem obedecer a características como, por exemplo, o mais curto, o mais rápido ou o mais seguro. A característica desejada irá definir o método utilizado no planejamento de trajetória, necessário para posterior execução da navegação autônoma.

Várias técnicas de Inteligência Artificial podem ser empregadas no planejamento de trajetória, para a seleção do melhor caminho em um mapa se destaca a utilização dos Algoritmos de Busca, principalmente os algoritmos A Estrela e de Dijkstra. Os Algoritmos de Busca serão apresentados na próxima seção deste referencial teórico.

2.4. ALGORITMOS DE BUSCA

Alguns problemas são muito complexos para serem solucionados através da programação tradicional e devem ser abordados sob o escopo da Inteligência Artificial (IA). A utilização de IA para a navegação de sistemas robóticos autônomos tem sido muito difundida, principalmente com o emprego de algoritmos de busca. Segundo Whitby (2004) em IA, “busca” descreve uma técnica de procura de soluções para um problema.

Estes algoritmos, através de suas lógicas, exploram sistematicamente as alternativas e apresentam como solução o caminho a ser percorrido. Conforme Russell (2003) um algoritmo de busca recebe um problema como entrada e retorna uma solução sob a forma de uma sequência de ações.

Algoritmos de busca são técnicas de IA que pertencem a uma área chamada de Busca em Espaço de Estados. O espaço de estados de um problema é (RUSSELL, 2003) o conjunto de todos os estados acessíveis a partir do estado inicial. Ou seja, o espaço de estados representa todas as possíveis posições intermediárias entre o estado inicial e o estado final ou estado-objetivo. Por exemplo, em um sistema robótico autônomo o estado inicial é a posição onde o robô se encontra em um determinado momento e o estado final é o objetivo da busca, que pode ser encontrar um alvo específico ou mapear uma área desconhecida. Os estados intermediários são todos os caminhos possíveis de serem percorridos pelo robô entre a sua posição atual e o objetivo.

A definição do caminho até o objetivo é dada por uma função chamada de “custo do caminho”, que atribui um custo numérico a cada caminho gerado. De acordo com Russell (2003) o custo de um caminho pode ser descrito como a soma dos custos das ações individuais ao longo do caminho. Um caminho que parte do estado inicial e alcança o estado-objetivo é uma solução para o problema. Porém a função de custo de caminho determina a qualidade desta solução. A solução ótima é aquela que tem, entre todas as possíveis soluções, o menor custo de caminho.

Podemos avaliar os algoritmos de busca sob quatro aspectos:

- Completeza – O algoritmo encontra a solução se ela existir?
- Otimização – O algoritmo encontra a solução de menor custo?
- Tempo – Quanto tempo o algoritmo leva para encontrar a solução?
- Espaço – Quanto o algoritmo consome de memória para executar a busca?

Essencialmente, uma busca consiste em seguir uma opção e deixar as demais reservadas para verificações posteriores, uma vez que a opção selecionada pode não conduzir a uma solução do problema. Desta forma continua-se escolhendo uma nova opção, testando se ela não corresponde ao estado-objetivo e expandindo a busca, gerando novos estados a partir do estado atual. E assim sucessivamente até que seja encontrada uma solução ou não exista mais estados a serem expandidos.

A escolha de qual estado deve ser expandido é definida pela “estratégia de busca”, é ela que determina a caminho a seguir. As estratégias de busca podem ser divididas em busca sem informação e busca com informação.

2.4.1. BUSCA SEM INFORMAÇÃO

As estratégias de busca sem informação, também conhecida como busca cega ou *blind search*, são aquelas que recebem apenas as informações disponíveis na definição do problema. Conforme Bittencourt (2001) uma estratégia é dita “cega” se ela não leva em conta informações específicas sobre o problema a ser resolvido. Estas estratégias são baseadas em tentativas de solução por força bruta, ou seja, como não há qualquer informação que auxilie a exploração dos estados são testados todos os caminhos possíveis até alcançar o objetivo.

O método de busca mais simples de ser implementado é a aplicação exaustiva de um procedimento que garanta a resolução do problema, caso ela exista. Os métodos de busca sem informação mais utilizados são a busca em profundidade e a busca em largura. Basicamente o que diferencia estas estratégias de busca é a ordem em que os estados são expandidos.

2.4.1.1. BUSCA EM LARGURA

A busca em largura (LUGER, 2004) explora o espaço nível por nível. Apenas quando não houver mais estados a serem explorados num determinado nível é que o algoritmo se moverá para o próximo. Ou seja, é um método onde todos os nós de um determinado nível são verificados antes de prosseguir para um nível mais baixo, o que será feito somente se a solução não for encontrada no nível atual. Também denominada como busca em extensão ou busca em amplitude, nela a ordem de expansão dos nós é definida como a seguir:

1. Nó raiz.
2. Todos os nós de profundidade 1.
3. Todos os nós de profundidade 2.
4. Todos os nós de profundidade 3.
5. E assim sucessivamente até os nós de maior profundidade.

A figura 2.5 abaixo ilustra esta estratégia de busca.

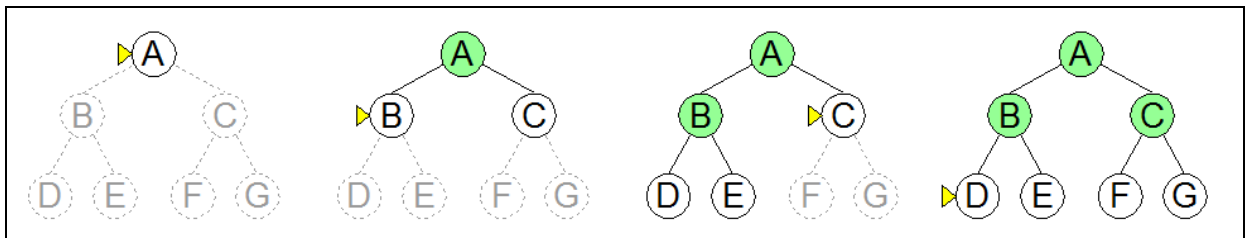


Figura 2.5 – Busca em Largura. Em cada fase o nó a ser expandido em seguida é indicado por um marcador (RUSSELL, 2003).

Na busca em largura a solução é encontrada percorrendo-se o menor número de níveis possível. Com relação aos quatro critérios de avaliação dos algoritmos de busca citados anteriormente, a busca em largura é **completa**, pois sempre encontra uma solução se ela existir, e é **ótima** se o custo dos passos forem idênticos, pois sempre encontra a solução mais “rasa”.

Outra avaliação deve ser feita em relação à quantidade de tempo e de memória necessários para completar a busca. Na busca em largura a complexidade de **tempo** e **espaço** é exponencial no tamanho do caminho, pois segundo Russell (2003),

[...] em um espaço de estados hipotético no qual cada estado tem b sucessores. A raiz da árvore de busca gera b nós no primeiro nível, cada um dos quais gera b outros nós, totalizando b^2 no segundo nível. Cada um desses outros nós gera b outros nós, totalizando b^3 no terceiro nível e assim por diante.

Assim, esta estratégia demanda muito tempo e memória, uma vez que todo nó deve ser mantido na memória, pois faz parte do caminho ou é um descendente de um nó do caminho. Por estas razões a busca em largura só é recomendada quando a árvore de busca não é muito profunda.

2.4.1.2. BUSCA DE CUSTO UNIFORME

Como foi visto, a estratégia de busca em largura sempre encontra a solução no estado mais raso, porém esta solução pode não ser a de menor custo. A busca de custo uniforme é uma variação da busca em largura que, em vez de escolher o nó mais raso, sempre expande o nó com custo de caminho mais baixo. Ou seja, ela encontra a solução de menor custo de caminho. Se em toda árvore de busca os custos dos passos tiverem o mesmo valor esta busca será idêntica à busca em largura. Assim como na busca em largura na busca de custo uniforme todo nó gerado deve permanecer em memória.

Um algoritmo proposto pelo cientista da computação Edsger W. Dijkstra em 1959, conhecido como Algoritmo de Dijkstra, calcula o caminho de custo mínimo entre um nó de origem e um nó de destino. Para isso ele utiliza a função de avaliação de custo de caminho:

$$f(n) = g(n)$$

Onde $g(n)$ é o custo do caminho já percorrido partindo da raiz até o nó n .

2.4.1.3. BUSCA EM PROFUNDIDADE

Segundo Luger (2004) na busca em profundidade quando um estado é examinado, todos os seus filhos e os descendentes deles são examinados antes de qualquer um de seus irmãos. Ou seja, é um método que expande o nó no nível mais

profundo no caminho atual da árvore. Quando a busca atinge o nó mais profundo, aquele que não tem mais sucessores, sem encontrar a solução ela regressa ao nó mais raso que ainda não foi explorado, para expandir um novo ramo da árvore. A ordem de expansão dos nós na busca em profundidade é definida como a seguir:

1. Nó raiz.
2. Primeiro nó de profundidade 1.
3. Primeiro nó de profundidade 2.
4. Primeiro nó de profundidade 3.
5. E assim sucessivamente até o nó de maior profundidade.

A figura 2.6 abaixo ilustra esta estratégia de busca.

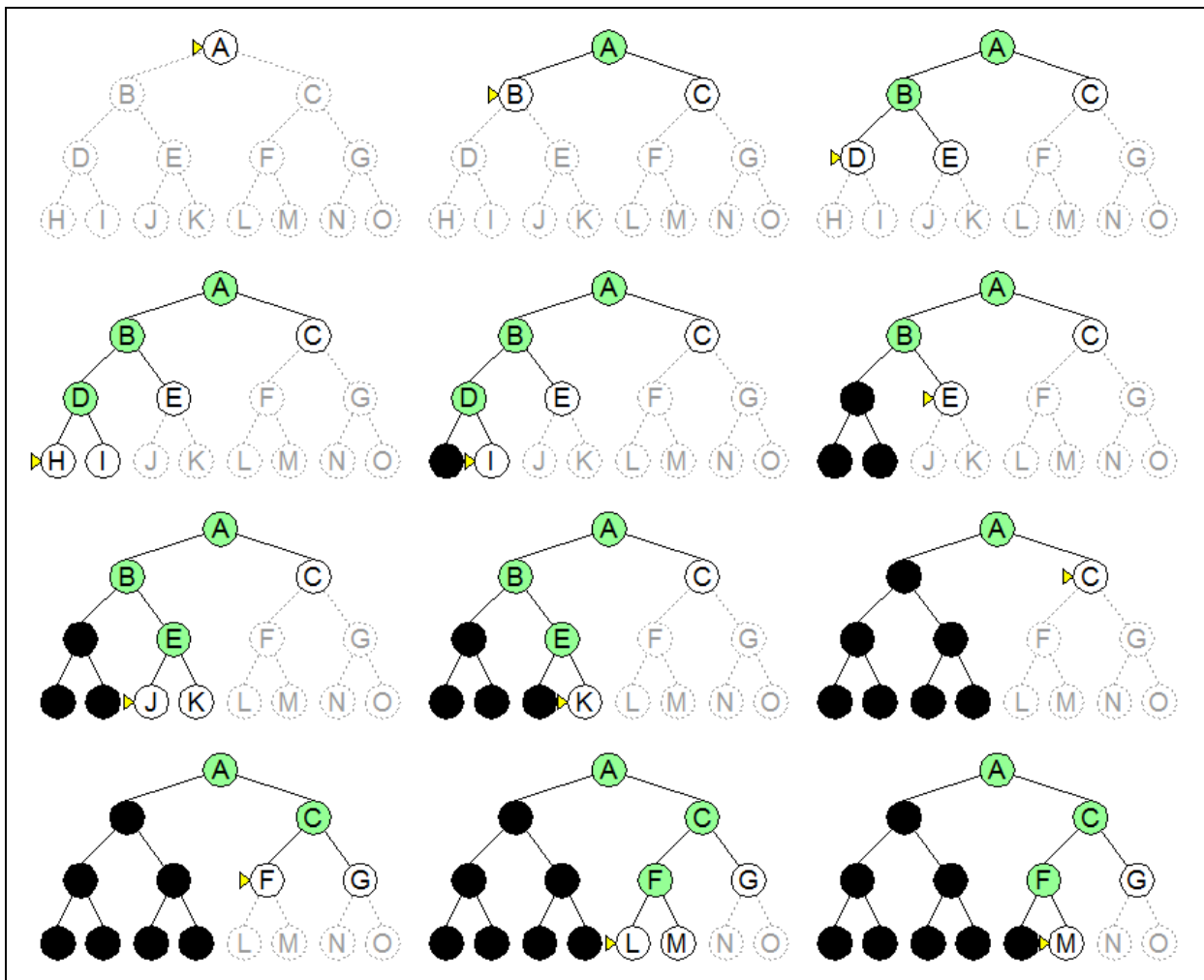


Figura 2.6 – Busca em Profundidade. Os nós que foram expandidos e não têm descendentes na borda podem ser removidos da memória; eles são mostrados em cor preta. Supomos que os nós na profundidade 3 não têm sucessores e que M é o único nó objetivo (RUSSELL, 2003).

A busca em profundidade não é **ótima**, pois uma escolha errada pode levar a uma solução em uma profundidade muito grande, enquanto em outro ramo da árvore de busca existe uma solução mais próxima da raiz. Ela também não é **completa**, pois se o ramo que está sendo explorado tiver caminhos ou laços infinitos e não possuir nenhuma solução a busca não terminará nunca.

A busca em profundidade precisa armazenar apenas um único caminho da raiz até um nó folha, isto é, um nó sem sucessores, além dos nós a serem expandidos neste caminho. Assim ela exige pouca memória, ou seja, sua **complexidade de espaço** é pequena. Porém a **complexidade de tempo** é exponencial no tamanho do caminho, assim como na busca em largura. Esta estratégia deve ser evitada em problemas que geram árvores muito profundas ou caminhos infinitos.

2.4.1.4. BUSCA EM PROFUNDIDADE LIMITADA

A busca em profundidade limitada possui um funcionamento semelhante ao da busca em profundidade, porém nela é definido um limite fixo na profundidade máxima de um caminho na árvore de busca. A busca em profundidade limitada resolve o problema de caminhos infinitos e garante encontrar uma solução se ela existir dentro da profundidade limite. Na situação descrita esta estratégia é **completa**, mas não é **ótima**, uma vez que é difícil definir um bom limite na profundidade máxima de busca.

2.4.1.5. BUSCA DE APROFUNDAMENTO ITERATIVO

É uma estratégia que encontra o melhor limite de profundidade e assim evita o problema de caminhos muito longos ou infinitos da busca em profundidade. Inicialmente ela limita a profundidade em 0 e busca a solução, depois incrementa este limite para 1, para 2, para 3 e assim progressivamente até encontrar a primeira solução.

A estratégia de busca de aprofundamento iterativo combina as vantagens das buscas em largura e a busca em profundidade. Ela é **ótima** e **completa** como na

busca em largura e tem **complexidade de espaço** como na busca em profundidade, ou seja, exige pouca memória. Apesar de a cada nova tentativa ter que recomeçar a busca do primeiro nível a **complexidade de tempo** não é tão alta quanto na busca em largura. Este é o método de busca sem informação mais utilizado em problemas onde o espaço de busca é grande e a profundidade da solução não é conhecida.

2.4.2. BUSCA COM INFORMAÇÃO

Anteriormente foram vistos os métodos de busca sem informação ou busca cega, que encontram soluções simplesmente gerando estados e os testando contra o objetivo, sem qualquer informação que *a priori* auxilie esta exploração. Em algumas situações estas buscas geram um grande número de nós a serem expandidos antes de se obter uma solução. Devido às limitações de tempo e espaço de memória, a utilização destas estratégias pode se tornar impraticável ou ineficiente.

A busca com informação ou busca heurística procura minimizar os custos computacionais empregando, além da definição do problema, conhecimento específico a cerca do problema em questão. Estas informações orientam e limitam a complexidade da busca de uma solução baseada em uma função de avaliação $f(n)$. De acordo com Whitby (2004) o uso da heurística permite que as buscas sejam orientadas até certo ponto pelas *suposições* acerca da natureza do mundo real.

Uma boa heurística permite encontrar boas soluções, mas não garante a obtenção da solução ótima. Segundo Rich (1993) a heurística é uma técnica que melhora a eficiência da busca, possivelmente sacrificando pretensões de completeza. Porém, na maioria das vezes não é necessário encontrar a melhor resposta e sim uma solução próxima da ideal que satisfaça os requisitos do sistema que está realizando a busca.

Um método geral de busca com informação é a **busca pela melhor escolha** (*best-first search*), este método utiliza uma função de avaliação $f(n)$ para definir o próximo nó a ser expandido, escolhendo aquele que tiver a menor estimativa de custo até o estado-objetivo.

[...] como os algoritmos de busca em profundidade e em amplitude, a busca pela melhor escolha usa listas para manter estados: **abertos**, para registrar a fronteira atual da busca, e **fechados**, para armazenar os estados já visitados. Um passo adicional do algoritmo ordena os estados em **abertos** de acordo com alguma estimativa heurística da sua “proximidade” em relação ao objetivo. Assim, cada iteração do laço considera o estado mais “promissor” da lista **abertos** (LUGER, 2004).

Apesar desta estratégia encontrar rapidamente um caminho, nem sempre este será o melhor. Duas abordagens básicas de busca pela melhor escolha são a Busca Gulosa (*Greedy Search*) e o Algoritmo A* (*A* Search*).

2.4.2.1. BUSCA GULOSA PELA MELHOR ESCOLHA

De acordo com Russell (2003) a busca gulosa pela melhor escolha tenta expandir o nó mais próximo a meta, na suposição de que isso provavelmente levará a uma solução rápida. Isto é feito baseado apenas na estimativa feita pela função heurística:

$$f(n) = h(n)$$

Onde $h(n)$ é o custo estimado do caminho para se chegar ao objetivo partindo-se do nó n . Este algoritmo tem custo de busca mínimo, pois não expande nós fora do caminho. A busca gulosa pela melhor escolha é semelhante à busca em profundidade e, como tal, não é ótima e nem completa. Não é ótima porque escolhe o caminho que aparentemente está mais próximo do objetivo, mas esta estimativa é feita através de uma função heurística que normalmente não é precisa. Também não é completa, pois pode entrar em um caminho ou laço infinito e não retornar para avaliar outras opções. As complexidades de tempo e espaço são exponenciais, pois armazena todos os nós expandidos na memória, mas de acordo com a qualidade da heurística elas podem ser bastante minimizadas.

2.4.2.2. BUSCA A*

A estratégia de **busca de custo uniforme**, implementada no algoritmo de Dijkstra, minimiza o custo do caminho, $g(n)$. Ela é ótima e completa, mas é uma busca cega e pode ser ineficiente. Já a **busca gulosa pela melhor escolha** procura

reduzir o custo estimado até o estado-objetivo, $h(n)$. Porém ela não é ótima e nem completa. A busca A^* (pronuncia-se A estrela) combina estas duas estratégias minimizando o custo total do caminho. Segundo Russell (2003) A^* é a forma mais amplamente conhecida da busca pela melhor escolha. Ela avalia nós combinando $g(n)$, o custo para alcançar cada nó, e $h(n)$, o custo para ir do nó até o objetivo:

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

Onde $g(n)$ é custo do caminho do nó inicial até o nó n e $h(n)$ é o custo estimado do caminho entre o nó inicial e o nó objetivo. Assim a função $f(n)$ representa o custo estimado da solução mais barata passando por n e chegando no estado-objetivo. Além disto, a busca A^* também utiliza os conceitos de listas abertas e fechadas. A lista aberta armazena todos os nós que foram gerados, mas ainda não foram examinados, ou seja, que não contém sucessores. Todos os nós da lista aberta são candidatos a expansão, a função heurística é que irá determinar o mais promissor. A lista aberta é mantida como uma fila de prioridades e de acordo com Lester (2005) este é de fato o elemento que consome a maioria do tempo do algoritmo A^* de *pathfinding*. Toda vez você acessa a lista aberta, você precisa achar o quadrado que tem o mais baixo valor de F . Para maior rapidez na inserção e acesso ao nó de menor custo de $f(n)$ a lista aberta é ordenada utilizando uma estrutura de dados denominada *binary heap*, que será explicada com mais detalhes na próxima seção deste referencial teórico. A lista fechada armazena os nós que já foram analisados.

O algoritmo A^* é completo e é ótimo, desde que a função heurística seja admissível, ou seja, desde que a função $h(n)$ não superestime o custo até um estado-objetivo. Conforme Russell (2003) heurísticas admissíveis são otimistas por natureza, pois imaginam que o custo da resolução do problema seja menor do que ele é na verdade. As complexidades de tempo e espaço são exponenciais com o comprimento da solução, mas uma boa heurística diminui significativamente estes custos. O tempo de execução do algoritmo não é o maior problema e sim o consumo excessivo de memória, à medida que a profundidade no espaço de busca aumenta o número de nós que devem ser armazenados cresce rapidamente. Apesar disso o A^* normalmente gera menos nós que outros métodos de busca que são ótimos.

2.4.3. COMPARAÇÃO ENTRE ALGORITMOS DE BUSCA DE CUSTO UNIFORME (DIJKSTRA), GULOSA PELA MELHOR ESCOLHA E A*

Foi implementado um programa computacional que simula a execução de algoritmos que realizam busca de caminho entre dois pontos em um cenário se desviando de obstáculos. Com este *software* é possível comparar o desempenho de três dos algoritmos de busca apresentados: Custo Uniforme (Dijkstra); Gulosa pela Melhor Escolha e Busca A*.

Os algoritmos de Dijkstra e de Busca Gulosa têm como função de avaliação, respectivamente, $f(n) = g(n)$ e $f(n) = h(n)$. O algoritmo de Busca A* utiliza como função de avaliação a soma destas duas, ou seja, $f(n) = g(n) + h(n)$. Assim para a codificação de um *software* que compare estes três algoritmos foi implementado o algoritmo A* e depois com simples modificações foi possível simular os outros dois.

Algoritmos que determinam o melhor trajeto entre dois pontos se desviando de obstáculos são conhecidos como algoritmos de busca de caminho ou *pathfinding*. Entre algoritmos de *pathfinding* a busca A* é a mais utilizada, devido ser completa, ótima e eficiente. Segundo Bourg (2004, tradução minha), no desenvolvimento de jogos, o algoritmo A* é provavelmente um dos mais, senão o mais utilizado algoritmo de *pathfinding*. Ele também tem sido muito usado em sistemas robóticos autônomos e foi descrito pela primeira vez em 1968 por Peter Hart, Nils Nilsson, e Bertram Raphael (ALGORITMO, 2009).

O programa desenvolvido para a comparação das três estratégias de busca foi baseado na implementação descrita em Lester (2005) e foi codificado na linguagem de programação C# (lê-se C *sharp*). Segundo Deitel (2003) o C# é uma linguagem de programação visual dirigida por eventos e totalmente orientada a objetos, na qual os programas são criados usando-se um IDE (*Integrated Development Environment* – Ambiente de Desenvolvimento Integrado).

A implementação do algoritmo A* em C# será descrita mais detalhadamente no capítulo 3 desta dissertação (Implementação), pois, devido suas vantagens, este foi o algoritmo de *pathfinding* utilizado na construção do *software* do projeto

(Aplicação Pathfinding e Árvore de Decisão para Navegação de Sistema Robótico Autônomo). Na apresentação do programa desenvolvido para a comparação entre as três estratégias, serão tratados apenas os aspectos referentes à criação da área de busca do caminho e a definição dos valores de $g(n)$ e $h(n)$ para cada n . Considerando que o propósito desta seção é a execução dos algoritmos, a fim de analisar o desempenho de cada estratégia de busca no que se refere à qualidade do caminho encontrado, o tempo e o consumo de recursos do sistema.

Quando a busca de caminho é realizada em um ambiente contínuo o número de nós gerados é muito grande e isto pode tornar o custo computacional da busca elevado. Normalmente a área de busca é simplificada representando o ambiente através de uma grade. De acordo com Leste (2005) este método particular reduz nossa área de procura a uma ordem simples bidimensional. A figura 2.7 mostra a interface gráfica do *software*, nela pode-se observar que a área de busca foi discretizada em quadrados de 40x40 *pixels*. Assim cada quadrado representa um nó na área de busca. Como o painel gráfico tem 640x480 *pixels*, a área de busca pode ser associada a uma matriz bidimensional de 16x12 posições. Sendo que cada posição pode assumir os estados de livre (passável), obstáculo (não-passável), origem ou destino.

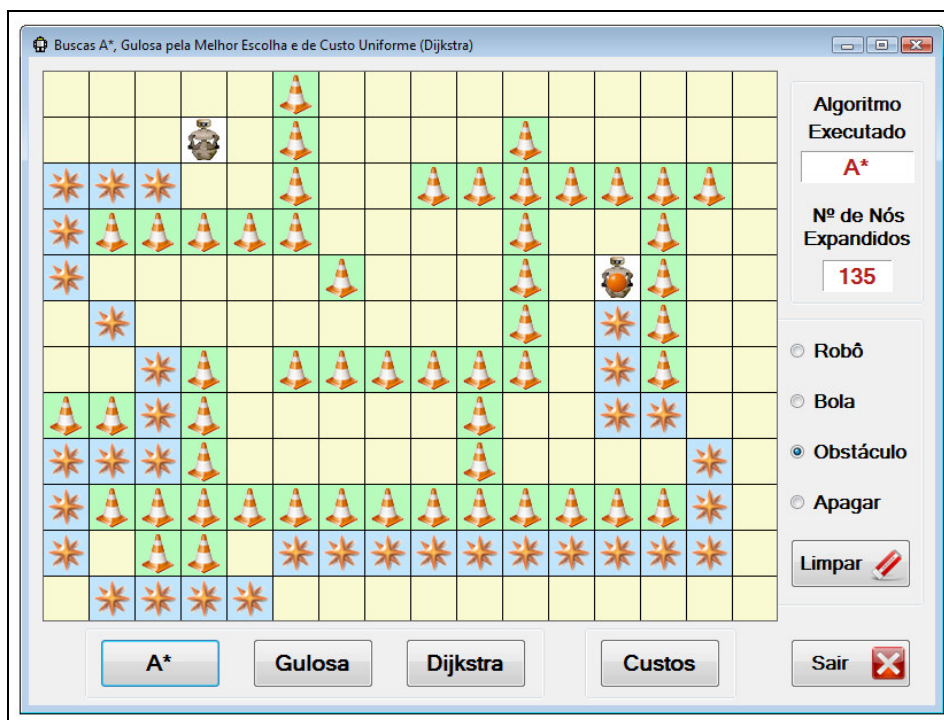


Figura 2.7 – Interface Gráfica.

Na interface cada um dos quatro estados são representados por imagens, assim como outras duas imagens representam o trajeto realizado e a chegada ao objetivo, conforme ilustra a tabela 2.1.

Tabela 2.1 – Estados da Área de Busca.

Estado	Valor	Imagem	Descrição
Livre	0		Área por onde o robô pode se locomover.
Obstáculo	1		Área por onde o robô não pode se locomover.
Início	2		Posição inicial do robô.
Objetivo	3		Posição do objetivo, o local onde está a bola.
Caminho			Caminho realizado pelo robô.
Objetivo OK			Representa que o robô chegou ao objetivo.

Como visto anteriormente a função de avaliação de A^* é $f(n) = g(n) + h(n)$. Assim, a próxima etapa é definir os valores de $g(n)$ e $h(n)$. Para $g(n)$, que é o custo de se mover do ponto inicial até um determinado quadrado da área de busca, foram definidos dois valores:

- $g(n) = 10$, para o deslocamento ortogonal (vertical ou horizontal) de um quadrado para outro.
- $g(n) = 14$, para o deslocamento diagonal de um quadrado para outro.

Esta definição não foi arbitrária, conforme Lester (2005) a distância real para se mover diagonalmente é a raiz quadrada de 2, ou aproximadamente 1,414 vezes o custo de se mover horizontalmente ou verticalmente. Nós usamos 10 e 14 por causa da simplicidade.

A função heurística $h(n)$ é o custo de se mover de um determinado quadrado da área de busca até o quadrado de destino. Existem várias maneiras de se calcular

o valor de $h(n)$, nesta implementação foi utilizado uma métrica chamada de **distância Manhattan**¹ que segundo Lester (2005),

[...] calcula o número total de quadrados movidos horizontalmente e verticalmente para alcançar o quadrado alvo a partir do quadrado atual, ignorando movimento diagonal, e ignorando qualquer obstáculo que pode estar no caminho. Multiplicamos o total então por 10.

A figura 2.8 abaixo mostra, na interface gráfica do *software* de comparação, os cálculos de $g(n)$ e $h(n)$.

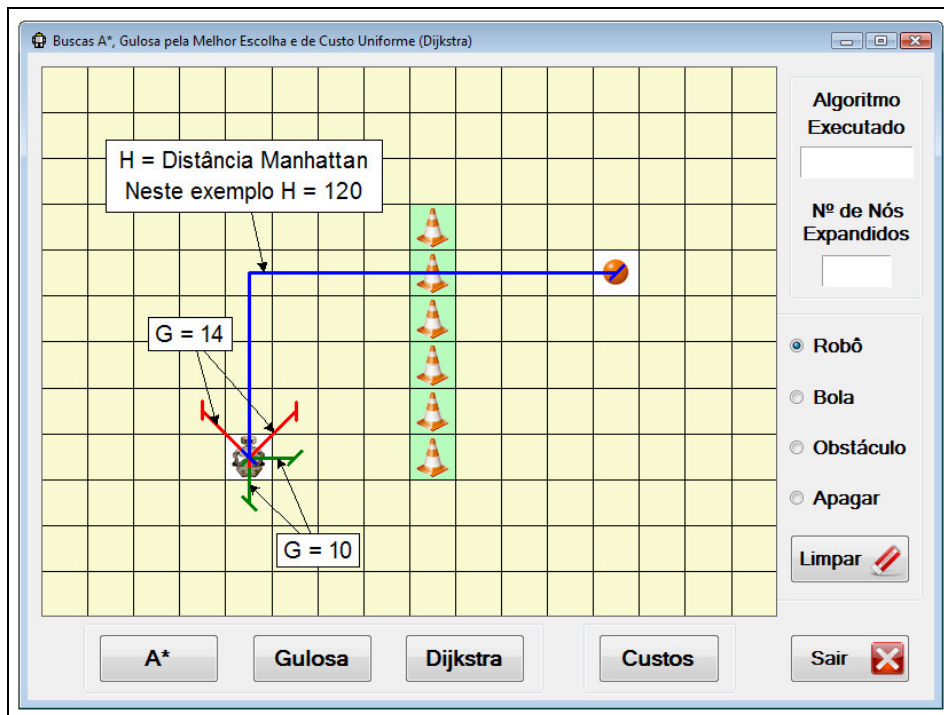


Figura 2.8 – Cálculo dos valores de $h(n)$ e $g(n)$.

Foram criados cenários para analisar os caminhos encontrados por cada estratégia de busca. Inicialmente será feito uma comparação visual para verificar a qualidade do caminho encontrado. A figura 2.9 a seguir mostra o caminho realizado pelo algoritmo A*.

¹ Também conhecida como distância do Taxista, faz alusão à disposição das ruas na ilha de Manhattan (Nova York), formada por quarteirões, por onde não se pode atravessar diagonalmente.

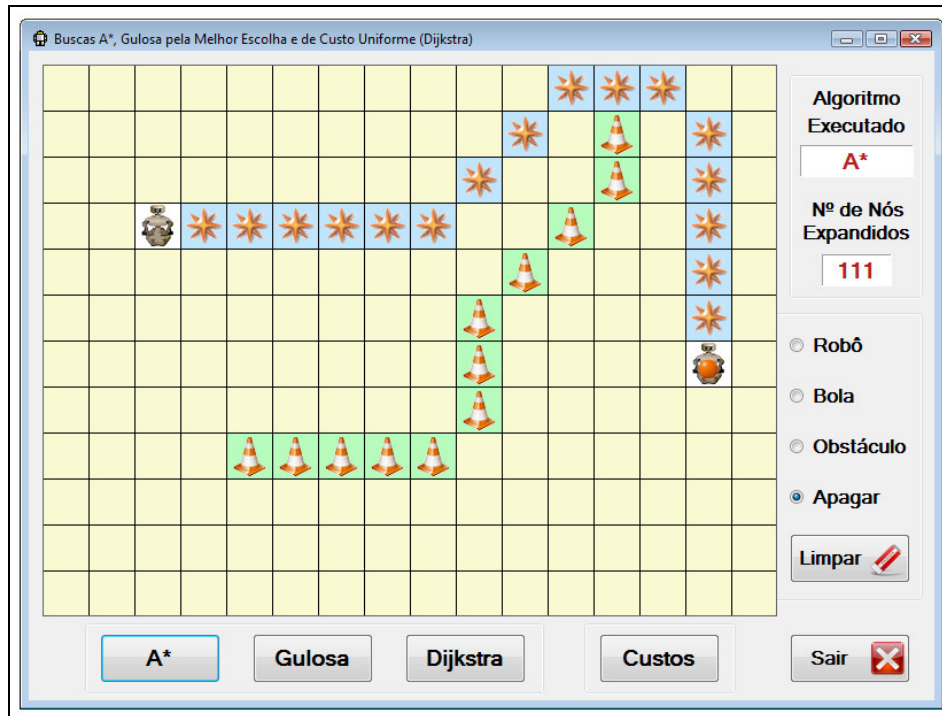


Figura 2.9 – Execução do algoritmo A*.

A figura 2.10 abaixo mostra o caminho resultante da execução do algoritmo de busca Gulosa.

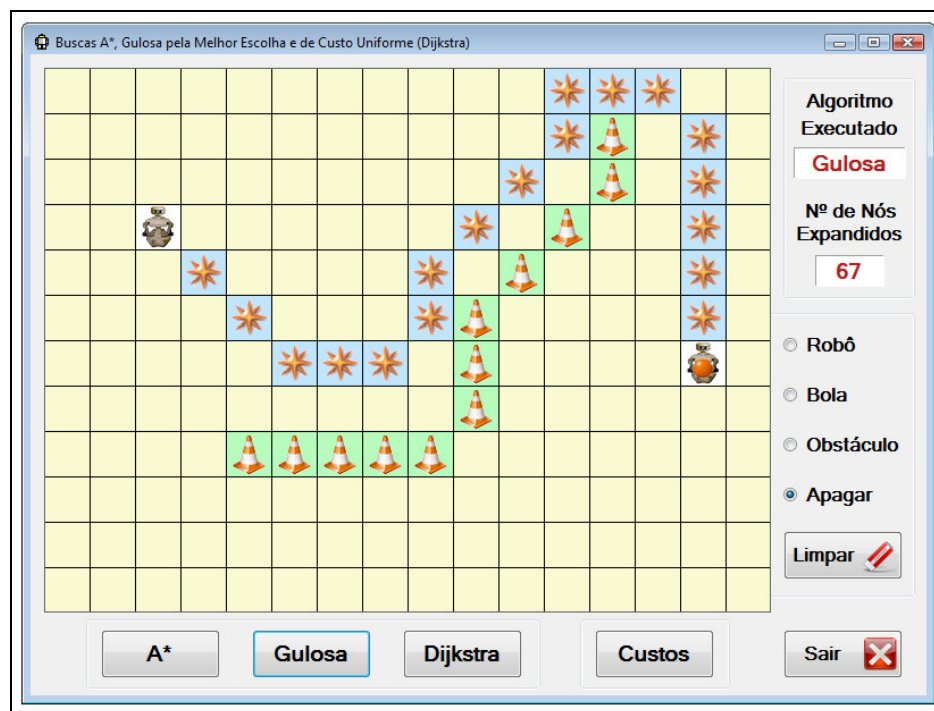


Figura 2.10 – Execução do algoritmo de busca Gulosa.

A figura 2.11 abaixo mostra o caminho encontrado pelo algoritmo de Dijkstra.

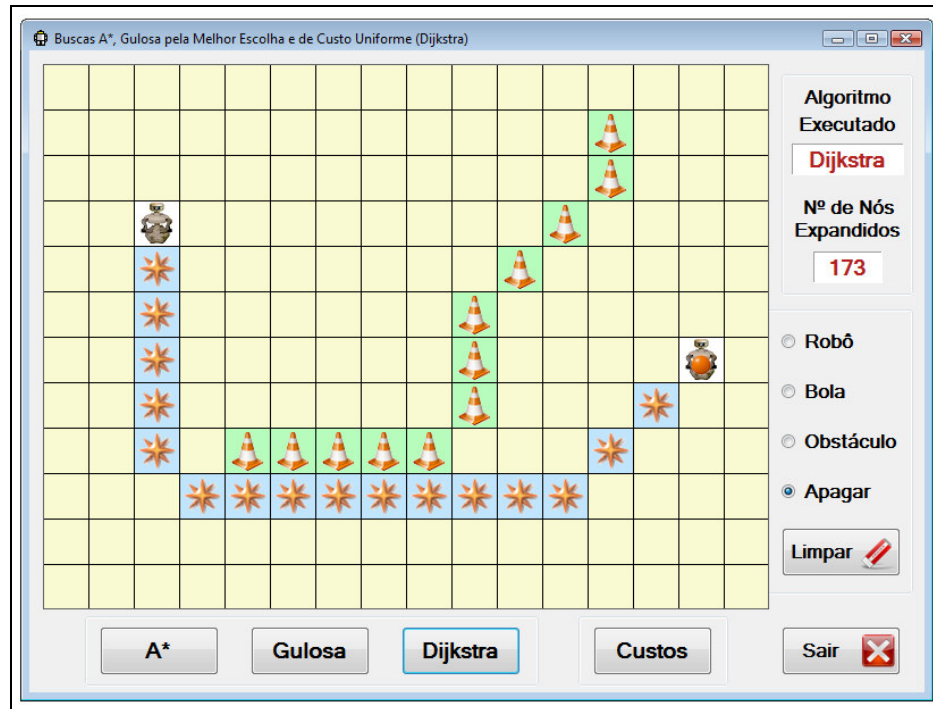


Figura 2.11 – Execução do algoritmo de Dijkstra.

A busca A^* produz uma solução ótima, portanto o caminho mostrado na figura 2.9 é o caminho mais curto entre a origem (robô) e o destino (bola). Analisando o caminho realizado na busca Gulosa pela Melhor Escolha, da figura 2.10, nota-se que no princípio ela seguiu um caminho para baixo e depois mudou de direção e realizou praticamente o mesmo trajeto da busca A^* . Confirmando, como visto anteriormente, que a busca Gulosa não é ótima. A figura 2.11 mostra que o algoritmo de Dijkstra (busca de Custo Uniforme) encontrou um caminho totalmente diferente dos obtidos pelas outras estratégias. Assim como no algoritmo A^* o algoritmo de Dijkstra realiza uma busca ótima, visualmente pode-se perceber que os caminhos definidos pelos dois algoritmos são distintos, porém os custos $g(n)$ dos dois caminhos são os mesmos.

A comparação anterior foi mensurada de forma empírica, já que foi baseada em percepção visual. No que tange à complexidade de espaço, ou seja, o consumo de memória do sistema, o *software* permite conclusões mais consistentes. As figuras 2.12, 2.13 e 2.14 a seguir mostram, em um novo cenário de comparação, os caminhos encontrados na execução de cada estratégia de busca, com os respectivos valores das funções de avaliação.

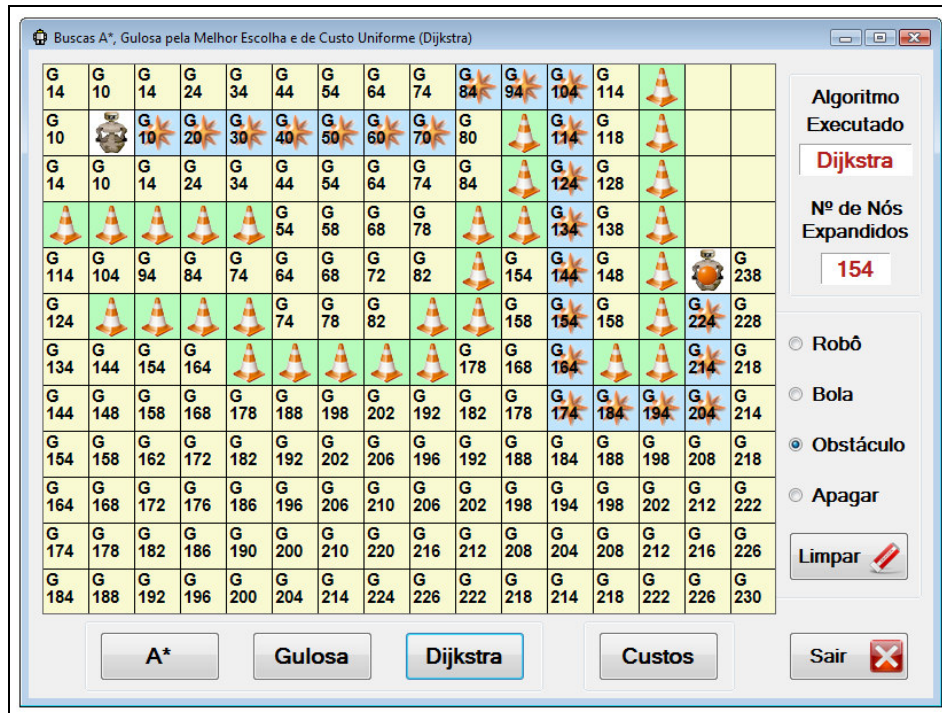


Figura 2.14 – Algoritmo de Dijkstra com os valores de $g(n)$ para cada nó expandido.

Pelas figuras 2.12, 2.13 e 2.14 pode-se notar que o algoritmo de Dijkstra expandiu um maior número de nós (154 nós) que os algoritmos de busca A* (82 nós) e de busca Gulosa (69 nós). Como foi visto anteriormente, no algoritmo de Dijkstra o valor da função de avaliação, $g(n)$, de cada nó expandido deve ser armazenado em memória, assim conclui-se que sua complexidade de espaço é maior que nos outros dois algoritmos. A diferença do número de nós expandidos nas buscas Gulosa e A* não é muito significativa, mas a busca gulosa normalmente expande um menor número de nós. Estas deduções ratificam a afirmação, feita anteriormente, de que uma boa heurística, $h(n)$, pode minimizar em muito a complexidade de espaço.

O programa de comparação possui uma área de busca reduzida e, conseqüentemente, um pequeno **espaço de estados**, que segundo Russell (2003) é o conjunto de todos os estados acessíveis a partir do estado inicial. Nesta situação os três algoritmos são bastante eficientes em relação à complexidade de tempo, ou seja, ao tempo gasto na busca do objetivo. Principalmente com o ganho de velocidade de execução dos algoritmos proporcionado pela utilização da técnica de *binary heap* na ordenação da lista aberta, como será visto na próxima seção deste referencial teórico. Devido a estes fatores, neste *software*, a medida da velocidade

de execução de cada algoritmo não retornou valores que permitissem uma análise satisfatória em relação à complexidade de tempo.

Conforme Lima (2002) a .NET fornece a classe *Process* que é usada para manipular processos do Windows (um processo é um programa em execução). Com a classe *Process* podemos criar novos processos ou gerenciar os que já estão em execução. A classe *Process* tem propriedades de avaliação de desempenho que podem medir o tempo de execução real de uma rotina em um programa, porém, devido aos motivos citados anteriormente, nesta aplicação retornaram valores na ordem de milésimos de segundos e que não permitiram uma comparação consistente.

Os resultados comparativos entre os algoritmos corroboraram com a fundamentação teórica, que aponta a estratégia A* como melhor em qualidade e em desempenho, quando comparada às buscas Gulosa e de Custo Uniforme (Dijkstra). Com esta ferramenta de análise confirmou-se que a busca A* é ótima, pois realmente determina o caminho mais curto entre a origem e o destino, demanda menor quantidade de memória e em reduzido espaço de tempo, ou seja, tem menor custo computacional. Por esta razão este foi o algoritmo de *pathfinding* utilizado na implementação do projeto apresentado nesta dissertação.

2.5. HEAP BINÁRIO EM APLICAÇÃO PATHFINDING

Ordenação é o ato de se colocar os elementos de uma sequência de informações, ou dados, em uma ordem predefinida. O termo técnico em inglês para ordenação é *sorting*, cuja tradução literal é "classificação" (ORDENAÇÃO, 2008). A ordenação facilita a recuperação de informação.

Existem vários métodos de ordenação, entre eles os mais utilizados são: ordenação por inserção (*insertion-sort*); ordenação por seleção (*selection-sort*); ordenação por flutuação (*bubble-sort*); ordenação por mistura (*merge-sort*), ordenação por *heap* (*heap-sort*) e *quick-sort*. Cada um deles com suas características, vantagens e desvantagens. Não se pode dizer que um seja melhor

que o outro, depende da aplicação. No caso específico do algoritmo de busca A^* , é necessário estar constantemente ordenando uma lista da qual é retirado o elemento que tem o mais baixo valor de F . Esta lista é chamada de lista aberta e é mantida como uma fila de prioridades, porém como se deseja obter apenas o menor valor da lista a ordenação não precisa ser completa, ou seja, não é necessário ordenar todos os elementos da lista. Segundo Goodrich (2002),

Uma implementação eficiente de uma fila de prioridade usa uma estrutura de dados chamada *heap*. Esta estrutura de dados permite que realizemos inserções e deleções em tempo logarítmico. A medida pela qual um *heap* obtém este avanço é abandonando a idéia de armazenar os elementos e as chaves em uma sequência e, ao invés disso, armazená-los em uma árvore binária.

Um *heap* binário pode ser máximo ou mínimo, ele é máximo quando o nó pai tem valor maior ou igual ao valor dos nós filhos, neste caso ele coloca no topo do monte (*heap*) o maior valor contido na lista, e é mínimo quando o nó pai tem valor menor ou igual ao valor dos nós filhos, e assim coloca no topo do monte o menor valor da lista. Na implementação do algoritmo A^* foi empregado um *heap* mínimo, pois conforme Lester (2003, tradução minha) nós, na verdade, não nos importamos se o item de número 127 é inferior ao número 128 na lista. Tudo que realmente queremos é que o item de custo F mais baixo esteja facilmente acessível no topo da lista.

A figura 2.15 a seguir mostra a disposição de elementos ordenados em uma árvore por *heap* binário mínimo. O elemento de menor valor (10) está no topo do monte, no nível imediatamente inferior estão os filhos com valores maiores, à esquerda com valor de 30 e à direita com valor de 20. Na segunda camada abaixo do topo estão os filhos dos elementos de valor 30 (34 e 38) e de valor 20 (30 e 24). Pode-se notar que neste *heap* um elemento de valor 24 está no segundo nível enquanto que um elemento de valor 30 está no primeiro nível abaixo do topo. Isto acontece porque em um *heap* binário mínimo cada elemento precisa apenas ser maior ou igual ao valor de seu pai.

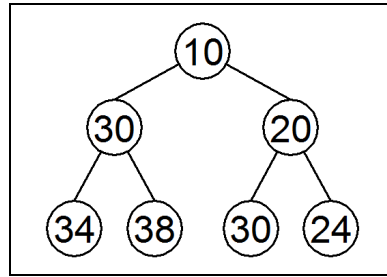


Figura 2.15 – *Heap* binário mínimo (LESTER, 2003).

Normalmente a estrutura utilizada para guardar os elementos de uma lista, como a lista aberta do *pathfinding A**, é o vetor. A figura 2.16 abaixo mostra um *heap* binário como uma árvore completa e sua respectiva representação em um vetor. Cada elemento do vetor armazena o valor de um nó da árvore. A raiz da árvore (topo do *heap*) é armazenada na posição de índice 1 do vetor, não deve ser colocado no índice 0 do vetor, pois os índices serão usados para o cálculo da posição dos filhos. O filho à esquerda é colocado a 2 vezes a posição do pai e o filho à direita é colocado a 2 vezes a posição do pai + 1. Ou seja:

- Raiz da árvore ou o topo do *heap*: posição 1 do vetor;
- Filhos de um nó na posição i : a esquerda ($2 * i$) e a direita ($2 * i + 1$);
- Pai de um nó na posição i : posição $\lfloor i / 2 \rfloor$.

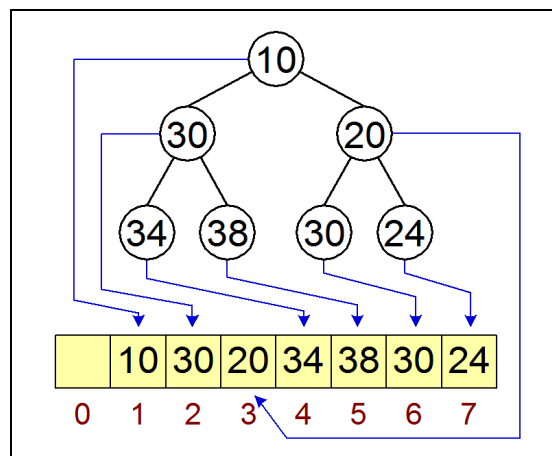


Figura 2.16 – *Heap* em árvore e respectiva representação em vetor (LESTER, 2003).

2.5.1. ADICIONAR UM NOVO ITEM NO *HEAP*

Para se acrescentar um novo item no *heap* este deve ser inserido no final do vetor. Em seguida ele precisa ser comparado com o seu pai, que está no índice que

corresponde ao número de elementos no vetor dividido por 2, é utilizado o piso resultado da divisão, ou seja, apenas o seu quociente inteiro. Se o valor de F do item inserido for menor que o valor do pai estes dois itens devem ser trocados de posição. Agora em uma nova posição no *heap* mais uma vez é necessário buscar a posição do pai do novo item, ou seja, divide-se o índice da sua posição atual por 2. Em seguida precisa ser feita outra comparação e se o valor de F do filho for menor que o valor do pai deve-se trocá-los novamente. Este procedimento é repetido até que o novo item encontre sua posição adequada, ou seja, até que seu valor de F seja maior do que o valor de seu pai ou até que ele atinja o topo do *heap*, que é a posição 1 do vetor.

Por exemplo, foi adicionado um item com valor de F igual a 17 no vetor mostrado na figura 2.16. O processo para reordenação do *heap* é mostrado na figura 2.17 abaixo.

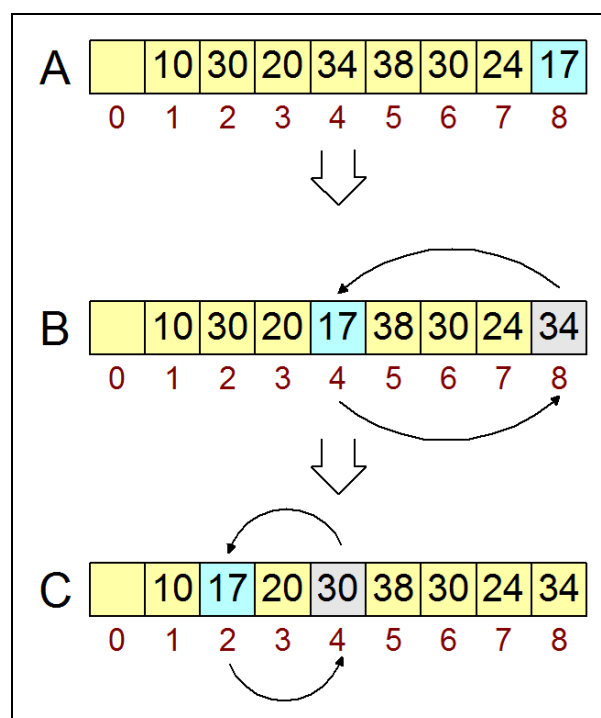


Figura 2.17 – Adicionado um novo item no *heap* (LESTER, 2003).

Inicialmente o novo item foi inserido na posição 8 do vetor, como pode ser visto na figura 2.17A acima. Dividindo este índice por 2 encontra-se o índice do pai que é 4. O valor de F do filho (17) é menor que o do pai (34) e então eles devem ser trocados (figura 2.17B). Na sua nova posição, índice 4, o seu pai é o elemento do índice 2, que também tem um valor de F maior que ele (30) e novamente devem ser

trocados (figura 2.17C). Agora o item adicionado está no índice 2 do vetor e o seu novo pai, no índice 1, tem valor de F menor (10) e então não devem ser trocados. Assim, o *heap* binário foi reordenado e o novo item assumiu a sua posição apropriada.

2.5.2. RETIRAR UM ITEM NO *HEAP*

O algoritmo de *pathfinding* A* remove o item que está no topo do *heap*, aquele que tem o menor valor de F e que está na posição de índice 1 do vetor. Assim esta posição fica vazia, conforme pode ser visto na figura 2.18A abaixo.

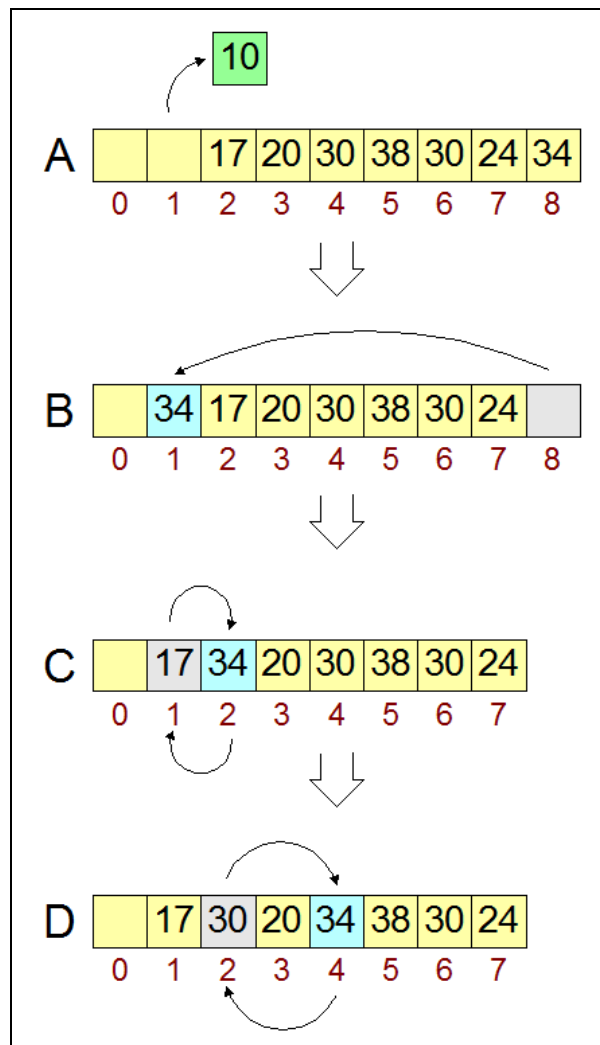


Figura 2.18 – Retirando um item do *heap* (LESTER, 2003).

O procedimento para recompor o *heap* consiste, inicialmente, em transferir o item que está na última posição do vetor para a posição de índice 1 (figura 2.18B).

Em seguida são realizadas as etapas que reordenam o vetor por *heap* binário, primeiro compara-se o item que está no topo do *heap* com os seus dois filhos, os filhos estão localizados nas posições calculadas por 2 vezes e por 2 vezes + 1 o índice do pai. Se o valor do pai for maior que de um dos filhos eles devem ser trocados, senão o vetor fica como está. Na figura 2.18B pode observar que o valor do pai (34) é maior que o do filho que está na posição 2 (índice do pai igual a $1 * 2 = 2$) que tem valor 17 e então eles devem ser trocados, conforme ilustra a figura 2.18C. Mais uma vez deve-se comparar o item que está sendo verificado no *heap*, agora na posição 2, com seus filhos que são os elementos de índice 4 ($2 * 2$) e índice 5 ($2 * 2 + 1$). Na primeira verificação observa-se que o valor do pai (34) é maior que o do filho da posição 4 (30) e então eles também devem ser trocados de posição (figura 2.18D). Novamente deve ser comparado o item corrente, na posição 4, com os seus filhos nas posições 8 ($4 * 2$) e 9 ($4 * 2 + 1$). Porém neste vetor não existem elementos nos índices 8 e 9, assim a reordenação do *heap* é encerrada.

Como pode ser visto na figura 2.18D o vetor está parcialmente ordenado, pois se o objetivo é ordenar os elementos na ordem crescente de valor, pode-se notar, por exemplo, que o valor no elemento de índice 2 (30) é maior que o valor do elemento subsequente de índice 3 (20). Entretanto, como foi colocado anteriormente, para um melhor desempenho do algoritmo de *pathfinding* A^* é importante o acesso rápido e fácil ao item de menor valor de F do vetor e não é necessária a ordenação completa do vetor. A ordenação por *heap* binário atende satisfatoriamente a este requisito, pois o elemento de menor valor do conjunto é disponibilizado na primeira posição do vetor, que corresponde ao topo do *heap*, com poucas iterações da rotina de ordenação. Segundo Lester (2003, tradução minha) na minha experiência o uso de um *heap* binário fará o seu algoritmo de *pathfinding* 2-3 vezes mais rápido na maioria dos casos, e ainda mais em trajetos mais longos.

3

IMPLEMENTAÇÃO DO SIMULADOR

Em que é apresentado a forma como foi construído o simulador. Neste capítulo são descritos a interface gráfica e sua usabilidade, o desenvolvimento do algoritmo que executa a busca A, as técnicas utilizadas na representação gráfica que mostra a exploração do ambiente pelo robô virtual e as rotinas de planejamento de trajetória e tomadas de decisão.*

3.1. INTRODUÇÃO

O simulador apresenta as estratégias desenvolvidas para que um robô virtual realize o mapeamento e a navegação em ambientes previamente conhecidos ou desconhecidos. Estas estratégias têm como base o algoritmo de *pathfinding* A*.

O *software* de simulação foi desenvolvido sobre a plataforma Microsoft Visual Studio 2008 na linguagem de programação Visual C# (pronuncia-se C *sharp*). O C# é uma linguagem totalmente orientada a objetos que une o poder e a flexibilidade das linguagens C e C++ com a produtividade de um IDE (*Integrated Development Environment* – ambiente de desenvolvimento integrado) bastante interativo para a implementação de programas computacionais utilizando recursos gráficos. Segundo Deitel (2003) o C# tem raízes em C, C++ e JAVA, adaptando os melhores recursos de cada linguagem e acrescentando novas capacidades próprias.

O simulador foi desenvolvido em um computador portátil que tem como principais especificações um processador Intel Centrino de 1,6 GHz, 1 GByte de memória RAM e 2 MBytes de memória cache. Gerenciado pelo sistema operacional Microsoft Windows XP.

3.2. A INTERFACE GRÁFICA

A interface gráfica do simulador foi construída com três áreas de visualização e vários componentes para o controle das funcionalidades do *software*. Como pode ser visto na figura 3.1 abaixo.

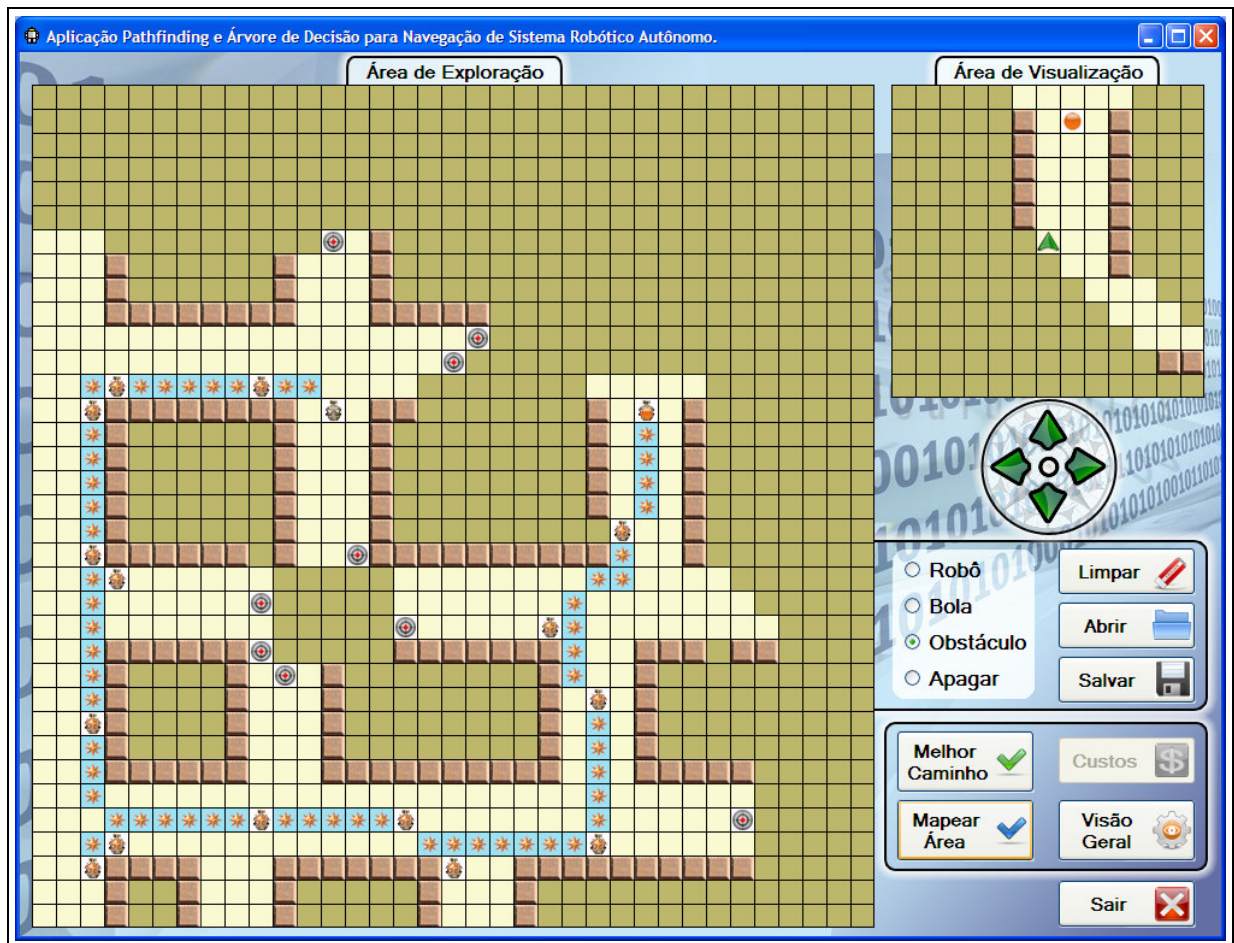


Figura 3.1 – Interface Gráfica.

A seguir será descrita a finalidade de cada componente gráfico da interface.

3.2.1. ANIMAÇÃO DO SIMULADOR

Para fazer uma demonstração gráfica do mapeamento do ambiente realizado pelo robô foram utilizados 3 painéis na interface do sistema.

3.2.1.1. ÁREA DE EXPLORAÇÃO DO ROBÔ

Simula a área por onde o robô pode se locomover na busca do objetivo. Ela é representada por um painel de formato quadrado com dimensões laterais de 700 *pixels*. Este painel é discretizado em uma malha quadrada de 20 *pixels*. Como pode ser visto na figura 3.2 abaixo. A definição do tamanho de cada célula da Área de Exploração corresponde ao espaço ocupado pelo robô virtual. Para a implementação deste painel foi utilizado o componente *PictureBox*, que é um componente próprio para se manipular imagens estáticas (figuras) ou dinâmicas (vídeos) na plataforma .NET.



Figura 3.2 – Imagem parcial da Área de Exploração.

3.2.1.2. ÁREA DE VISUALIZAÇÃO DO ROBÔ

Simula o sensoriamento que o robô faz através da visão computacional, ou seja, a detecção de obstáculos, espaços livres ou o objetivo através de imagens geradas por um sistema mono-ocular (a partir de uma única câmera) ou estéreo (duas câmeras).

Mostra a área que o robô consegue visualizar, ou seja, representa a distância máxima que ele pode "enxergar" a partir do local onde se encontra na Área de Exploração. Esta visualização também pode ser limitada pela proximidade com os perímetros da Área de Exploração ou por obstáculos. A Área de Visualização do robô é representada por um painel quadrado com dimensões laterais de 260 *pixels* e é dividida em áreas menores com células de 20x20 *pixels*. Como pode ser visto na

figura 3.3 abaixo. Para a implementação deste painel também foi utilizado o componente *PictureBox* da plataforma .NET.

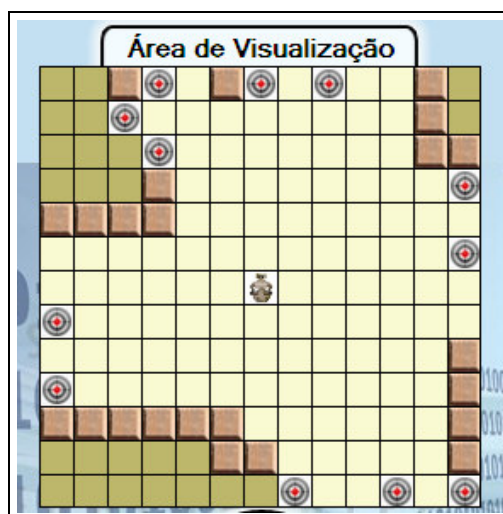


Figura 3.3 – Área de Visualização.

3.2.1.3. DIREÇÃO DE VISUALIZAÇÃO DO ROBÔ

Este painel mostra, através da representação de uma seta, em que direção o robô está visualizando. Uma vez que o robô virtual utilizado no simulador é capaz de girar sobre o próprio eixo sem sair do lugar e assim visualizar 360°. Como pode ser visto na figura 3.4 abaixo. Para a implementação deste painel também foi utilizado o componente *PictureBox*.



Figura 3.4 – Indicação de Direção de Visualização.

3.2.2. FORMAÇÃO DO CENÁRIO

Os controles utilizados para a criação, armazenamento e recuperação de cenários para a simulação foram agrupados em uma região da interface gráfica.

Como pode ser visto na figura 3.5 abaixo. A seguir será descrita a finalidade de cada controle.

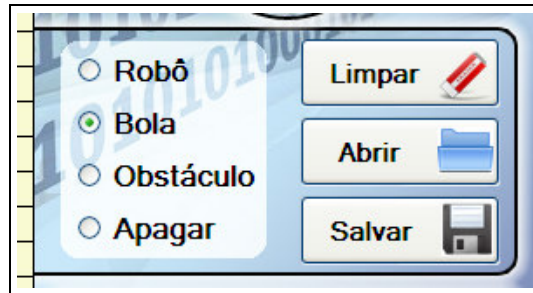


Figura 3.5 – Controles de Formação do Cenário.

3.2.2.1. GRUPO DE SELEÇÃO “ROBÔ” / “BOLA” / “OBSTÁCULO” / “APAGAR”

Estes controles são utilizados para posicionar o robô, a bola (objetivo), obstáculos ou apagar algum destes objetos na Área de Exploração do robô. Para a implementação deste controle foi utilizado o componente *RadioButton*, que é um componente que permite ao usuário fazer uma escolha entre um grupo de opções.

3.2.2.2. BOTÃO “LIMPAR”

Comando utilizado para remover todos os objetos (obstáculos, o robô e o alvo) da Área de Exploração do robô, para iniciar a formação de um novo cenário de simulação. Para a implementação deste controle foi utilizado o componente *Button*, que é um componente que executa uma rotina quando ocorre o evento *click* do *mouse* sobre ele.

3.2.2.3. BOTÃO “ABRIR”

Comando em que o usuário pode recuperar cenários que foram armazenados anteriormente e carregá-los na Área de Exploração do robô. Para a implementação deste controle foi utilizado o componente *Button* do C#.

3.2.2.4. BOTÃO “SALVAR”

Comando em que o usuário pode salvar o cenário que foi formado na Área de Exploração do robô em um arquivo digital. Para a implementação deste controle foi utilizado o componente *Button* do C#.

3.2.3. BUSCA DO OBJETIVO

Os comandos relacionados à realização da busca do objetivo pelo robô estão agrupados em outro conjunto de controles da interface gráfica do sistema. Todos os comandos desta seção foram implementados utilizando o componente *Button* do C#. Como pode ser visto na figura 3.6 abaixo.



Figura 3.6 – Controles de Busca do Objetivo.

3.2.3.1. BOTÃO “MELHOR CAMINHO”

Comando que calcula e mostra o caminho mais curto (*pathfinding*) entre o robô e o alvo (bola), considerando que o robô conhece todo o cenário. Ou seja, o robô tem a informação de onde estão localizados todos os obstáculos e o alvo na Área de Exploração e assim calcula o melhor caminho entre o robô e a bola.

3.2.3.2. BOTÃO “CUSTOS”

Comando que mostra na Área de Exploração do robô os cálculos realizados na busca do caminho mais curto até o alvo. Mostra os custos F, que indicam os valores do deslocamento total para cada célula verificada pelo algoritmo de *pathfinding*. Este comando está associado ao botão “Melhor Caminho”, ou seja,

somente é habilitado quando o simulador está mostrando o melhor caminho na Área de Exploração do robô.

3.2.3.3. BOTÃO “MAPEAR ÁREA”

Comando que inicia a busca do objetivo considerando que o robô não conhece a Área de Exploração. Ou seja, o robô não tem nenhuma informação do cenário em que foi inserido e, sendo assim, para mapear o ambiente ele visualiza 360º da área ao seu redor, gera possíveis pontos de busca, escolhe um destes pontos e se desloca até ele. O robô repete este procedimento até encontrar o objetivo (bola) ou até que na Área de Exploração não exista mais nenhuma célula desconhecida acessível.

3.2.3.4. BOTÃO “VISÃO GERAL”

Este comando mostra na Área de Exploração uma imagem que contém os caminhos percorridos pelo robô durante o mapeamento do ambiente e o melhor caminho entre o robô e a bola. Este botão está associado aos comandos de “Melhor Caminho” e de “Mapear Área”, ou seja, somente é habilitado quando estas duas buscas já foram concluídas.

3.3. ALGORITMO DE *PATHFINDING A**

Para a realização do *pathfinding A**, o melhor caminho entre dois pontos, foi implementada uma classe de nome *Pathfinding.cs* que usou como modelo um algoritmo desenvolvido na linguagem Visual C++, conforme referenciado em Lester (2005).

A linguagem de programação C# não é totalmente compatível, mas possui uma grande familiaridade com a linguagem C++. Esta característica facilitou portar o algoritmo escrito em Visual C++ para a implementação em C#.

A grande maioria dos algoritmos de *pathfinding* A* são desenvolvidos para utilização em jogos de computadores, porém para a aplicação em um sistema robótico autônomo devem ser observadas algumas particularidades. O programa de *pathfinding* implementado atende a requisitos específicos para este simulador, os quais serão destacados na apresentação das rotinas mais relevantes da classe *Pathfinding.cs*, que será feita a seguir.

3.3.1. MÉTODO CONSTRUTOR (*PATHFINDING*)

Diferente do código modelo, na declaração das variáveis globais os *arrays* (vetores e matrizes) são declarados, mas não são dimensionados. Como pode ser visto em parte do código demonstrada no apêndice A.1.

O número de elementos de cada um dos *arrays* é definido neste método, ele recebe como parâmetros as dimensões da área onde será realizado o *pathfinding*, o tamanho das células em que esta área será dividida e o número de robôs que irão procurar o melhor caminho, no caso desta aplicação, apenas um robô. Dividindo-se o tamanho das laterais da Área de Exploração pelo tamanho da célula pode-se dimensionar os *arrays*. Este procedimento torna a classe mais flexível, permitindo que o desenvolvedor defina estes parâmetros na aplicação. No apêndice A.2 são mostrados fragmentos do método *Pathfinding*.

3.3.2. MÉTODO *FINDPATH*

O método que realmente calcula o melhor caminho é o *FindPath*, ele recebe como parâmetros a posição onde está o robô e a posição onde está o alvo (bola). Inicialmente ele faz uma verificação rápida porque em duas circunstâncias nenhum caminho precisa ser gerado. A primeira, quando a posição inicial e o objetivo estão na mesma posição, e a segunda, quando a posição de objetivo é um obstáculo (*unwalkable*), nestes casos o método retorna que o caminho não existe.

Em seguida o código do método *FindPath* acrescenta a posição inicial na lista aberta, que é mantida como um *binary heap*, e entra em um laço de repetição que é

executado até que seja encontrado um caminho ou que este seja considerado inexistente.

Dentro do laço de repetição é retirado o primeiro item da lista aberta e o *heap* binário é reordenado, assim o item com menor custo de F é levado ao topo da lista aberta. Como mostra o apêndice A.3. Depois de reordenado a lista aberta o método *FindPath* verifica as posições adjacentes (os filhos) à posição corrente. Antes de adicionar um “filho” à lista aberta o algoritmo confere se ele não está fora da Área de Exploração, se não está na lista fechada (itens da lista fechada já foram considerados e agora podem ser ignorados) e se não é um obstáculo.

Constatado que a posição adjacente é uma área livre, o código faz uma última verificação para evitar que o robô atravesse o canto de um obstáculo no deslocamento do ponto atual para esta área. Esta é uma situação que não é tratada pela maioria dos algoritmos de *pathfinding* estudados. Como eles normalmente são desenvolvidos para jogos de computadores, que tem uma movimentação dinâmica, este problema não é notado pelo jogador. Mas é um requisito imprescindível para o *pathfinding* que irá determinar a navegação de um sistema robótico autônomo, pois o seu não cumprimento significa a colisão do robô com o canto do obstáculo. No apêndice A.4 é mostrado o trecho do método *Findpath* que realiza esta tarefa. A figura 3.7 abaixo ilustra como seria a movimentação do robô sem o código do apêndice A.4, ela também destaca quais seriam os pontos de colisão. A figura 3.8 mostra o caminho realizado após este tratamento.

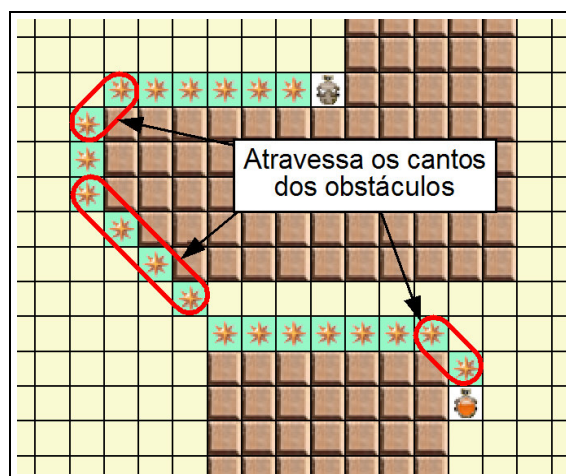


Figura 3.7 – O robô atravessa os cantos dos obstáculos.

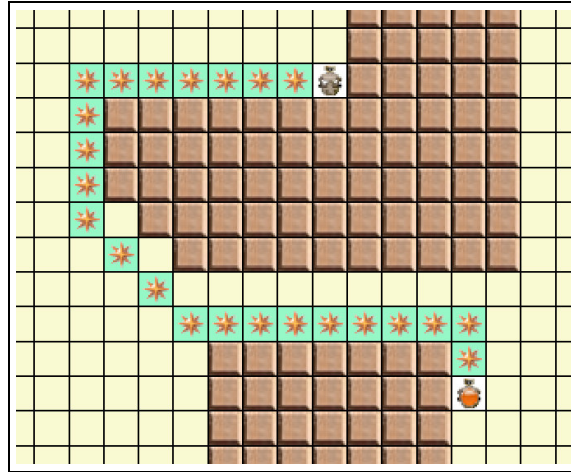


Figura 3.8 – O robô não atravessa os cantos dos obstáculos.

Uma vez constatado que a posição adjacente à posição corrente é uma área livre por onde o robô pode se deslocar, o método testa se esta posição já está na lista aberta. Se não está ele coloca esta posição na lista, calcula os custos F, G e H (veja o apêndice A.5) e reordena a lista aberta por *heap* binário.

Porém se a posição adjacente já está na lista aberta, o método precisa verificar se o caminho que está sendo calculado é melhor do que aquele que colocou esta posição na lista anteriormente. Se for melhor, o código modifica o pai da posição, calcula os custos F, G e H do novo caminho possível e reordena a lista aberta.

No método *FindPath* o código que procura o melhor caminho é executado dentro de uma estrutura de repetição. Existem apenas duas situações capazes de interromper este *loop*: a primeira, quando a lista aberta estiver vazia, o que representa que não existe um caminho possível até o objetivo, e a segunda, quando o objetivo é adicionado à lista aberta, o que indica que o melhor caminho foi encontrado. Como pode ser visto no apêndice A.6.

Uma vez encontrado o melhor caminho a parte final deste método armazena as informações a respeito deste caminho. O algoritmo retrocede do objetivo para a posição inicial verificando cada célula pai e calculando o comprimento do percurso. A informação do tamanho do caminho é utilizada para dimensionar uma matriz que irá armazenar as coordenadas X e Y ao longo deste caminho de forma ordenada, do primeiro passo para o último.

3.3.3. MÉTODO *LOADWALKABILITY*

Este método recebe a informação do estado de cada posição da Área de Exploração. Este estado pode ser definido como área livre (*walkable* = 0) ou obstáculo (*unwalkable* = 1). Como pode ser visto no apêndice A.7.

3.3.4. MÉTODOS DE LEITURA DE DADOS

Na classe original do algoritmo de *pathfinding*, escrita em Visual C++, foram implementados métodos que retornam as informações essenciais referentes ao caminho encontrado, basicamente as coordenadas X e Y ao longo do caminho. Porém neste desenvolvimento também foram implementados métodos que retornam os custos F, G e H, o comprimento do percurso e as coordenadas X e Y de todas as posições avaliadas durante o *pathfinding*. Eles possibilitam que estes dados também estejam disponíveis para aplicações que necessitem destas informações, como será visto posteriormente na explicação do desenvolvimento do simulador. Os métodos de leitura de dados são:

3.3.4.1. MÉTODO *READPATHLENGTH*

Método que retorna o comprimento do melhor caminho. Veja apêndice A.8.

3.3.4.2. MÉTODO *READPATHX*

Método que retorna as coordenadas X do caminho e faz a conversão destas para as coordenadas da Área de Exploração (em *pixels*). Veja apêndice A.9.

3.3.4.3. MÉTODO *READPATHY*

Método que retorna as coordenadas Y do caminho e faz a conversão destas para as coordenadas da Área de Exploração (em *pixels*). Veja apêndice A.10.

3.3.4.4. MÉTODO *READFCOST*

Método que retorna os valores de custo F calculados durante o *pathfinding*. Veja apêndice A.11.

3.3.4.5. MÉTODO *READGCOST*

Método que retorna os valores de custo G calculados durante o *pathfinding*. Veja apêndice A.12.

3.3.4.6. MÉTODO *READHCOST*

Método que retorna os valores de custo H calculados durante o *pathfinding*. Veja apêndice A.13.

3.3.4.7. MÉTODO *READOPENX*

Método que retorna as coordenadas X de todas as posições avaliadas durante o *pathfinding*. Veja apêndice A.14.

3.3.4.8. MÉTODO *READOPENY*

Método que retorna as coordenadas Y de todas as posições avaliadas durante o *pathfinding*. Veja apêndice A.15.

O diagrama de classe da classe *PathFinding.cs* é apresentado no apêndice A.16 deste trabalho.

3.4. IMPLEMENTAÇÃO DO SIMULADOR GRÁFICO

A operação do simulador foi baseada em eventos gerados pelas ações do usuário sobre os diversos componentes da interface gráfica. Para o desenvolvimento da representação visual do simulador foi utilizado a biblioteca GDI+ (*Graphics Device Interface*) da arquitetura .NET, que permite a manipulação de objetos gráficos no C# através de classes contidas no *namespace* System.Drawing.

Na implementação do programa inicialmente são instanciados objetos, declaradas variáveis, vetores e matrizes globais, objetos gráficos e carregado imagens que serão utilizadas durante a execução.

A seguir serão apresentadas as principais funcionalidades do simulador e são destacados os detalhes mais relevantes da implementação.

3.4.1. FORMAÇÃO DO CENÁRIO

Antes de se iniciar a simulação da navegação do robô em um ambiente conhecido ou desconhecido, é necessário construir o cenário onde o robô e o objetivo (bola) serão inseridos. O aspecto visual do cenário de simulação é montado na Área de Exploração do robô, representada no simulador por um componente *PictureBox* de nome *picImagem*, e a informação do estado da posição é armazenada em uma matriz bidimensional de nome *mtzCenario*.

3.4.1.1. INSERÇÃO DE OBJETOS NA ÁREA DE EXPLORAÇÃO

Inicialmente todas as células da Área de Exploração estão no estado de “livre”, ou seja, são áreas por onde o robô pode se deslocar. Através de um conjunto de botões de seleção (*RadioButtons*) o usuário escolhe qual objeto ele deseja posicionar na Área de Exploração. As opções são o robô, a bola, obstáculos ou apagar algum destes objetos.

Selecionado uma das opções o usuário posiciona o objeto clicando ou clicando e arrastando o cursor do *mouse* sobre a Área de Exploração. Estas

operações vão disparar, respectivamente, os eventos *MouseClicked* ou *MouseMove* do *PictureBox picImagem*. Qualquer que seja o evento disparado será executada uma mesma rotina, que realiza basicamente os seguintes procedimentos:

- Capturar as coordenadas X e Y de onde aconteceu o evento no *PictureBox picImagem*;
- Converter estas coordenadas em índices correspondentes na matriz que representa a Área de Exploração do robô (*mtzCenario*);
- Armazenar nesta posição da matriz o valor que representa o objeto;
- Atualizar a imagem no *PictureBox picImagem*, mostrando a figura que representa o objeto que foi inserido.

No apêndice A.17 são mostrados trechos do código que é executado quando ocorrem os eventos *MouseClicked* ou *MouseMove* na Área de Exploração.

Na formação do cenário cada célula da matriz *mtzCenario* pode assumir o valor de “livre” (0), “obstaculo” (1), “inicio” (2) ou “objetivo” (3).

A atualização da imagem no *PictureBox* da Área de Exploração é feita através da instrução *picImagem.Invalidate(...)*. O método *Invalidate* faz com que o *PictureBox picImagem* seja repintado através do seu evento *Paint*, as partes principais do código deste evento serão explicadas posteriormente.

3.4.1.2. LIMPAR A ÁREA DE EXPLORAÇÃO

O código que é executado no evento *Click* do botão “Limpar” (*btnLimpar*) remove todos os objetos (obstáculos, o robô e o alvo) da Área de Exploração e da Área de Visualização do robô, limpa as variáveis, as matrizes e ajusta todos os controles para iniciar a formação de um novo cenário de simulação. No apêndice A.18 são mostrados trechos do código que limpa a Área de Exploração.

3.4.1.3. RECUPERAR UM CENÁRIO

O código que é executado no evento *Click* do botão “Abrir” (*btnAbrir*) recupera um cenário armazenado anteriormente e o recarrega na Área de Exploração. Para esta finalidade foi utilizado o componente *OpenFileDialog* do C#, que exibe uma caixa de diálogo que retorna o caminho e o nome do arquivo selecionado pelo usuário. Conforme pode ser visto na figura 3.9 abaixo.

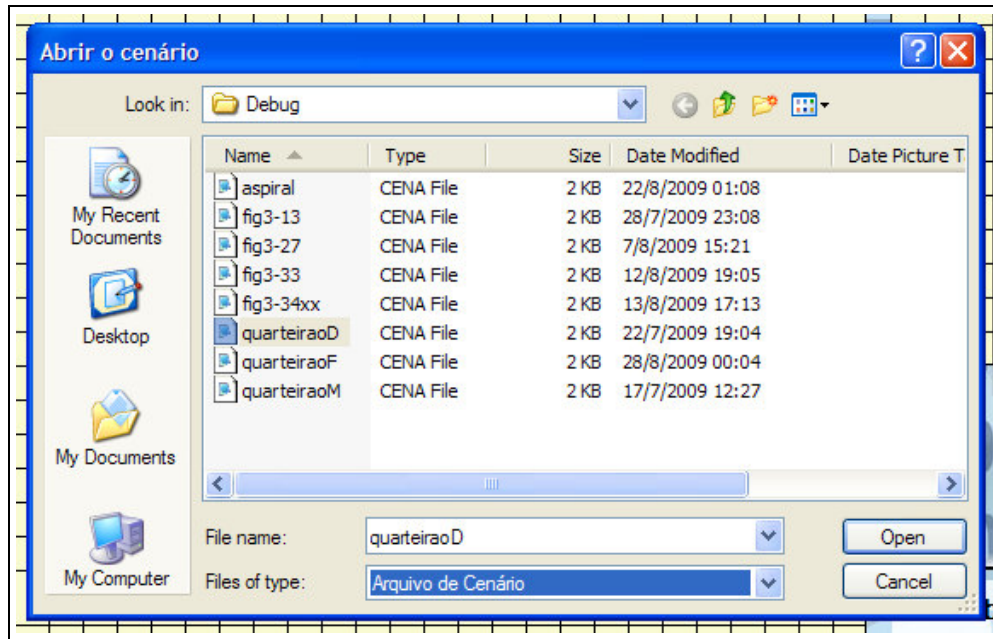


Figura 3.9 – Recupera um cenário armazenado.

Para abrir e ler o arquivo foi utilizada a classe *FileStream* passando três argumentos: o nome do arquivo, a operação (*Open*) e o tipo de acesso (*Read*). Os valores lidos do arquivo são atribuídos à matriz *mtzCenario*, que representa a Área de Exploração do robô, às variáveis *inicioX* e *inicioY*, que indicam as coordenadas da posição onde está o robô, e às variáveis *objetivoX* e *objetivoY*, que indicam as coordenadas da posição onde está a bola. Finalmente o código atualiza a representação gráfica do simulador chamando os eventos *Paint* da Área de Exploração e da Área de Visualização do robô. No apêndice A.19 são mostrados partes do código que recupera um cenário armazenado.

3.4.1.4. SALVAR UM CENÁRIO

O código que é executado no evento *Click* do botão “Salvar” (*btnSalvar*) armazena o cenário que foi formado na Área de Exploração em um arquivo digital. Para esta finalidade foi utilizado o componente *SaveFileDialog* do C#, que exibe uma caixa de diálogo que retorna o caminho e o nome do arquivo escolhido pelo usuário. Conforme pode ser visto na figura 3.10 abaixo.

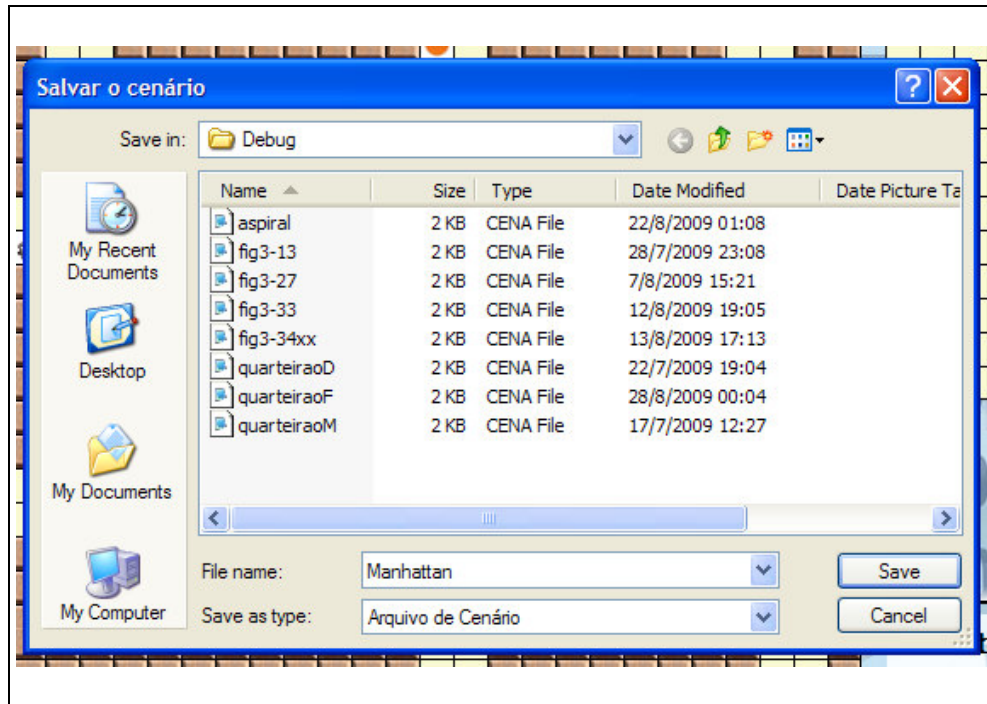


Figura 3.10 – Armazena um cenário criado na Área de Exploração.

Para criar e salvar o arquivo foi utilizada a classe *FileStream* passando três argumentos: o nome do arquivo, a operação (*Create*) e o tipo de acesso (*Write*). Os valores lidos da matriz *mtzCenario*, que representa a Área de Exploração do robô, das variáveis *inicioX* e *inicioY*, que indicam as coordenadas da posição onde está o robô, e das variáveis *objetivoX* e *objetivoY*, que indicam as coordenadas da posição onde está a bola, são escritas no arquivo. Finalmente o código atualiza a representação gráfica do simulador chamando os eventos *Paint* da Área de Exploração e da Área de Visualização do robô. No apêndice A.20 são mostrados partes do código que armazena um cenário.

3.4.2. BUSCA DO OBJETIVO

Nesta fase do projeto são implementados os módulos que executam as estratégias desenvolvidas para que um robô virtual realize o mapeamento e a navegação em ambientes previamente conhecidos ou desconhecidos. Estas estratégias foram elaboradas baseadas no algoritmo de *pathfinding* A* descrito anteriormente.

No desenvolvimento do simulador o algoritmo de *pathfinding* A* será utilizado em três situações distintas:

1. Na navegação do robô em um ambiente conhecido, ou seja, quando o robô tem um conhecimento prévio da sua localização, a posição da bola e dos obstáculos na Área de Exploração. Neste caso o algoritmo de *pathfinding* A* é utilizado para determinar o melhor caminho entre o robô e a bola.
2. Uma segunda situação é quando o robô está explorando um ambiente desconhecido e precisa fazer o mapeamento da Área de Exploração. À medida que o robô visualiza a região ao seu redor o programa gera possíveis pontos para onde o robô pode se deslocar para continuar a busca do alvo (bola). Neste caso o algoritmo de *pathfinding* A* é utilizado para a tomada de decisão sobre qual o próximo ponto de busca para onde o robô deve se locomover.
3. O terceiro emprego do algoritmo de *pathfinding* A* não está relacionado à navegação do robô e sim à sua visualização. Durante o mapeamento da Área de Exploração o robô visualiza o ambiente ao seu redor, quando ele “enxerga” uma área livre ou a bola, o algoritmo de *pathfinding* é utilizado para confirmar se esta região visualizada está realmente acessível ao robô. Em uma situação específica, áreas livres ou a bola são visualizadas, mas não estão acessíveis. Na explicação da parte do código que simula a visão computacional do robô será abordada esta situação com mais detalhes.

3.4.2.1. NAVEGAÇÃO PELO MELHOR CAMINHO

O botão “Melhor Caminho” (*btnMelhor*) dispara a execução de um código que simula a navegação do robô em um ambiente conhecido. Ele mostra na Área de Exploração o melhor caminho entre o robô e a bola. Como pode ser visto na figura 3.11 abaixo.



Figura 3.11 – Melhor caminho entre o robô e a bola.

Inicialmente o programa instancia um objeto da classe *Pathfinding* de nome *PathRobo*, passando como parâmetros para o método construtor a largura e a altura da Área de Exploração, o tamanho de cada célula em que ela é subdividida e o número de robôs no ambiente. Em seguida é verificado se o robô e a bola estão inseridos no ambiente, se estiverem o código transfere as informações sobre o ambiente, que estão na matriz *mtzCenario*, para o objeto *PathRobo* através do método *LoadWalkability*. Feito isto o programa calcula o melhor caminho entre o robô e a bola através do método *FindPath*, passando como parâmetros o número de robôs no ambiente (no caso apenas um), as coordenadas da posição do robô na Área de Exploração (*inicioX* e *inicioY*) e as coordenadas da posição da bola (*objetivoX* e *objetivoY*).

O método *FindPath* retorna um valor inteiro que é atribuído à variável *pathMelhor* e que pode assumir dois valores: 1 (encontrado), que indica que foi encontrado o melhor caminho, ou o valor 2, que indica que não existe um caminho entre o robô e o seu alvo (bola). Se o melhor caminho foi encontrado o próximo

passo do código é ler qual é este caminho, o apêndice A.21 demonstra partes do código que realizam as funções descritas acima.

Outro botão na interface gráfica do simulador, de nome “Custos” (*btnCustos*), mostra os custos F de cada célula da Área de Exploração que foi verificada pelo algoritmo de *pathfinding* no cálculo do melhor caminho. Posteriormente será explicado o funcionamento das partes relevantes do código deste botão, mas os dados que serão utilizados são lidos neste módulo do programa. Como pode ser visto no apêndice A.22.

Uma vez capturados os dados do melhor caminho o simulador precisa fazer a representação gráfica deste caminho, ela é feita através do evento *Paint* do *PictureBox* da Área de Exploração (*picImagem*), as partes principais do código deste evento serão explicadas posteriormente.

Da primeira vez que o melhor caminho é apresentado na tela, o simulador faz uma animação que mostra passo a passo o caminho percorrido pelo robô até a bola e emite uma indicação sonora no instante da chegada ao objetivo. Esta animação é realizada dentro do evento *Tick* de um componente *Timer* de nome *tmrMelhorCaminho*. O apêndice A.23 mostra o código que executa estes procedimentos.

A figura do botão “Melhor Caminho” é alterada para indicar que, para o cenário corrente, já foi realizado o cálculo do melhor caminho, conforme pode ser visto nas figuras 3.12 e 3.13 abaixo. Enquanto o cenário não for modificado, as próximas vezes em que o usuário clicar no botão “Melhor Caminho” será apresentado apenas a imagem final e não mais a animação.



Figura 3.12 – Botão antes de executar o Melhor Caminho.

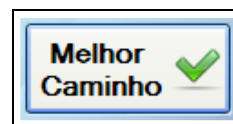


Figura 3.13 – Botão depois de executar o Melhor Caminho.

3.4.2.2. MOSTRA OS CUSTOS F DO MELHOR CAMINHO

O botão “Custos” (*btnCustos*) mostra na Área de Exploração do robô os custos F, que indicam os valores do deslocamento total para cada célula verificada pelo algoritmo de *pathfinding*. Este comando está associado ao botão “Melhor Caminho”, ou seja, é habilitado somente quando o melhor caminho está sendo mostrado na Área de Exploração, como pode ser visto na figura 3.14 abaixo. A representação gráfica destes custos é feita através do evento *Paint* do *PictureBox* da Área de Exploração, as partes principais do código deste evento serão explicadas posteriormente.

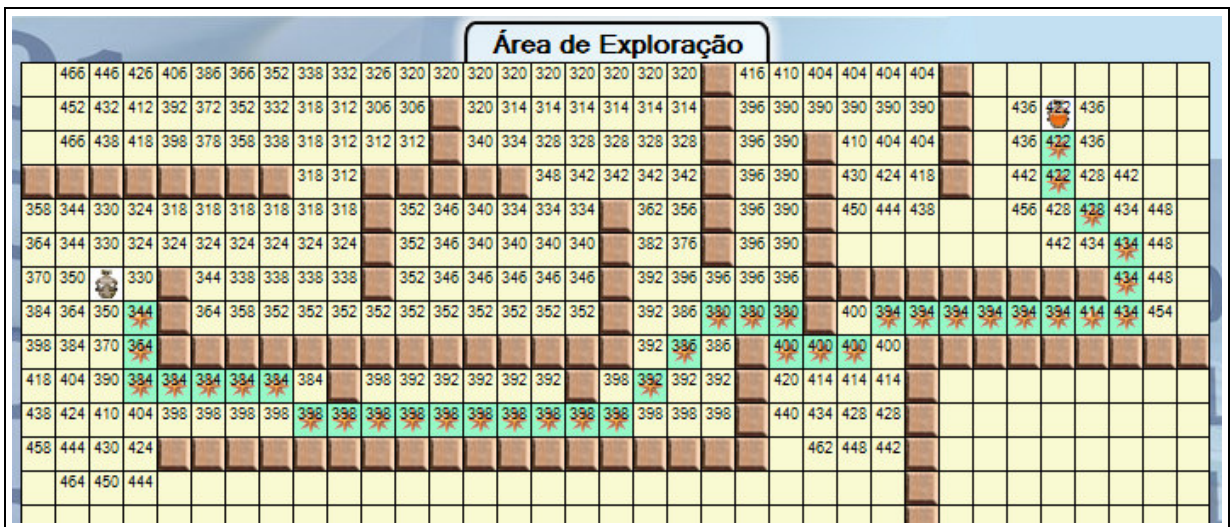


Figura 3.14 – Custos F do *pathfinding* do Melhor Caminho.

3.4.2.3. MAPEAMENTO DO AMBIENTE

O botão “Mapear Área” (*btnMapear*) inicia a simulação do mapeamento de um ambiente desconhecido para a navegação autônoma de um robô, baseado na utilização de visão computacional para a detecção de obstáculos, espaços livres e o objetivo (bola).

A realização desta tarefa foi dividida em 5 módulos, cada um com uma função específica. Os módulos são:

1. Evento *Click* do botão “Mapear Área” (*btnMapear*) – Faz a verificação inicial da presença do robô e da bola no ambiente, interagindo com o usuário através de caixas de mensagens (*MessageBox*), e inicia a simulação.

2. Função *ProcuraAlvo* – Inicia a representação gráfica e ajusta as matrizes e variáveis que serão utilizadas no mapeamento do ambiente.
3. Função *Visualizar* – Simula o sistema de visão computacional do robô, detectando no ambiente os obstáculos, espaços livres e o alvo (bola).
4. Função *PontosBusca* – Baseado na região visualizada pelo robô, gera possíveis pontos para onde o robô deve se locomover na busca do objetivo.
5. Função *ProximoPonto* – Define entre os possíveis pontos de busca para qual o robô deve se deslocar para realizar uma nova visualização do ambiente. Neste módulo também são armazenados os caminhos percorridos pelo robô durante o mapeamento do ambiente e informado o seu resultado final, ou seja, se o alvo foi encontrado ou não.

Durante a execução de todos estes módulos o programa cria uma animação gráfica mostrando, na Área de Exploração e na Área de Visualização, o progresso do mapeamento do ambiente. Esta animação é apresentada somente na primeira vez que o robô faz o mapeamento de um determinado ambiente. Enquanto o cenário não for alterado, nas próximas vezes em que o usuário clicar no botão “Mapear Área” será apresentado apenas a imagem final do mapeamento e não mais a animação, como pode ser visto na figura 3.15 abaixo.

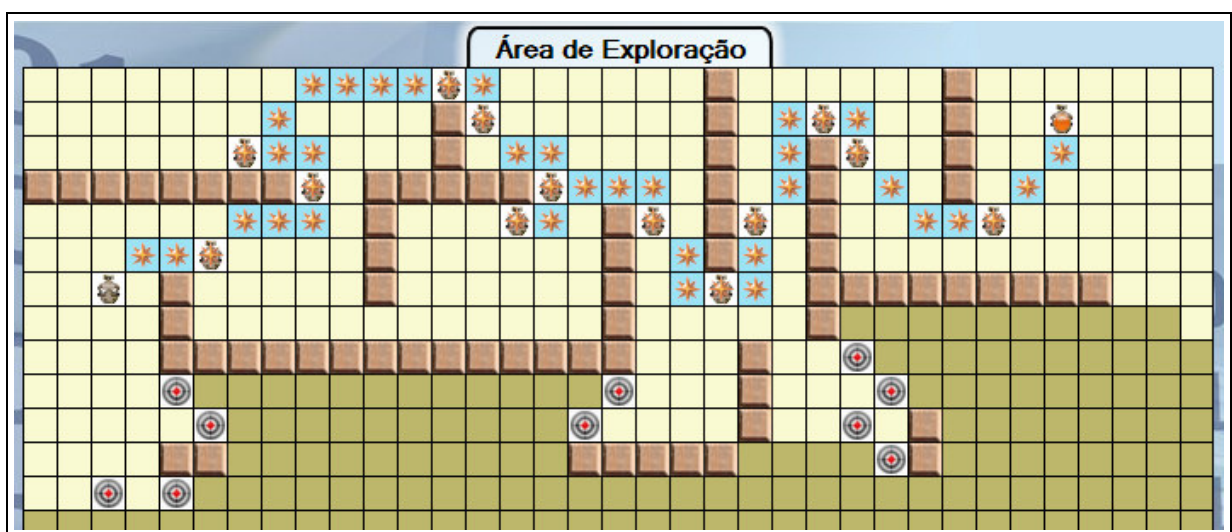


Figura 3.15 – Mapeamento do ambiente.

A figura do botão “Mapear Área” também é alterada para indicar que, para o cenário corrente, já foi realizado o mapeamento, conforme pode ser visto nas figuras 3.16 e 3.17 abaixo.



Figura 3.16 – Botão antes de executar o Mapear Área.



Figura 3.17 – Botão depois de executar o Mapear Área.

3.4.2.3.1. Evento *Click* do botão “Mapear Área”

Antes do mapeamento de um ambiente desconhecido o algoritmo verifica se o robô e a bola foram inseridos na Área de Exploração, se um dos dois não está presente o programa avisa o usuário através de caixas de mensagens (*MessageBox*), porém, diferente do código da navegação no “Melhor Caminho”, neste módulo o simulador permite que o usuário opte pelo robô fazer o mapeamento do ambiente mesmo que nele não esteja presente um alvo (bola). Desta forma, não havendo um alvo no ambiente, o robô pode ser compelido a mapear toda a Área de Exploração. Em seguida o código chama o próximo módulo do mapeamento, a função *ProcuraAlvo*.

3.4.2.3.2. Função *ProcuraAlvo*

Esta função inicia a representação gráfica do mapeamento do ambiente e também define os valores iniciais de algumas matrizes e variáveis. Uma matriz bidimensional de nome *mtzOculto* será utilizada para armazenar as informações coletadas no mapeamento da Área de Exploração do robô, ela tem as mesmas dimensões da matriz *mtzCenario*, que armazena o ambiente que será utilizado na simulação, ou seja, o estado de cada posição da Área de Exploração. Para inicializar o mapeamento a matriz *mtzOculto* tem todas as posições carregadas com o estado de “desconhecido” (valor 4), exceto a posição onde está o robô, que é colocado na matriz de mapeamento no mesmo local onde foi inserido no ambiente. Na representação gráfica é mostrado um fundo escuro na Área de Exploração e a

imagem do robô na sua posição inicial. Como pode ser visto nas figuras 3.18 e 3.19 abaixo.

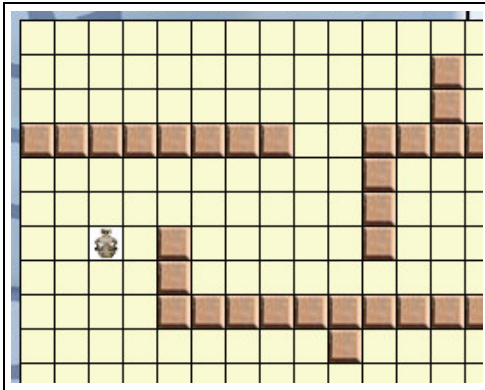


Figura 3.18 – Parte do ambiente que será explorado pelo robô.

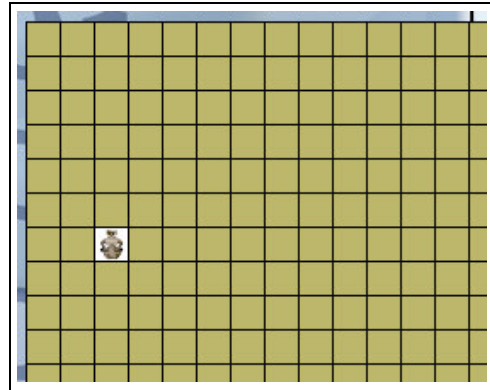


Figura 3.19 – Início do mapeamento da Área de Exploração.

No final, o código chama a função *Visualizar*, passando como parâmetros a posição do robô. O apêndice A.24 destaca as principais linhas de código desta função.

3.4.2.3.3. Função *Visualizar*

Este módulo simula o sensoriamento que o robô faz do ambiente através de um sistema de visão computacional. Assim como em um sistema de visão computacional real, no simulador também é definido um limite para o campo de visão do robô, ou seja, a distância máxima na qual a imagem adquirida é suficientemente boa para detecção de objetos no ambiente. A definição deste campo de visão foi baseada em um sistema real que utiliza câmeras de vídeo que capturam imagens com profundidade de campo de 3 metros, ou seja, a imagem tem uma nitidez aceitável para detecção de obstáculos da posição onde está o robô até uma distância máxima de 3 metros. Para transformar esta informação em uma representação visual no simulador, a profundidade de campo em metros deve ser convertida em *pixels* ou em um número de células na Área de Exploração. Como visto anteriormente, o tamanho da célula é de 20x20 *pixels* e corresponde ao espaço ocupado pelo robô virtual, mas não houve uma definição das dimensões deste robô em metros. Assim, para se estabelecer uma correspondência direta entre o tamanho da célula e a medida em metros foi necessário definir o tamanho do robô. Ele foi considerado como um robô de médio porte e que ocupa uma área de 0,5x0,5

metros. Esta equivalência determina que a profundidade de campo de 3 metros corresponde a 6 células da Área de Exploração.

Uma vez que o robô virtual é capaz de girar sobre o próprio eixo sem sair do lugar e assim visualizar nas 4 direções (norte, sul, leste e oeste) a Área de Visualização foi dimensionada da seguinte forma:

- Eixo vertical (orientação norte-sul)
6 células na direção norte + 1 célula do robô + 6 células na direção sul
- Eixo horizontal (orientação leste-oeste)
6 células na direção leste + 1 célula do robô + 6 células na direção oeste

Isto determina que a Área de Visualização do robô tenha as dimensões de 13x13 células ou 260x260 *pixels*. Como pode ser visto na figura 3.20 abaixo.

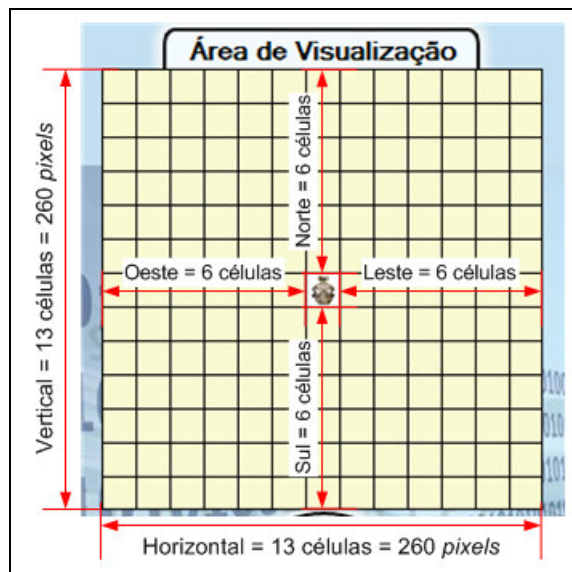


Figura 3.20 – Dimensões da Área de Visualização.

De forma análoga, a Área de Exploração com dimensões de 700x700 *pixels* é dividida em células de 20x20 *pixels*, o que corresponde a uma malha de 35x35 células.

Na função *Visualizar* são inicializados os objetos gráficos, matrizes e variáveis que serão utilizadas no algoritmo de visualização. Na representação gráfica o programa traça na Área de Exploração um quadrado ou um retângulo, dependendo

da posição do robô na Área de Exploração, com bordas vermelhas mostrando a área que o robô está visualizando. Como pode ser visto na figura 3.21 abaixo.

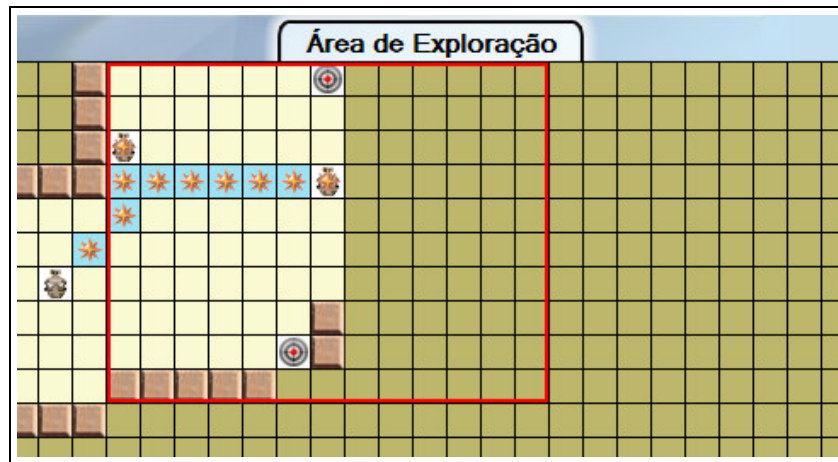


Figura 3.21 – Borda que mostra a área que o robô está visualizando.

Em seguida duas matrizes, nas dimensões que correspondem à Área de Visualização do robô, são utilizadas: a matriz *mtzVisual*, que copia da matriz do cenário (ambiente) o estado das células que estão dentro do campo de visão do robô, e a matriz *mtzVisOculta*, onde serão armazenados os estados das áreas visualizadas pelo robô e que inicialmente tem todas as posições carregadas com o estado de “desconhecido” (valor 4), exceto na posição central onde é colocado o robô. Como pode ser visto no apêndice A.25.

A representação gráfica mostra no *PictureBox* da Área de Visualização um fundo escuro e coloca no centro a imagem indicando que o robô irá iniciar a visualização pela direção leste. Como pode ser visto na figura 3.22 abaixo.



Figura 3.22 – Imagem inicial da Área de Visualização.

Na sequência do código, da função *Visualizar*, é que se faz propriamente a visualização do ambiente ao redor do robô. Dentro de um laço de repetição que é executado 4 vezes, uma para cada direção (norte, sul, leste e oeste) que o robô irá visualizar a partir da sua posição atual.

Em cada direção o robô “enxerga” um ângulo de visão de 90°. Como estratégia para simular a visão computacional foi realizada uma varredura da Área de Visualização em ângulos de +1 a +45° e depois de -1 a -45°, com intervalos de 1 grau. Sempre partindo do centro para os perímetros da Área de Visualização. Como pode ser visto na figura 3.23 abaixo.

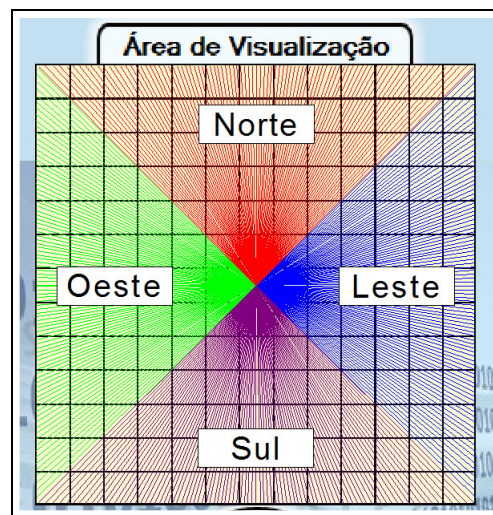


Figura 3.23 – Visualização em cada direção.

Pontos de verificação do estado da célula são gerados fazendo-se um deslocamento sobre uma linha que parte do centro para os limites da Área de Visualização em um determinado ângulo. Cada novo ponto é definido pela seguinte relação trigonométrica do triângulo retângulo:

Simbolicamente:

$$\tan \theta = \frac{\text{cateto oposto a } \theta}{\text{cateto adjacente a } \theta}$$

Ou seja: *A tangente do ângulo θ é dada pela razão entre a medida do cateto oposto a ele e a medida do cateto que lhe é adjacente.*

Para se gerar um novo ponto de verificação define-se o ângulo da “linha de visão” (θ) e o deslocamento perpendicular à direção de visualização do robô, que corresponde ao cateto adjacente. Com estes dados pode-se calcular o

deslocamento paralelo à direção de visualização do robô, que corresponde ao cateto oposto, utilizando a relação trigonométrica acima. Porém reescrita da seguinte forma:

$$\text{cateto oposto a } \theta = \tan \theta * \text{cateto adjacente a } \theta$$

Como foi visto, o robô é colocado no centro da Área de Visualização que tem as dimensões de 260x260 *pixels*, o que implica que o centro está nas coordenadas 130x130 *pixels*. Por exemplo, se em um dado momento o robô está visualizando na direção leste em uma linha de visão com ângulo de 30° e um deslocamento perpendicular de 100 *pixels*, de acordo com a equação acima o valor do deslocamento paralelo à direção de visualização do robô será igual a:

$$\text{cateto oposto} = \tan 30^\circ * 100 = 0,577 * 100 = 57,7$$

Isto implica que o novo ponto de verificação está a um deslocamento perpendicular de 100 *pixels* e a um deslocamento paralelo de, aproximadamente, 58 *pixels* da posição central da Área de Visualização. Ou seja, o ponto de verificação está nas coordenadas:

$$X = 130 + 100 = 230$$

$$Y = 130 + 58 = 188$$

Como é ilustrado na figura 3.24 abaixo.

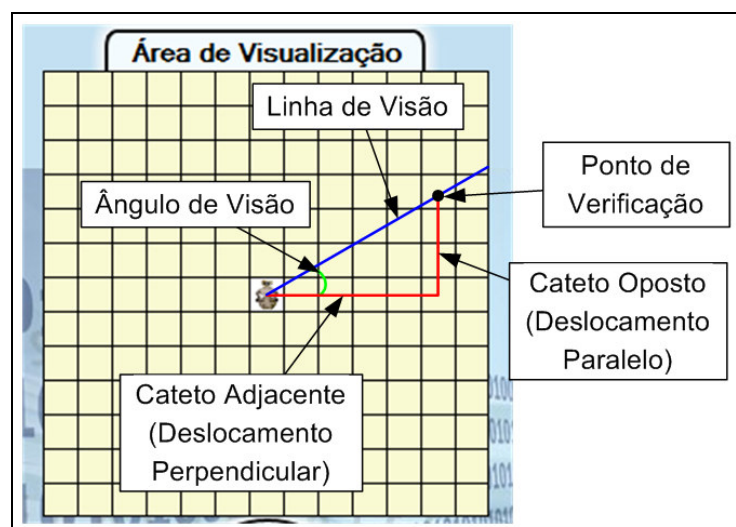


Figura 3.24 – Cálculo de um Ponto de Verificação.

O intervalo do ângulo entre as linhas de visão e o deslocamento perpendicular à direção de visualização definem a acuidade da visão do robô, ou seja, estão relacionadas à qualidade com que o robô visualiza a região ao seu redor. Quanto menores estes valores melhor a visualização do ambiente, porém maior o custo computacional para se executar o algoritmo, ou seja, maior a demanda de recursos do sistema. Na relação custo-benefício estes valores foram definidos em um intervalo de 1 grau com um deslocamento perpendicular à direção de visualização do robô de 5 *pixels*. Estes valores resultaram em uma visão ótima, valores maiores proporcionaram um resultado aceitável e causaram um impacto menor ao sistema computacional. Porém em um sistema robótico o tempo de execução do programa não é tão crítico, se comparado ao tempo que o robô despende para se locomover ou para girar sobre o próprio eixo. Inclusive no simulador em vários momentos é utilizado a instrução *Thread.Sleep(tempo)*, que atrasa a execução do programa com o intuito de melhor representar um sistema real. Onde o argumento (*tempo*) define o tempo de atraso em milissegundos. O apêndice A.26 mostra a parte do código que faz os cálculos descritos acima.

Definido um ponto de verificação o seu valor em *pixels* é transformado em um valor inteiro, que corresponde aos índices das matrizes bidimensionais de visualização (*mtzVisual* e *mtzVisOcultas*). A visualização do robô consiste em utilizar estes índices para ler o estado da posição na matriz *mtzVisual*, que é uma cópia do ambiente ao redor do robô, e atribuir este estado à mesma posição da *mtzVisOcultas*, que vai armazenar os estados das células que o robô efetivamente conseguiu “enxergar” dentro de seu campo de visão. Após esta atribuição, de acordo com o estado da posição visualizada serão executados procedimentos distintos e é isso que vai diferenciar o conteúdo final da matriz *mtzVisOcultas* do conteúdo da matriz *mtzVisual*.

Na visualização as células podem assumir 6 estados, conforme ilustra a tabela 3.1 a seguir.

Tabela 3.1 – Estados da Área de Visualização.

Estado	Valor	Imagem	Descrição
<i>livre</i>	0		Área por onde o robô pode se locomover.
<i>obstaculo</i>	1		Área por onde o robô não pode se locomover.
<i>inicio</i>	2		Posição atual do robô.
<i>objetivo</i>	3		Posição do objetivo, o local onde está a bola.
<i>desconhecido</i>	4		O robô não conhece o estado desta célula.
<i>fora</i>	5		Delimita a região de exploração do robô.

Na visualização foi observado que em determinados momentos a visão atravessava dois obstáculos diagonais adjacentes e “enxergava” áreas livres ou o próprio objetivo, mas estes não estavam acessíveis ao robô. Conforme pode ser visto na figura 3.25 abaixo.



Figura 3.25 – A visão do robô atravessa os obstáculos.

Isto acontecia devido ao deslocamento perpendicular à direção de visualização do robô de 5 *pixels*, que nestes casos atravessava o canto de uma célula com obstáculo. A solução mais simples e definitiva para o problema seria diminuir o deslocamento perpendicular para apenas 1 *pixel*, mas, como visto anteriormente, esta solução acarretaria um custo computacional demasiadamente alto e inviável, deixando o algoritmo muito lento.

Outra consideração feita é que em sistemas reais a situação de o robô visualizar uma área livre ou o objetivo através de uma “fresta” entre obstáculos pode acontecer. Assim, ao invés de se evitar que o problema ocorra, a abordagem adotada foi de tratar esta situação como factível e providenciar que o algoritmo verifique se a área livre ou o objetivo visualizado está ou não acessível ao robô. Se não estiver acessível, a visualização da célula é desconsiderada e ela é mantida no estado de “desconhecido”, tanto na Área de Visualização como no mapeamento da Área de Exploração. No algoritmo da função *Visualizar* esta verificação é realizada de 2 formas:

1. Verifica se a visualização não atravessou dois obstáculos diagonais adjacentes, passando a informação correta da posição de uma área livre ou do objetivo, mas que não está acessível ao robô.
2. Verifica, utilizando o algoritmo de *pathfinding*, se a área livre ou o objetivo visualizado está acessível ao robô. Neste caso o algoritmo de *pathfinding* foi empregado para examinar se há um caminho até a célula visualizada e não propriamente para saber qual é este caminho.

A figura 3.26 abaixo mostra, para a mesma situação da figura 3.25, o resultado da visualização após os cuidados citados acima.

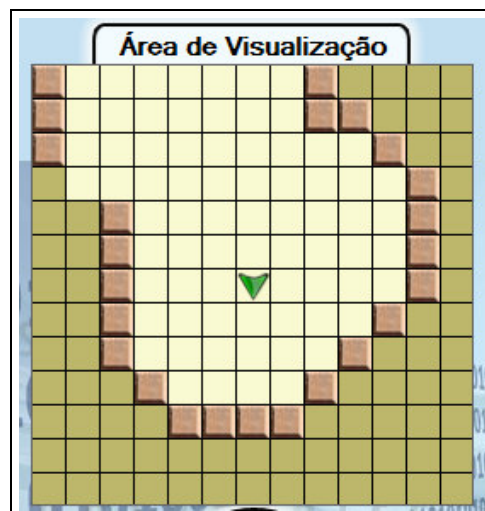


Figura 3.26 – A visão do robô não atravessa os obstáculos.

O apêndice A.27 destaca as principais linhas do código que executa estes procedimentos.

Uma vez validado a célula visualizada pelo robô o algoritmo irá salvar na matriz que armazena os estados das áreas visualizadas pelo robô (*mtzVisOcultas*) e na matriz de mapeamento da Área de Exploração (*mtzOcultas*) o estado da célula visualizada. Também será atualizado o *PictureBox* da Área de Visualização, representando na célula visualizada a imagem correspondente.

A partir de então, os procedimentos se diferem dependendo da última célula visualizada:

- Se foi visualizada uma área livre o algoritmo continua a visualização no mesmo ângulo de visão e faz um novo deslocamento perpendicular à direção de visualização do robô de 5 *pixels*. Ou seja, enquanto a área estiver livre ele gera pontos de verificação, na mesma linha de visão, até atingir o limite do campo de visão do robô.
- Se a célula visualizada foi o objetivo (bola) o programa interrompe a execução da função *Visualizar* e chama a função *ProximoPonto*, que irá finalizar o mapeamento da Área de Exploração.
- Se foi visualizado um obstáculo o código irá interromper a visualização no ângulo de visão corrente e passar para o ângulo seguinte. Isto representa que o robô não é capaz de “enxergar” além dos obstáculos presentes no ambiente.
- Se foi visualizado o limite da Área de Exploração o código irá interromper a visualização no ângulo de visão corrente e passar para o ângulo seguinte.

No apêndice A.28 são mostrados trechos do código que executa os procedimentos descritos acima.

No final da visualização da área ao redor do robô podem ocorrer duas situações distintas:

- Se o objetivo foi encontrado o algoritmo chama a função *ProximoPonto*, que neste caso irá finalizar o mapeamento da Área de Exploração.
- Se o objetivo não foi encontrado é chamada a função *PontosBusca*, que gera possíveis pontos para onde o robô deve se locomover na busca do objetivo.

3.4.2.3.4. Função *PontosBusca*

Esta função marca na Área de Exploração pontos para buscas futuras, baseadas na última visualização que o robô fez do ambiente. Os pontos gerados nesta função serão utilizados pela função *ProximoPonto* na tomada de decisão sobre o próximo ponto para onde o robô deve se deslocar na busca do objetivo.

Teoricamente qualquer célula próxima a uma área desconhecida pode ser considerada um possível ponto de busca. Porém um número indiscriminado de pontos de busca tornaria a tarefa de tomada de decisão da função *ProximoPonto* muito “pesada”, acarretando um alto custo computacional ao sistema. Assim o código desta função tem o compromisso de gerar uma quantidade de pontos de busca suficiente para o bom mapeamento da Área de Exploração sem, contudo, demandar uma quantidade excessiva de recurso do sistema computacional.

Para armazenar a informação da posição do ponto de busca na matriz da visão do robô (*mtzVisOculto*) e na matriz do mapeamento da Área de Exploração (*mtzOculto*) foi criado mais um estado chamado *alvoBusca* e sua respectiva representação gráfica, conforme ilustra a tabela 3.2.

Tabela 3.2 – Ponto de Busca.

Estado	Valor	Imagem	Descrição
<i>alvoBusca</i>	6		Possível ponto de busca.

Inicialmente a função cria pontos de busca no perímetro da Área de Visualização, porém não em todas as células livres, pois em um ambiente com poucos obstáculos seriam gerados muitos pontos de busca. Conforme pode ser visto na figura 3.27 a seguir.

Para gerar pontos de busca o algoritmo percorre todo o perímetro da visão do robô. Ele faz este percurso no sentido horário e partindo da primeira célula, ou seja, da célula do canto superior esquerdo da Área de Visualização e que corresponde à posição (0, 0) da matriz *mtzVisOculto*. Os pontos são gerados considerando os seguintes critérios:

- Cria um ponto de busca na primeira área livre detectada no perímetro da visão.
- Mede a extensão da área livre contígua no perímetro da visão do robô até detectar o primeiro obstáculo e marca um ponto de busca no centro do perímetro livre. Porém em áreas com pouco ou nenhum obstáculo este procedimento pode gerar um número muito pequeno de pontos de busca. Assim, se este perímetro livre for maior que um dos lados da Área de Visualização do robô o programa marca mais de um ponto.

Este procedimento é repetido até ser percorrido todo o perímetro da Área de Visualização. Os pontos de busca gerados são mostrados na Área de Visualização do robô, conforme pode ser visto na figura 3.28 abaixo.

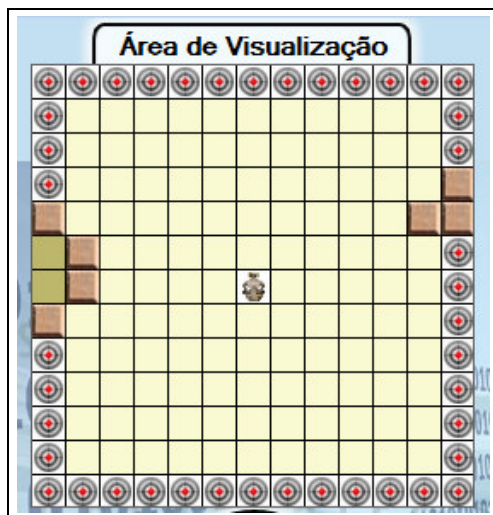


Figura 3.27 – Pontos de busca em todo o perímetro da Área de Visualização.

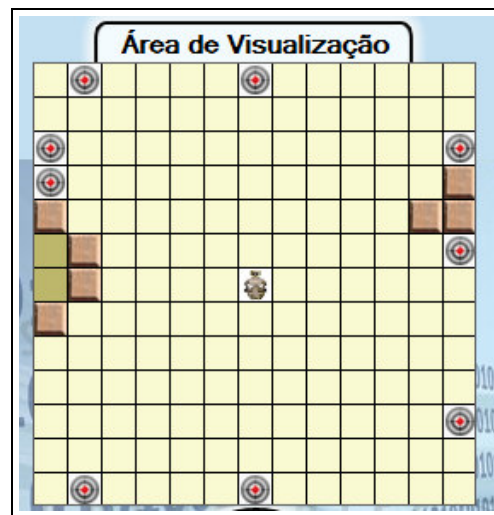


Figura 3.28 – Pontos de busca seguindo os critérios descritos.

O apêndice A.29 destaca as principais linhas desta parte do código.

Em alguns casos, devido a obstáculos, a visão do robô é obstruída e não chega a visualizar os perímetros do seu campo de visão, e, assim, pelo algoritmo anterior não seria gerado nenhum ponto de busca para o mapeamento do ambiente. Sendo assim, foi implementado uma rotina que cria pontos de busca adjacentes a áreas desconhecidas dentro do campo de visão do robô. As figuras 3.29 e 3.30 a seguir ilustram esta situação. O apêndice A.30 mostra parte do código que gera os pontos de busca descritos acima.

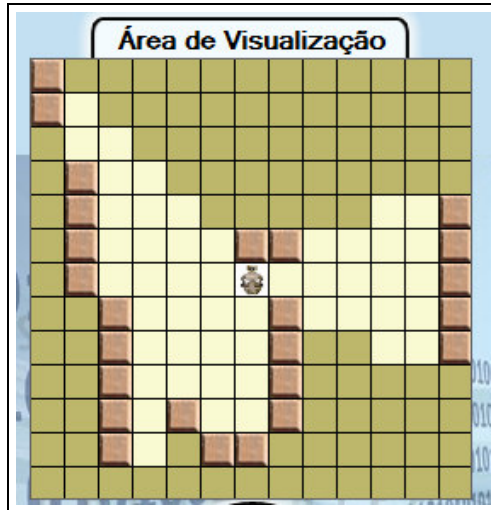


Figura 3.29 – O robô tem a visão obstruída dentro do seu campo de visão.

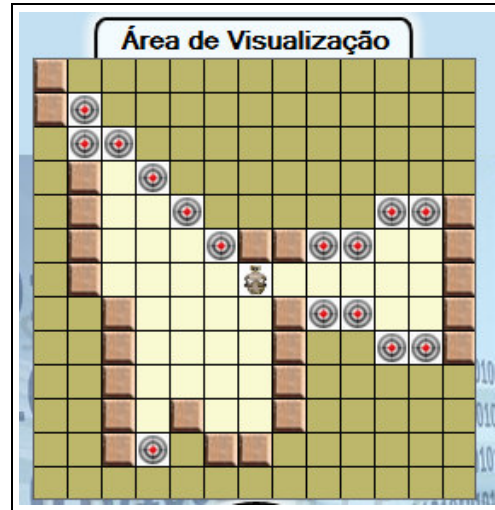


Figura 3.30 – Pontos de busca gerados dentro da Área de Visualização.

À medida que o robô mapeia a Área de Exploração, um antigo ponto de busca pode ficar dentro de uma região conhecida deixando de ser uma opção interessante para buscas futuras. Assim, o algoritmo elimina pontos de busca que não estão adjacentes a uma área desconhecida. As figuras 3.31 e 3.32 ilustram esta situação. A figura 3.31 representa a primeira visualização do ambiente feita pelo robô e mostra que ela gerou 5 possíveis pontos de busca. A figura 3.32 mostra para qual ponto de busca o robô se deslocou e que nesta segunda visualização o programa gerou 4 novos pontos e excluiu 4 pontos antigos, que passaram a estar dentro de uma área conhecida. O apêndice A.31 mostra o trecho da função que realiza esta tarefa.



Figura 3.31 – Primeira visualização.



Figura 3.32 – Segunda visualização.

Para tornar mais leve o algoritmo de tomada de decisão que irá escolher o próximo ponto de busca, são eliminados pontos perpendiculares adjacentes a outros pontos de busca, permanecendo apenas 1. A figura 3.33 a seguir mostra os pontos

de busca gerados na Área de Visualização e a figura 3.34 mostra os pontos que são inseridos na Área de Exploração, já com os pontos perpendiculares adjacentes excluídos. O apêndice A.32 destaca as principais linhas do código que executa esta tarefa.

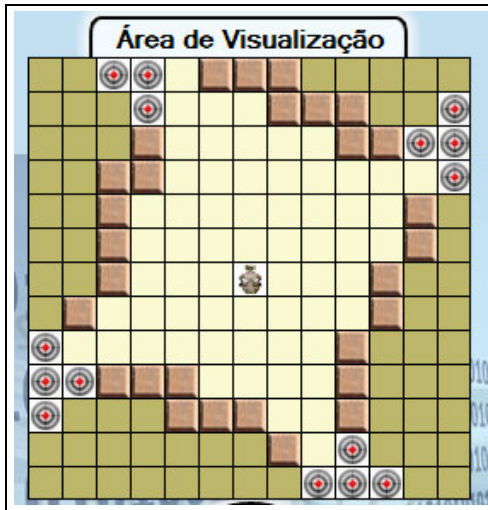


Figura 3.33 – Pontos de busca gerados na Área de Visualização.

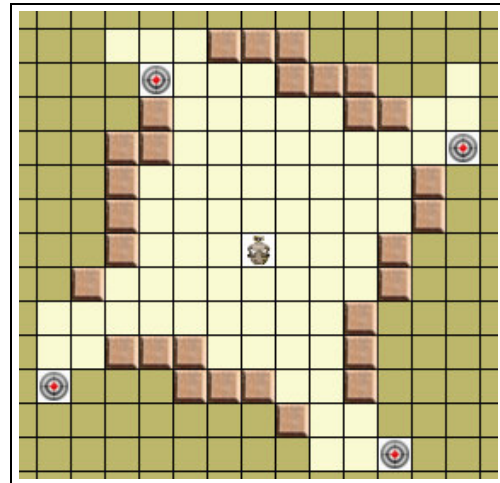


Figura 3.34 – Pontos de busca inseridos na Área de Exploração.

Em seguida o algoritmo da função *PontosBusca* calcula o número de pontos de busca remanescentes na Área de Exploração. Se este número for igual a zero significa que toda a área foi mapeada e o objetivo (bola) não foi encontrado. Porém, para garantir que o robô realmente verificou todos os pontos possíveis, uma rotina de segurança examina se ainda existe alguma área desconhecida na Área de Exploração que está acessível ao robô, se existir ela gera um ponto de busca contíguo a esta área desconhecida.

Finalmente a função *PontosBusca* faz a representação gráfica que mostra na Área de Visualização os pontos de busca gerados e chama a função *ProximoPonto*.

3.4.2.3.5. Função *ProximoPonto*

Este é o módulo de tomada de decisão do simulador. Esta função determina:

- Para qual dos pontos de busca, gerados no ambiente, o robô deve se deslocar para realizar uma nova visualização;

- Encerra o mapeamento do ambiente se não houver mais pontos de busca e o robô não encontrou o objetivo;
- Finaliza a busca quando o robô encontra o objetivo.

A função *ProximoPonto* também armazena os caminhos percorridos pelo robô durante o mapeamento da Área de Exploração. Para registrar estes caminhos foram criados mais dois estados chamados *caminho* e *pontoVisao*, assim como suas respectivas representações gráficas, conforme ilustra a tabela 3.3 a seguir.

Tabela 3.3 – Estados dos caminhos percorridos pelo robô.

Estado	Valor	Imagem	Descrição
<i>caminho</i>	7		Representa o caminho percorrido pelo robô.
<i>pontoVisao</i>	8		Representa um ponto onde o robô realizou uma visualização do ambiente.

Inicialmente o algoritmo instancia um objeto da classe *Pathfinding.cs* de nome *PathPonto*. Se o objetivo ainda não foi encontrado, o objeto *PathPonto* é utilizado para calcular o melhor caminho entre a posição atual do robô e cada ponto de busca registrado na Área de Exploração. Destes cálculos o algoritmo extrai para qual dos pontos de busca o custo de deslocamento do robô é menor e, então, eleger este ponto como a próxima posição para onde o robô deve se locomover para continuar o mapeamento. A figura 3.35 a seguir mostra 5 pontos de busca com os respectivos custos F do deslocamento do robô até eles, assim como destaca o ponto escolhido. O apêndice A.33 mostra as principais linhas desta parte do código.

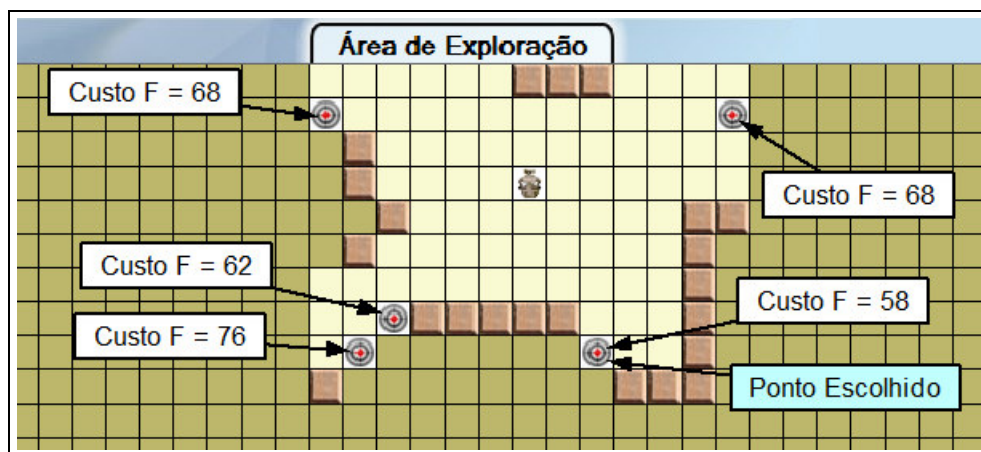


Figura 3.35 – Custos F até cada ponto de busca.

A função *PontosBusca*, descrita anteriormente, gera os pontos de busca e atualiza a imagem na Área de Visualização, mas não na Área de Exploração do robô. A função *ProximoPonto* é que atualiza o *PictureBox* da Área de Exploração, mostrando em cada célula a imagem que representa o seu estado. Esta imagem pode ser a de uma área livre (fundo claro), um obstáculo, o robô, a bola, uma área desconhecida (fundo escuro), a imagem de uma estrela com fundo azul, que representa o caminho percorrido pelo robô, ou a imagem do robô com uma estrela sobreposta, que representa um ponto onde o robô realizou a visualização da Área de Exploração. O apêndice A.34 mostra partes do código que faz a atualização da imagem.

No mapeamento da Área de Exploração o robô percorre vários caminhos. Para armazenar cada caminho foi implementado uma classe de nome *caminhos.cs*, que armazena a informação da ordem do caminho em relação aos outros e das coordenadas X e Y de cada posição ao longo deste caminho. Entretanto, o número de caminhos difere de acordo com o ambiente e as posições onde são inseridos o robô e a bola. Assim, para armazenar estes caminhos foi utilizado um componente do C# chamado *ArrayList*, que cria uma lista de vetores cujo tamanho é incrementado dinamicamente conforme a necessidade, ou seja, conforme a quantidade de caminhos que foram gerados no mapeamento do ambiente. O apêndice A.35 mostra o código da classe *caminhos.cs*.

Se a posição da bola foi encontrada ou se foi definido o próximo ponto de busca para onde o robô deve se deslocar, o algoritmo calcula o melhor caminho entre o ponto onde se encontra o robô e o novo ponto de busca selecionado ou até o seu objetivo (bola). Realizado o *pathfinding* o caminho é armazenado na lista de vetores *CaminhosBusca*. O apêndice A.36 mostra a parte do código que salva os caminhos percorridos pelo robô.

De posse das informações do caminho o programa faz a animação que mostra, na Área de Exploração, o caminho percorrido pelo robô até um novo ponto de busca ou o objetivo. O apêndice A.37 mostra o trecho da função que realiza esta tarefa.

Em seguida é executada uma rotina de tomada de decisão da função, que analisa 3 situações distintas:

1. Enquanto o objetivo não for encontrado e ainda houver algum ponto de busca não verificado na Área de Exploração, o programa retorna à função *Visualizar* para continuar o mapeamento do ambiente. A representação gráfica do simulador mostra quando o robô chega ao seu novo ponto de visualização, conforme ilustra a figura 3.36 abaixo. O apêndice A.38 mostra o trecho da função que realiza esta tarefa.

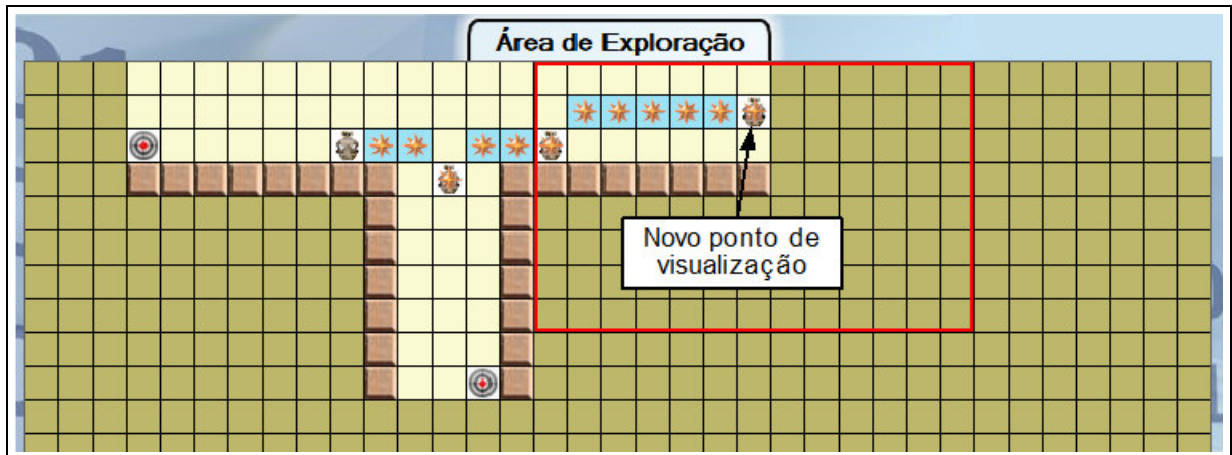


Figura 3.36 – Robô em um novo ponto de visualização.

2. Se o objetivo foi encontrado o simulador encerra o mapeamento da Área de Exploração, faz a representação gráfica do robô chegando ao seu objetivo, conforme pode ser visto na figura 3.37, e emite uma indicação sonora de que a busca foi bem sucedida. O apêndice A.39 mostra o trecho da função que realiza esta tarefa.



Figura 3.37 – O robô encontra o seu objetivo.

3. Se o objetivo não foi encontrado e o número de possíveis pontos de busca é igual a zero, o programa informa que não foi encontrada a bola e encerra a busca. Conforme pode ser visto na figura 3.38. O apêndice A.40 mostra o trecho da função que realiza esta tarefa.



Figura 3.38 – O objetivo não foi encontrado.

3.4.2.4. VISÃO GERAL

O botão “Visão Geral” (*btnGera*) mostra, na Área de Exploração, os caminhos percorridos pelo robô durante o mapeamento do ambiente e o melhor caminho entre o robô e a bola simultaneamente. Este botão está associado aos comandos de “Melhor Caminho” e de “Mapear Área”, ou seja, somente é habilitado quando já foi calculado o melhor caminho e o objetivo foi encontrado durante o mapeamento do ambiente.

A atualização da imagem no *PictureBox* da Área de Exploração é feita através do seu evento *Paint*, as partes principais do código deste evento serão explicadas posteriormente. Na figura 3.39 a seguir a imagem de uma estrela com o fundo verde representa o melhor caminho e a imagem da estrela com o fundo azul mostra os caminhos percorridos no mapeamento da Área de Exploração.



Figura 3.39 – Melhor caminho e o resultado do mapeamento do ambiente.

3.4.2.5. ATUALIZA A ÁREA DE EXPLORAÇÃO

A imagem da Área de Exploração é exibida no *PictureBox picImagem* e é atualizada pelo código associado ao seu evento *Paint*. No simulador este evento é ativado sempre que a Área de Exploração precisa ser redesenhada. O evento *Paint* é lançado pelo método *OnPaint* quando um controle (como *TextBox* ou *PictureBox*) é exibido, porém para disparar o evento *Paint* explicitamente deve ser chamado o método *Invalidate*.

No simulador ocorrem 5 situações que demandam a atualização da Área de Exploração e em cada uma delas é executado um procedimento distinto, porém todos eles foram implementados dentro do evento *Paint* do *PictureBox picImagem*. A seguir são descritas cada uma destas situações:

1. A imagem na Área de Exploração é atualizada quando na formação do cenário é inserido um objeto (o robô, a bola, obstáculos ou apagado algum destes objetos) ou é recuperado o cenário que foi armazenado anteriormente. O apêndice A.41 mostra a parte do código do evento *Paint* que realiza este procedimento.
2. Mostra na Área de Exploração o melhor caminho entre o robô e a bola. A parte do código do evento *Paint* que realiza este procedimento é mostrada no apêndice A.42.

3. Mostra na Área de Exploração os custos F de todas as células verificadas no cálculo do melhor caminho. O apêndice A.43 destaca a parte do código que realiza esta tarefa.
4. Mostra na Área de Exploração a representação gráfica do mapeamento realizado pelo robô. A parte do código do evento *Paint* que realiza este procedimento é mostrada no apêndice A.44.
5. Mostra os caminhos percorridos pelo robô no mapeamento do ambiente e o melhor caminho simultaneamente. Como pode ser visto em parte do código demonstrada no apêndice A.45.

3.4.2.6. ATUALIZA A ÁREA DE VISUALIZAÇÃO

A imagem da Área de Visualização é exibida no *PictureBox picVisualizar* e é atualizada pelo código associado ao seu evento *Paint*, que apenas traça as linhas que separam as células na Área de Visualização do robô. Como pode ser visto no apêndice A.46.

4

APLICAÇÃO E RESULTADOS

Neste capítulo será demonstrado um exemplo de aplicação do sistema desenvolvido, trata-se de um simulador que tem como cenário o campus da UNIFEI e a sua execução mostra a forma como um veículo autônomo mapeia e navega neste ambiente enquanto procura pelo objetivo. A interface gráfica mostra uma imagem via satélite do campus da UNIFEI extraída do *software* Google Earth, um serviço *online* e gratuito que permitiu visualizar e capturar a imagem usada no simulador, conforme pode ser visto na figura 4.1 abaixo.

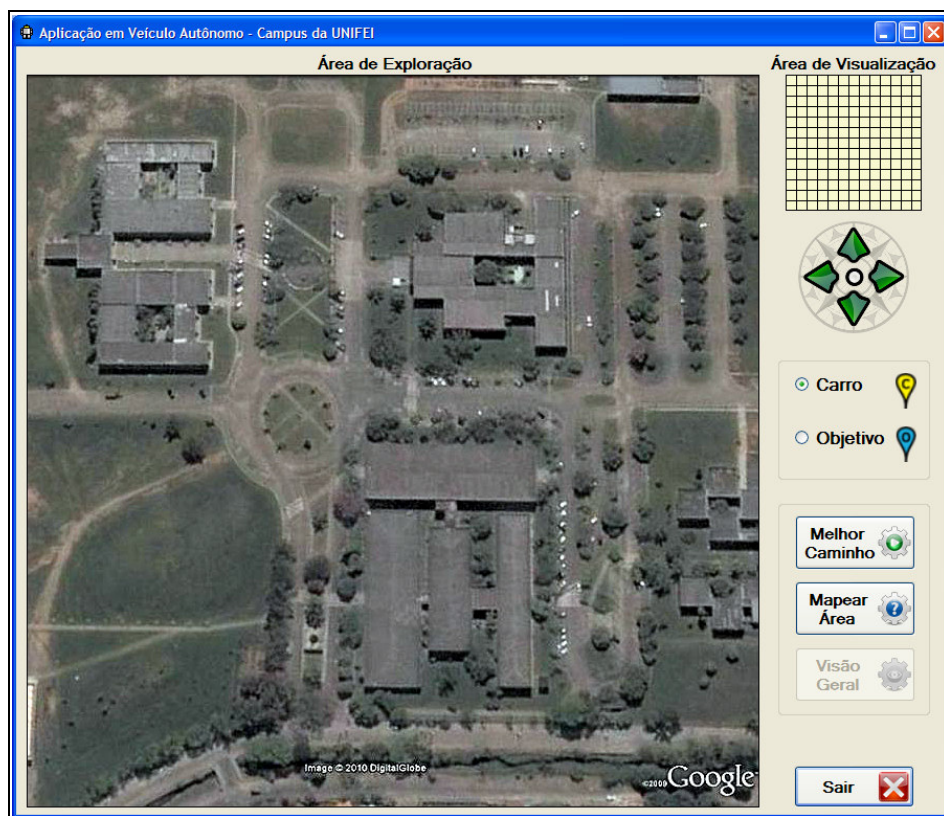


Figura 4.1 – Interface Gráfica.

O veículo autônomo virtual só pode se locomover nas vias internas do campus da UNIFEI, pois em *background*, ou seja, atrás da imagem que é mostrada ao usuário na Área de Exploração, foram marcadas apenas as ruas e os estacionamentos como áreas livres, todas as demais regiões foram assinaladas como obstáculos. O simulador também não permite que o carro e o objetivo sejam posicionados em regiões que não estejam marcadas como áreas livres.

Não foi fornecido ao sistema robótico nenhuma informação a respeito das regras de circulação nas vias como, por exemplo, sentido único ou duplo, sentido na rotatória, proibição de conversão, parada obrigatória, faixa de pedestres e outros. O sensoramento do veículo se restringe à simulação de um sistema de visão computacional, que realiza apenas a detecção de obstáculos, espaços livres e o objetivo. Consequentemente, nos testes realizados por várias vezes o veículo autônomo transitou na contramão.

Para avaliar o funcionamento do simulador foi realizado um experimento em que o carro foi posicionado próximo à portaria da UNIFEI e o objetivo inserido na entrada de serviço do CRTi (Centro de Referência em Tecnologia da Informação). Conforme pode ser visto na figura 4.2 abaixo.

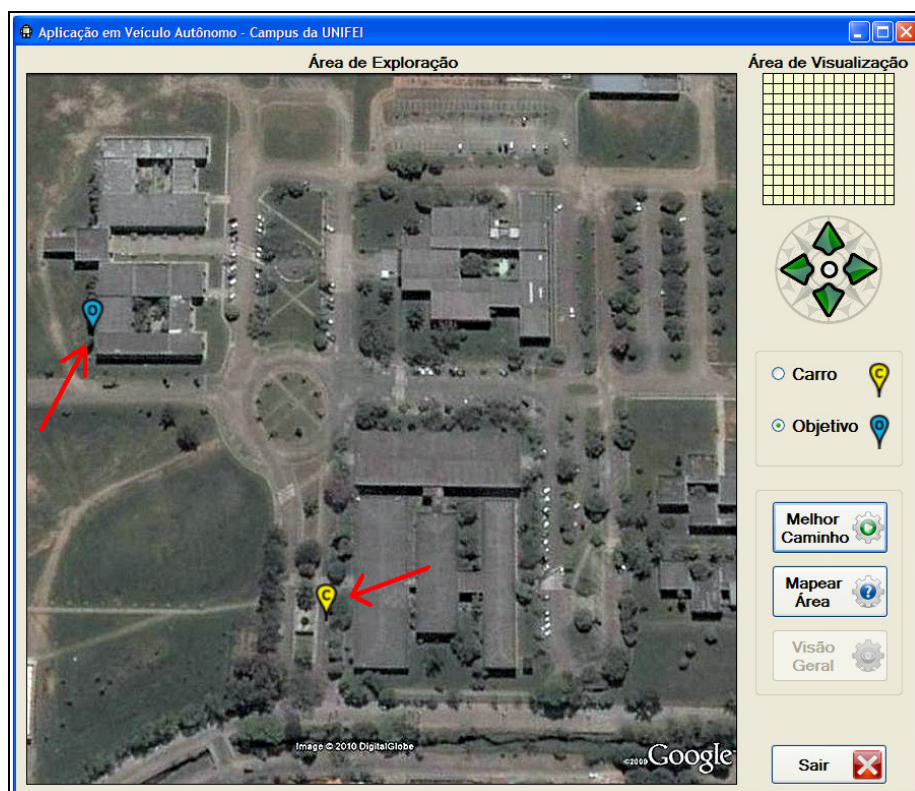


Figura 4.2 – Posicionando o carro e o objetivo.

O simulador permite que seja mostrado o melhor caminho entre o carro e o objetivo. Neste caso é representada a navegação do veículo em um ambiente conhecido, ou seja, ele tem a informação prévia da sua localização, a posição do objetivo e dos obstáculos na Área de Exploração. Os pontos verdes na figura 4.3 abaixo mostram o caminho mais curto entre o carro e o objetivo.

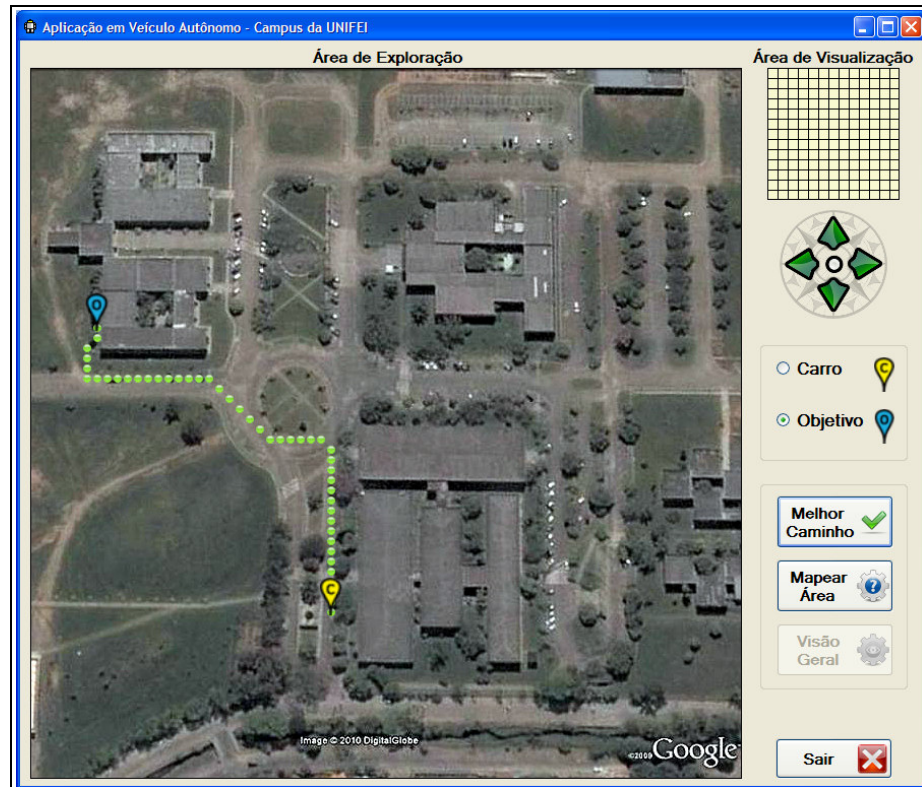


Figura 4.3 – Melhor caminho entre o carro e o objetivo.

Porém o objetivo deste trabalho é desenvolver estratégias para a exploração de ambientes desconhecidos por robôs móveis autônomos e inteligentes. Assim, é simulada a situação em que um veículo autônomo não tem qualquer informação sobre o ambiente a ser explorado e apenas a informação de procurar o objetivo ou, se não for marcado um objetivo, mapear todo o campus da UNIFEI. Para realizar o mapeamento do ambiente o veículo visualiza a região ao seu redor, a área máxima que ele consegue visualizar é representada por um quadrado com bordas vermelhas na Área de Exploração e o resultado da visualização é mostrado na Área de Visualização, conforme pode ser visto na figura 4.4 a seguir. Em um sistema real esta tarefa pode ser desempenhada utilizando-se várias câmeras de vídeo colocadas em pontos estratégicos do carro ou com o uso de uma única câmera *pan-tilt*, que é capaz de se movimentar nos sentidos horizontal (*pan*) e vertical (*tilt*), permitindo que sua lente possa ser apontada para qualquer ponto no espaço.

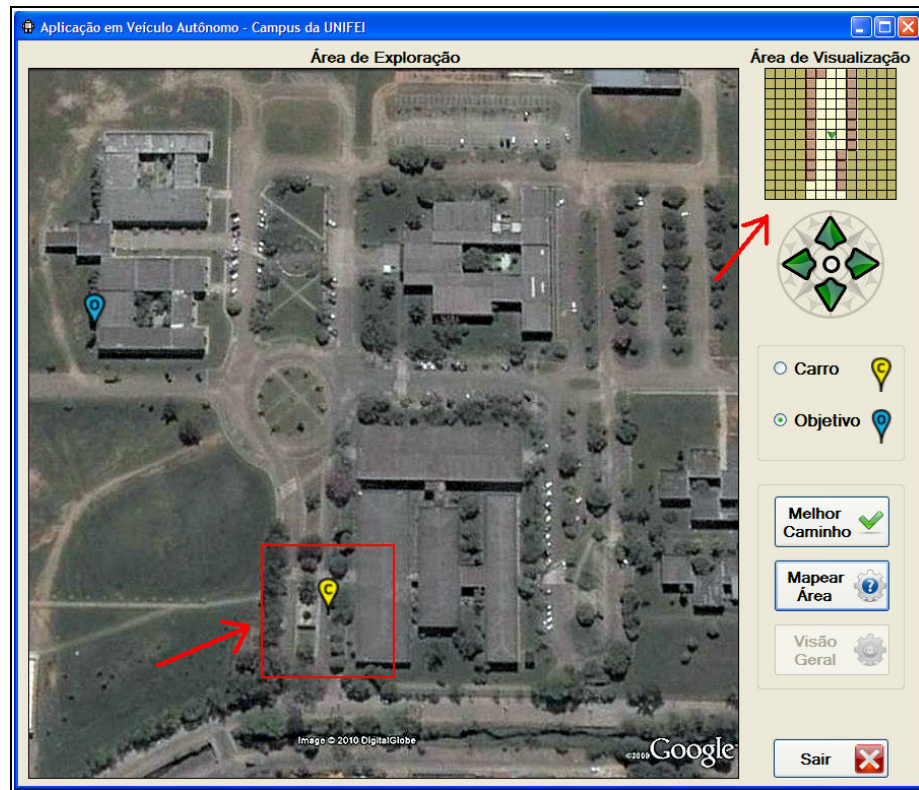


Figura 4.4 – Mostra a visualização na Área de Exploração e na Área de Visualização.

Em seguida são gerados possíveis pontos para onde o veículo pode se deslocar para continuar a busca do objetivo, conforme pode ser visto na figura 4.5.

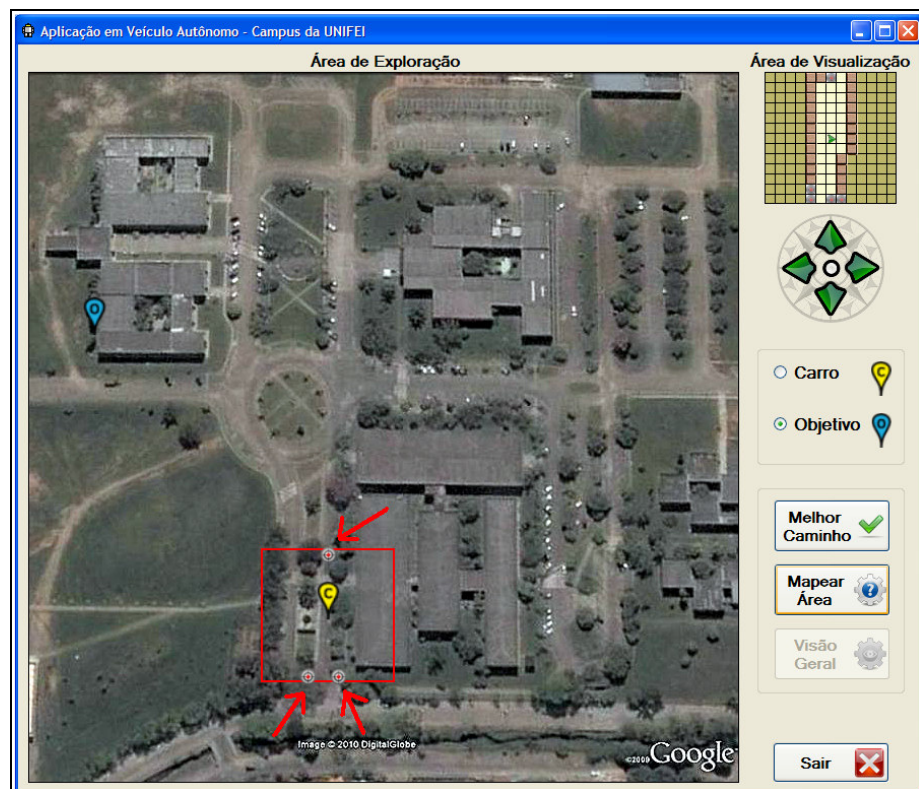


Figura 4.5 – Possíveis pontos de busca.

Neste momento, o módulo de estratégia de exploração escolhe para qual ponto de busca o veículo deve se deslocar e o módulo de planejamento de trajetória calcula o melhor caminho entre o ponto onde se encontra o carro e o ponto de busca selecionado. Com estas informações o módulo de navegação envia comandos que possibilitam o veículo autônomo virtual percorrer eficientemente o ambiente desconhecido. Os pontos azuis na figura 4.6 abaixo mostram o caminho realizado pelo carro partindo de seu ponto inicial até o primeiro ponto de busca selecionado.

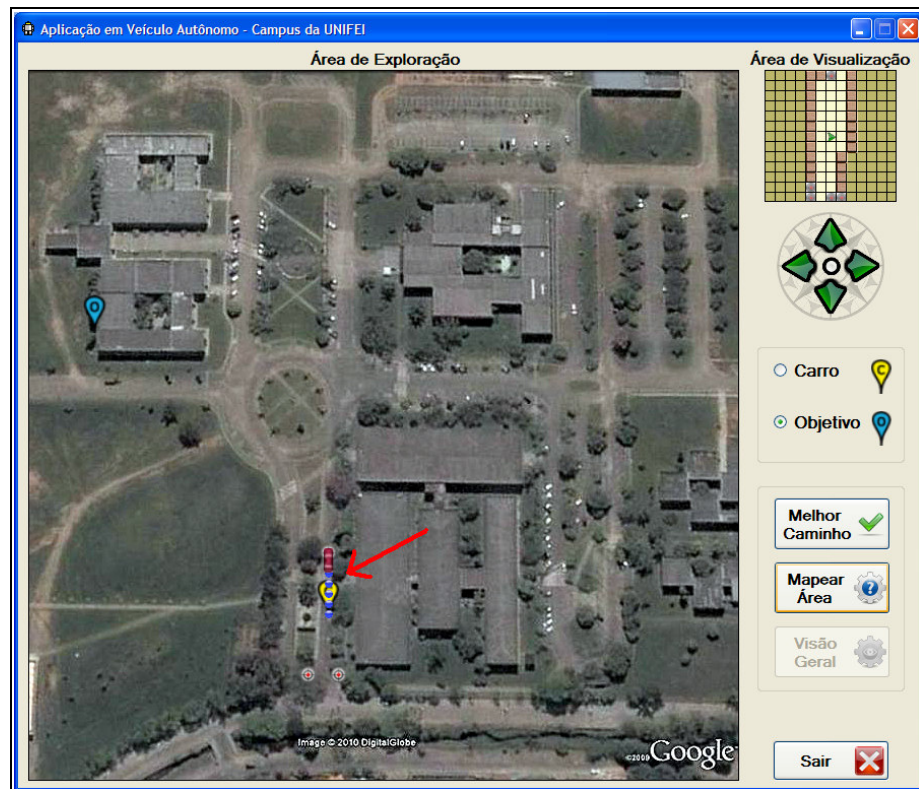


Figura 4.6 – Deslocamento do veículo no ambiente.

Os procedimentos descritos anteriormente são repetidos até que o sistema robótico encontre o objetivo ou até que o ambiente esteja totalmente mapeado, ou seja, não exista mais nenhuma área desconhecida acessível.

A seguir é apresentada uma sequência de imagens que ilustram a busca realizada pelo veículo autônomo até ele encontrar o objetivo, colocado próximo à “garagem” do laboratório do CRTi. As figuras mostram momentos onde o carro está percorrendo o campus em busca do objetivo, parado visualizando e mapeando o ambiente e o instante em que ele atinge o seu objetivo. Em todos os testes realizados o veículo foi capaz de alcançar o objetivo de modo autônomo, inteligente e seguro.

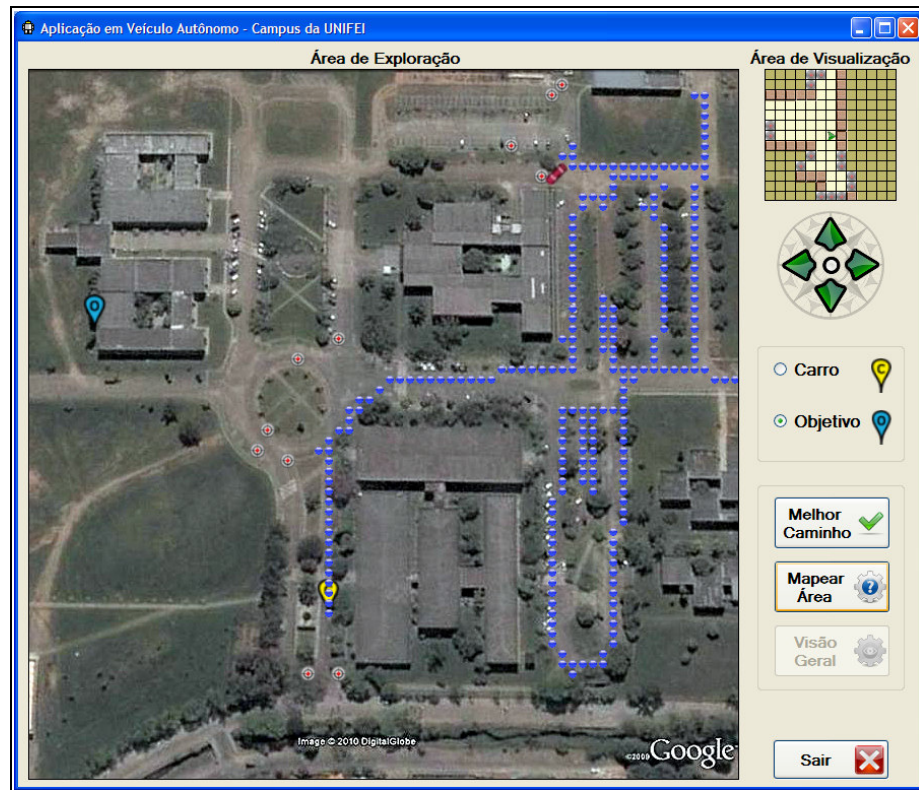


Figura 4.7 – Percorrendo o campus em busca do objetivo.

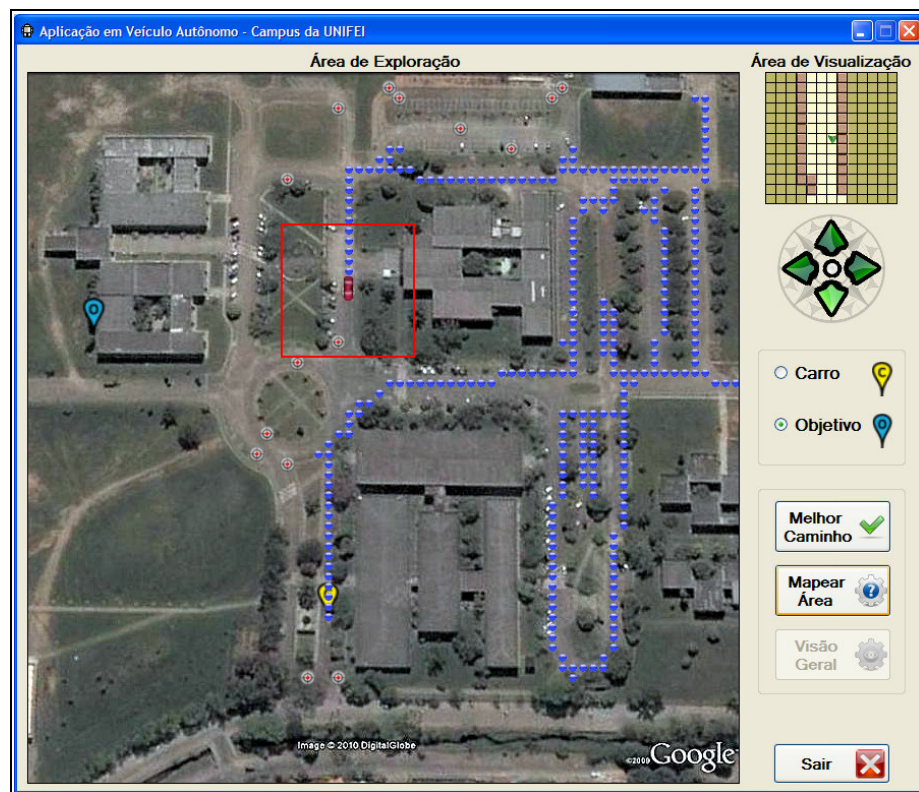


Figura 4.8 – Visualizando e mapeando o ambiente.

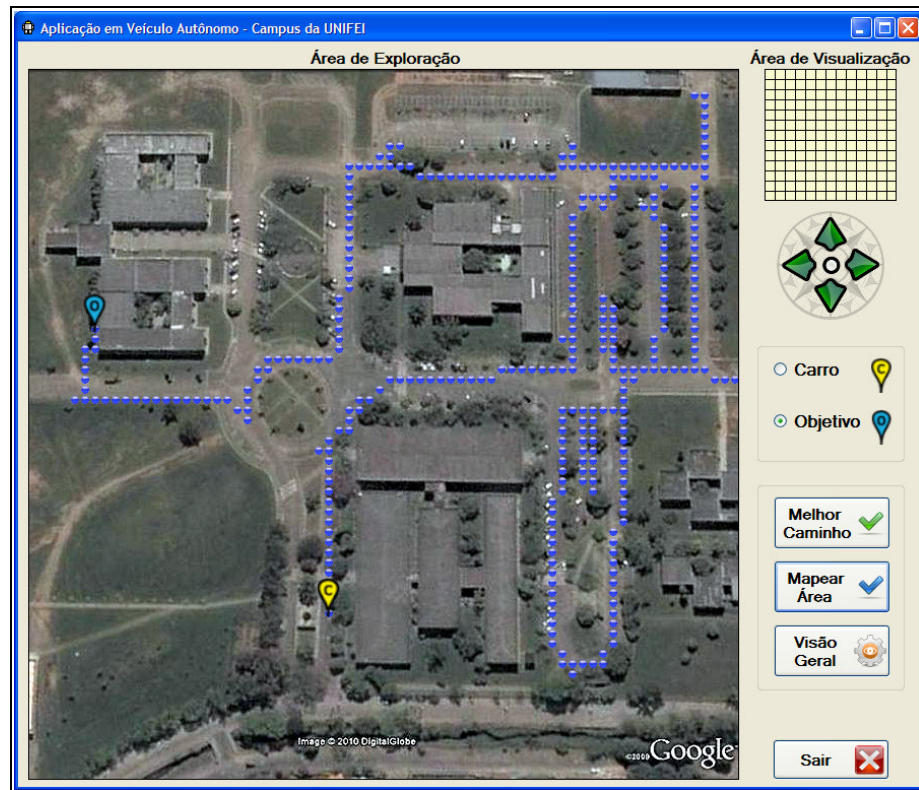


Figura 4.9 – O veículo autônomo chega ao seu objetivo.

Também é possível a comparação entre o percurso realizado durante o mapeamento do ambiente e o melhor caminho, conforme ilustra a figura 4.10.

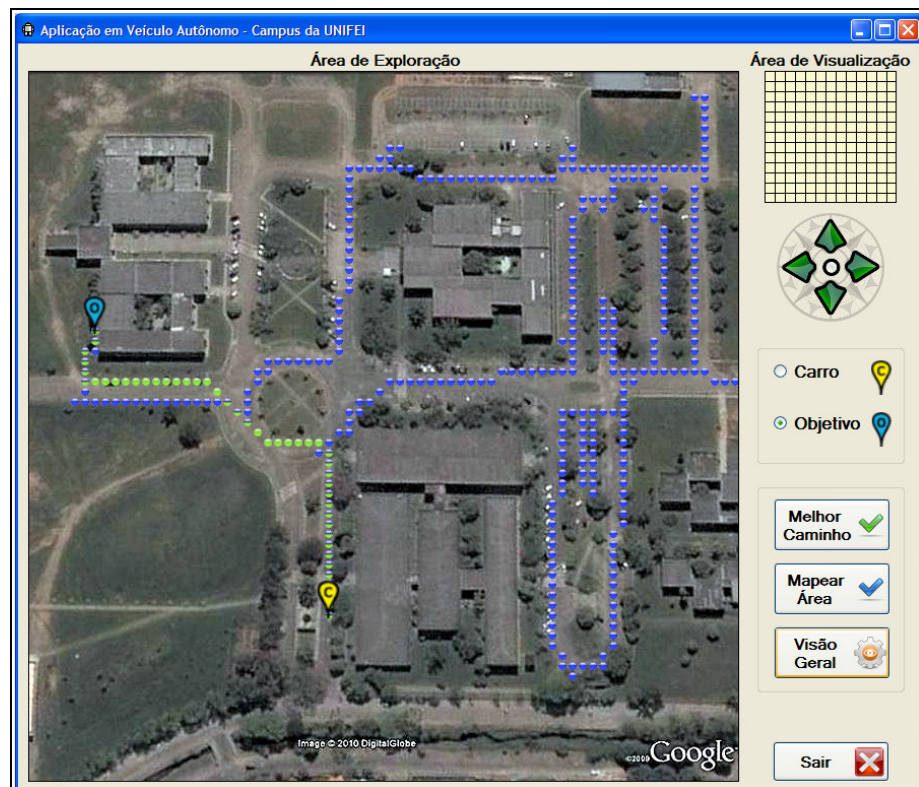


Figura 4.10 – O mapeamento do ambiente desconhecido e o melhor caminho simultaneamente.

5

CONCLUSÃO

Este capítulo apresenta as conclusões finais, os resultados alcançados e propostas para desenvolvimentos futuros.

5.1. CONSIDERAÇÕES FINAIS

O futuro da robótica móvel autônoma tem se mostrado bastante promissor, principalmente com o uso de técnicas cada vez mais avançadas de Inteligência Artificial e sofisticados sistemas de sensoriamento como, por exemplo, sonares, lasers, visão computacional, GPS, etc.

Nesta dissertação foram desenvolvidas estratégias com o intuito de habilitar um sistema robótico a realizar a exploração autônoma de ambientes desconhecidos. Para o robô alcançar um bom nível de autonomia é necessário que ele se locomova de forma inteligente dentro do ambiente em que está inserido. Assim foi implementado um sistema de mapeamento e navegação baseado em técnicas de Inteligência Artificial chamadas de Busca em Espaço de Estados. Conforme apresentado no levantamento bibliográfico, o estudo dos fundamentos que embasam estas técnicas apontou a Busca A* como a mais apropriada para o desenvolvimento do sistema. Esta definição foi ratificada através de uma análise comparativa entre métodos de busca, feita utilizando uma ferramenta de *software* desenvolvida pelo autor, que comprovou as características de completeza, otimização e de baixa complexidade de espaço e de tempo do algoritmo de busca A*.

O simulador utilizado para validar e testar as estratégias propostas foi implementado na linguagem de programação C#, da plataforma Microsoft Visual Studio 2008. A interface gráfica do sistema desenvolvido permite visualizar a construção do mapa à medida que o robô navega pelo ambiente, assim como mostra o sensoriamento que o robô realiza através da simulação de um sistema de visão computacional.

Para a representação do ambiente mapeado foi utilizada a técnica de grades de ocupação, desta forma as informações podem ser armazenadas em uma matriz de duas dimensões. Este método torna o sistema bastante genérico e pode ser empregado em diferentes tipos de sensores que os robôs utilizam para percepção do ambiente. Porém o tamanho da célula da grade de ocupação define um compromisso entre precisão, eficiência e custo computacional do sistema. Segundo Siegwart (2004, tradução minha), a complexidade da representação do mapa tem impacto direto na complexidade computacional do raciocínio sobre o mapeamento, a localização e a navegação.

Os experimentos realizados no simulador demonstraram que o sistema proposto atingiu plenamente os objetivos, atendendo aos requisitos levantados no início do trabalho. Pelos resultados obtidos o sistema mostrou-se confiável e eficiente, visto que, sem nenhum conhecimento prévio do ambiente e apenas com a informação da tarefa a ser realizada (procurar uma bola ou mapear todo o ambiente), em todos os experimentos realizados o robô foi capaz de alcançar o objetivo de modo autônomo, inteligente e seguro. Em nenhuma situação o robô ficou preso em um local ou colidiu com obstáculos.

O *software* cria uma animação gráfica que representa a aquisição de imagens do sistema de visão computacional, o mapeamento e a movimentação do robô no ambiente. Para melhor representar um sistema real o simulador utiliza instruções de atraso de tempo durante a apresentação da animação. Porém os tempos despendidos no mapeamento, que cria a representação interna do ambiente através dos dados recebidos do sistema de sensoriamento, na execução das rotinas de tomada de decisões, que definem as estratégias de exploração e fazem o planejamento de trajetória, e na geração dos comandos aos atuadores, que orientam a movimentação do robô, são bastante pequenos. A execução de uma chamada de

qualquer destes procedimentos é muito rápida e, por isso, não foi possível mensurar de forma precisa os tempos gastos individualmente. Para se ter um parâmetro de avaliação do tempo de execução das rotinas do sistema proposto, foi criado um cenário de teste, com toda a representação gráfica o robô demorou 15,5 minutos para percorrer e mapear todo o ambiente. No mesmo cenário, retirando-se toda a animação gráfica e as instruções de atraso, o tempo gasto para a apresentação do resultado final foi de 1,5 minuto. Através deste experimento foi possível concluir que os tempos gastos nas rotinas do sistema desenvolvido não irão impactar significativamente na performance de um sistema robótico real. O tempo de execução do programa não é tão crítico, se comparado ao tempo que o robô depende para se locomover e para fazer o sensoriamento do ambiente.

Por fim, pode-se comprovar que a simulação virtual é uma boa alternativa para se implementar, testar, aperfeiçoar e validar sistemas robóticos antes de realizar experimentos em um ambiente real. As simulações em computador agilizam o processo de desenvolvimento e minimizam os custos do sistema.

5.2. SUGESTÕES PARA TRABALHOS FUTUROS

A partir do sistema desenvolvido pode-se vislumbrar possibilidades de trabalhos futuros sob dois aspectos: no simulador desenvolvido para a validação do sistema e na aplicação das estratégias propostas em um sistema robótico real.

No que tange ao simulador, para tornar o mapeamento do ambiente mais eficiente, sugere-se a representação virtual de outros sensores que realizam a percepção do ambiente como, por exemplo, lasers, sonares, infravermelhos, GPSs, bússolas digitais, etc. Inclusive permitindo que o usuário do *software* possa equipar o robô virtual com os sistemas de sensoriamento que melhor lhe convier. Também propõe-se que no planejamento de trajetória sejam utilizados métodos para suavizar o caminho a ser percorrido, removendo vértices não necessários e produzindo um caminho mais realista. O sistema foi desenvolvido para ambientes estáticos, outra sugestão é capacitar o robô a interagir com elementos dinâmicos no ambiente,

mesmo que estes elementos não estejam representados no mapa construído pelo robô. Uma possibilidade para trabalhos futuros diz respeito à utilização de cenários extraídos do mundo real, no próprio simulador seria possível, por exemplo, capturar imagens acessando o Google Earth, um serviço *online* e gratuito que permite visualizar lugares, cidades, construções, etc. Também seria interessante fazer a representação do simulador em três dimensões, construindo um ambiente virtual 3D associado ao ambiente real. Por fim, robôs reais lidam com erros e imprecisões nas leituras dos sensores e no controle dos atuadores, assim uma proposta para a extensão deste trabalho é introduzir ruídos que tornem a representação do simulador mais fiel a um sistema real, em seguida deve-se incluir métodos que façam a avaliação e a correção destes erros.

A partir do estágio atual do sistema proposto ou incrementado com os aprimoramentos sugeridos acima, propõe-se a sua aplicação em um robô real. Acredita-se que, os resultados obtidos no simulador tornam possível e viável a implementação do sistema em um ambiente do mundo real.

A.2 - Método construtor, dimensionamento dos *arrays*.

```
public Pathfinding(int mapaLarg, int mapaAlt, int tamCelula, int numRobo)
{
    // Transforma a largura do mapa (em pixels) em número de posições.
    mapWidth = mapaLarg/tamCelula;
    // Transforma a altura do mapa (em pixels) em número de posições.
    mapHeight = mapaAlt/tamCelula;
    // Recebe o tamanho das células.
    tileSize = tamCelula;
    // Recebe o número de robôs que procuram o caminho.
    numberPeople = numRobo;
    // Ajusta a matriz de status da posição.
    walkability = new int[mapWidth, mapHeight];
    // Ajusta o vetor com os ID dos itens da lista aberta.
    openList = new int[mapWidth * mapHeight + 2];
    // Ajusta a matriz que registra se a célula está na lista aberta ou
    // na lista fechada.
    whichList = new int[mapWidth + 1, mapHeight + 1];
    // Ajusta o vetor da posição X de um item na lista aberta.
    openX = new int[mapWidth * mapHeight + 2];
    // Ajusta o vetor da posição Y de um item na lista aberta.
    openY = new int[mapWidth * mapHeight + 2];
    // Ajusta a matriz que fornecer o pai de cada célula (X).
    parentX = new int[mapWidth + 1, mapHeight + 1];
    // Ajusta a matriz que fornecer o pai de cada célula (Y).
    parentY = new int[mapWidth + 1, mapHeight + 1];
    // Ajusta o vetor do custo F de uma célula na lista aberta.
    Fcost = new int[mapWidth * mapHeight + 2];
    // Ajusta a matriz do custo G de uma célula na lista aberta.
    Gcost = new int[mapWidth + 1, mapHeight + 1];
    // Ajusta o vetor do custo H de uma célula na lista aberta.
    Hcost = new int[mapWidth * mapHeight + 2];
    // Ajusta o vetor do comprimento do path encontrado.
    pathLength = new int[numberPeople + 1];
    // Ajusta o vetor da posição corrente no caminho escolhido.
    pathLocation = new int[numberPeople + 1];
    // Ajusta a matriz das coordenadas do caminho encontrado.
    pathBank = new int[numberPeople + 1, 1, 1];
}
```

A.3 - Ordenação da lista aberta por *heap* binário.

```
.....
// Se a lista aberta não está vazia, retira a primeira célula da lista.
// Esta é a célula como o mais baixo custo de F.
if (numberOfOpenListItems != 0)
{
    Retira o primeiro item da lista aberta.
    parentXval = openX[openList[1]]; // Registra as coordenadas da célula
    parentYval = openY[openList[1]]; // do item.
    // Acrescenta o item à lista fechada.
    whichList[parentXval, parentYval] = onClosedList;
    // Lista Aberta = Binary Heap: Exclui este item da lista aberta, que
    // é mantida como um heap binário.
    // Reduz o número de itens da lista aberta de 1.
}
```

```

numberOfOpenListItems = numberOfOpenListItems - 1;
// Exclui o item no topo do heap binário e reordenar o heap, assim
// o item com menor custo de F é levado ao topo.
// Move o último item do heap para a posição 1.
openList[1] = openList[numberOfOpenListItems + 1];
v = 1;
// Repete a rotina a seguir até que o novo item na posição 1 desça
// para a sua posição no heap.
do
{
    u = v;
    // Se ambos os filhos existem.
    if (2 * u + 1 <= numberOfOpenListItems)
    {
        // Verifica se o custo F do pai é maior do que o de cada filho.
        // Seleciona o menor dos dois filhos.
        if (Fcost[openList[u]] >= Fcost[openList[2 * u]])
            v = 2 * u;
        if (Fcost[openList[v]] >= Fcost[openList[2 * u + 1]])
            v = 2 * u + 1;
    }
    else
    {
        // Se somente o filho 1 existe.
        if (2 * u <= numberOfOpenListItems)
        {
            // Verifica se o custo F do pai é maior do que o do filho 1.
            if (Fcost[openList[u]] >= Fcost[openList[2 * u]])
                v = 2 * u;
        }
    }
    // Se o custo F do pai é maior que um dos seus filhos eles são
    // trocados.
    if (u != v)
    {
        temp = openList[u];
        openList[u] = openList[v];
        openList[v] = temp;
    }
    // Caso contrário, sai do loop.
    else
        break;
}
while (true);

.....

```

A.4 - Evita que o robô atravesse o canto de um obstáculo.

```

.....
// Não permite atravessar o canto de um obstáculo.
corner = walkable;
if (a == parentXval - 1)
{
    if (b == parentYval - 1)
    {
        if (walkability[parentXval - 1, parentYval] == unwalkable ||
            walkability[parentXval, parentYval - 1] == unwalkable)

```

```

        corner = unwalkable;
    }
    else if (b == parentYval + 1)
    {
        if (walkability[parentXval, parentYval + 1] == unwalkable ||
            walkability[parentXval - 1, parentYval] == unwalkable)
            corner = unwalkable;
    }
}
else if (a == parentXval + 1)
{
    if (b == parentYval - 1)
    {
        if (walkability[parentXval, parentYval - 1] == unwalkable ||
            walkability[parentXval + 1, parentYval] == unwalkable)
            corner = unwalkable;
    }
    else if (b == parentYval + 1)
    {
        if (walkability[parentXval + 1, parentYval] == unwalkable ||
            walkability[parentXval, parentYval + 1] == unwalkable)
            corner = unwalkable;
    }
}
}
.....

```

A.5 - Cálculo dos custos F, G e H para um novo item da lista aberta.

```

.....
// Calcula o seu custo G.
if (Math.Abs(a - parentXval) == 1 && Math.Abs(b - parentYval) == 1)
    addedGCost = 14; // Custo de se movimentar na diagonal.
else
    addedGCost = 10; // Custo de não se movimentar na diagonal.
Gcost[a, b] = Gcost[parentXval, parentYval] + addedGCost;

// Calcula seu custo H. Distância Manhattan (horizontal + vertical).
Hcost[openList[m]] = 10*(Math.Abs(a - targetX) + Math.Abs(b - targetY));

// Calcula o seu custo F (custo G + custo H).
Fcost[openList[m]] = Gcost[a, b] + Hcost[openList[m]];

// Atribui as coordenadas do pai da célula corrente.
parentX[a, b] = parentXval;
parentY[a, b] = parentYval;
.....

```

A.6 - Resultado final da busca do melhor caminho.

```

.....
do
{
    // Se a lista aberta não está vazia, procura o melhor caminho.
}

```

```

if (numberOfOpenListItems != 0)
{
    // Código que procura o caminho mais curto.
    .....
}
// Se a lista aberta estiver vazia, então não há caminho.
else
{
    path = nonexistent;
    break;
}

// Se o objetivo é adicionado à lista aberta, então o melhor caminho
// foi encontrado.
if (whichList[targetX, targetY] == onOpenList)
{
    path = found;
    break;
}
}
while (true); // Executa esta rotina até que o caminho seja
              // encontrado ou considerado inexistente.
.....

```

A.7 - Método *LoadWalkability*.

```

public void LoadWalkability(int x, int y, int status)
{
    x = x / tileSize;
    y = y / tileSize;
    walkability[x, y] = status;
}

```

A.8 - Método *ReadPathLength*.

```

public int ReadPathLength(int pathfinderID)
{
    return pathLength[pathfinderID];
}

```

A.9 - Método *ReadPathX*.

```

public int ReadPathX(int pathfinderID, int pathLocation)
{
    return pathBank[pathfinderID, pathLocation, 0] * tileSize;
}

```

A.10 - Método *ReadPathY*.

```
public int ReadPathY(int pathfinderID, int pathLocation)
{
    return pathBank[pathfinderID, pathLocation, 1] * tileSize;
}
```

A.11 - Método *ReadFcost*.

```
public int ReadFcost(int arrayF)
{
    return Fcost[arrayF];
}
```

A.12 - Método *ReadGcost*.

```
public int ReadGcost(int arrayG_a, int arrayG_b)
{
    return Gcost[arrayG_a, arrayG_b];
}
```

A.13 - Método *ReadHcost*.

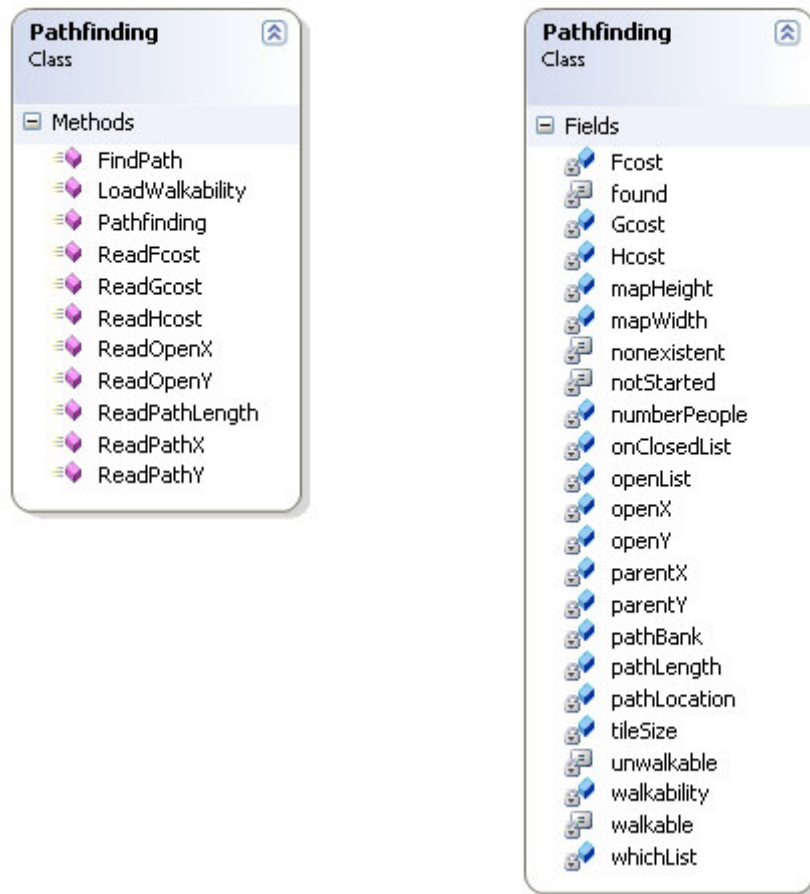
```
public int ReadHcost(int arrayH)
{
    return Hcost[arrayH];
}
```

A.14 - Método *ReadOpenX*.

```
public int ReadOpenX(int arrayX)
{
    return openX[arrayX];
}
```

A.15 - Método *ReadOpenY*.

```
public int ReadOpenY(int arrayY)
{
    return openY[arrayY];
}
```

A.16 - Diagrama de classe de *PathFinding.cs*.**A.17 - Parte do código dos eventos *MouseClicked* ou *MouseMove* na Área de Exploração.**

```
private void picImagem_MouseClick(object sender, MouseEventArgs e)
{
    .....
    int px = e.X; // Captura a posição onde ocorreu o evento MouseClick
    int py = e.Y; // ou MoveMove na Área de Exploração (em pixels).

    // Testa se estão dentro dos limites do componente picImagem.
    if (px <= 0 || px >= largura || py <= 0 || py >= altura)
        return;
    else
    {
        coordX = (int)px / celula; // Transforma as coordenadas para
        coordY = (int)py / celula; // uma posição na matriz mtzCenario.
    }

    .....

    // Chama o evento Paint do PictureBox picImagem.
    picImagem.Invalidate();
    // Chama o evento Paint do PictureBox picVisualizar.
    picVisualizar.Invalidate();
}
```

```

// Marca obstáculos na matriz que representa a Área de Exploração
// do robô (mtzCenario).
if (radObstaculo.Checked == true)
{
    // Se nesta posição estava o robô ele é retirado.
    if (mtzCenario[coordX, coordY] == inicio)
        robo = false;
    // Se nesta posição estava a bola ela é retirada.
    else if (mtzCenario[coordX, coordY] == objetivo)
        bola = false;

    // Armazena a informação de obstáculo na matriz mtzCenario.
    mtzCenario[coordX, coordY] = obstaculo;

    // Atualiza a imagem colocando um obstáculo na posição onde houve
    // o evento.
    picImagem.Invalidate(new Rectangle(coordX * celula, coordY * celula,
                                        celula, celula));
}

// Posiciona o robô na matriz que representa a Área de Exploração.
else if (radRobo.Checked == true)
{
    .....
}

```

A.18 - Parte do código do evento *Click* do botão “Limpar”.

```

private void btnLimpar_Click(object sender, EventArgs e)
{
    .....

    // Limpa as matrizes relacionadas a Área de Exploração do robô.
    for (int i = 0; i < larg; i++)
    {
        for (int j = 0; j < alt; j++)
        {
            mtzCenario[i, j] = 0;
            mtzOculata[i, j] = 0;
        }
    }

    // Limpa as matrizes relacionadas a Área de Visualização do robô.
    for (int i = 0; i < largVis; i++)
    {
        for (int j = 0; j < altVis; j++)
        {
            mtzVisual[i, j] = 0;
            mtzVisOculata[i, j] = 0;
        }
    }

    // Limpa a lista de vetores que recebe os caminhos percorridos no
    // mapeamento do ambiente.
    CaminhosBusca.Clear();

    .....
}

```

A.19 - Parte do código do evento *Click* do botão “Abrir”.

```

private void bntAbrir_Click(object sender, EventArgs e)
{
    // Determina o diretório inicial onde será aberto o arquivo do cenário.
    ofdAbrir.InitialDirectory = Application.StartupPath;
    // Coloca um título na caixa de diálogo.
    ofdAbrir.Title = "Abrir o cenário";
    // Determina o tipo de arquivo que pode ser aberto e que será mostrado
    // na caixa de diálogo.
    ofdAbrir.Filter = "Arquivo de Cenário|*.cena";

    // Exibe a caixa de diálogo de abrir arquivo, retorna o caminho e o
    // nome do arquivo selecionado pelo usuário.
    if (ofdAbrir.ShowDialog(this) == DialogResult.OK)
    {
        // Se o nome do arquivo não estiver vazio.
        if (ofdAbrir.FileName != "")
        {
            // Limpa a Área de Exploração, chamando o código do evento click
            // do botão btnLimpar.
            btnLimpar_Click(sender, e);

            // Classe utilizada para abrir um arquivo, com o nome do arquivo,
            // a operação e o tipo de acesso.
            FileStream arq = new FileStream(ofdAbrir.FileName,
                                           FileMode.Open, FileAccess.Read);

            // Lê no arquivo o estado de cada célula do cenário.
            for (int y = 0; y < larg; y++)
                for (int x = 0; x < alt; x++)
                    mtzCenario[x, y] = arq.ReadByte();

            // Se foi inserido o robô no cenário, retorna do arquivo a sua
            // posição.
            if (arq.ReadByte() == 1)
            {
                robo = true; // Informa que tem um robô no cenário.
                inicioX = arq.ReadByte() * celula; // Coordenada X do robô.
                inicioY = arq.ReadByte() * celula; // Coordenada Y do robô.
            }
            else
                robo = false; // Informa que não tem um robô no cenário.

            // Se foi inserido a bola no cenário, retorna do arquivo a sua
            // posição.

            .....

            arq.Close(); // Fecha a Stream do arquivo.
            arq.Dispose(); // Libera todos os recursos utilizados pela Stream.
        }
    }
    // Chama o evento Paint do PictureBox picImagem.
    picImagem.Invalidate();
    // Chama o evento Paint do PictureBox picVisualizar.
    picVisualizar.Invalidate();
}

```

A.20 - Parte do código do evento *Click* do botão “Salvar”.

```

private void btnSalvar_Click(object sender, EventArgs e)
{
    // Determina o diretório inicial onde será salvo o arquivo do cenário.
    sfdSalvar.InitialDirectory = Application.StartupPath;
    // Coloca um título na caixa de diálogo.
    sfdSalvar.Title = "Salvar o cenário";
    // Determina o tipo de arquivo que pode ser salvo e que será mostrado
    // na caixa de diálogo.
    sfdSalvar.Filter = "Arquivo de Cenário|*.cena";

    // Exibe a caixa de diálogo de salvar arquivo, retorna o caminho e
    // o nome do arquivo escolhido pelo usuário.
    if (sfdSalvar.ShowDialog(this) == DialogResult.OK)
    {
        // Se o nome do arquivo não estiver vazio.
        if (sfdSalvar.FileName != "")
        {
            // Classe utilizada para criar um arquivo, com o nome do arquivo,
            // a operação e o tipo de acesso.

            FileStream arq = new FileStream(sfdSalvar.FileName,
                                           FileMode.Create, FileAccess.Write);

            // Escreve no arquivo o estado de cada célula do cenário.
            for (int y = 0; y < larg; y++)
                for (int x = 0; x < alt; x++)
                    arq.WriteByte((byte)mtzCenario[x, y]);

            // Se foi inserido o robô no cenário, escreve a sua posição.
            if (robo == true)
            {
                arq.WriteByte(1); // Informa que tem um robô no cenário.
                arq.WriteByte((byte)(inicioX / celula)); // Coord. X do robô.
                arq.WriteByte((byte)(inicioY / celula)); // Coord. Y do robô.
            }
            else
                arq.WriteByte(0); // Informa que não tem um robô no cenário.

            // Se foi inserido a bola no cenário, escreve a sua posição.

            .....

            arq.Close(); // Fecha a Stream do arquivo.
            arq.Dispose(); // Libera todos os recursos utilizados pela Stream.
        }
    }
    // Chama o evento Paint do PictureBox picImagem.
    picImagem.Invalidate();
    // Chama o evento Paint do PictureBox picVisualizar.
    picVisualizar.Invalidate();
}

```

A.21 - Cálculo e leitura do melhor caminho entre o robô e a bola.

```

.....

// Faz a busca do melhor caminho para o objetivo (bola).
Pathfinding PathRobo = new Pathfinding(largura, altura, celula, numRobo);

```

```

.....
// Testa se o robô foi posicionado no cenário.
if (robo == true)
{
    // Testa se a bola foi posicionada no cenário.
    if (bola == true)
    {
        .....

        // Carrega o cenário para o pathfinding, definindo os pontos livres e
        // de obstáculo, através do método LoadWalkability da classe
        // Pathfinding, para o objeto PathRobo.
        for (int i = 0; i < larg; i++)
        {
            for (int j = 0; j < alt; j++)
            {
                if (mtzCenario[i, j] == livre)
                    PathRobo.LoadWalkability(i * celula, j * celula, livre);
                else if (mtzCenario[i, j] == obstaculo)
                    PathRobo.LoadWalkability(i * celula, j * celula, obstaculo);
            }
        }
        // Calcula o melhor caminho entre o ponto de origem do robô e o seu
        // objetivo (bola).
        pathMelhor = PathRobo.FindPath(numRobo, inicioX, inicioY,
                                       objetivoX, objetivoY);

        // Se o melhor caminho foi encontrado armazena as informações
        // relacionadas a este caminho.
        if (pathMelhor == encontrado)
        {
            .....

            // Recebe o comprimento do caminho encontrado.
            pathComprimento = PathRobo.ReadPathLength(1);
            // Dimensiona a matriz que irá armazenar as coordenadas do caminho
            // encontrado.
            pathDados = new int[numRobo + 1, pathComprimento, 2];
            // Lê as coordenadas X e Y do caminho encontrado e armazena na matriz
            // pathDados.
            for (int i = 0; i < pathComprimento; i++)
            {
                pathDadosX = PathRobo.ReadPathX(numRobo, i);
                pathDadosY = PathRobo.ReadPathY(numRobo, i);
                pathDados[numRobo, i, 0] = pathDadosX;
                pathDados[numRobo, i, 1] = pathDadosY;
            }
        }
        .....

```

A.22 - Leitura dos custos F, G e H do *pathfinding* realizado.

```

.....
// Lê os custos F, G e H de todas as posições que foram verificadas
// pelo algoritmo de pathfinding.
for (int i = 0; i < larg; i++)
{
    for (int j = 0; j < alt; j++)

```

```

    {
        custoG[i, j] = PathRobo.ReadGcost(i, j);
        custoF[f] = PathRobo.ReadFcost(f);
        custoH[f] = PathRobo.ReadHcost(f);
        abertoX[f] = PathRobo.ReadOpenX(f);
        abertoY[f] = PathRobo.ReadOpenY(f);
        f++;
    }
}

.....

```

A.23 - Animação do caminho percorrido pelo robô.

```

private void tmrMelhorCaminho_Tick(object sender, EventArgs e)
{
    // Cria um objeto gráfico que desenha em tempo real na superfície do
    // PictureBox picImagem, porém a imagem não é persistente.
    Graphics ge = picImagem.CreateGraphics();

    // Faz a simulação do robô se deslocando da origem até o objetivo.
    if (pathMelhor == encontrado)
    {
        int k;
        // Mostra na Área de Exploração do robô o caminho encontrado.
        for (k = 0; k < pathDados.GetLength(1) - 1; k++)
        {
            // Tempo que o robô gasta para se deslocar por uma célula.
            Thread.Sleep(100);
            // Desenha o caminho percorrido pelo robô.
            ge.DrawImage(imgEstrelaMelhor, pathDados[numRobo, k, 0],
                        pathDados[numRobo, k, 1]);
        }
        // Mostra nesta célula do PictureBox a imagem de que o robô chegou
        // ao seu objetivo.
        ge.DrawImage(imgRoBola, pathDados[numRobo, k, 0],
                    pathDados[numRobo, k, 1]);
        // Indicação sonora de chegada ao objetivo.
        SystemSounds.Asterisk.Play();
    }

    tmrMelhorCaminho.Enabled = false; // Desabilita este timer.
    // Libera todos os recursos utilizados pelos objetos gráficos.
    ge.Dispose();
}

```

A.24 - Função ProcuraAlvo.

```

private void ProcuraAlvo()
{
    bolaOK = false; // Define inicialmente o objetivo como não encontrado.

    // Cria um objeto gráfico que desenha na superfície do PictureBox.
    Graphics gp = picImagem.CreateGraphics();
    // Limpa o PictureBox picImagem e coloca o fundo escuro.
    gp.Clear(Color.DarkKhaki);
}

```

```

.....
// Carrega uma matriz, nas dimensões que representam a Área de
// Exploração, como desconhecida e posiciona o robô nesta matriz.
for (int i = 0; i < larg; i++)
{
    for (int j = 0; j < alt; j++)
    {
        // Define todas as posições da matriz do mapeamento da Área de
        // Exploração como desconhecido.
        mtzOculta[i, j] = desconhecido;
        // Posiciona o robô na matriz de mapeamento e no PictureBox da
        // Área de Exploração.
        if (mtzCenario[i, j] == inicio) // Posição do robô no cenário.
        {
            mtzOculta[i, j] = inicio;
            roboX = i * celula;
            roboY = j * celula;
            gp.FillRectangle(ye, roboX, roboY, celula - 1, celula - 1);
            gp.DrawImage(imgRobo, roboX, roboY);
        }
    }
}
// Libera todos os recursos utilizados pelos objetos gráficos.
gp.Dispose();
// Chama a função Visualizar e passa a posição corrente do robô.
Visualizar(roboX, roboY);
}

```

A.25 - Carrega valores iniciais em matrizes que representam a Área de Visualização.

```

.....
for (int i = 0; i < largVis; i++)
{
    for (int j = 0; j < altVis; j++)
    {
        x0 = i - 6 + roboX / celula;
        y0 = j - 6 + roboY / celula;

        // Carrega a posição da matriz de visão do robô como desconhecida.
        mtzVisOculta[i, j] = desconhecido;

        // Testa se a célula a ser copiada para a matriz do campo de visão
        // do robô está fora dos limites da Área de Exploração.
        if (x0 < 0 || x0 >= larg || y0 < 0 || y0 >= alt)
        {
            // Carrega a posição da matriz como fora dos limites (valor 5).
            mtzVisual[i, j] = fora;
        }
        else
        {
            // Copia parte da Área de Exploração para dentro da matriz do
            // campo de visão do robô.
            mtzVisual[i, j] = mtzCenario[x0, y0];
            // Se a posição copiada é onde está o robô.
            if (mtzVisual[i, j] == inicio)

```

```

    {
        // Coloca o robô no centro da matriz da visualização.
        mtzVisOculta[i, j] = inicio;
        roboVisuX = i * celula;
        roboVisuY = j * celula;
        centroX = roboVisuX + 10;
        centroY = roboVisuY + 10;
        // Coloca no centro de picVisualizar a imagem indicando que
        // o robô irá iniciar a visualização pela direção leste.
        gv.FillRectangle(ye, roboVisuX, roboVisuY, celula - 1,
                        celula - 1);
        gv.DrawImage(imgLeste, roboVisuX, roboVisuY);
    }
}
}
}
.....

```

A.26 - Cálculo de um Ponto de Verificação.

```

.....
// Faz a visualização do robô nas 4 direções(leste, norte, oeste e sul).
foreach (int o in orientacao)
{
    // Faz um atraso de 1,5 segundos na mudança da direção de visualização.
    Thread.Sleep(1500);
    // Mostra nos PictureBoxes picVisualizar e picBussola em que direção
    // o robô está visualizando.
    gv.FillRectangle(ye, roboVisuX, roboVisuY, celula - 1, celula - 1);
    if (o == leste)
    {
        gv.DrawImage(imgLeste, roboVisuX, roboVisuY);
        gb.DrawImage(imgBusLeste, 0, 0, 110, 110);
    }
    else if (o == norte)
    {
        .....
        // Visualiza um ângulo de visão de 90° dentro de cada orientação.
        graus = 0;
        // Faz a varredura da Área de Visualização com intervalos de 1 grau.
        while (graus > -45)
        {
            if (graus >= 0)
            {
                graus += 1;
                if (graus > 45)
                    graus = -1;
            }
            else
                graus -= 1;

            pX = centroX; // Coordenadas X e Y do centro da Área de
            pY = centroY; // de Visualização, em pixels.
            cX = centroX / celula; // Coordenadas X e Y do centro da matriz
            cY = centroY / celula; // da Área de Visualização.
            dX = 0; // Mede o deslocamento no eixo X na Área de Visualização.
            dY = 0; // Mede o deslocamento no eixo Y na Área de Visualização.

```



```

        PathVisao.LoadWalkability(i * celula, j * celula, obstaculo);
    }
}

// Verifica se existe um caminho entre o ponto onde está o robô e
// a área livre ou o objetivo visualizado.
pathLivre = PathVisao.FindPath(numRobo, roboVisuX, roboVisuX,
                               x * celula, y * celula);

.....

```

A.28 - Procedimentos após a visualização de uma célula.

```

.....

// Define na matriz de mapeamento da Área de Exploração que esta posição
// é uma área livre e mostra isto no PictureBox de visualização.
if (mtzVisual[x, y] == livre)
{
    mtzOculto[x0, y0] = livre;
    gv.FillRectangle(ye, x * celula, y * celula, celula - 1, celula - 1);
}

// Define na matriz de mapeamento da Área de Exploração que nesta posição
// está o objetivo (bola) e mostra isto no PictureBox de visualização.
else if (mtzVisual[x, y] == objetivo)
{
    mtzVisOculto[x, y] = objetivo;
    mtzOculto[x0, y0] = objetivo;
    gv.FillRectangle(ye, x * celula, y * celula, celula - 1, celula - 1);
    gv.DrawImage(imgBola, x * celula, y * celula);
    bolaOK = true; // Bola (objetivo) encontrado.
}

.....

// Define na matriz de mapeamento da Área de Exploração que esta posição
// é um obstáculo e mostra isto no PictureBox de visualização.
else if (mtzVisual[x, y] == obstaculo)
{
    mtzVisOculto[x, y] = obstaculo;
    mtzOculto[x0, y0] = obstaculo;
    gv.DrawImage(imgObstaculo, x * celula, y * celula);
    gv.DrawRectangle(p, x * celula - 1, y * celula - 1, celula, celula);
    break; // Interrompe a visualização neste ângulo.
}

// Define na matriz das áreas visualizadas pelo robô que a posição está
// fora da Área de Exploração e mostra isto na Área de Visualização.
else if (mtzVisual[x, y] == fora)
{
    mtzVisOculto[x, y] = fora;
    gv.FillRectangle(cz, x * celula, y * celula, celula - 1, celula - 1);
    break; // Interrompe a visualização neste ângulo.
}

.....

```

A.29 - Marca pontos de busca no perímetro da Área de Visualização.

```

.....
// Cria pontos de busca no perímetro da Área de Visualização do robô.
while (cont <= tamPeri)
{
    // Percorre o perímetro da Área de Visualização do robô.
    if (cont >= 0 && cont < tamLateral)
        x++;
    else if (cont >= tamLateral && cont < tamLateral * 2)
        y++;
    else if (cont >= tamLateral * 2 && cont < tamLateral * 3)
        x--;
    else if (cont >= tamLateral * 3 && cont < tamLateral * 4)
        y--;
    // Carrega uma matriz com as coordenadas XY do perímetro da visão
    // do robô.
    if (cont < tamPeri)
    {
        mtzPeri[cont, 0] = x;
        mtzPeri[cont, 1] = y;
    }

    // Mede a extensão da área livre contígua no perímetro da visão do
    // robô. Marca um ponto para uma nova busca no centro do perímetro
    // livre. Se este perímetro livre for maior que um dos lados da
    // matriz de visão do robô marca mais de um ponto.
    if (mtzVisOculata[x, y]==livre && cont<tamPeri && medePeri<tamLateral)
    {
        // Marca um novo ponto de busca na primeira célula livre do perímetro
        // e inicia a contagem da extensão da área livre contígua.
        if (periMatriz == false)
        {
            celuIni = cont;
            periMatriz = true;
            // Marca na matriz de visão do robô um novo ponto de busca.
            mtzVisOculata[mtzPeri[cont, 0], mtzPeri[cont, 1]] = alvoBusca;
            // Marca na matriz de exploração do robô um novo ponto de busca.
            x0 = mtzPeri[cont, 0] - 6 + roboX / celula;
            y0 = mtzPeri[cont, 1] - 6 + roboY / celula;
            mtzOculata[x0, y0] = alvoBusca;
        }
        medePeri++; // Mede a extensão da área livre contígua.
    }
    else
    {
        celuFim = cont - 1;
        if (periMatriz == true)
        {
            // Marca um ponto no meio do perímetro de espaços livres contíguos.
            alvo = celuIni + (celuFim - celuIni) / 2;
            periMatriz = false;
            // Marca na matriz de visão do robô um novo ponto de busca.
            mtzVisOculata[mtzPeri[alvo, 0], mtzPeri[alvo, 1]] = alvoBusca;
            // Marca na matriz de exploração do robô um novo ponto de busca.
            x0 = mtzPeri[alvo, 0] - 6 + roboX / celula;
            y0 = mtzPeri[alvo, 1] - 6 + roboY / celula;
            mtzOculata[x0, y0] = alvoBusca;
        }
        medePeri = 0;
    }
}

```

```

    }
    cont++;
}
.....

```

A.30 - Marca pontos de busca dentro do campo de visão do robô.

```

.....
// Cria pontos de busca adjacentes a áreas desconhecidas dentro do campo
// de visão do robô. Isto acontece quando a visão do robô está obstruída
// e não alcança seus limites.
for (i = 0; i < largVis; i++)
{
    for (j = 0; j < altVis; j++)
    {
        // Marca um ponto de busca em toda célula livre perpendicular
        // adjacente a uma área desconhecida.
        if (mtzVisOculata[i, j] == desconhecido)
        {
            // Cria um ponto de busca a direita da área desconhecida.
            try
            {
                if (mtzVisOculata[i + 1, j] == livre)
                {
                    mtzVisOculata[i + 1, j] = alvoBusca;
                    xO = i - 5 + roboX / celula;
                    yO = j - 6 + roboY / celula;
                    mtzOculata[xO, yO] = alvoBusca;
                }
            }
            catch
            { }
            // Cria um ponto de busca a esquerda da área desconhecida.
            try
            {
                .....
            }
        }
    }
}

```

A.31 - Elimina pontos de busca que não estão adjacentes a uma área desconhecida.

```

.....
// Elimina possíveis pontos de busca que não estão adjacentes a uma
// área desconhecida. A medida que o robô mapeia a Área de Exploração
// um antigo ponto de busca pode ficar no meio de uma região conhecida
// deixando de ser uma opção interessante para buscas futuras.
for (i = 0; i < larg; i++)
{
    for (j = 0; j < alt; j++)
    {
        if (mtzOculata[i, j] == alvoBusca)
            if (!((i + 1 < larg && mtzOculata[i + 1, j] == desconhecido) ||
                (j + 1 < alt && mtzOculata[i, j + 1] == desconhecido) ||
                (i - 1 >= 0 && mtzOculata[i - 1, j] == desconhecido) ||
                (j - 1 >= 0 && mtzOculata[i, j - 1] == desconhecido)))
            {
                .....
            }
    }
}

```

```

        mtzOcultas[i, j] = livre;
    }
}
.....

```

A.32 - Elimina pontos de busca perpendiculares adjacentes a outros pontos de busca.

```

.....
// Elimina pontos de busca contíguos a outros pontos de busca. Para
// tornar mais leve o algoritmo de tomada de decisão que irá escolher
// o próximo ponto de busca, são eliminados pontos de busca
// perpendiculares adjacentes, permanecendo apenas 1.
for (i = 0; i < larg; i++)
{
    for (j = 0; j < alt; j++)
    {
        if (mtzOcultas[i, j] == alvoBusca)
        {
            if ((i + 1 < larg && mtzOcultas[i + 1, j] == alvoBusca) ||
                (j + 1 < alt && mtzOcultas[i, j + 1] == alvoBusca) ||
                (i - 1 >= 0 && mtzOcultas[i - 1, j] == alvoBusca) ||
                (j - 1 >= 0 && mtzOcultas[i, j - 1] == alvoBusca))
                mtzOcultas[i, j] = livre;
        }
    }
}
.....

```

A.33 - Determina o próximo ponto de busca para onde o robô deve se deslocar.

```

.....
// Define para qual ponto de busca o custo do deslocamento do robô é
// menor, se o objetivo não foi encontrado.
for (i = 0; i < larg; i++)
{
    for (j = 0; j < alt; j++)
    {
        // Calcula o custo F do deslocamento do robô até um ponto de busca.
        if (bolaOK == false)
        {
            if (mtzOcultas[i, j] == alvoBusca)
            {
                // Calcula o Pathfinding entre o ponto onde se encontra o robô
                // e o ponto de busca que está sendo verificado.
                pathPto = PathPonto.FindPath(numRobo, roboX, roboY,
                                                i * 20, j * 20);

                if (pathPto == encontrado)
                {
                    // Lê o custo F para o ponto de busca. Como F=G+H e no ponto de
                    // objetivo do pathfinding H=0 então F=G. Com as coordenadas X

```

```

        // e Y do ponto de busca é mais fácil obter o valor de G.
        aux = PathPonto.ReadGcost(i, j);
        // Compara o custo F do ponto de busca verificado com o melhor
        // ponto já encontrado.
        if (aux < melhorPonto)
        {
            melhorPonto = aux;
            // Armazena o ponto para onde o robô deve se deslocar.
            buscaX = i * 20;
            buscaY = j * 20;
        }
    }
}
}
}
.....

```

A.34 - Atualização da imagem da Área de Exploração do robô.

```

.....
// Atualiza a imagem da Área de Exploração do robô, o PictureBox
// picImagem.
for (i = 0; i < larg; i++)
{
    for (j = 0; j < alt; j++)
    {
        // Mostra nesta célula do PictureBox uma imagem de área livre.
        if (mtzOculata[i, j] == livre)
            gp.FillRectangle(ye, i*celula, j*celula, celula-1, celula-1);
        // Mostra nesta célula do PictureBox uma imagem de obstáculo.
        .....
        // Mostra nesta célula do PictureBox a imagem de uma estrela,
        // que representa o caminho percorrido.
        else if (mtzOculata[i, j] == caminho)
            gp.DrawImage(imgEstrelaBusca, i * celula, j * celula);
        // Mostra nesta célula do PictureBox a imagem do robô com uma
        // estrela sobreposta, que representa um ponto onde o robô
        // realizou a visualização da Área de Exploração.
        else if (mtzOculata[i, j] == pontoVisao)
            gp.DrawImage(imgRoboEstr, i * celula, j * celula);
    }
}
.....

```

A.35 - Classe *caminhos.cs*.

```

// Registra um caminho percorrido pelo robô no mapeamento do ambiente.
class Caminhos
{
    public string caminho;
    public int x, y;
}

```

```

// Recebe o nome e as coordenadas X e Y ao longo do caminho.
public Caminhos(string nCaminho, int nx, int ny)
{
    caminho = nCaminho;
    x = nx;
    y = ny;
}
}

```

A.36 - Armazena os caminhos percorridos pelo robô.

```

.....
// Se ainda existir pontos de busca não verificados ou se a bola foi
// encontrada.
if (numPtoBusca != 0 || bolaOK == true)
{
    // Calcula o Pathfinding entre o ponto onde se encontra o robô e o
    // novo ponto de busca selecionado ou até o seu objetivo (bola).
    pathPto = PathPonto.FindPath(numRobo, roboX, roboY, buscaX, buscaY);

    // Carrega a lista de vetores que armazena os caminhos percorridos
    // no mapeamento do ambiente.
    if (pathPto == encontrado)
    {
        // Testa se o ArrayList CaminhosBusca está vazio e armazena o
        // ponto inicial do robô.
        if (CaminhosBusca.Count == 0)
        {
            // Inicia o caminho percorrido pelo robô na busca do objetivo.
            Caminhos caminhos = new Caminhos("PtoInicial", roboX, roboY);
            // Adiciona o ponto inicial no ArrayList.
            CaminhosBusca.Add(caminhos);
        }

        // Carrega, na matriz original, o ponto onde estava o robô como
        // área livre.
        mtzCenario[roboX / celula, roboY / celula] = livre;

        // Posiciona o robô no ponto de busca selecionado ou no objetivo.
        roboX = buscaX;
        roboY = buscaY;

        // Posiciona, na matriz original, o robô em seu novo local de
        // visualização ou no objetivo.
        mtzCenario[roboX / celula, roboY / celula] = inicio;

        // Faz um atraso de 1,5 segundos antes do robô se deslocar para um
        // novo ponto.
        Thread.Sleep(1500);

        // Recebe o comprimento do caminho encontrado.
        pathComprimento = PathPonto.ReadPathLength(1);
        // Lê as coordenadas X e Y do caminho encontrado e armazena no
        // ArrayList CaminhosBusca.
        for (i = 0; i < pathComprimento; i++)
        {
            pathDadosX = PathPonto.ReadPathX(numRobo, i);
            pathDadosY = PathPonto.ReadPathY(numRobo, i);
        }
    }
}

```

```

        // Armazena o ponto inicial de um novo caminho percorrido pelo
        // robô no mapeamento do ambiente.
        if (i == pathComprimento - 1)
            NomeCaminho = "NovoCaminho";
        // Armazena cada célula do deslocamento do robô até um novo ponto
        // de busca ou o objetivo.
        else
            NomeCaminho = "Caminho " + contBusca;

        // Cria um objeto com o nome e as coordenadas do caminho.
        Caminhos caminhos = new Caminhos(NomeCaminho, pathDadosX,
                                           pathDadosY);

        // Adiciona o ponto no ArrayList.
        CaminhosBusca.Add(caminhos);
    }
    .....

```

A.37 - Mostra o caminho percorrido pelo robô até um novo ponto de busca ou o objetivo.

```

    .....
    // Mostra, na Área de Exploração, o caminho percorrido pelo robô até um
    // novo ponto de busca ou o objetivo.
    for (k = CaminhosBusca.Count - pathComprimento;
         k < CaminhosBusca.Count - 1; k++)
    {
        // Atraso que simula o deslocamento do robô de uma célula para outra.
        Thread.Sleep(150);
        // Mostra na célula do PictureBox a imagem de uma estrela, que
        // representa o caminho percorrido.
        gp.DrawImage(imgEstrelaBusca, ((Caminhos)CaminhosBusca[k]).x,
                                           ((Caminhos)CaminhosBusca[k]).y);
        // Registra o caminho percorrido pelo robô na matriz de mapeamento
        // da Área de Exploração.
        mtzOculta[((Caminhos)CaminhosBusca[k]).x / celula,
                  ((Caminhos)CaminhosBusca[k]).y / celula] = caminho;
    }
    .....

```

A.38 - Se o objetivo não foi encontrado retorna à função *Visualizar*.

```

    .....
    // Se o objetivo ainda não foi encontrado, mostra quando o robô chega
    // ao novo ponto de visualização.
    if (bolaOK == false)
    {
        // Mostra neste PictureBox uma imagem que representa o robô em um
        // novo ponto de visualização.
        gp.DrawImage(imgRoboEstr, ((Caminhos)CaminhosBusca[k]).x,
                                           ((Caminhos)CaminhosBusca[k]).y);
        // Registra o ponto para onde o robô se deslocou para fazer uma
        // nova visualização.
        mtzOculta[((Caminhos)CaminhosBusca[k]).x / celula,

```

```

        ((Caminhos)CaminhosBusca[k]).y / celula] = pontoVisao;
// Faz o atraso de 1 seg. antes do robô iniciar uma nova visualização.
Thread.Sleep(1000);
// Chama a função Visualizar, para continuar o mapeamento.
Visualizar(roboX, roboY);
}
.....

```

A.39 - Se o objetivo foi encontrado encerra o mapeamento do ambiente.

```

.....
// Se o objetivo foi encontrado, mostra quando o robô chega a seu
// objetivo (bola).
else
{
    // Mostra nesta célula do PictureBox a imagem de que o robô chegou
    // ao seu objetivo.
    gp.DrawImage(imgRoBola, ((Caminhos)CaminhosBusca[k]).x,
                  ((Caminhos)CaminhosBusca[k]).y);

    // Registra que o mapeamento da Área de Exploração já foi realizado.
    pathBusca = encontrado;
    // Se o pathfinding do melhor caminho já foi realizado habilita
    // o botão "Visão Geral".
    if (pathMelhor == encontrado)
        btnGeral.Enabled = true;

    // Indicação sonora de que a busca foi bem sucedida.
    SystemSounds.Asterisk.Play();
}
.....

```

A.40 - Informa que o objetivo não foi encontrado e encerra o mapeamento do ambiente.

```

.....
// Se o número de possíveis pontos de busca é igual a zero o programa
// informa que não foi encontrado o objetivo (bola) e encerra a busca.
else
{
    // Mensagem informando que não foi possível encontrar o objetivo.
    MessageBox.Show("Não foi encontrada a bola!!!", "Atenção!!!",
                   MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
}

// Recoloca o robô e a bola em seus pontos iniciais.
mtzCenario[roboX / celula, roboY / celula] = livre;
mtzCenario[inicioX / celula, inicioY / celula] = inicio;
if (bola == true)
{
    mtzCenario[objetivoX / celula, objetivoY / celula] = objetivo;
}

```

```
// Registra que o mapeamento da Área de Exploração já foi realizado.
pathBusca = encontrado;

// Finalizado o mapeamento da Área de Exploração, muda a figura do
// botão "Mapear Área".
btnMapear.Image = imgBuscaOK;

.....
```

A.41 - Atualiza a Área de Exploração quando na formação do cenário.

```
.....

// Cria um objeto que contém métodos relacionados a elementos gráficos.
Graphics ge = e.Graphics;
// Limpa o PictureBox picImagem e coloca o fundo amarelo.
ge.Clear(Color.LightGoldenrodYellow);

// Atualiza a imagem da Área de Exploração do robô.
for (int i = 0; i < larg; i++)
{
    for (int j = 0; j < alt; j++)
    {
        // Mostra nesta célula do PictureBox uma imagem de obstáculo.
        if (mtzCenario[i, j] == obstaculo)
            ge.DrawImage(imgObstaculo, i * celula, j * celula);
        // Mostra nesta célula do PictureBox a imagem do robô.
        else if (mtzCenario[i, j] == inicio)
            ge.DrawImage(imgRobo, i * celula, j * celula);
        // Mostra nesta célula do PictureBox a imagem da bola.
        else if (mtzCenario[i, j] == objetivo)
            ge.DrawImage(imgBola, i * celula, j * celula);
    }
}

// Pinta uma célula na Área de Exploração do robô como área livre.
if (mtzCenario[coordX, coordY] == livre)
    ge.FillRectangle(ye, coordX * celula, coordY * celula, celula - 1,
                    celula - 1);
// Mostra nesta célula do PictureBox uma imagem de obstáculo.
else if (mtzCenario[coordX, coordY] == obstaculo)
    ge.DrawImage(imgObstaculo, coordX * celula, coordY * celula);
// Mostra nesta célula do PictureBox a imagem do robô.
else if (mtzCenario[coordX, coordY] == inicio)
    ge.DrawImage(imgRobo, coordX * celula, coordY * celula);
// Mostra nesta célula do PictureBox a imagem da bola.
else if (mtzCenario[coordX, coordY] == objetivo)
    ge.DrawImage(imgBola, coordX * celula, coordY * celula);

.....
```

A.42 - Mostra o melhor caminho na Área de Exploração.

```
.....

// Mostra na Área de Exploração do robô o melhor caminho.
if (pathMelhor == encontrado && desenhaMelhor == true)
{
```

```

// Na primeira vez que mostra o melhor caminho a animação gráfica é
// feita dentro do timer tmrMelhorCaminho.
if (desMelhorPrim == true)
{
    tmrMelhorCaminho.Enabled = true;
    desMelhorPrim = false;
}
// Mostra o melhor caminho sem usar o timer tmrMelhorCaminho.
else
{
    int k;
    // Mostra na Área de Exploração do robô o caminho encontrado.
    for (k = 0; k < pathDados.GetLength(1) - 1; k++)
    {
        // Desenha o caminho percorrido pelo robô.
        ge.DrawImage(imgEstrelaMelhor, pathDados[numRobo, k, 0],
                    pathDados[numRobo, k, 1]);
    }
    // Mostra nesta célula do PictureBox a imagem de que o robô
    // chegou ao seu objetivo.
    ge.DrawImage(imgRoBola, pathDados[numRobo, k, 0],
                pathDados[numRobo, k, 1]);
}
.....

```

A.43 - Mostra os custos F do melhor caminho na Área de Exploração.

```

.....
// Mostra os custos de cada célula verificada pelo pathfinding.
Font fonte = new Font("Arial", 7);
String valor;
if (custos == true)
{
    int h = 0;
    for (int i = 0; i < larg; i++)
    {
        for (int j = 0; j < alt; j++)
        {
            // Mostra o custo F (G + H) de cada célula verificada pelo
            // pathfinding.
            if (custoF[h] != 0)
            {
                // Valor do custo F de uma determinada célula.
                valor = custoF[h].ToString();
                // Define, em pixels, o local onde será mostrado o custo F
                // no PictureBox picImagem.
                PointF localF = new PointF((float)abertoX[h] * celula,
                                           (float)abertoY[h] * celula);

                // Escreve o custo F da célula.
                ge.DrawString(valor, fonte, bk, localF);
            }
            .....
        }
        h++;
    }
}
}
.....

```

A.44 - Mostra a imagem do mapeamento do ambiente.

```

.....
if (pathBusca == encontrado && desenhaBusca == true)
for (int i = 0; i < larg; i++)
{
    for (int j = 0; j < alt; j++)
    {
        // Mostra nesta célula uma imagem de área livre (fundo claro).
        if (mtzOculata[i, j] == livre)
            ge.FillRectangle(ye, i * celula, j * celula, celula - 1,
                               celula - 1);

        // Mostra nesta célula do PictureBox uma imagem de obstáculo.
        else if (mtzOculata[i, j] == obstaculo)
            ge.DrawImage(imgObstaculo, i * celula, j * celula);
        // Mostra nesta célula do PictureBox a imagem do robô.
        else if (mtzOculata[i, j] == inicio)
            ge.DrawImage(imgRobo, i * celula, j * celula);
        // Mostra nesta célula do PictureBox a imagem de que o robô chegou
        // ao seu objetivo.
        else if (mtzOculata[i, j] == objetivo)
            ge.DrawImage(imgRoBola, i * celula, j * celula);
        // Mostra nesta célula do PictureBox uma imagem de área desconhecida
        // (fundo escuro).
        else if (mtzOculata[i, j] == desconhecido)
            ge.FillRectangle(dk, i * celula, j * celula, celula - 1,
                               celula - 1);

        // Apaga desta célula a imagem de um possível ponto de busca.
        else if (mtzOculata[i, j] == alvoBusca)
            ge.FillRectangle(ye, i * celula, j * celula, celula - 1,
                               celula - 1);

        // Mostra nesta célula do PictureBox a imagem de uma estrela, que
        // representa o caminho percorrido pelo robô.
        else if (mtzOculata[i, j] == caminho)
            ge.DrawImage(imgEstrelaBusca, i * celula, j * celula);
        // Mostra nesta célula do PictureBox a imagem do robô com uma
        // estrela sobreposta, que representa que foi um ponto onde o
        // robô realizou a visualização da Área de Exploração.
        else if (mtzOculata[i, j] == pontoVisao)
            ge.DrawImage(imgRoboEstr, i * celula, j * celula);
    }
}
.....

```

A.45 - Mostra a imagem do mapeamento do ambiente e o melhor caminho.

```

.....
// Mostra os caminhos percorridos pelo robô no mapeamento do ambiente
// e o melhor caminho simultaneamente.
if (desenhaGeral == true)
{
    int l;

    // Mostra os caminhos percorridos pelo robô durante o mapeamento do
    // ambiente.
    for (l = 0; l < CaminhosBusca.Count; l++)
    {

```

```

// Mostra nesta célula do PictureBox a imagem do robô.
if (((Caminhos)CaminhosBusca[l]).caminho == "PtoInicial")
    ge.DrawImage(imgRobo, ((Caminhos)CaminhosBusca[l]).x,
                  ((Caminhos)CaminhosBusca[l]).y);
// Mostra nesta célula do PictureBox uma imagem que representa o
// robô em um novo ponto de busca.
else if (((Caminhos)CaminhosBusca[l]).caminho == "NovoCaminho")
    ge.DrawImage(imgRoboEstr, ((Caminhos)CaminhosBusca[l]).x,
                  ((Caminhos)CaminhosBusca[l]).y);
// Mostra a imagem de uma estrela com fundo azul, que representa
// o caminho percorrido no mapeamento da Área de Exploração.
else
    ge.DrawImage(imgEstrelaBusca, ((Caminhos)CaminhosBusca[l]).x,
                  ((Caminhos)CaminhosBusca[l]).y);
}

// Mostra na Área de Exploração do robô o melhor caminho encontrado.
for (l = 0; l < pathDados.GetLength(1) - 1; l++)
{
    // Mostra nesta célula a imagem de uma estrela pela metade com o
    // fundo verde, que representa uma célula percorrida tanto no
    // melhor caminho como no mapeamento do ambiente.
    if (mtzOculta[pathDados[numRobo, l, 0] / celula,
                  pathDados[numRobo, l, 1] / celula] == caminho ||
        mtzOculta[pathDados[numRobo, l, 0] / celula,
                  pathDados[numRobo, l, 1] / celula] == pontoVisao)
        ge.DrawImage(imgEstrMelhorMeia, pathDados[numRobo, l, 0],
                      pathDados[numRobo, l, 1] + celula / 2 - 1);
    // Mostra nesta célula a imagem de uma estrela com fundo verde,
    // que representa o melhor caminho.
    else
        ge.DrawImage(imgEstrelaMelhor, pathDados[numRobo, l, 0],
                      pathDados[numRobo, l, 1]);
}

// Mostra nesta célula do PictureBox a imagem de que o robô chegou
// ao seu objetivo.
ge.DrawImage(imgRoBola, pathDados[numRobo, l, 0],
              pathDados[numRobo, l, 1]);
}

.....

```

A.46 - Atualiza a Área de Visualização do robô.

```

.....
// Cria um objeto que contém métodos relacionados a elementos gráficos.
Graphics gv = e.Graphics;
// Traça as linhas que separam as células na Área de Visualização.
for (int i = celula - 1; i < altVisual; i += celula)
    gv.DrawLine(p, 0, i, largVisual, i);
for (int i = celula - 1; i < largVisual; i += celula)
    gv.DrawLine(p, i, 0, i, altVisual);

base.OnPaint(e);

.....

```

REFERÊNCIAS BIBLIOGRÁFICAS

A* Search Algorithm. In: WIKIPEDIA, The Free Encyclopedia. Wikimedia, 2009. Disponível em: http://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=320206157>. Acesso em: 22 jul. 2009.

ALGORITMO A*. In: WIKIPÉDIA, a enciclopédia livre. Wikimedia, 2009. Disponível em: http://pt.wikipedia.org/w/index.php?title=Algoritmo_A*&oldid=16058354>. Acesso em: 30 set. 2009.

BIANCHI, Reinaldo. **Robótica**. Centro Universitário da FEI, 2007. Disponível em: <http://www.fei.edu.br/~rbianchi/robotica/ROBOTICA-MS-07-A-Robos%20Moveis%20Intro.ppt#256.1,Robótica>>. Acesso em: 08 nov. 2009.

BITTENCOURT, Guilherme. **Inteligência Artificial: Ferramentas e Teorias**. 2ª ed. Florianópolis: Editora da UFSC, 2001.

BOURG, David M.; SEEMANN Glenn. **AI for Game Developers**. Cambridge: O'Reilly Media, 2004.

COX, I. J. **Blanche - an experiment in guidance and navigation of an autonomous robot vehicle**. *IEEE Transactions on Robotics and Automation*, vol. 7, n.2, pp. 193-204, April 1991.

DECCÓ, Cláudia C. G. **Construção de mapas de ambiente para navegação de robôs móveis com visão omnidirecional estéreo**. Tese de Doutorado - Escola Politécnica, Universidade de São Paulo, São Paulo, 2004.

DEITEL, Harvey M., et al. **C# - Como Programar**. São Paulo: Makron Books, 2003.

FOX, D.; BURGARD, W.; THRUN, S. **Markov localization for reliable robot navigation and people detection.** *Modelling and Planning for Sensor-Based Intelligent Robot Systems*. Berlin: Springer, 1999.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Projeto de Algoritmos - Fundamentos, Análise e Exemplos da Internet.** Porto Alegre: Editora Bookman, 2002.

LESTER, Patrick. **A* Pathfinding for Beginners.** Policyalmanac, 2005. Disponível em: <<http://www.policyalmanac.org/games/aStarTutorial.htm>>. Acesso em: 13 jan. 2009.

LESTER, Patrick. **A* Pathfinding para Iniciantes.** Policyalmanac, 2005. Disponível em: <http://www.policyalmanac.org/games/aStarTutorial_port.htm>. Acesso em: 13 jan. 2009.

LESTER, Patrick. **Using Binary Heaps in A* Pathfinding.** Policyalmanac, 2003. Disponível em: <<http://www.policyalmanac.org/games/binaryHeaps.htm>>. Acesso em: 14 jan. 2009.

LIMA, Edwin; REIS, Eugênio. **C# e .NET - Guia do Desenvolvedor.** Rio de Janeiro: Editora Campus, 2002

LIPPMAN, Stanley B. **C # Um Guia Pratico.** Porto Alegre: Editora Bookman, 2003.

LOPES, Arthur Vargas. **Estruturas de Dados para a Construção de Software – vol. 2.** Canoas: Editora Ulbra, 1999.

LUGER, George F. **Inteligência Artificial. Estruturas e estratégias para a solução de problemas complexos.** 4ª ed. Porto Alegre: Editora Bookman, 2004.

NASA. **Mars Exploration Rover Mission.** NASA.gov, 2007. Disponível em: <<http://marsrover.nasa.gov/gallery/artwork/>>. Acesso em: 10 nov. 2009.

NEHMZOW, Ulrich. **Mobile Robotics: A Pratical Introduction.** 2nd ed. London: Springer, 2003.

ORDENAÇÃO (Computação). In: WIKIPÉDIA, a enciclopédia livre. Wikimedia, 2008. Disponível em:

<[http://pt.wikipedia.org/w/index.php?title=Ordena%C3%A7%C3%A3o_\(computa%C3%A7%C3%A3o\)&oldid=9905757](http://pt.wikipedia.org/w/index.php?title=Ordena%C3%A7%C3%A3o_(computa%C3%A7%C3%A3o)&oldid=9905757)>. Acesso em: 30 set. 2009.

PEDROSA, Diogo P. F. **Mapeamento de Ambientes Estruturados com Extração de Informações Geométricas Através de Dados Sensoriais**. Tese de doutorado, UFRN, Natal, 2006.

PEGDEN, C. D.; SHANNON, R. E.; SADOWSKI, R. P. **Introduction to Simulation using SIMAN**. 2th Ed. New York: McGraw-Hill, 1995.

RICH, Elaine; KNIGHT Kevin. **Inteligência Artificial**. 2ª ed. São Paulo: Makron Books, 1993.

RUSSELL, Stuart; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. 2th Ed. New Jersey: Prentice Hall, 2003.

RUSSELL, Stuart; NORVIG, Peter. **Inteligência Artificial**. 3ª ed. Rio de Janeiro: Editora Campus, 2004.

SIEGWART, Roland, NOURBAKHSH, Illah R. **Introduction to Autonomous Mobile Robots**. Cambridge, Massachusetts: MIT Press, 2004.

TAYLOR, Alan. **The Big Picture: News Stories in Photographs**. Boston.com, 2009. Disponível em: <<http://www.boston.com/bigpicture/2009/03/robots.html>>. Acesso em: 15 nov. 2009.

TOMADA de Decisão. In: WIKIPÉDIA, a enciclopédia livre. Wikimedia, 2009. Disponível em: <http://pt.wikipedia.org/w/index.php?title=Tomada_de_decis%C3%A3o&oldid=16208123>. Acesso em: 10 out. 2009.

UNIVERSIDADE FEDERAL DE SANTA CATARINA. **Algoritmo de Dijkstra para cálculo do Caminho de Custo Mínimo**. Disponível em: <<http://www.inf.ufsc.br/grafos/temas/custo-minimo/dijkstra.html>>. Acesso em: 23 mar. 2009.

ULLRICH, Robert A. **Robótica uma introdução: o porquê dos robôs e seu papel no trabalho.** Rio de Janeiro: Campus, 1987.

WHITBY, Blay. **I.A. Inteligência Artificial: um guia para iniciantes.** São Paulo: Editora Madras, 2004.