



Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Traveller: Um Framework de Definição de Trajetórias para Robôs Móveis

Autor: Rodrigo Lopes Rincon
Orientador: Prof. Dr Maurício Serrano

Brasília, DF
2015



Rodrigo Lopes Rincon

Traveller: Um Framework de Definição de Trajetórias para Robôs Móveis

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr Maurício Serrano

Coorientador: Profa. Dra Milene Serrano

Brasília, DF

2015

Rodrigo Lopes Rincon

Traveller: Um Framework de Definição de Trajetórias para Robôs Móveis/
Rodrigo Lopes Rincon. – Brasília, DF, 2015-
107 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr Maurício Serrano

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2015.

1. robótica móvel. 2. definição de trajetória. I. Prof. Dr Maurício Serrano. II.
Universidade de Brasília. III. Faculdade UnB Gama. IV. Traveller: Um Framework
de Definição de Trajetórias para Robôs Móveis

CDU

Rodrigo Lopes Rincon

Traveller: Um Framework de Definição de Trajetórias para Robôs Móveis

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Prof. Dr Maurício Serrano
Orientador

Profa. Dra Milene Serrano
Coorientadora

Profa. Dra Carla Silva Rocha Aguiar
Convidado 1

Prof. Dr Daniel Mauricio Muñoz
Arboleda
Convidado 2

Brasília, DF
2015

Resumo

A robótica móvel procura lidar com máquinas capazes de se locomover de forma autônoma. Para viabilizar o tratamento desse desafio, esse foi dividido em desafios (lê-se objetivos) menores. Um desses objetivos é contemplado nesse trabalho, Trata-se da definição da trajetória do caminho a ser percorrido. Considerando que o ambiente em volta já é conhecido, o robô precisa definir um caminho seguro para chegar aonde deseja, sem se chocar com os obstáculos existentes. Diversos algoritmos com esse intuito foram desenvolvidos, sendo o objetivo deste trabalho prover um *framework* que forneça tais soluções já prontas ao programador. Através de padrões de projetos, o Traveller Framework é projetado para a fácil troca entre os algoritmos, evitando afetar o resto do código. Além da implementação destes algoritmos, é projetada uma arquitetura que permita a portabilidade para o maior número de plataformas, preocupando-se tanto com a reutilização quanto com a legibilidade do código. O Traveller comunica-se com os outros módulos de movimentação do robô por uma interface simples e comum a todos os algoritmos. É importante ressaltar que o módulo de controle do hardware do robô bem como a movimentação dele de fato não fazem parte do escopo deste trabalho. Na verdade, é preocupação desse trabalho a comunicação com o módulo de controle do hardware do robô, fornecendo a esse módulo uma série de coordenadas para onde o robô deve ir. Este trabalho é *open-source*, desenvolvido em linguagem Java, e testado em robôs *differential steering* do kit educacional da LEGO Mindstorm NXT.

Palavras-chaves: robótica móvel. definição de trajetória. desenvolvimento de framework. kit educacional da LEGO Mindstorm. Grafo de Visibilidade. Voronoi. Quadtree. Wavefront.

Abstract

The mobile robotics field is concerned in developing machines which move themselves autonomously. The movimentation problem was divided into the challenges, understood as goals, lower. One of these challenges, the path planning, is addressed in this paper. Whereas the environment around is already known, the robot needs to define a safe path to go where it is headed without colliding with any obstacles. Several algorithms for this were developed, such this paper aims to develop a framework that provides this ready-made solutions to the programmer. Through design patterns, the Traveller Framework is designed for easy switching between algorithms without affecting the rest of the code. Apart from the implementation of these algorithms, it is designed an architecture that allows portability to as many different platforms, worrying about reuse and code readability. The Traveller communicates with the other module robots by a simple and common interface to all algorithms. Importantly, the robot hardware control module and move is not part of the scope of this paper, but chat with this module and gives it a series of coordinates for where it should go. This work is open-source, developed in Java and tested on differential steering robots of the LEGO Mindstorm NXT educational robotics kit.

Key-words: mobile robotics. path planning. framework development. LEGO Mindstorm educational kit. Visibility Graph. Voronoi. Quadtree. Wavefront.

Listas de ilustrações

Figura 1 – Braço robótico usado na robótica industrial (http://sicmscj.com.br/links, 2014)	20
Figura 2 – a) roda padrão, b) roda castor, c) roda suéca, d) roda esférica (SIEGWART; NOURBAKHSH, 2004)	21
Figura 3 – Ackerman Steering (SECCHI, 2008)	22
Figura 4 – Camadas de execução na navegação do robô (VIEIRA, 2005)	26
Figura 5 – Arquitetura SPA (MATARIC, 2007)	27
Figura 6 – diferentes tipos de mapeamento: a) mapa real b) mapa geométrico c) grade de ocupação d) topológico (SIEGWART; NOURBAKHSH, 2004)	29
Figura 7 – Diagrama de sequência da criação do caminho	30
Figura 8 – Grafos de visibilidade a) tradicional b) com expansão dos obstáculos (HOLDGAARD-THOMSEN, 2010) e (Site do MRIT, 2014)	32
Figura 9 – Diagrama de Voronoi (SIEGWART; NOURBAKHSH, 2004)	33
Figura 10 – a) decomposição celular usando regiões de tamanho fixo b) Quadtree encontrando um caminho estreito entre dois obstáculos (SIEGWART; NOURBAKHSH, 2004)	34
Figura 11 – Algoritmo <i>Wavefront</i> , com grafo em azul (Site do MRIT, 2014)	35
Figura 12 – Exemplo de arquitetura em camadas, (LARMAN, 2005)	37
Figura 13 – <i>Hot-spot</i> em um <i>framework</i> caixa-preta e caixa-branca. (FAYAD; JOHNSON; SHMIDT, 1999)	41
Figura 14 – Estruturas dos subsistemas <i>hot-spot</i> . (FAYAD; JOHNSON; SHMIDT, 1999)	42
Figura 15 – Processo de desenvolvimento orientado a <i>hot-spot</i> . (FAYAD; JOHNSON; SHMIDT, 1999)	43
Figura 16 – <i>Hot-spot card.</i> (FAYAD; JOHNSON; SHMIDT, 1999)	44
Figura 17 – Foto do robô utilizado	50
Figura 18 – Exemplos de estruturas <i>differential steering</i> (http://www.enigmaindustries.com/configur 2014)	51
Figura 19 – Fluxo de atividades propostas	55
Figura 20 – Estrutura do funcionamento do <i>framework</i>	58
Figura 21 – Arquitetura de navegação considerada, (NEHMZOW, 2003)	58
Figura 22 – Diagrama de pacotes do sistema	59
Figura 23 – Componente embarcado no robô	60
Figura 24 – Componente que se comunica com o robô	61
Figura 25 – Componente Controller	62
Figura 26 – Diagrama de sequência do componente controller	63

Figura 27 – Componente Map	64
Figura 28 – Componente PathPlanner	64
Figura 29 – Componente Graph	65
Figura 30 – Componente BestPath	66
Figura 31 – Código do <i>framework</i> sendo executado na versão protótipo	67
Figura 32 – Código do <i>framework</i> sendo executado na versão final	68
Figura 33 – Ordem em que os dados devem ser enviados	70
Figura 34 – Obstáculo real em preto e a expansão incrementada em amarelo (Site do MRIT, 2014)	72
Figura 35 – Teste inicial do <i>framework</i>	78
Figura 36 – teste de stress, tempo de execução	79
Figura 37 – teste de stress, métodos que mais foram utilizados	79
Figura 38 – Cobertura de código do <i>framework</i>	83
Figura 39 – Trajetórias para o mapa 1 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront	84
Figura 40 – Trajetórias para o mapa 2 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront	84
Figura 41 – Trajetórias para o mapa 3 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront	85
Figura 42 – Trajetórias para o mapa 4 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront	85
Figura 43 – Trajetórias para o mapa 5 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront	86
Figura 44 – Trajetórias para o mapa 6 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront	86
Figura 45 – tempo de processamento do mapa 1 com LinkedList	89
Figura 46 – tempo de processamento do mapa 1 com ArrayList	89
Figura 47 – tempo de processamento do mapa 3 com Quadtree	90
Figura 48 – tempo de processamento do mapa 3 com Grafo de Visibilidade	90
Figura 49 – tempo de processamento do mapa 3 com Voronoi	91
Figura 50 – tempo de processamento do mapa 3 com Wavefront	91
Figura 51 – mapa exemplo do <i>hot-spot card</i> PathPlanner	103
Figura 52 – resultado do exemplo 1 do <i>hot-spot card</i> PathPlanner	104
Figura 53 – mapa exemplo 2 do <i>hot-spot card</i> PathPlanner	104
Figura 54 – resultado do exemplo 2 do <i>hot-spot card</i> PathPlanner	104
Figura 55 – grafo exemplo do <i>hot-spot card</i> BestPath	105
Figura 56 – grafo do exemplo 2 do <i>hot-spot card</i> BestPath	106

Sumário

1	INTRODUÇÃO	15
1.1	O Problema	15
1.2	Questão de Pesquisa	15
1.3	Justificativa	16
1.4	Objetivos	16
1.4.1	Objetivo Geral	16
1.4.2	Objetivos Específicos	16
1.5	Organização do Trabalho	17
2	REFERENCIAL TEÓRICO	19
2.1	A Base da Robótica	19
2.1.1	Definição de Robótica	19
2.1.2	Robôs Industriais e Móveis	19
2.1.3	Estruturas Físicas de Robôs Com Rodas	20
2.2	Desafios da Robótica Móvel	22
2.2.1	Modelos de movimentação	23
2.2.2	Localização	23
2.2.3	Mapeamento	23
2.2.4	Paradigmas de definição de trajetória	24
2.2.5	Visão Geral Sobre Navegação	25
2.3	Mapeamento e Modelagem dos Obstáculos	27
2.3.1	Tipos de Mapas	28
2.3.2	Otimização do Processo de Mapeamento	28
2.3.3	Definindo Percursos em um Mapa	29
2.4	Algoritmos Globais de Definição de Trajetória	30
2.4.1	Grafos de Visibilidade	31
2.4.2	Voronoi	32
2.4.3	Quadtree	33
2.4.4	Wavefront	34
2.5	Frameworks e padrões de projeto	35
2.5.1	Arquitetura e Componentes	36
2.5.2	Padrões de Projeto	38
2.5.3	<i>Framework Caixa-preta, Caixa-branca e Caixa-cinza</i>	39
2.5.4	<i>Hot-spot e Frozen-spot</i>	40
2.5.5	Processo de Desenvolvimento Orientado a <i>Hot-Spots</i>	42

2.5.6	Abordagem de Projetos	44
2.6	Trabalhos Relacionados	45
2.6.1	Carmen	45
2.6.2	ROS	46
2.6.3	Tese de Morten Strandberg	46
2.7	Resumo do Capítulo	47
3	SUPORTE TECNOLÓGICO	49
3.1	Linguagem de programação	49
3.2	Ferramentas e plugins	49
3.3	Lego NXT	50
3.4	Differential Steering	50
3.5	Resumo do Capítulo	51
4	METODOLOGIA	53
4.1	Processo de produção	53
4.2	Atividades e fluxo de atividades	54
4.3	Resumo do Capítulo	56
5	O TRAVELLER FRAMEWORK	57
5.1	A Arquitetura	58
5.1.1	EmbeddedCommunicator	60
5.1.2	RobotCommunicator	61
5.1.3	Main	61
5.1.4	Controller	62
5.1.5	Map	63
5.1.6	Path Planner	64
5.1.7	Graph	65
5.1.8	Best Path	65
5.2	O Protótipo	66
5.3	A Versão Final	67
5.4	<i>Hot-spots do Framework</i>	70
5.4.1	PathPlanner	70
5.4.2	BestPath	71
5.4.3	EmbeddedCommunicator	71
5.4.4	<i>Hot-spot Descartados: O Mapa</i>	71
5.5	Os Obstáculos	72
5.6	Boas Práticas de Programação	73
5.7	Resumo do Capítulo	75
6	RESULTADOS OBTIDOS	77

6.1	Testes iniciais	77
6.2	Testes de desempenho	77
6.2.1	Testes de Stress - Simulação	78
6.2.2	Testes de Stress no Robô	80
6.2.3	Decisões tomadas	80
6.3	Testes unitários e de Integração	81
6.3.1	Testes unitários e cobertura de código	81
6.3.2	Testes de Integração: Comparações Com o Simulador MRIT	82
6.3.3	Testes de Integração: Desempenho	88
6.4	Testes Finais Com o Robô	90
6.5	Observações Sobre o Funcionamento dos Algoritmos	92
6.6	Resumo do Capítulo	93
7	CONCLUSÃO	95
7.1	Sugestão de Trabalhos Futuros	95
	Referências	97
	APÊNDICES	101
	APÊNDICE A – HOST-SPOT CARDS	103
A.1	PathPlanner	103
A.2	BestPath	105
A.3	EmbeddedCommunicator	106
	APÊNDICE B – CÓDIGO FONTE	107

1 Introdução

O ramo da robótica costuma atrair a atenção de leigos e profissionais, explorando um campo onde a imaginação trabalha ao máximo. A robótica móvel em específico tem levado as máquinas para mais perto da população e trazendo desafios aos desenvolvedores destas máquinas. Para construir uma máquina autônoma há uma série de dificuldades, cada uma fonte de pesquisas e trabalho. Para mover uma máquina entre dois pontos é preciso o controle de motores, mapeamento e sensoriamento da região, estudo da cinemática do robô, planejamento da trajetória e monitoramento do movimento.

Produzir um robô exige conhecimento tanto mecânico quanto eletrônico e de software. Kits de robótica entregam a máquina pronta, com manuais de uso e um *firmware* para facilitar a programação. Entretanto, cada kit funciona de forma diferente e costumamente seus códigos são complicados de entender, não seguindo as boas práticas de programação.

1.1 O Problema

Atualmente, existe pouco conteúdo *open-source* para quem busca desenvolver na área, tendo muitas vezes de implementar tudo desde as funções mais básicas. Parte disso acontece pelas diferenças físicas entre as máquinas e devido ao fato de, na área, pouco se desenvolver orientado à reutilização de software e às boas práticas de programação. As soluções existentes funcionam apenas no kit no qual foram desenvolvidas e com interfaces diferentes, muitas vezes impedindo que o desenvolvedor reutilize seu código em outra plataforma.

1.2 Questão de Pesquisa

Considerando essas dificuldades, este trabalho busca ajudar a comunidade de robótica em uma das áreas de navegação: a definição da trajetória, com um *framework open-source* que permita a implementação destes algoritmos nos mais variados tipos de robôs móveis.

Para tanto é preciso analisar o cenário existente de robótica e se perguntar: é possível abstrair a definição de trajetória do hardware do robô, tornando essa prática independente da plataforma utilizada? Para responder adequadamente a essa questão, é relevante analisar o quanto é necessário conhecer do hardware e dos sensores e com que outros módulos da navegação a definição de trajetória está diretamente ligado.

1.3 Justificativa

O objetivo deste trabalho é ajudar a comunidade de robótica a alcançar maior reutilização e portabilidade em seus códigos, deixando-os menos acoplados ao hardware utilizado. Como são poucas as bibliotecas e ferramentas projetadas para funcionarem em vários kits, cada sistema fica acoplado às bibliotecas de seus respectivos kits. Este problema diminui a reusabilidade e quase anula a chance de portabilidade.

Conforme ([LARMAN, 2005](#)), ([GOODLIFFE, 2007](#)) e ([MCCONNEL, 2004](#)), alto acoplamento é indesejado, pois torna o código sensível a falhas e de difícil manutenção. Segundo eles, a reutilização facilita e agiliza o desenvolvimento, diminuindo o esforço para tarefas repetitivas e/ou que possam ser isoladas.

O Traveller Framework é um módulo independente que procura desacoplar a definição de trajetória do resto do sistema. Um *framework* de código aberto permite ainda o uso de um código comum a toda a comunidade, induzindo a reutilização e procurando explorar um possível padrão de utilização à área contemplada. Tudo isso ainda acaba por facilitar a compreensão de outros códigos. A implementação do Traveller *framework* terá a visão de fazê-lo funcionar na maioria das plataformas e deixá-lo extensível para inclusão posterior de novas plataformas e algoritmos. A própria comunidade pode contribuir para sua expansão, tornando-o compatível com mais hardwares e implementando novos algoritmos para traçar rotas.

1.4 Objetivos

Os objetivos deste trabalho, tanto de forma geral quanto mais específicos, são listados a seguir.

1.4.1 Objetivo Geral

Este trabalho de conclusão de curso visa desenvolver um *framework* manutenível e extensível sobre algoritmos de definição de trajetória para robôs móveis. Serão utilizados algoritmos clássicos para um ambiente previamente conhecido (algoritmos globais) como descrito por ([GUZMÁN et al., 2008](#)). A implementação compreende em demonstrar um robô *differential steering* em funcionamento utilizando o *framework* proposto.

1.4.2 Objetivos Específicos

Este trabalho tem como objetivos mais específicos, principalmente:

1. Investigar abordagens para desenvolvimento de *frameworks*;

2. Explorar algoritmos de definição de trajetória considerando a estratégia de algoritmos globais;
3. Aplicar boas práticas de modelagem e programação ao *framework*, como alta coesão, baixo acoplamento, funções atômicas, comentários e nomenclatura correta de variáveis;
4. Usar padrões de projeto para maior robustez do código, como prover uma interface comum à todos os algoritmos e a fácil troca entre eles, além de maior portabilidade da solução proposta para outros tipos de hardware;
5. Aplicar testes, principalmente unitários, na solução proposta;
6. Prover um teste prático do *framework* usando como base o kit educacional da LEGO;

1.5 Organização do Trabalho

O trabalho está dividido em 7 capítulos principais:

1. Introdução: este capítulo, visando explicar o problema que gerou este trabalho.
2. Referencial Teórico: explana sobre os conceitos importantes da robótica e questões mecânicas pertinentes à locomoção do robô, além de uma rápida análise quanto aos módulos de navegação disponíveis na literatura.
3. Suporte Tecnológico: explica sobre as tecnologias utilizadas para o desenvolvimento tanto do *framework* quanto da prova de conceito, listando ferramentas, *plugins* e kits utilizados.
4. Metodologia: análise das dificuldades e riscos do projeto, descrição e fluxo das atividades, cronograma e métodos utilizados.
5. O Traveller Framework: detalhamento do funcionamento do *framework*: explicação da arquitetura e padrões usados na modelagem, técnicas de programação utilizadas e algoritmos implementados
6. Resultados Obtidos: apresentação dos testes e análises feitas, além das escolhas realizadas a partir dos dados coletados.
7. Conclusão: apresentação dos resultados finais, propostas de melhorias e estudos futuros.

2 Referencial Teórico

Neste capítulo será abordado como funciona os temas tratados neste trabalho, bem como introduzir conceitos importantes para a melhor compreensão do tema.

2.1 A Base da Robótica

O termo robô foi criado em 1921 pelo dramaturgo tcheco Karel Čapek, em sua obra *Rossum's Universal Robots*, sendo o termo uma variação da palavra *robota* (trabalho forçado) (MATARIC, 2007). Na obra, os robôs eram máquinas humanoides que realizavam todo tipo de trabalho. Entretanto, o conceito do que é um robô tem sofrido variações ao longo do tempo.

2.1.1 Definição de Robótica

Alguns autores, como (MATARIC, 2007) e (ABREU, 2001), consideram máquinas movidas por engrenagens e vapor, séculos antes, como ancestrais dos robôs atuais por suas articulações e movimentos por vezes independentes de interação humana. Hoje ainda não há um consenso do que exatamente pode se enquadrar como robô. Segundo a RIA, Associação das Indústrias de Robótica ((Site do RIA, 2015)), um robô industrial é um “manipulador reprogramável, multifuncional, projetado para mover material, ferramentas ou dispositivos especializados através de movimentos programáveis variados para desenvolver uma variedade de tarefas”. (ABREU, 2001) apresenta mais definições de robôs industriais.

(MATARIC, 2007) apresenta uma definição mais genérica, que abrange não só robôs industriais, mas todos os grupos, segundo ele “um robô é um sistema autônomo que existe no mundo físico, que reconhece o ambiente a volta e pode agir sobre ele para alcançar suas metas”.

Segundo (SECCHI, 2008), existem três tipos de robôs: industriais, médicos e móveis; apesar das demais bibliografias fazerem referência também a robôs humanoides, médicos, militares, de entretenimento, domésticos, entre outros.

2.1.2 Robôs Industriais e Móveis

A robótica industrial foi a que impulsionou a pesquisa e desenvolvimento na área de robótica. Ela criou um mercado bilionário ao redor do mundo desenvolvendo soluções

automatizadas para a indústria. Esses robôs, em grande parte, eram braços robóticos que realizavam trabalhos perigosos e/ou repetitivos com precisão e velocidade.

Também chamados de manipuladores, os braços robóticos eram fixos em uma posição, formados por uma estrutura articulada que move a ponta para o local desejado para trabalhar. Os graus de liberdade das articulações e o comprimento dos braços que os ligavam definem a área de atuação do robô, que são classificadas pelo formato alcançado por ele (cartesiano, cilíndrico, esférico e outros). Normalmente, as máquinas possuem seis graus de liberdade (um para cada articulação), três para posicionar a ponta de trabalho no local certo e mais três para movê-la ao executar o serviço (ABREU, 2001). A Figura 01 apresenta um exemplo de robô industrial.



Figura 1 – Braço robótico usado na robótica industrial (<http://sicmscj.com.br/links, 2014>)

Já a robótica móvel, por sua vez, trata de máquinas capazes de se movimentar independentemente. Com maior liberdade de movimentação, este grupo de máquinas possui uma maior gama de trabalhos. Um robô móvel pode ser terrestre, aquático, voador ou até espacial (VIEIRA, 2005), sendo movido por rodas, esteiras, patas, hélices, dentre outros. Com essa ampla área de atuação e o dom da mobilidade, podemos encontrar os robôs móveis aplicados nas mais diversas áreas. (PEREIRA, 2003) lista cinco grandes áreas aonde esses robôs são mais utilizados: indústria, serviços, pesquisa, campo e entretenimento.

2.1.3 Estruturas Físicas de Robôs Com Rodas

Existe um grande campo de pesquisa para cada forma de deslocamento do robô. No caso dos robôs com rodas, o tipo de roda e a estrutura do chassi são consideradas para a mobilidade que a máquina terá. (SIEGWART; NOURBAKHSH, 2004) afirma que é possível dar estabilidade ao robô com duas ou três rodas e ao usar quatro ou mais é necessário um sistema de suspensão para permitir que todas as rodas toquem o chão quando o robô estiver em terreno acidentado.

(SECCHI, 2008) lista quatro tipos de roda mais usados: roda fixa para tração, roda orientável centralizada para direção e tração, roda castor (ou roda louca) para estabilidade e a suéca para mobilidade. Já (SIEGWART; NOURBAKHS, 2004) lista outros quatro tipos de roda: a roda padrão (semelhante a orientável centralizada), a castor, a suéca e a esférica, que assim como a suéca permite maior mobilidade. A Figura 02 mostra um esboço de alguns tipos de rodas.

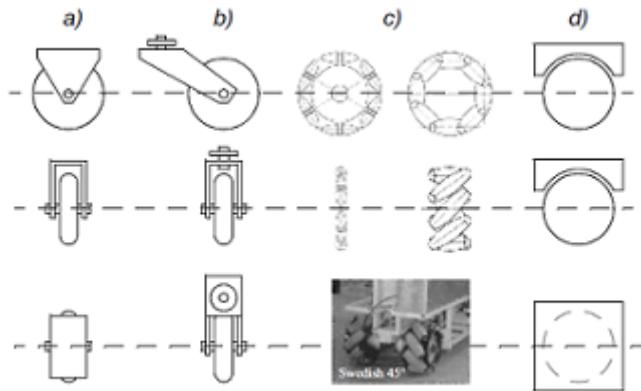


Figura 2 – a) roda padrão, b) roda castor, c) roda suéca, d) roda esférica (SIEGWART; NOURBAKHS, 2004)

A definição das rodas determina a quantidade de liberdade de movimento da estrutura. Para fazer um robô omnidirecional (que se movimenta para todos os lados sem reorientação) por exemplo, costuma-se usar rodas suécas ou esféricas, porém também é possível montar com rodas orientáveis centralizadas (SECCHI, 2008). Ele se destaca por sua capacidade de se mover em qualquer sentido (frente, trás, lados e girar em torno do próprio eixo) usando um motor para cada roda. Robôs do tipo *Synchro Drive* também podem se mover para qualquer direção, porém ao contrário do omnidirecional que tem um motor para cada roda, esse usa três rodas ligadas ao mesmo motor, girando-os com a mesma velocidade e mesma direção (BORENSTEIN; EVERETT; FENG, 1996). Esses tipos de robôs são chamados holonômicos.

(SECCHI, 2008) e (BORENSTEIN; EVERETT; FENG, 1996) também apresentam máquinas do modelo triciclo, que possuem uma roda orientável centralizada afrente e duas convencionais presas ao mesmo eixo atrás, como as rodas de trás de um carro. A roda da frente tem função de direcionar e de tração, enquanto as duas traseiras apenas acompanham o movimento. Outro modelo semelhante é o modelo de quadriciclo, que funciona igual a um carro, com duas rodas de tração-direção na frente ligadas ao mesmo motor e atrás duas rodas de tração também presas ao mesmo eixo.

Ambos os modelos mencionados anteriormente sofrem erros de odometria (estimação da posição do robô) e apresentam problemas de derrapagem em curvas. Essa derrapagem ocorre pelo fato da roda interna à curva fazer um arco menor que a roda externa,

obrigando essa a derrapar e desgastar o pneu. Esse problema pode ser consertado com um modelo chamado *Ackerman Steering*. Segundo (BORENSTEIN; EVERETT; FENG, 1996), (SECCHI, 2008) e (BAGNALL, 2011), o sistema *Ackerman* inclina mais uma roda que a outra para compensar a diferença de espaço percorrido, fazendo com que o eixo de cada roda sempre esteja apontando para o centro da curva, conforme a Figura 3 mostra.

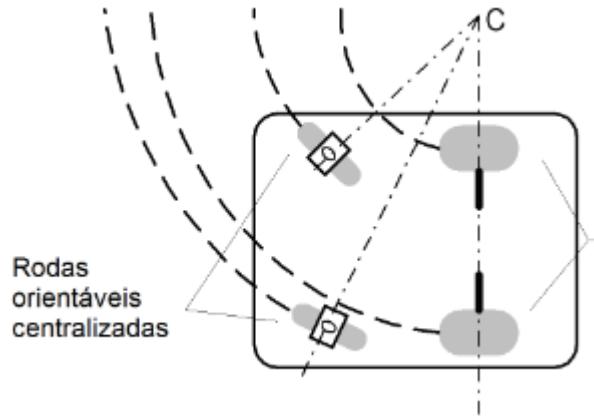


Figura 3 – Ackerman Steering (SECCHI, 2008)

Robôs do tipo *differential steering* possuem três rodas: duas fixas para guiar o robô e uma roda castor atrás para dar estabilidade. O grande diferencial desse modelo é que cada roda de direção possui um motor separado. As duas rodas estão alinhadas no mesmo eixo, onde cada uma é independente, com seu próprio motor. Porém, essa dinâmica é diferente de um carro, onde ambas as rodas estão ligadas ao mesmo motor. Isso permite ao robô girar em torno do próprio eixo e não depender de manobras em arco como o modelo de carro (BAGNALL, 2011) (SECCHI, 2008) (MATARIC, 2007).

2.2 Desafios da Robótica Móvel

Na área de software, a robótica móvel apresenta diversos desafios a serem superados. Dotar uma máquina com a capacidade de locomover-se sozinha esconde diversas dificuldades, que advém do fato de que a navegação deve integrar sensoriamento, atuação, planejamento, arquitetura, hardware, eficiência e computação (SOUZA, 2008). A navegação do robô por um terreno envolve mapeamento da região, controle dos motores, sensoriamento, auto-localização e definição de trajetória. Uma visão geral sobre cada uma é dada a seguir para auxiliar a compreensão do trabalho de navegação como um todo e a importância de cada um.

2.2.1 Modelos de movimentação

A primeira dificuldade da navegação de um robô é a correta locomoção do robô. Podemos determinar o movimento do robô com cálculos complexos para o controle mecânico dos motores, considerando os tipos de rodas e distribuição do peso. O controle matemático do robô é dividido em controle cinemático (não considera efeitos dinâmicos provocados pelo movimento do robô, como atrito e derrapagem) e dinâmico (considera efeitos provocados pelo movimento, como atrito e derrapagem). A determinação do movimento e a orientação da máquina são feitas por uma série de equações e por operações lineares e vetoriais, considerando os planos cartesianos e a direção para onde aponta o robô (SIEGWART; NOURBAKHSH, 2004). Ambos os modelos se baseiam em cálculos geométricos e nas propriedades do robô para realizar uma movimentação correta e atualizar a posição atual do robô.

2.2.2 Localização

Outro desafio é definir a real posição da máquina. A locomoção nem sempre ocorre como esperado. Derrapagem, atoleiros e forças externas que empurram o robô geram uma alteração da sua real posição com a esperada inicialmente. Tanto o modelo cinemático como no dinâmico são sensíveis a estas falhas devido a eles necessitarem da posição atual do robô.

A odometria procura obter incrementalmente informações do movimento do robô, lendo a quantidade de giros que cada roda realizou para determinar a localização final. Entretanto, essa solução possui alto índice de erros cumulativos que a tornam inadequada para longas distâncias (PEREIRA, 2003). Outras soluções são a de *beacons* (faróis), que se comunicam com o robô dando sua coordenada dentro daquela região mapeada. O sistema de *beacon* mais abrangente é o de GPS, porém faróis locais também são comuns para localização dentro de áreas pré-determinadas. Marcas no chão para auto localização pelo robô também são muito usadas em ambientes fechados. Outra solução é através de processamento de imagens, onde o robô conhece alguns objetos ou locais únicos e conhece previamente suas posições no mapa. Ao passar por estes referenciais é capaz de inferir sua posição. (BORENSTEIN; EVERETT; FENG, 1996) explica cada um dos algoritmos de localização.

2.2.3 Mapeamento

O mapeamento consiste na modelagem do ambiente que contém o robô através do uso de mapas obtidos pelo sistema sensorial ou previamente armazenados (SOUZA, 2008). Considerando o sensoriamento, a forma de escanear a região depende muito dos tipos de sensores usados e seu alcance. Uma representação do mundo real dentro da máquina

facilita nas tomadas de decisão apesar de ocupar muita memória.

2.2.4 Paradigmas de definição de trajetória

Há duas estratégias mais abordadas para a definição de trajetórias: uma em que o robô não conhece nada ao seu redor (algoritmos locais) e uma em que o robô conhece previamente todo o ambiente (algoritmos globais).

(GUZMÁN et al., 2008) explica que os algoritmos locais, o robô procura seguir em linha reta até um ponto desejado. Ao encontrar um obstáculo, o robô contorna o objeto até que não detecte mais nada entre ele e o ponto final. Neste tipo de algoritmo, a comunicação com os sensores de presença faz-se essencial. Um problema deste paradigma ocorre ao encontrar um mínimo local, onde o robô entra, mas não consegue sair (SOUZA, 2008). (SECCHI, 2008) explica o funcionamento de alguns métodos locais.

Dentre os algoritmos locais, o algoritmo potencial é o mais comentado, citado por (SECCHI, 2008), (SOUZA, 2008), (GUZMÁN et al., 2008), (CHOSET et al., 2005), (SIEGWART; NOURBAKHSH, 2004) e (HOLDGAARD-THOMSEN, 2010). Este algoritmo funciona sem uma rota pré-definida e, ao invés disso, usa uma ideia de atração e repulsão para se locomover. O objetivo tem um efeito atrativo, puxando o robô para sua direção enquanto os obstáculos tem efeito repulsivo, distanciando o robô de si. Como o ponto final é conhecido desde o início, ele tem seu efeito atrativo funcionando a todo instante; enquanto o efeito repulsivo dos obstáculos só ocorre quando o obstáculo está ao alcance dos sensores.

Nos algoritmos globais, o robô possui um mapa interno com os obstáculos presentes e define rotas entre eles para chegar ao outro ponto (GUZMÁN et al., 2008) (SOUZA, 2008). Esta estratégia não precisa conhecer os sensores, porém envolve maior modelagem do ambiente a sua volta e dos objetos que tem nele. Uma outra camada que, essa sim, conhece os sensores, pode realizar o sensoriamento e entregar o mapa pronto ao módulo. Os algoritmos globais serão melhor explicados na seção 2.4.

Ambos os algoritmos mencionados podem ser usados juntos para obter melhores resultados. Nesse caso usa-se o global para definir o caminho completo, traçando longas distâncias, e o local para a movimentação a curta distância, verificando mudanças no ambiente próximo com os sensores para correções naquela parte (SOUZA, 2008). Outros autores como (SECCHI, 2008) também apresentam a mesma ideia.

(MATARIC, 2007), por sua vez, apresenta quatro tipos de algoritmos para planejamento de trajetória: os deliberativos, reativos, híbridos e o baseado em comportamento. (GUZMÁN et al., 2008) descreve a navegação como sendo a mesma abordagem dos algoritmos globais e a reativa como sendo a mesma abordagem dos algoritmos locais.

Os algoritmos deliberativos conhecem o mapa previamente e fazem toda a análise

antes de agir. Essa análise prévia é descrita como o grande problema desse paradigma, pois o robô precisa ficar muito tempo parado para tomar a decisão. “Se o problema exige uma grande dose de planejamento antecipado e não envolve nenhuma pressão de tempo, e, ao mesmo tempo apresenta um ambiente estático e baixa incerteza na execução, então esse paradigma atende bem ao caso” (MATARIC, 2007).

Os reativos possuem a mesma descrição que os algoritmos locais definidos por (GUZMÁN et al., 2008), (SOUZA, 2008) e os demais autores. “Ele é baseado em uma estreita ligação entre os sensores e atuadores do robô. Sistemas puramente reativos não usam quaisquer representações internas do ambiente. Eles operam em uma escala de tempo curto e reagem às informações sensoriais atuais.” (MATARIC, 2007). Tal prática confere agilidade de processamento, porém não garante confiabilidade.

A navegação híbrida é uma mistura dos dois. A ideia básica por trás deste paradigma é obter o melhor dos dois mundos: a velocidade de controle reativo e os cérebros de controle deliberativo. Isso é alcançado através de três camadas, uma planejadora deliberativa, uma reativa que comunica com os atuadores e uma intermediária ligando as duas. Essa solução usa os algoritmos deliberativos para longas distâncias (definindo o percurso de forma macro até o ponto final) e os reativos para distâncias curtas, entre cada ponto do percurso. Como já dito anteriormente, (SOUZA, 2008) oferece uma descrição parecida de navegação híbrida, apesar de não usar essa nomenclatura.

Por fim, os algoritmos baseados em comportamentos são diferentes dos dois anteriores, dividindo suas funcionalidades em módulos que executam ações, as quais realizam um comportamento desejado, como andar para frente até encontrar um obstáculo ou seguir uma parede. O objetivo deste paradigma é montar ações que executam esse comportamento e fazer esses módulos se comunicarem. Esse paradigma, assim como o reativo, também precisa conhecer os sensores, que são a principal entrada dos módulos.

Há outros métodos de definição de trajetória. (GUZMÁN et al., 2008) comenta rapidamente sobre a existência de métodos probabilísticos e (STRANDBERG, 2004) descreve o método probabilístico baseado em *roadmaps*.

2.2.5 Visão Geral Sobre Navegação

(VIEIRA, 2005) define uma arquitetura de navegação de robôs móveis, onde cada camada é uma área de pesquisa e juntas realizam a navegação do robô (Figura 04). A primeira é a percepção do ambiente pelo robô, através de sensores. Após tratar esses dados e saber o que o cerca, é executada a camada de decisão, onde a máquina avalia a informação e decide que ações tomar. Essa camada é o cérebro do robô e pode possuir algoritmos de inteligência artificial.

A camada de “Planejamento de Caminho” é a que define por onde o robô irá passar



Figura 4 – Camadas de execução na navegação do robô ([VIEIRA, 2005](#))

para chegar à posição desejada. É nesta camada que o trabalho proposto irá agir. Com o ambiente já reconhecido pelas camadas acima e o controle do hardware pelas camadas abaixo, o sistema proposto (assim como a camada descrita por ([VIEIRA, 2005](#))) define um percurso por onde a máquina não chocará com os obstáculos detectados considerando o tamanho do robô. Existe uma série de algoritmos que fazem a escolha do melhor caminho livre de obstáculos. No *framework* foram implementados alguns deles para uso comum a todos os desenvolvedores interessados.

A camada “Geração de Trajetória” é a de definição do percurso. Ela recebe o caminho definido pela camada “Planejamento de Caminho” e define que ações devem ser feitas sobre o hardware para percorrer aquele percurso. Essa camada deve conhecer a estrutura do robô e suas limitações físicas de movimento. Ela define a velocidade e a direção em que o robô irá andar, repassando aos motores os sinais necessários a cada instante do percurso.

A última camada é a que atua diretamente no hardware, garantindo que os atuadores estão recebendo o sinal certo para girar conforme desejado.

([BAGNALL, 2011](#)) também apresenta uma arquitetura de navegação muito interessante baseada na navegação de um navio. Ele define que um sistema de navegação deve possuir: um veículo (hardware), piloto (controlador dos motores), navegador (gerador do caminho), provedor de posição (algoritmos de auto-localização), mapas, planejador de trajetória (algoritmos de definição de trajetória) e planejador da missão.

O “veículo” é o hardware do robô, que precisa ser manipulado para sair de um ponto a outro. Assim como cada navio é diferente, cada hardware também é, tanto por sua estrutura quanto pelos motores e sensores utilizados.

O “piloto” é quem controla o veículo diretamente. No caso da robótica será a camada mais baixa, que controla os motores e realiza o controle cinemático ou dinâmico.

O “navegador” já não conhece detalhes do navio (hardware), apenas atua como

homem-do-meio entre o “piloto” e os “líderes do navio”. Ele que diz ao “piloto” que devem se mover do ponto X ao ponto Y, transformando o sistema de coordenadas conhecido em comandos que o piloto (máquina) saiba interpretar. Ele não funciona sem um sistema de coordenadas definido e sem saber sua posição atual. Ele seria a penúltima camada da arquitetura apresentada por ([VIEIRA, 2005](#)).

O “provedor de posição” é quem opera a bússola do navio, especialista em responder aonde estão. No robô os algoritmos de localização como odometria e uso de *beacons*, fazendo parte do sensoriamento (camada de percepção).

O mapa informa os obstáculos e o sistema de coordenadas para a tripulação, estando lá para ser analisado sempre que for preciso.

O “planejador de trajetória” é quem trabalha acima do “navegador”. O “navegador” sabe como ir de um ponto a outro, gerando uma trajetória apenas entre dois pontos com movimentos mecânicos. Para realizar uma viagem grande, evitando obstáculos do mapa, é preciso um “planejador de trajetória”. Ele estuda o mapa e desenha um percurso seguro com os pontos que devem ser atravessados.

Por fim o “planejador de missão” é o capitão do navio. Na robótica, pode ser o cérebro do robô, caso haja uma inteligência artificial ou o próprio programador, que define para onde se deve ir e o porquê.

([MATARIC, 2007](#)) define uma arquitetura para cada paradigma de definição de trajetória que apresenta. Para algoritmos globais (deliberativos), ele apresenta a arquitetura SPA (*sensor-plan-act*), que funciona em três camadas. A camada “*sensor*” cuida do mapeamento e sensoriamento (como a primeira camada de ([VIEIRA, 2005](#))), a camada “*plan*” define a trajetória e pesa os critérios de decisão (como a segunda e terceira camada) e a camada “*act*” que se comunica com os motores e realiza o movimento (como as últimas camadas).

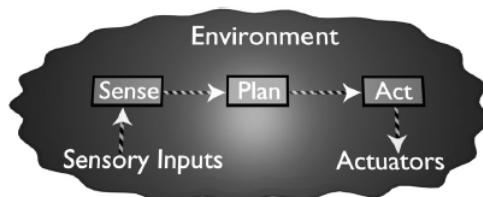


Figura 5 – Arquitetura SPA ([MATARIC, 2007](#))

2.3 Mapeamento e Modelagem dos Obstáculos

Um mapa é uma representação simplificada do mundo real dentro da memória do robô, eliminando informações desnecessárias à máquina, como cor e textura. Isso permite

um conhecimento prévio da região e agiliza a tomada de decisão, além de permitir um planejamento mais a longo prazo (MATARIC, 2007).

2.3.1 Tipos de Mapas

(CHOSET et al., 2005) descreve três formas de construir um mapa: topologia, geometria e por malhas. Um mapa topológico é baseado em grafo, onde cada nó representa um local distingível e as arestas são o caminho entre um local e outro. Qualquer ponto que possa ser diferenciável (uma sala, uma encruzilhada, uma marca no chão, entre outros) pode ser um nó. (MATARIC, 2007) descreve os mapas topológicos como um mapa baseado em “*landmarks*” (marcas). Por ser baseado nas características físicas do ambiente, as arestas ligando estes pontos não necessariamente são uma linha reta ou um corredor a ser seguido, mas uma série de movimentos que o leva até aquele ponto (segue reto, vira na primeira à direita e à esquerda na junção em “T”).

O modelo geométrico usa formas geométricas para representar o ambiente (CHOSET et al., 2005). Ele detecta o ambiente e os obstáculos com maior detalhamento e define um objeto com o formato mais próximo do real. (SECCHI, 2008) afirma que um mapa geométrico representa os objetos de acordo com suas relações geométricas absolutas como um mapa de polígonos e linhas. Ele também afirma que este modelo é empregado em navegação baseada em uma planificação (como uma planta baixa) com base em coordenadas geométricas e trajetórias pré-fixadas.

O modelo de malha de ocupação (*occupancy grid*) divide o espaço em uma matriz, onde cada célula representa um pequeno pedaço do ambiente e seu valor a probabilidade daquela célula estar ocupada. (BORENSTEIN; EVERETT; FENG, 1996) lista as vantagens e desvantagens da malha de ocupação, essa abordagem permite maior densidade de dados, requer menos processamento ao longo da definição de trajetória e é mais fácil de criar, porém possui áreas de incerteza, tem dificuldades com obstáculos dinâmicos e exige um processo de estimativa complexo.

2.3.2 Otimização do Processo de Mapeamento

O problema do mapeamento é o gasto de memória, que nem sempre é algo muito disponível em um robô. Uma forma de diminuir isso é mapear apenas aquilo que os sensores são capazes de detectar e desconsiderar detalhes que não estejam ligados com a posição. Relevo só é considerado quando a locomoção do robô for impedida por ele. Outra forma de economizar espaço, comentada por (SIEGWART; NOURBAKHSH, 2004) e (GUZMÁN et al., 2008) é descartar áreas côncavas nos obstáculos, ligando os vértices mais próximos que formem uma região convexa e considerando toda a área livre entre os pontos como região de risco e, portanto, intransponível.

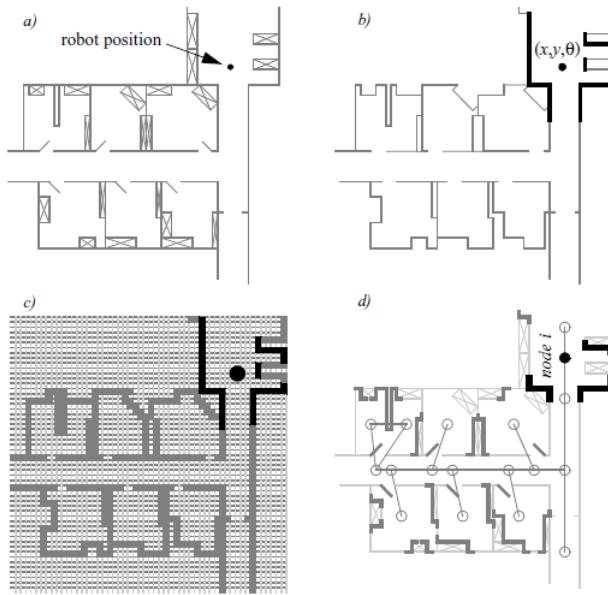


Figura 6 – diferentes tipos de mapeamento: a) mapa real b) mapa geométrico c) grade de ocupação d) topológico ([SIEGWART; NOURBAKHSH, 2004](#))

Porém, mesmo abstraindo o máximo de informações, o espaço ocupado na memória ainda será proporcional à quantidade de obstáculos na região. Assim, quanto mais denso de objetos for o mapa mais memória será ocupada e mais processamento será exigido ([SIEGWART; NOURBAKHSH, 2004](#)).

2.3.3 Definindo Percursos em um Mapa

Há duas abordagens para a busca pelo menor caminho nos mapas, os *roadmaps* e a decomposição celular.

Um *roadmap* é um mapa com um conjunto de posições específicas livres (nós). O *roadmap* liga duas posições através de um caminho livre, não necessariamente linear (aresta). ([CHOSET et al., 2005](#)) define um *roadmap* como uma classe de mapa topológico e o compara a um mapa de uma auto-estrada, que liga locais específicos por caminhos que se possam trafegar formando um grafo.

A decomposição celular funciona dividindo o mapa em regiões livres, aonde qualquer lugar dentro daquela região é navegável. As regiões vizinhas são deslocáveis de uma para outra, podendo passar pela borda entre eles sem problemas. Assim, os algoritmos que usam essa abordagem ligam as regiões vizinhas como nós adjacentes em um grafo, dizendo que aquelas duas regiões são trafegáveis entre si. As regiões podem ter todos o mesmo tamanho ou variar drasticamente, mas o robô deve sempre caber em seu interior.

A decomposição celular é dividida em dois sub-gupos: a exata, aonde cada região tem exatamente o formato da área livre, sem deixar nenhuma parte do mapa fora de uma

das regiões; e a aproximada, aonde cada região tem um formato pré-determinado (embora o tamanho possa ser variável) e pode desconsiderar pequenas partes do mapa, causando possíveis perdas de trajetos (SOUZA, 2008).

2.4 Algoritmos Globais de Definição de Trajetória

Como comentado anteriormente, o foco do trabalho é na definição de trajetória usando algoritmos globais. Este grupo de algoritmos já conhecem o ambiente, tendo um mapa da região e dos obstáculos presentes. A partir deste mapeamento é definida uma trajetória para o robô percorrer sem colidir com obstáculo algum.

Para fazer essa análise, os algoritmos transformam o mapa em um grafo e depois rodam um algoritmo de melhor caminho nele (como por exemplo Djikstra ou A*) para definir a trajetória. Cada algoritmo gera um grafo diferente e, portanto, uma trajetória diferente. Porém, após definido o grafo, todos funcionam de forma semelhante, rodando o algoritmo de Djikstra ou semelhante para solucioná-lo e, criando o percurso (GUZMÁN et al., 2008). A Figura 07 mostra o diagrama de sequência que ilustra esse funcionamento, onde a classe *PathPlanning* representa a classe do sistema projetado que abstrai o funcionamento.

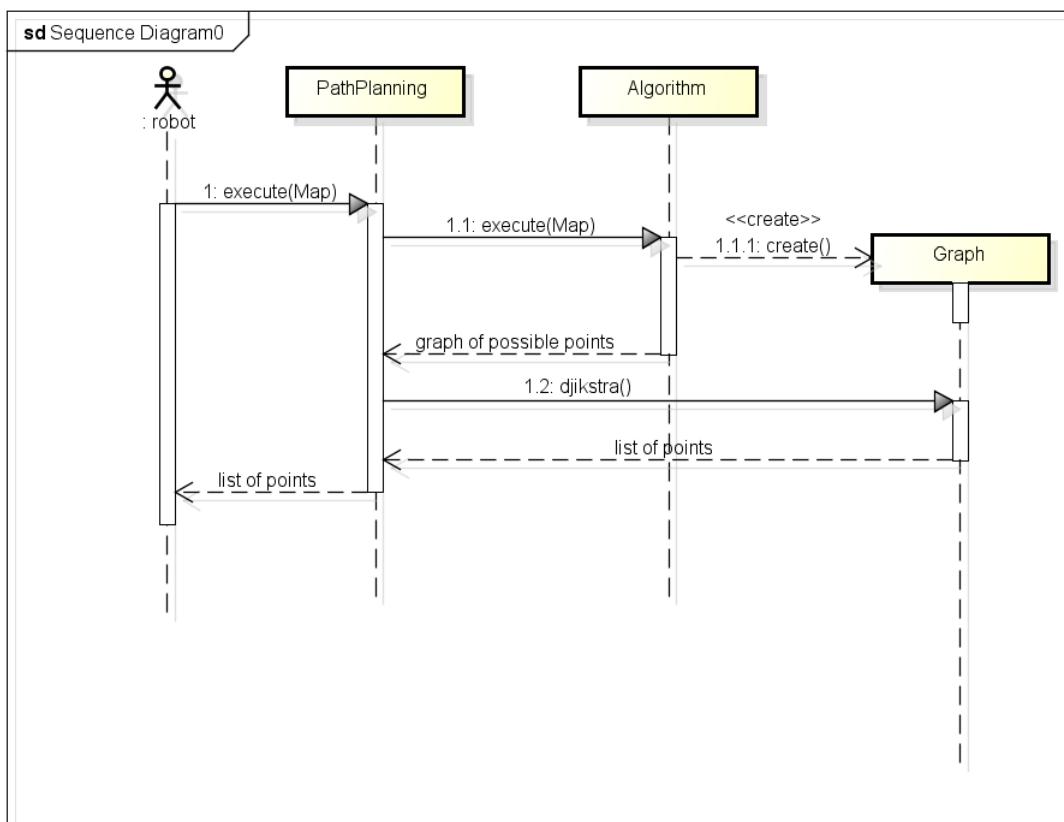


Figura 7 – Diagrama de sequência da criação do caminho

Assim, a camada de planejamento de trajetória recebe das camadas acima um mapa com obstáculos, deve ter acesso à estrutura do robô (como seu tamanho) e passa para as camadas abaixo uma lista de pontos que o robô deverá seguir. Será esta lista de pontos (nós do grafo definidos pelo algoritmo de melhor caminho) que formarão a trajetória, cabendo à camada abaixo o controle dos motores para ir de um ponto ao outro.

Os algoritmos globais estão divididos em dois grupos, cada um utilizando uma abordagem para trabalhar com o mapa: os *roadmaps* e a decomposição celular.

O *roadmap* (Grafo de Visibilidade, Voronoi) é um mapa em forma de grafo que representa um conjunto de caminhos livres, ligando posições específicas do mapa. Cada nó representa uma posição ou área livre e suas arestas representam um caminho livre entre os mesmos.

A decomposição celular (*Quad Tree* e *Wavefront*) em que o espaço é dividido em regiões geométricas menores totalmente livres ou totalmente ocupadas e liga as regiões vizinhas se for possível se deslocar entre elas (SOUZA, 2008).

Os algoritmos explicados a seguir foram implementados neste trabalho e estão disponíveis para uso no *framework*.

2.4.1 Grafos de Visibilidade

A definição de um Grafo de Visibilidade é baseado no conceito de pontos visíveis, que são pontos (inicial, final e vértices dos obstáculos) que podem ser ligados por uma linha reta sem passar por obstáculo algum (GUZMÁN et al., 2008). Em outras palavras, se alguém estiver parado em um ponto, ele estará ligado a todos os pontos que conseguir enxergar a partir desse ponto. (SOUZA, 2008) diz que a complexidade de um Grafo de Visibilidade depende da complexidade da geometria de seus obstáculos.

Este método permite alcançar sempre o menor caminho, porém ele passa sempre o mais perto possível dos obstáculos. Como o robô é considerado apenas um ponto (seu centro de rotação que, no caso dos robôs *differential steering*, é o centro do eixo das rodas dianteiras) ele se colidiria com os vértices dos obstáculos. Para evitar isso, (SOUZA, 2008), (SIEGWART; NOURBAKHSH, 2004) e (HOLDGAARD-THOMSEN, 2010) dizem que os obstáculos devem ser expandidos, sendo considerados maior do que realmente são. Essa margem de erro deve ser maior do que metade da largura do robô para que não haja contato.

Segundo (HOLDGAARD-THOMSEN, 2010) e (CHOSET et al., 2005), a complexidade deste algoritmo é $O(n^3)$, sendo $O(n)$ para saber se dois vértices são visíveis ou não e $O(n^2)$ para percorrer todos os vértices e executar este algoritmo em todos os possíveis vizinhos. (HOLDGAARD-THOMSEN, 2010) diz que essa complexidade pode ser reduzida

para $O(n^2 \log(n))$. (MEDEIROS; SILVA; YANASSE, 2011) sugere um método para descartar nós e arestas sem prejudicar o resultado final. Já (CHOSET et al., 2005) apresenta um método para descarte de arestas dentro de uma região. (SIEGWART; NOURBAKHS, 2004) comenta que por gerar demasiados vértices e arestas, este algoritmo funciona melhor em ambientes esparsos, gerando menor gasto de processamento.

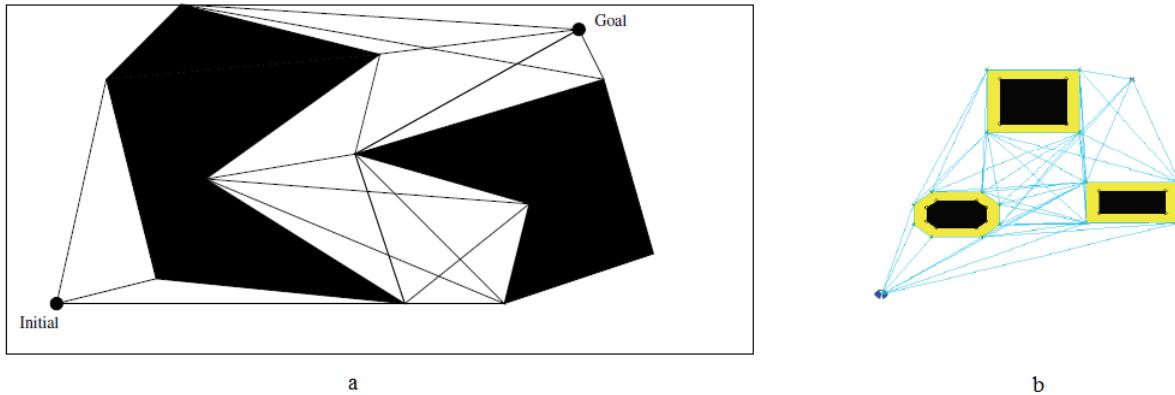


Figura 8 – Grafos de visibilidade a) tradicional b) com expansão dos obstáculos (HOLDGAARD-THOMSEN, 2010) e (Site do MRIT, 2014)

2.4.2 Voronoi

O Diagrama de Voronoi é utilizado em diversos contextos. É possível encontrar aplicações dele na geofísica, química, meteorologia, biologia, entre outros (Site do Voronoi, 2014).

O diagrama consiste em dividir a região em áreas poligonais menores chamadas células, em que cada ponto de uma célula esteja mais próximo do seu centro do que do centro de outra célula. Cada célula é definida a partir deste ponto central, que deve estar igualmente distante dos obstáculos e dos limites do mapa (GUZMÁN et al., 2008). O ponto central das células são os vértices do grafo e ligados a todos os vértices vizinhos (com os quais sua célula faz fronteira). Devido aos vértices estarem o mais distante possível dos obstáculos, o algoritmo gera um caminho maior entre os pontos inicial e final. Entretanto, garante que o robô passará longe de qualquer obstáculo, dando maior prioridade à segurança do que ao percurso.

(SIEGWART; NOURBAKHS, 2004) e (CHOSET et al., 2005) comentam que essa abordagem possui um risco agregado. Por buscar sempre o caminho mais distante dos obstáculos, caso o robô não possua sensores de longo alcance, o robô não poderá confirmar sua posição exata. Sem obstáculos e objetos para captar, o robô não terá o que usar para estimar sua posição atual e terá de percorrer o percurso sem ter certeza de onde exatamente está ou se permanece no percurso correto.

Para montar as células, o algoritmo primeiramente precisa receber um conjunto de pontos, que são os vértices dos obstáculos. Com estes pontos, o algoritmo executa um algoritmo de triangulação para determinar o ponto central do triângulo e a partir deles o centro de cada célula. A triangulação de Delaunay é a opção mais comum ao Diagrama de Voronoi ([SOUZA, 2008](#)). Após realizar a triangulação, são excluídas as interseções com obstáculos e triângulos formados dentro dos mesmos. Por fim, são adicionados os pontos inicial e final e ligados aos centros das células próximas. Ao rodar um algoritmo de menor percurso (Dijkstra ou A*), ele passará por estes pontos, sempre pelo meio das células até o objetivo.

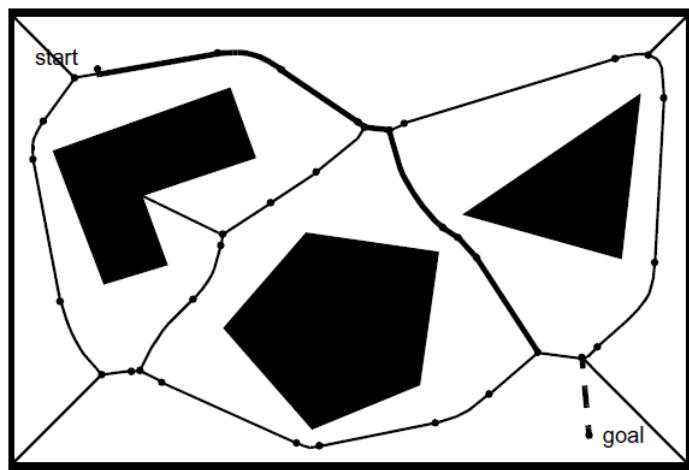


Figura 9 – Diagrama de Voronoi ([SIEGWART; NOURBAKHSH, 2004](#))

2.4.3 Quadtree

A decomposição aproximada, também chamada de Quadtree, divide o mapa em regiões e executa um método recursivo para continuar dividindo cada região em áreas cada vez menores ([HOLDGAARD-THOMSEN, 2010](#)). Essa divisão recursiva pode parar de duas formas: ou quando chegar em um tamanho mínimo ou quando toda sua região estiver livre ou ocupada. Ainda segundo ([HOLDGAARD-THOMSEN, 2010](#)), esse método recebe este nome porque uma região é dividida em quatro células menores de mesma forma cada vez que se decompõe.

Assim, o método do Quadtree divide o mapa em áreas de formato pré-determinado, porém com tamanhos diferentes de acordo com o espaçamento entre os obstáculos. Isso pode gerar a perda de caminhos estreitos por serem colocados na mesma área que uma parte do obstáculo. Isso pode ser resolvido diminuindo a área mínima de cada região, porém acarretará no aumento de memória gasta e tempo de processamento da solução do trajeto. ([SIEGWART; NOURBAKHSH, 2004](#)) exemplifica este caso na Figura 10.

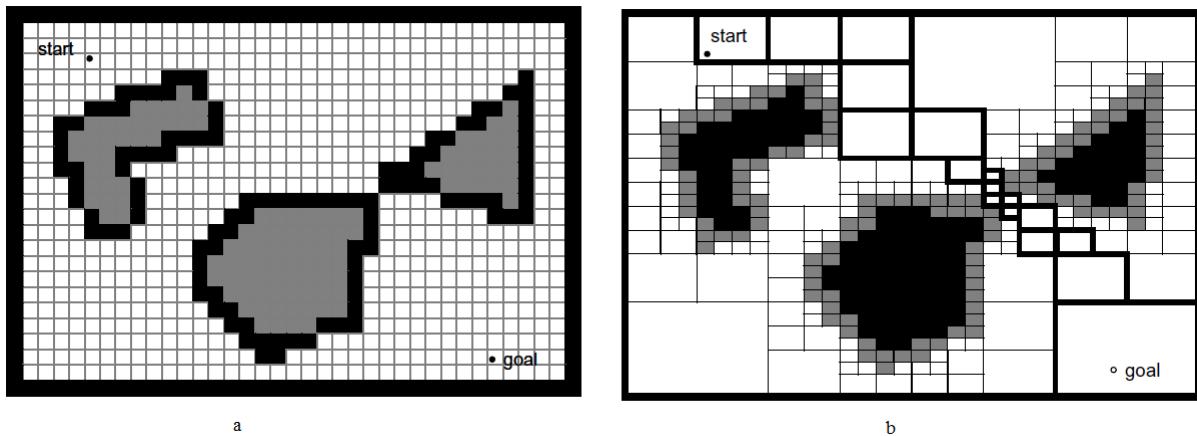


Figura 10 – a) decomposição celular usando regiões de tamanho fixo b) Quadtree encontrando um caminho estreito entre dois obstáculos ([SIEGWART; NOUR-BAKHSH, 2004](#))

Por fim, cada área será um vértice do grafo retornado e estará ligado a cada área adjacente livre. Os vértices ocupados por obsáculos são retirados, permanecendo apenas os trajetos livres de uma área adjacente a outra.

2.4.4 Wavefront

O algoritmo Wavefront divide o mapa em pequenas células do mesmo tamanho e igualmente distribuídas, formando uma matriz de posições. Esse algoritmo funciona perfeitamente com mapas do tipo malha de ocupação, pois já está dividido em *grids*. Cada célula possuirá um valor que representará o quão perto está da posição final.

O algoritmo começa na posição final e dá um valor inicial qualquer à célula. Todas as casas em volta dele recebem o mesmo valor mais 1. Por sua vez, as casas ao redor dessas casas acrescentadas de 1 recebem o valor delas mais 1, e assim em diante. Assim, cada célula terá o valor inicial mais o número de passos para chegar até ele. Quando uma célula encontra um vizinho com um valor já estipulado ela troca seu valor apenas se o valor atual do seu vizinho for maior que o dela mesma mais um.

Este método permite chegar ao objetivo a partir de qualquer célula. Células com obstáculos são desconsiderados pelos vizinhos.

Ao finalizar, um grafo é feito ligando o ponto inicial à todas as células vizinhas com menor valor. Essas células por sua vez são ligadas às vizinhas com menor valor e assim por diante até todas as células passarem por essa função. Isso fará com que nem todas as células entrem no grafo, sendo inserido nele apenas os caminhos por onde o valor diminui. A Figura 11 exemplifica a montagem do grafo.

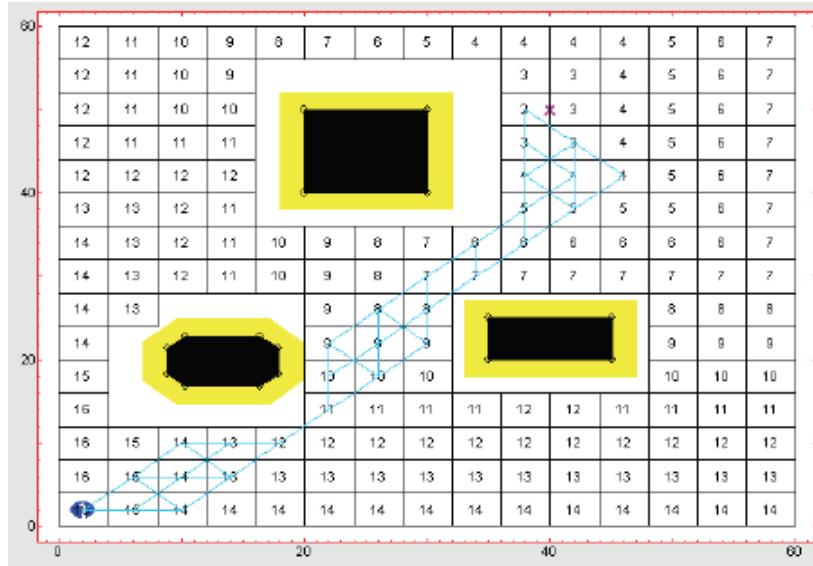


Figura 11 – Algoritmo *Wavefront*, com grafo em azul ([Site do MRIT, 2014](#))

2.5 Frameworks e padrões de projeto

Escrever um código reutilizável e extensível não é fácil. Para garantir que o mesmo código possa ser replicado em diferentes contextos com nenhuma ou mínima alteração, é preciso uma arquitetura bem projetada e uma correta modularização das tarefas do sistema. Uma vez que é conhecido o escopo com que irá trabalhar, um planejamento da melhor forma de estruturá-lo é necessário para garantir que todas as funcionalidades sejam separadas e suas comunicações o menos acopladas possível. Isso leva ao estudo de padrões arquiteturais, encapsulamento e padrões de projeto, que nos ajudam a realizar essa tarefa.

A definição de *framework*, segundo ([SZYPERSKI, 2002](#)), é um conjunto de classes cooperativas, algumas dos quais podem ser abstratas, que criam um projeto reutilizável para um específico nicho de software. Outra definição mais simples dada por ([FAYAD; JOHNSON; SHMIDT, 1999](#)) é “uma composição de classes cujas colaborações e responsabilidades são especificadas”. Dentro de um *framework* há um conjunto de módulos que são responsáveis por funcionalidades e tarefas específicas, encapsulando dentro de si alguma responsabilidade e isolando sua implementação para a fácil utilização pelo usuário do *framework*. Porém, pela sua característica de extensibilidade, o *framework* deve permitir que parte de sua estrutura seja aberta e visível ao programador para extender e configurar essas classes. Espera-se que haja sempre classes que implementem a execução básica de uma ação, das quais o usuário possa herdar para especificar ou configurar um comportamento sem se preocupar em escrever todo o trabalho. Frequentemente um *framework* provê uma implementação padrão.

([LARMAN, 2005](#)) lista algumas características de um *framework*, como:

- ter um conjunto coeso de interfaces e classes que colaboram para fornecer um serviço;
- implementar as funções básicas e invariantes do sistema e facilitar a implementação da parte específica de variante do escopo;
- conter classes concretas e (especialmente) abstratas, que definem interfaces a serem seguidas e,
- definir o que é imutável e recorrente em todos os sistemas deste tipo, funções e comportamentos gerais e separá-los do que é específico da aplicação.

(FAYAD; JOHNSON; SHMIDT, 1999) diz que, apesar do que possa parecer a primeira vista, o benefício primário do *framework* não é uma implementação reutilizável, mas a estrutura reutilizável que ele descreve. Um *framework* provê reutilização em três níveis: de implementação, de projeto e de análise. Ao fazer a análise de domínio ao criar um *framework*, são levantadas todas as questões pertinentes sobre o mesmo e aquilo que era comum a todos os casos já está analisado e documentado. Com isto, bastará fazer a análise das questões específicas do seu problema. A documentação do *framework* possui o projeto do sistema quase pronto, fornecendo o "esqueleto" do seu sistema. E o código do *framework* funcionando, com seus *hot-spots* e componentes funcionais entrega parte do sistema já desenvolvido e testado.

Nos tópicos a seguir são explicados os principais focos ao se construir um *framework*, seguindo as boas práticas da engenharia de software.

2.5.1 Arquitetura e Componentes

Construir um *framework*, assim como todo sistema grande e complexo, exige um planejamento e uma modelagem prévia. Esse planejamento leva à construção de uma arquitetura, definindo módulos, componentes e suas interações.

Uma resumida definição de arquitetura de software é “um conjunto de decisões significativas sobre a organização de um sistema de software, considerando as interfaces pela qual o sistema é composto, seus comportamentos e as colaborações entre os elementos” (LARMAN, 2005).

Dividir o projeto em grupos menores ajuda a organizar o código, garantir que todas as funcionalidades estão desenvolvidas e diminui o acoplamento entre as partes. Ao definir o que é visível em um módulo e como operá-lo (interface) e como os componentes se comunicam fica mais fácil testar, isolar funcionalidades e garantir o bom funcionamento de cada parte do sistema e dele como um todo.

Essa divisão de tarefas e responsabilidades levam ao estudo de técnicas de como fazer isso da melhor forma. Os princípios GRASP definem algumas preocupações que

devem ser consideradas ao se projetar uma arquitetura, como quem deve criar um objeto, quem deve conter dados e referências de uma determinada classe, como garantir que os módulos e classes são minimamente dependentes uns dos outros e se uma funcionalidade ou método está na classe correta. Essas preocupações levaram à criação dos padrões de projeto, que implementam soluções para esses e outros problemas. Para saber mais sobre os princípios GRASP veja em ([LARMAN, 2005](#)) capítulos 17, 18 e 25.

Existem diversas formas de se estruturar uma arquitetura, os mais comuns são a divisão por camadas ou por componentes. Segundo ([LARMAN, 2005](#)), na divisão por camadas (como já mostrado na seção 2.2.5 sobre arquitetura de navegação), cada camada realiza uma função distinta, presta serviço às camadas acima e se utiliza das camadas abaixo. Quanto mais baixa a camada, mais geral são suas funcionalidades e quanto mais alta, mais específica da aplicação se torna. Uma arquitetura por camadas pode ser rígida (se comunica apenas com a camada logo abaixo e atende apenas a camada logo acima) ou relaxada (se comunica com qualquer camada abaixo da sua). Normalmente sistemas de software usam camadas relaxadas em sua estrutura arquitetural.

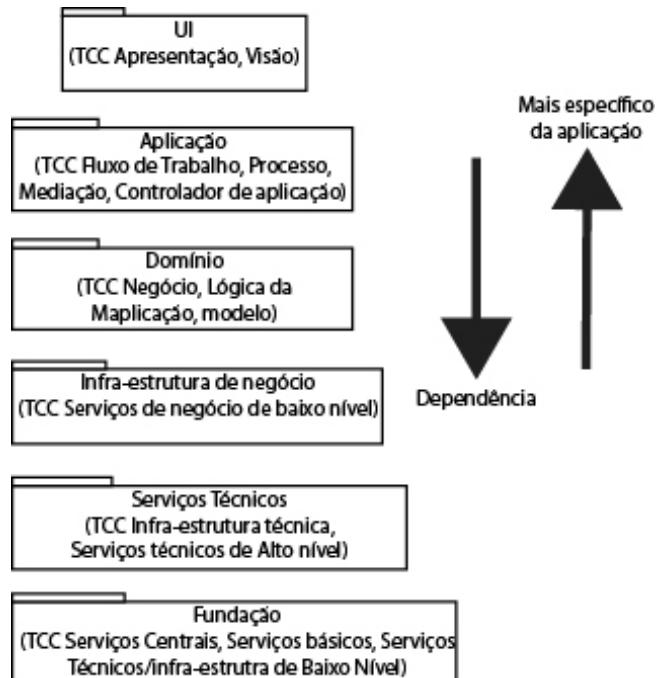


Figura 12 – Exemplo de arquitetura em camadas, ([LARMAN, 2005](#))

Já uma arquitetura orientada a componentes separa as funcionalidades em pacotes e as disponibiliza para uso por qualquer outro componente. Um componente, módulo ou pacote, é um sub-conjunto de classes que realizam uma tarefa específica e centralizam nela toda a responsabilidade por lidar com isso. O componente em geral trabalha de forma caixa-preta, não fornecendo acesso a detalhes de implementação da solução. ([SZYPERSKI, 2002](#)) define um componente como “uma entidade de composição com interfaces especificadas contratualmente e somente explícita no contexto da dependência.

Um componente de software pode ser desenvolvido pelo próprio usuário como parte do sistema ou um módulo de terceiros com o qual se comunica”. ([SZYPERSKI, 2002](#)) também defende que cada módulo deve ser o mais independente possível e não possuir um estado observável externamente.

([GOODLIFFE, 2007](#)) trás também outros dois padrões de arquitetura: cliente e servidor e *pipe-and-filter*. A arquitetura cliente-servidor é muito usada em redes e sistemas distribuídos, aonde um programa requisita dados e/ou serviços de outro. O servidor provê uma série de serviços bem definidos e uma interface conhecida para receber as requisições de serviço. O cliente consome esses serviços e deve conhecer a interface do servidor para saber como pedir um dado ou serviço. A arquitetura *pipe-and-filter* define todo componente como um filtro, que recebe um dado, trata-o e o repassa para outro filtro. Todo componente funciona com um dado de entrada e liberando um de saída. As interações entre os filtros são chamadas de *pipes*. Um exemplo é o terminal do Linux, onde pode dar vários comandos na mesma linha separados pela barra vertical (*pipe*) e o resultado de um serve de entrada para o seguinte.

Independente da abordagem, as boas práticas de projeto (GRASP) e os padrões de projeto de ([GAMMA et al., 1995](#)) são seguidos para garantir baixo acoplamento, alta coesão, reusabilidade e clareza do código. O uso de polimorfismo e interfaces é incentivado e a abstração é vista como ponto chave do projeto da arquitetura. ([SZYPERSKI, 2002](#)) diz que a abstração é talvez a mais poderosa ferramenta para um engenheiro de software. A abstração encapsula o problema em uma visão simplificada e assim deve se manter. “Encapsulamento diz que você não só está permitido a ter uma visão simplificada, diz que você não está permitido a ver mais detalhes que os fornecidos. O que você vê é tudo o que você tem” ([MCCONNEL, 2004](#)).

Também é comum à construção de qualquer arquitetura a divisão das tarefas em dois grupos: módulos (ou camadas) e suas interações. Definir como as funcionalidades estarão estruturadas e separá-las em módulos é só metade do trabalho. É preciso também haver um esforço em como elas se comunicarão e que essas comunicações sejam as mais simples possíveis. ([SZYPERSKI, 2002](#)) também diz que o maior esforço dos arquitetos de software consiste em diminuir a complexidade das interações dos objetos. Quanto maior a interação, mais o acoplamento cresce e é algo a ser evitado ao máximo.

2.5.2 Padrões de Projeto

Padrões de projeto são descrições de objetos e classes comunicantes que precisam ser personalizadas para resolver um problema geral de projeto num contexto particular ([GAMMA et al., 1995](#)). Esses padrões são soluções eficientes para problemas corriqueiros no desenvolvimento de software. ([GAMMA et al., 1995](#)) catalogou estes padrões e descreveu cada um, juntamente com seu problema, em seu livro, fornecendo uma biblioteca de

padrões bem aceitos e difundidos na comunidade.

Em um projeto de *framework* certos padrões se destacam. O padrão *Template Method* é destacado por (LARMAN, 2005) e (FAYAD; JOHNSON; SHMIDT, 1999) por fornecer pronta a estrutura padrão para esse tipo de operação com as características constantes já implementadas e permitir a implementação daquilo que é diferente no contexto do sistema. O padrão Fachada também é comentado por (LARMAN, 2005) por fornecer uma interface única e simples para uma tarefa que usa várias classes e objetos. O princípio de *Hollywood* ou princípio da Inversão de Controle (implementado pelo padrão Injeção de Dependência) também é mencionado por (LARMAN, 2005) por diminuir o acoplamento entre as camadas e forçar a comunicação por interfaces, que dá um comportamento mais geral e também é outro princípio de boa programação.

Outros padrões comentados por (LARMAN, 2005) e (SZYPERSKI, 2002) por serem muito utilizados em *frameworks* são o *Observer*, *Command*, *Composite*, *Strategy* e *Decorator*. (GAMMA et al., 1995) descreve detalhadamente todos estes padrões.

2.5.3 Framework Caixa-preta, Caixa-branca e Caixa-cinza

(SZYPERSKI, 2002) descreve *frameworks* e componentes como podendo ser caixa-preta, caixa-branca ou caixa-cinza. Um *framework* caixa-preta só revela suas interfaces e como usá-las. Neste tipo de abordagem não há tipo algum de informação sobre seu comportamento ou subclasses. Essa abordagem é raramente usada para *frameworks* e mal aconselhada pelo autor, se encaixando melhor para projeto de componentes. (SZYPERSKI, 2002) e (FAYAD; JOHNSON; SHMIDT, 1999) dizem que, ainda assim, o sistema caixa-preta não é todo fechado à extensão. Novos comportamentos podem ser adicionados através de *plugins* que extendem das interfaces e unidos via composição. Porém, (FAYAD; JOHNSON; SHMIDT, 1999) também comenta que este tipo de abordagem é mais difícil de desenvolver.

Já a abordagem caixa-branca revela sua estrutura, permite extensão de suas classes e personalização de seu comportamento. Ela é muito mais usada para *frameworks* e aconselhada por (SZYPERSKI, 2002). Já (FAYAD; JOHNSON; SHMIDT, 1999) diz que, apesar de ser largamente utilizada, a estrutura caixa-branca exige que o programador conheça profundamente a estrutura interna do *framework* e que o uso de herança é muito mais comum em caixa-branca, enquanto no caixa-preta usa-se mais composição e delegação. Em comparação com o modelo caixa-preta, este é mais fácil de se desenvolver, porém mais difícil de estender.

O padrão caixa-cinza revela alguns detalhes de seu funcionamento e permite extensão e colaboração com alguns de seus pacotes, porém mantém algumas partes totalmente fechadas em seus componentes. *Frameworks* caixa-cinza tem flexibilidade suficiente

e extensibilidade, e ainda tem a habilidade de esconder informações desnecessárias do desenvolvedor da aplicação (FAYAD; JOHNSON; SHMIDT, 1999).

2.5.4 *Hot-spot* e *Frozen-spot*

Frameworks são adaptados para personalização e extensão das estruturas que fornece. As partes do *framework* abertas para extensão e personalização são chamadas de *hot-spot* (FAYAD; JOHNSON; SHMIDT, 1999). *Hot-spots* expressam aspectos variantes do domínio e devem ser levantadas na fase de análise da construção do *framework*. *Hot-spots* são uma parte muito importante do *framework*, pois são neles que o usuário tem a chance de construir algo reutilizando as partes já desenvolvidas. *Hot-spots* deixam ganchos para que o usuário herde de classes abstratas ou implemente interfaces, e defina seus próprios comportamentos.

Frozen-spot, por sua vez, são blocos fechados do *framework*. São componentes que não permitem extensão e são apenas utilizados da forma como são apresentados. Enquanto um *hot-spot* está semi-pronto, o *frozen-spot* já está pronto para uso e não pode ser alterado. Quando padrão, o comportamento *default* é colocado aos métodos de uma classe abstrata.

Determinar quais são os *hot-spots* é uma das principais tarefas do desenvolvedor de *frameworks*. Cada *hot-spot* deve possuir uma interface para padronizar a comunicação e definir qual deve ser o comportamento daquele módulo. O *framework* pode ou não vir com alguma implementação padrão e básica da interface.

Cada *hot-spot* deve ser bem especificado pelo desenvolvedor do *framework* para que o usuário saiba como construir suas customizações. (FAYAD; JOHNSON; SHMIDT, 1999) defende que uma profunda análise do domínio e dos *hot-spots* devem ser realizados, documentando uma análise alto nível dos *hot-spots*, uma especificação mais aprofundada dessas tarefas, o desenho alto nível de seus subsistemas e por fim a codificação desse desenho em uma classe abstrata e generalizações (se houver). Em um *framework* caixa-preta deve haver uma série de implementações da interface já prontas para o usuário escolher qual utilizar. Já o *framework* caixa-branca o usuário deve desenvolver sua própria classe. Isso pode ser trabalhoso se a documentação da interface estiver mal feita ou incompleta. A Figura 13 ilustra a diferença entre as duas abordagens quanto aos ganchos *hot-spots*.

Um subsistema *hot-spot* pode conter apenas a classe base e suas extensões (subsistema baseado em herança) ou pode ter ainda classes adicionais e relacionamentos internos (subsistema baseado em composição). Na primeira opção costuma-se usar o Padrão Fábrica ou *Template* para comunicação externa, enquanto o subsistema por composição, que é mais complexo, exige um padrão mais complexo também, como o *Strategy*.

(FAYAD; JOHNSON; SHMIDT, 1999) classifica os subsistemas *hot-spot* em três categorias de acordo com o número e a estrutura das subclasses que possui. Essas es-

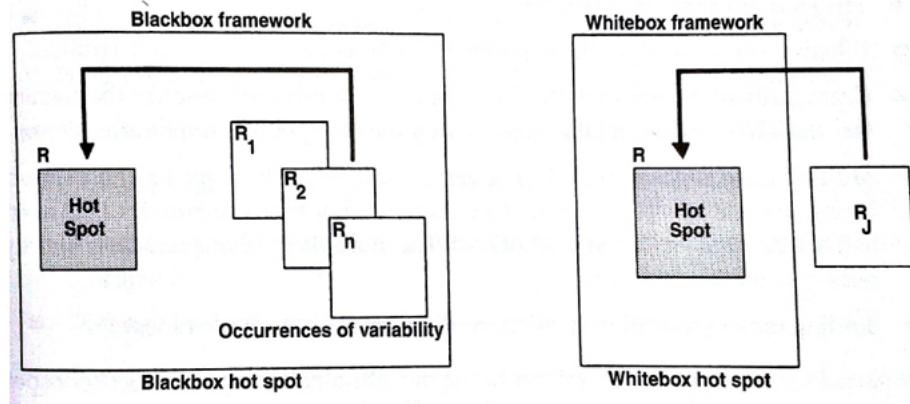


Figura 13 – *Hot-spot* em um *framework* caixa-preta e caixa-branca. (FAYAD; JOHNSON; SHMIDT, 1999)

truturas podem ser não-recursiva se o serviço é realizado por apenas uma subclasse (o caso normal), recursão estruturada em cadeia (*chain-structured recursive*) caso o serviço possa ser realizado por várias subclasses estruturadas em cadeia ou recursiva estruturada em árvore (*tree-structured recursive*) caso o serviço possa ser realizado por uma árvore de várias subclasses. Essa estrutura não é observável de fora do subsistema, que ao receber uma mensagem chama recursivamente os objetos (no caso de ser recursivo), através de um método *Template* até percorrer toda a lista de objetos filhos ou chegar na folha da árvore.

O tipo de um subsistema *hot-spot* é definido pelo padrão de projeto implementado. Cada padrão provê variabilidade e flexibilidade de um modo diferente, causando as diferenças entre as três categorias citadas anteriormente. Os não-recursivos são implementações de *Interface Inheritance*, *Abstract Factory*, *Builder*, Método Fábrica, Protótipo, *Strategy*, *Template*, *Visitor*, *Adapter*, *Bridge*, *Proxy*, *Command*, *Iterator*, *Mediator*, *Observer* e *State*. O recursivo estruturado em cadeia é implementado com *Chain of Responsibility* e *Decorator* e o recursivo estruturado em árvore é implementado com *Composite* e *Interpreter*.

Em resumo, um *framework* caixa-branca costuma ser baseado em herança ou interfaces, enquanto um caixa-preta é baseado em composição. Apesar das diferenças entre as duas formas, ambas possuem em sua estrutura interna superclasses e subclasses utilizadas para solucionar um problema, mudando como o usuário do *framework* pode se comunicar com essas classes (herdando ou compondo-as em suas próprias classes). Em ambos os casos, o padrão *Template Method* é fortemente usado. O uso deste padrão fornece ganchos para inserir seus próprios comportamentos, sendo por isso este padrão comumente ligado a *hot-spots*. Locais onde o comportamento varia de aplicação para aplicação e o *framework* se abre para que o usuário defina o comportamento é a definição de um *hot-spot*, e também exatamente o que o padrão permite realizar. Quando um módulo possui uma série de

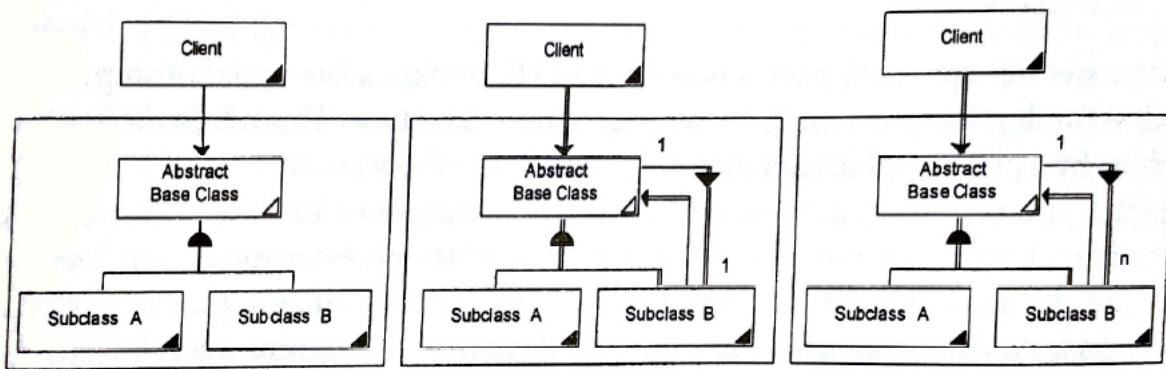


Figura 14 – Estruturas dos subsistemas *hot-spot*. (FAYAD; JOHNSON; SHMIDT, 1999)

objetos semelhantes ou com funções semelhantes o *framework* pode criar uma estrutura recursiva em cadeia ou árvore (ao invés da referência direta) para guardá-los e manter as chamadas a eles.

2.5.5 Processo de Desenvolvimento Orientado a *Hot-Spots*

A modelagem do domínio, análise dos pontos de variação e seu projeto são atividades iterativas, sendo sempre incrementadas e aperfeiçoadas. Assim, a modelagem de um *framework* é uma tarefa iterativa e incremental, sendo constantemente alterada, melhorada e aditivada em termos de novos *hot-spots*. Os *hot-spots* já existentes são avaliados se foram corretamente projetados e se fornecem abertura suficiente para personalização pelo usuário, mantendo o baixo acoplamento e a alta coesão. Todo esse processo gira em torno dos *hot-spots*, permanecendo até que todo o modelo de domínio tenha sido esmiuçado e os *hot-spots* levantados tenham sido analisados, projetados e implementados. A Figura 15 esboça o fluxo de trabalho do processo orientado aos *hot-spots*.

A definição de um modelo de objeto específico é a construção de seu modelo de domínio, realizado no levantamento de requisitos. A cada porção do domínio levantada são identificados *hot-spots*. Essa atividade é realizada tanto com o engenheiro de software quanto com um especialista no domínio tratado, que conhece das variâncias do assunto. A comunicação do engenheiro com o especialista pode ser complicada pelo especialista conhecer da área, porém desconhecer sobre classes, objetos, herança e *hot-spots*. Para facilitar a comunicação entre os dois e a identificação de novos *hot-spots* (FAYAD; JOHNSON; SHMIDT, 1999) sugere a utilização de *hot-spot cards*. Essa relação é muito semelhante a do analista de requisitos, que conhece do contexto do software buscando montar o modelo de negócio com o cliente, que nada conhece. Nesse caso, são utilizados *CRC Cards* e *User Stories*.

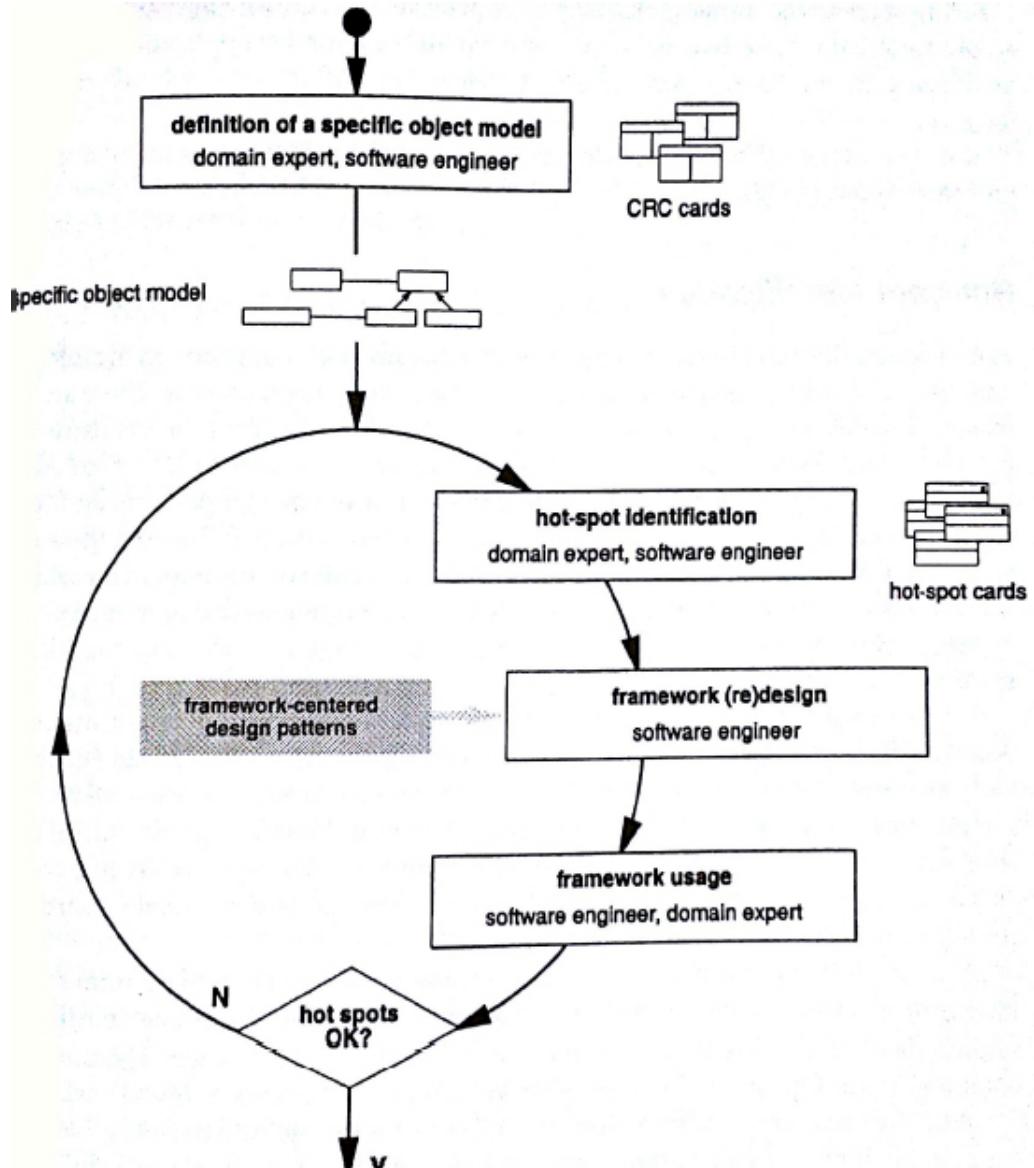


Figura 15 – Processo de desenvolvimento orientado a *hot-spot*. ([FAYAD; JOHNSON; SHMIDT, 1999](#))

O projeto e reprojeto do *framework* ocorre após os *hot-spots* tiverem sido identificados e documentados. Essa atividade trata de projetar a abstração do *hot-spot*, permitindo a extensibilidade no módulo e mantendo acoplamento baixo e a coesão alta. Padrões de projeto são utilizados para alcançar este fim. É nesta atividade que é analisado se o *hot-spot* será não-recursivo ou recursivo a partir das decisões de projeto. A constante reavaliação dos *hot-spots* e possíveis reprojetos podem ocorrer em futuras iterações se forem identificados problemas na modelagem atual.

A última atividade da iteração é por fim implementar as decisões arquiteturais e de projeto da atividade anterior e seu uso afim de detectar falhas no mesmo. Vale lembrar que o projeto e a implementação seguem os *hot-spot cards* como guia, que lhes dá uma forte tendência de como realizar aquele *hot-spot*.

Um *hot-spot card* contém informações sobre a semântica e o desejado grau de flexibilidade, mas não sobre qual classe o *hot-spot* pertence (FAYAD; JOHNSON; SHMIDT, 1999). Passar da análise para algo com valor arquitetural se torna relativamente simples com essa abordagem, pois o autor fornece uma relação de como implementar de acordo com o grau de flexibilidade. Porém essa integração depende de uma boa granularidade da função *hot-spot*. Uma função *hot-spot* com a granularidade correta implica que um método de gancho ou um grupo deles tem que ser adicionado. O local onde inserir o método gancho depende do grau de flexibilidade.

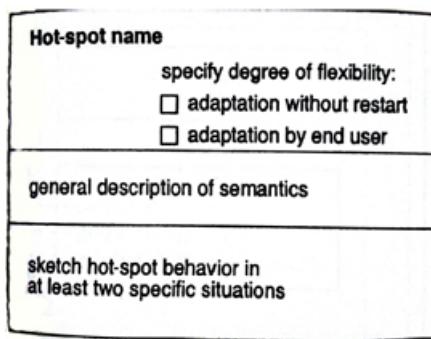


Figura 16 – *Hot-spot card*. (FAYAD; JOHNSON; SHMIDT, 1999)

Caso ambas as opções de grau de flexibilidade estejam desmarcadas, um método gancho deve ser adicionado (caso normal do método *Template*). Caso a primeira opção seja marcada (sem reinício), esse método gancho deve ser implementado em uma classe gancho separada (gerando indireção para uma melhor abstração). Caso a segunda opção seja marcada (adaptação pelo usuário final), deve-se realizar uma forma de configuração do uso do método ou classe gancho. Com os dois selecionados, ambos, configuração e indireção, devem ser feitos.

2.5.6 Abordagem de Projetos

(SZYPERSKI, 2002) mostra outra maneira de projetar um *framework*, mais simples e com menos foco em *hot-spots*. Segundo ele, o projeto de *frameworks* pode ter duas abordagens: *bottom up* (orientado a padrões) ou *top down* (orientado a alvos).

A abordagem *bottom up* funciona bem se o domínio já está bem entendido e conhecido. Por já se conhecer os problemas e ter uma boa visão de como será o final, os componentes vão sendo desenvolvidos e os problemas atacados sem demora. Padrões são usados para o desenvolvimento dos componentes e para suas interações, o que confere à orientação (orientado a padrões). Os módulos são desenvolvidos até ter resolvido todos os problemas e conectá-los corretamente. Isso pode levar a uma codificação exagerada, indo além do necessário e fornecendo um monte de soluções fragmentadas e sem foco definido.

A abordagem *top down* é preferível se o domínio ainda não foi suficientemente explorado, mas os problemas a serem resolvidos estão bem definidos. Em vez de focar na solução e sua implementação como o anterior, este foca nos problemas e vai construindo a solução a partir deles, por isso o nome orientação a alvos. Neste contexto, um alvo é definido como um conjunto de entidades e interações encontradas em *frameworks* já implementados que demonstram um bom resultado.

2.6 Trabalhos Relacionados

A ideia de criar um *framework* para abstrair o código e permitir a reutilização entre kits diferentes não é algo novo. A comunidade de robótica já sentiu a necessidade de uma forma de adaptar seus programas e criou sistemas semelhantes ao sugerido neste trabalho.

A seguir serão apresentados dois *frameworks* para desenvolvimento de robôs: o Carmen e o ROS, que implementam diversos módulos do funcionamento da robótica móvel, entre eles a navegação e definição de trajetória. É apresentada também uma tese de PhD que apresenta um *framework* para desenvolvimento e avaliação de algoritmos de definição de trajetória.

2.6.1 Carmen

O Carmen (Carnegie Mellon Robot Navigation Toolkit) é um *framework open-source* para controle de robôs móveis. Ele é focado em toda parte de navegação, dando uma coleção volumosa para o desenvolvimento de novos programas. Ele é escrito predominantemente em C e possui uma arquitetura modular que troca informações através de IPC (*inter process communication*). Suas principais funcionalidades incluem: sensoriamento, *logging*, desvio de obstáculos, localização, definição de trajetória e mapeamento ([Site do Carmen, 2014](#)).

O Carmen funciona em uma série de robôs diferentes, permitindo portabilidade entre os tipos de hardware com qual trabalha. Entre eles estão o IRobot, ActivMedia Pioneer, OrcBoard e SegWay.

O Carmen monta o mapa como uma malha de ocupação, formando uma matriz com os espaços ocupados e livres. Para definição de trajetória, o *framework* usa um algoritmo potencial chamado Konoliges Linear Programming Navigation *gradient method* ou LPN ([HOLDGAARD-THOMSEN, 2010](#)).

2.6.2 ROS

O ROS (Robot Operating System) é um *framework open-source* que trabalha como um sistema operacional, abstraindo o hardware para o usuário e fornecendo um controle de dispositivos de baixo nível. O *framework* fornece ainda uma fácil troca de mensagens entre processos, gerenciamento de pacotes e funcionamento em nós, permitindo a execução de código em vários computadores ([Site do ROS, 2014](#)). O *framework* também possui uma gama de ferramentas e bibliotecas para simulação e teste do código. O código está disponível em C++ e Python para desenvolvimento em qualquer uma das duas linguagens. Dentre os tipos de hardwares em que ele funciona estão o Aldebaran Nao, Robotnik Guardian, Neobotix, AscTec Quadrotor, entre outros.

2.6.3 Tese de Morten Strandberg

Morten ([STRANDBERG, 2004](#)) realiza em sua tese de PhD um *framework* que facilita a criação e avaliação de planejadores de caminho. Seu trabalho não se restringe apenas à robótica móvel, mas qualquer aplicação que use de planejadores de caminho, embora seu foco seja na robótica industrial. Ele considera o ambiente dinâmico, ou seja, os obstáculos podem se mover.

([STRANDBERG, 2004](#)) divide os conceitos em quatro grupos: representação geométrica dos obstáculos, a detecção de colisão, o planejamento da trajetória e os modelos de movimento do robô. Com isso é perceptível que o trabalho vai além de puramente definir a trajetória, pois lida com o sensoriamento constante e a atuação dos motores para movimentação do robô.

O *framework* foi construído a partir das técnicas de orientação a objetos e de padrões de projeto, visando a fácil extensão, usabilidade, portabilidade e o baixo acoplamento. Nesse sentido, o autor definiu um conjunto de componentes, cada um com uma responsabilidade específica. Cada componente possui uma hierarquia de classes de baixo acoplamento que realizam as tarefas, baseadas em uma interface comum a todo o componente. Essa interface facilita uma comparação justa entre diferentes implementações de definição de trajetória.

As métricas são feitas a partir de cálculos sobre a posição dos objetos e do ambiente. Quase todas as métricas são feitas a partir de uma equação como a métrica Manhattan ou a de corpo rígido. As métricas e suas variações também formam um módulo do sistema, com sua própria hierarquia de classes e uma classe abstrata mãe que serve de interface para o resto do sistema.

2.7 Resumo do Capítulo

A robótica, em linhas gerais, lida com a construção de máquinas autônomas, capazes de realizar uma tarefa sem intervenção humana. Essa descrição é bastante genérica, mas revela a principal característica dos robôs, autonomia. Os robôs mais comuns são os industriais, presentes nas linhas de montagens da indústria, e os móveis, presentes nos mais diversos nichos de mercado como entretenimento, pesquisa e militar. A principal diferença entre os dois é o robô móvel não ter uma base fixa, podendo se locomover em um ambiente para realizar um trabalho.

Dentre todas as dificuldades da robótica móvel, a navegação é uma das com maior destaque. Englobando todas as dificuldades para a locomoção do robô, essa área pode ser dividida em sensoriamento e mapeamento, cinemática do robô, definição de trajetória, auto-localização, entre outros.

A definição de trajetória pode ser dividida em dois grupos: algoritmos globais e locais. Os globais recebem um mapa de todo o ambiente e, através de algoritmos específicos, definem o melhor caminho para seguirem sem haver colisão com os obstáculos. Os locais trabalham com os sensores, sem conhecer o ambiente além do alcance dos sensores. Ele segue ao objetivo final desviando dos obstáculos conforme os detecta. Ambos os paradigmas podem trabalhar em conjunto, usando algoritmos globais para definir o caminho a longo prazo e os locais para garantir a segurança no trajeto entre cada par de pontos. Algoritmos com Grafo de Visibilidade, Voronoi, Quadtree e Wavefront definem um grafo com os caminhos livres de obstáculos. Com um algoritmo que solucione o melhor caminho do grafo (como Djikstra) é possível definir o percurso a ser seguido como uma lista de pontos até o objetivo final.

A implementação do trabalho inclui a construção de um *framework*, que é um conjunto de classes ou módulos que criam um projeto reutilizável para um específico nicho de software. A construção de um *framework* é fortemente baseada na arquitetura que terá. Desenvolver a arquitetura, assim como o *framework*, é projetar os componentes, dividir as responsabilidades e planejar as interações entre eles a fim de diminuir o acoplamento entre eles e torná-los mais independentes.

O projeto de um *framework* é baseado em padrões de projeto e *hot-spots*. Os padrões de projeto auxiliam na modularidade e reutilização do código, enquanto os *hot-spots* definem as áreas do código abertas para customização pelo usuário. Cada *hot-spot* deve ser bem definido e projetado para que novas funcionalidades possa ser inseridas sem interferir no fluxo de trabalho comum do sistema. O *framework* ainda precisa definir se será caixa-preta ou caixa-branca e sua escolha interfere na visibilidade das classes e como os *hot-spots* serão implementados pelo usuário final.

Foi ainda abordado um processo de desenvolvimento específico para *frameworks*,

orientado a *hot-spots*. Este processo procura desenvolver a partir dos pontos-chave do *framework*, escolhendo um para ser analisado, projetado e implementado a cada iteração. Toda iteração também realiza uma avaliação da arquitetura para garantir que as novas mudanças não estão afetando a arquitetura de forma indesejada.

3 Suporte Tecnológico

Nesse capítulo, são listados todos os programas, tecnologias, ferramentas e aparelhos que foram usados para a produção do Traveller Framework e do robô onde foi testado.

3.1 Linguagem de programação

A linguagem em que o *framework* foi implementado é Java. Isso exige que o robô em que o *framework* será rodado tenha uma máquina virtual Java instalada nele. A escolha da linguagem se deu ao kit disponível para o trabalho rodar esta linguagem.

3.2 Ferramentas e plugins

Como inspiração para o tema e base para o funcionamento dos algoritmos foi analisado o software MRIT (Mobile Robotics Interactive Tool) ([Site do MRIT, 2014](#)) desenvolvido por ([GUZMÁN et al., 2008](#)). Este software permite a simulação de algoritmos locais e globais, definindo a trajetória em meio aos obstáculos e considerando as características físicas do robô. Ele foi usado para estudo do tema e do comportamento dos algoritmos globais e foi aproveitado para comparações com os resultados feitos pelo *framework*, exemplificando visualmente o funcionamento dos algoritmos utilizados.

Para o desenvolvimento do projeto foi utilizada a IDE Eclipse Indigo ([Site do Eclipse, 2014](#)). O código, a compilação e a execução foram todos feitos através desta IDE, que contou com um *plugin* do kit utilizado (LEJOS NXT) para rodar as bibliotecas do kit e permitir a compilação devida e *upload* do código para o robô. O *plugin* pode ser encontrado em ([Site do plugin NXT, 2014](#)).

Para os testes e controle de qualidade do sistema foi usado o *plugin* JUnit ([Site do JUnit, 2014](#)) para testes unitários, que também foi integrado à IDE. Outro *plugin* é o Eclemma para cobertura de código ([Site do Eclemma, 2014](#)), também integrado à IDE. Testes de memória e desempenho foram feitos com a IDE Netbeans 8.0.2 através de sua função *profile*.

Para a comunicação com o robô, foi usado a biblioteca Bluecove, versão 2.1.1 ([Site do Bluecove, 2015](#)). Essa biblioteca externa, em formato .jar, permite que o computador troque informações com o robô via *bluetooth*. A versão do *bluetooth* utilizada é a 2.0.

3.3 Lego NXT

O kit utilizado para testar o *framework* foi o kit educacional da LEGO em sua segunda versão: o LEGO NXT. O robô foi montado com peças de lego e movido com os motores do kit. O LEGO NXT possui um processador Atmel ARM 32 bits com *clock* de 48MHz, HD de 256KB de memória *flash*, memória RAM de 64KB e sistema operacional proprietário.

O kit permite a instalação de um Java adaptado para sua plataforma, o LEJOS NXJ ([Site do Lejos, 2014](#)). O LEJOS não vem instalado por padrão no kit. Para inserir o *firmware* no robô, primeiramente, deve instalá-lo no computador, onde será programado. Posteriormente, deve ser executada a instalação via cabo USB no robô.



Figura 17 – Foto do robô utilizado

3.4 Differential Steering

A estrutura física da plataforma foi um *differential steering* montado com os dois motores e as peças do kit já referido. O modelo tem duas rodas fixas na frente, cada uma ligada a um motor e a iguais distâncias do centro do robô. Atrás, uma roda castor mantém o equilíbrio e permanece livre para girar conforme a estrutura se movimenta.

Por ter motores separados nas rodas, o movimento do robô torna-se mais flexível e variado, podendo girar em torno de cada roda ou do próprio eixo. Cada roda pode girar em uma velocidade diferente ou até para lados diferentes ou pode girar uma roda e manter a outra parada. Por essa liberdade e simplicidade este modelo é muito usado em robótica (([MATARIC, 2007](#))).

Há variações deste estilo, onde todas as rodas de um lado estão ligadas a um mesmo motor, como mostra a figura 18.

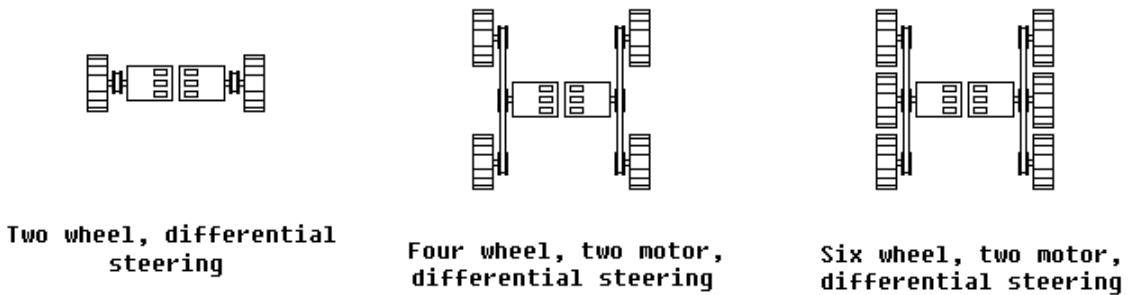


Figura 18 – Exemplos de estruturas *differential steering* (<http://www.enigmaindustries.com/configurations.htm>, 2014)

3.5 Resumo do Capítulo

O robô usado nos testes deste trabalho foi um Lego NXT, rodando o *firmware* para linguagem Java LEJOS. O ambiente de desenvolvimento foi o Eclipse Indigo com *plugin* para desenvolvimento para o kit LEGO. O robô é do modelo *differential steering*, que possui duas rodas independentes e uma roda de apoio atrás.

Os testes foram feitos com apoio das ferramentas JUnit e Eclemma, ambos com *plugins* para o ambiente de desenvolvimento utilizado.

Os resultados finais foram comparados com o simulador MRIT, que permite montar os obstáculos do ambiente e testar os algoritmos, mostrando o percurso gerado.

4 Metodologia

Neste trabalho foi seguido um modelo de pesquisa teórico, aplicado e experimental. Este trabalho foi embasado em pesquisa bibliográfica em cima dos temas definidos, que já foram bem abordados por diversos autores anteriormente. Com isso, já há uma metodologia a ser seguida (processo orientado a *hot-spots*) e pesquisas e/ou simuladores nas áreas abordadas (como o MRIT de (GUZMÁN et al., 2008) ou as teses de (SOUZA, 2008), (HOLDGAARD-THOMSEN, 2010) e (STRANDBERG, 2004)), servindo de guia para implementação e modelo de comparação para os testes realizados.

O tipo de pesquisa (em relação aos objetivos) foi majoritariamente exploratório, buscando conhecer e compreender o tema. Foi dispendido tempo com levantamento bibliográfico, estudando teses e artigos na área e analisando o que se encaixava no tema. Já o tipo de pesquisa para a abordagem (implementação) é híbrida, sendo quantitativa em relação às medidas de desempenho, coesão, acoplamento e qualidade dos resultados apresentados pelos algorítmos (os resultados podem ser comparados com outros estudos e medidos por testes) e qualitativa, quanto às métricas subjetivas como legibilidade do código.

4.1 Processo de produção

Com isso, a metodologia abordada neste trabalho envolve pesquisa bibliográfica sobre o tema, a fim de levantar materiais de estudo ligados ao tema e deles tirar o modo de desenvolver o *framework* e seus algoritmos, além de buscar métricas para comparar o funcionamento do sistema.

Com base nestes estudos, foi feito então uma prova de conceito, desenvolvendo a arquitetura do sistema e implementado os algoritmos Quadtree e Djikstra para validar a proposta levantada. Nesta prova de conceito, foram definidos os módulos do *framework*, as interfaces públicas para comunicação e implementado um algoritmo para testar a arquitetura modelada. Assim, ao fim da prova de conceito já era possível passar por todo o fluxo de atividades do *framework* e chegar na resposta correta para o conjunto de entradas dado. A partir dos resultados obtidos, a proposta foi refinada.

A partir deste refinamento, foi dado início à produção do *framework* por completo, seguindo o processo orientado a *hot-spot* apresentado por (FAYAD; JOHNSON; SHMIDT, 1999) em combinação com os princípios ágeis, como refatoração, testes automatizados, detalhamento das atividades apenas ao implementá-la, dentre outros. A abordagem foi *top-down*, focando cada iteração (ou *sprint*) em uma funcionalidade e no módulo, com as

classes correspondentes.

A cada *sprint* foi concluída uma tarefa que agregasse valor ao produto ou ao trabalho de conclusão de curso, podendo ser essa tarefa um algoritmo, um módulo de estrutura de dados (como o Graph ou Map), uma interface e as interações de um componente completa e testada ou um teste da implementação no robô em ambiente controlado. A cada início de ciclo, era escolhida uma tarefa a ser realizada, revisada e refinada sua análise, definido *hot-spots* e desenvolvida e revista sua comunicação com os demais módulos. Foi dedicado um tempo para a reavaliação dos *hot-spots* já desenvolvidos assim como a validação do produto entregue.

4.2 Atividades e fluxo de atividades

O fluxo de atividades deste trabalho foi baseado no processo mostrado na Figura 15. As inserções foram baseadas no processo do Scrum e nas atividades já realizadas ao longo deste trabalho. O diagrama de atividades completo é apresentado na figura 19.

As atividades iniciais compuseram a definição do tema, estudo do mesmo e execução da prova de conceito. Após seu refinamento foi criado o *backlog* do produto de software, preenchendo-o com todas as funcionalidades, *hot-spots*, pesquisas e tarefas correlatas a serem realizadas. A partir de então iniciou-se as *sprints*.

Cada *sprint* possuiu as seguintes atividades que foram realizadas:

- Selecionar as tarefas a serem feitas nesta iteração;
- Caso a atividade seja apenas pesquisa ou teste no robô, é executada a tarefa e ao seu fim verificado se há mais tarefas no backlog para selecioná-las;
- Caso seja trabalho sobre o *framework*, é analisado se há algum *hot-spot* nas funcionalidades selecionadas;
- Refinar o desenho do componente trabalhado caso necessário para receber a nova funcionalidade e o novo *hot-spot* (se houver);
- Implementar a funcionalidade nova e rodar a suíte de testes;
- Fazer uma revisão de todo o sistema para ver se está evoluindo para o fim esperado e se há alguma falha arquitetural e,
- Verificar se ainda há trabalho a ser feito e, caso haja, levantar mais tarefas, testes e pesquisas para alcançar seu fim.

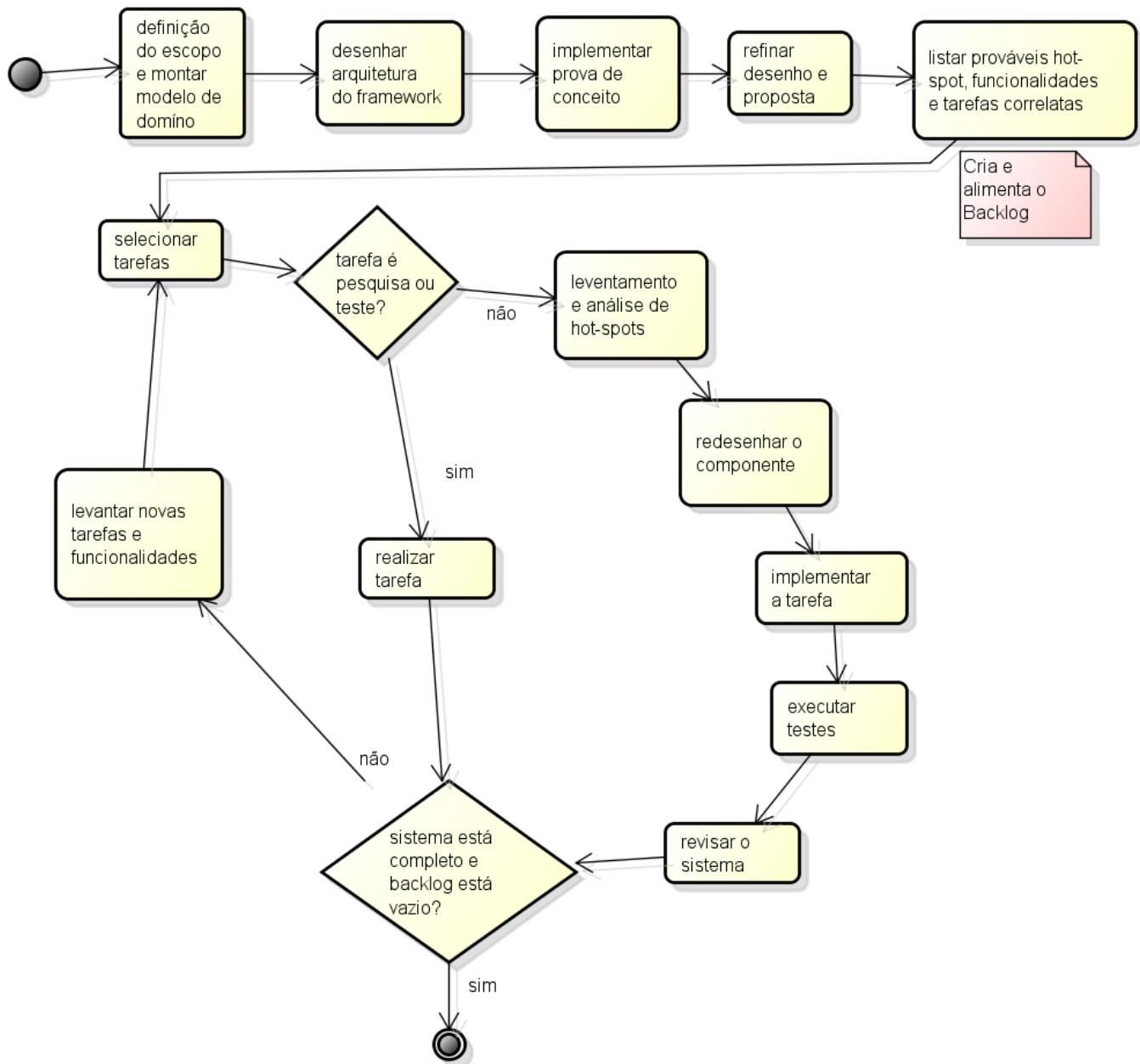


Figura 19 – Fluxo de atividades propostas

As tarefas realizadas nas *sprints* e que compuseram a construção deste trabalho foram as seguintes:

- Revisar a arquitetura e a implementação do algoritmo Quadtree;
- Revisar trabalho escrito;
- Implementar o algoritmo de menor caminho A* (A Estrela);
- Analisar outros algoritmos de menor caminho;

- Realizar teste do *framework* no robô;
- Testar memória e processamento do *framework* no robô;
- Criar ambientes de testes com base da ferramenta de simulação MRIT;
- Implementar algoritmo Wavefront;
- Implementar expansão dos obstáculos;
- Implementar algoritmo Voronoi;
- Implementar algoritmo Grafo de Visibilidade;
- Implementar mudanças a partir do tamanho do robô;
- Desenvolver serviço para comunicar o servidor com o robô;
- Desenvolver módulo embarcado no robô que se comunique com o serviço externo;
- Avaliar possibilidade do mapa ser um novo *hot-spot*;
- Otimizar implementação do algoritmo;
- Revisar código;
- Revisar testes e,
- Escrever trabalho escrito.

Vale ressaltar que algumas destas tarefas foram executadas mais de uma vez, sendo realizada de forma iterativa e incremental (como, por exemplo, as tarefas de revisão). Em geral, cada iteração levou pouco menos de uma semana, embora não tenham sido constantes ao longo do semestre.

4.3 Resumo do Capítulo

Para desenvolver o *framework* foi usado o processo orientado a *hot-spot* junto com uma adaptação de metodologia ágil. A implementação foi dividida em iterações (*sprints*) onde um *hot-spot* e/ou funcionalidade era selecionado para ser realizado. O *hot-spot* era então analisado, estudado e projetado mais adequadamente. Buscou-se que ao fim de cada *sprint* uma nova funcionalidade ou *hot-spot* tivesse sido desenvolvida e testada, além da arquitetura ser reavaliada para garantir que não houve depreciação da mesma com a implementação realizada.

A cada *sprint*, os *hot-spots* foram avaliados para assegurar sua boa implementação e projeto. Adicionalmente, novas funcionalidades e *hot-spots* foram sendo levantados para alimentar o *backlog*.

5 O Traveller Framework

A proposta deste trabalho foi a construção de um *framework* de definição de trajetórias para robôs móveis, mais especificamente, algoritmos globais de definição de trajetória. Foram implementados os quatro algoritmos descritos no capítulo 2 (Grafo de Visibilidade, Voronoi, Quadtree e Wavefront). Um mapa do ambiente é recebido das camadas acima da camada do *framework* (como mostrado na Figura 4) e o algoritmo cria um grafo a partir dele, definindo os caminhos livres que podem ser percorridos. Após isso, é executado um algoritmo de melhor caminho (como Djikstra ou A*, por exemplo) para definir o caminho a ser percorrido pelo robô.

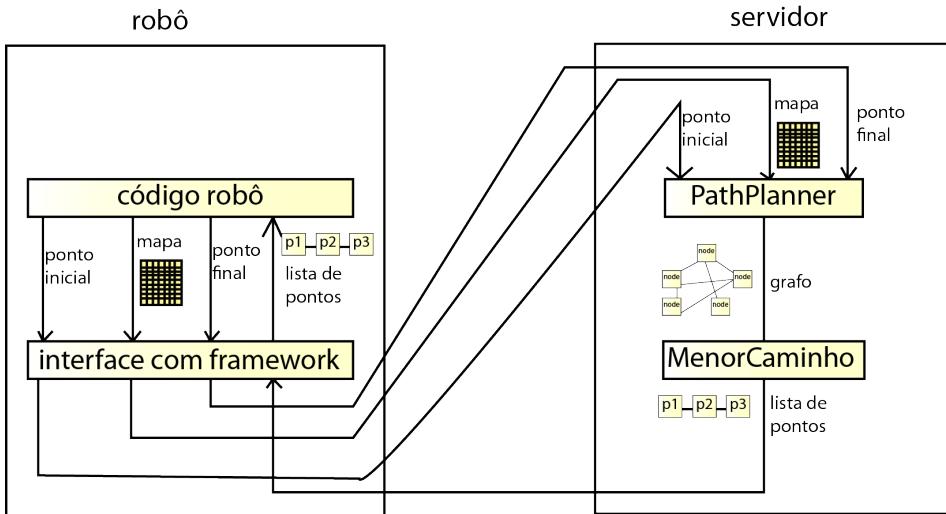
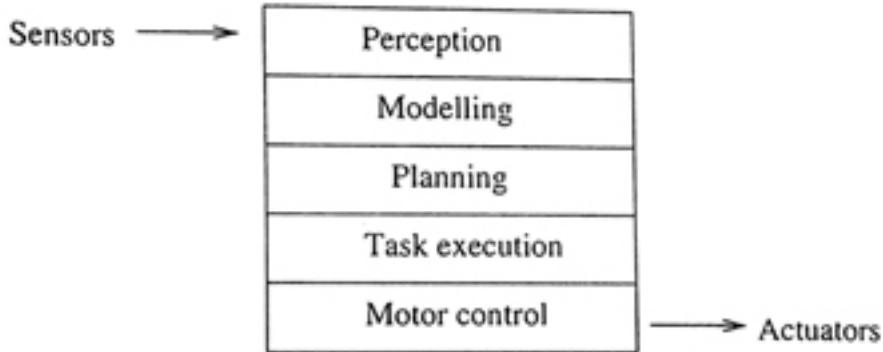
É importante ressaltar que o Traveller não realiza o mapeamento, apenas o recebe pronto de outro módulo do robô e o utiliza.

O núcleo do *framework* roda em um computador fora do robô, como em um servidor local, e se comunica com o robô via comunicação *bluetooth*. A escolha de desenvolver a solução fora da máquina será explicada no capítulo 6. No robô, uma classe encapsula a comunicação com o servidor e realiza a troca de mensagens. O servidor recebe os dados de entrada e opera os algoritmos para devolver ao robô a lista de pontos a serem percorridos.

A Figura 20 demonstra as tarefas principais do *framework*, bem como suas entradas e saídas. Essas entradas e saídas definem como o *framework* se comunica com as demais camadas do robô bem como as dependências desse framework para cumprir seu funcionamento. Cada tarefa apresentada representa um dos módulos principais do *framework*, e as interações entre eles representam as estruturas de dados utilizadas para se comunicarem. Estes dados são recebidos pela classe embarcada no robô. Essa classe se comunica com o servidor. Os dados ainda são repassados para esse servidor sem alterações. Todo o processamento é então feito fora do robô, retornando uma resposta à classe embarcada. Por fim, essa classe responde ao sistema do robô.

A arquitetura geral do robô foi considerada como na Figura 21, orientada de acordo com a visão apresentada em (NEHMZOW, 2003). Todas as abordagens da arquitetura de navegação seguiam uma lógica semelhante e apresentavam pequenas diferenças entre si. Assim sendo, esta estrutura mantém as características apresentadas no capítulo 2.

A definição de trajetória e o *framework* participam da terceira camada. Como as camadas acima (*Perception* e *Modelling*) são responsáveis pelo mapeamento e entregam o mapa completo, os algoritmos globais e o Traveller Framework não precisam se preocupar com a comunicação com os sensores. As camadas abaixo (*Task execution* e *Motor control*) recebem a lista de coordenadas para onde o robô deve ir e cuidam do controle dos motores para chegar nesses pontos, considerando os graus de liberdade do robô (que varia de

Figura 20 – Estrutura do funcionamento do *framework*Figura 21 – Arquitetura de navegação considerada, ([NEHMZOW, 2003](#))

acordo com sua estrutura física). Assim, a camada a ser criada não se comunica nem com os sensores nem com os motores, o que confere a essa camada maior independência das configurações de robô. Baseado nessa abordagem, é esperado que o *framework* funcione em variados tipos de robôs, não apenas no modelo *differential steering* testado.

A seguir é definido como funciona em detalhes o Traveller e sua arquitetura.

5.1 A Arquitetura

A arquitetura de navegação costuma ser em camadas, como mostrado no capítulo 2.2.5. O Traveller trabalha dentro da camada de planejamento de trajetória e é uma estrutura baseada em componentes. Uma arquitetura em camadas dentro de outra poderia gerar indireções desnecessárias, além de uma abordagem por componentes suprir de forma mais adequada as necessidades do *framework*.

Cada funcionalidade é efetuada por um componente diferente, projetado para con-

sumir o mínimo de memória e depender o mínimo de outros componentes e de bibliotecas externas (incluindo as próprias do Java). Cada módulo tem a preocupação de ser o mais auto-suficiente possível, tendo apenas as interações necessárias para o seu funcionamento. Cada um dos módulos foi constantemente avaliado quanto a seu acoplamento, coesão, simplicidade e facilidade de compreensão. Estas características serão mais bem definidas na seção 5.6.

O *framework* segue o padrão caixa-cinza, pois é desejável que o mesmo seja transparente e configurável pelo usuário. O usuário do *framework* é livre para usar as implementações já prontas dos algoritmos de definição de trajetória e melhor caminho ou criar as suas próprias a partir das classes abstratas fornecidas. Além disso, o fluxo de execução e alguns comportamentos são mantidos estáticos.

O diagrama de componentes da Figura 22 mostra os módulos do *framework* e suas interações. São apresentadas as interfaces e as classes conhecidas pelos outros componentes. Levando em conta o princípio de programar para interfaces, as classes conhecidas serão sempre as interfaces e classes abstratas (exceto nos componentes de estrutura de dados e que determinam o fluxo de dados). Para simplificar o diagrama, as classes concretas foram retiradas, bem como foram criados diagramas específicos para detalhar cada módulo.

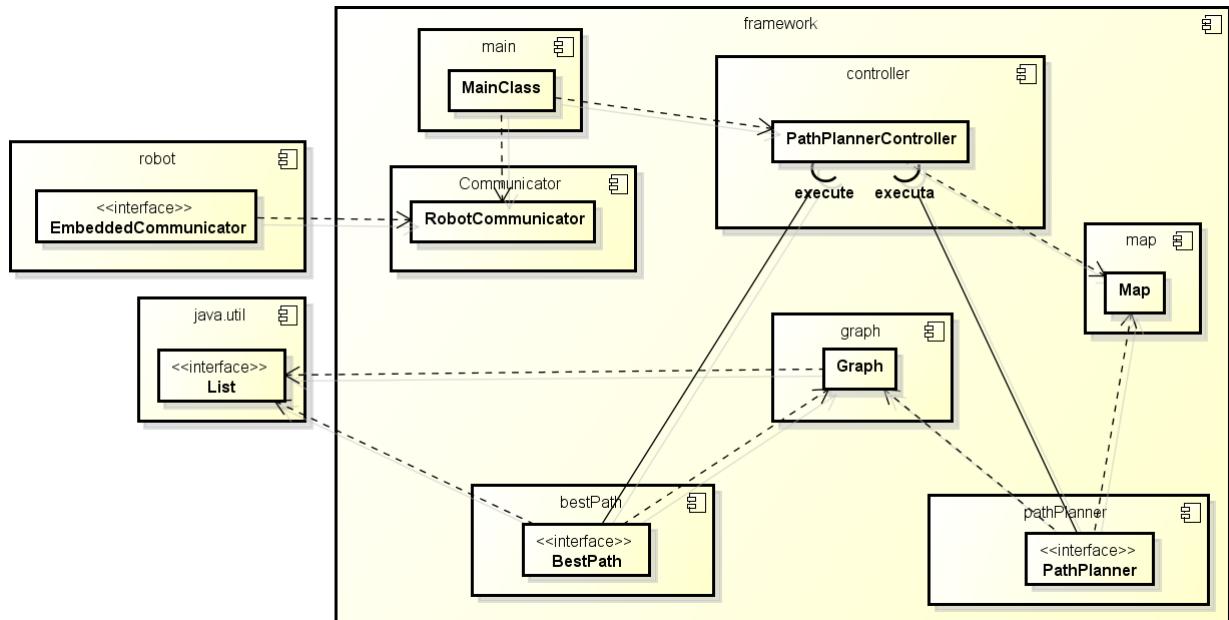


Figura 22 – Diagrama de pacotes do sistema

Nas seções subsequentes, cada componente é descrito detalhadamente.

5.1.1 EmbeddedCommunicator

Este componente é o único que será executado dentro do robô e sua função é se comunicar com o servidor que esteja rodando o *framework*, realizando as trocas de mensagens. Como cada kit possui uma programação, cada um pode realizar a comunicação externa de forma diferente. As classes e argumentos entre cada robô podem mudar, por isso é definida uma interface genérica de comunicação e uma classe concreta para o Lego NXT. A Figura 23 mostra a estrutura do pacote.

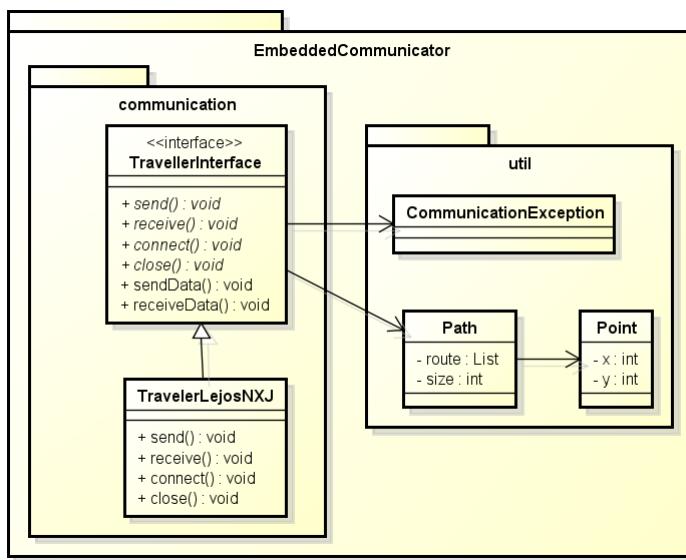


Figura 23 – Componente embarcado no robô

O módulo possui dois pacotes: *communication*, que realiza a comunicação e possui a classe abstrata e suas implementações; e *util*, que possui classes utilitárias para a mesma. Na classe abstrata são implementados os métodos que recebem os dados do robô e que devolvem a lista para o mesmo, além de organizar as trocas de mensagens para o correto envio de informações. Essa classe abstrata define o padrão de troca de informações através do padrão de projeto Template, estabelecendo a ordem em que as mensagens devem ser enviadas, porém não define como os métodos *send* e *receive* funcionam. Esses métodos são definidos pela classe filha, que deve ser construída pelo usuário de acordo com as bibliotecas do kit que utiliza.

No pacote *util* é definido uma classe de exceção que é a única retornada pela classe abstrata e suas implementações, abstraindo os erros possíveis para uma única classe. O pacote possui também a classe retornada pela comunicação, a classe *Path*. A classe *Path* possui o tamanho total do percurso entre o ponto inicial e final pelo percurso retornado e a lista de pontos (coordenadas em *x* e *y*) que o robô deverá seguir.

As únicas restrições feitas pela interface são: (i) que a comunicação seja *bluetooth* para que o servidor consiga receber os dados, e (ii) que possam ser enviados *strings* e

arrays de inteiros e booleanos pelo kit.

5.1.2 RobotCommunicator

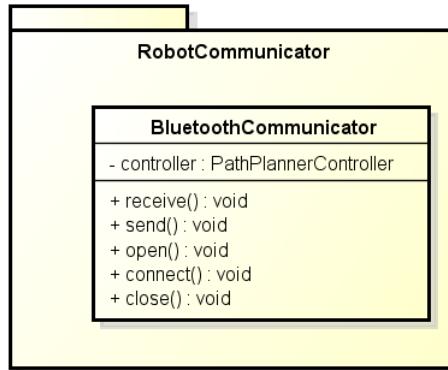


Figura 24 – Componente que se comunica com o robô

Este módulo apenas recebe os dados do robô e os repassa adiante. Esse módulo possui apenas funções de iniciar e concluir a comunicação e de leitura e escrita pelo *bluetooth*. Assim, toda comunicação e tratamento do *bluetooth* fica encapsulado dentro dele e qualquer alteração na forma de comunicação não impactará no *framework*. Este módulo também reorganiza os dados do mapa em uma nova matriz de booleanos para então repassá-los ao controlador. A Figura 24 mostra as principais funções do módulo.

Para a comunicação *bluetooth*, é utilizado o Bluecove versão 2.1.1 como citado no capítulo 3. A versão do *bluetooth* é a versão 2.0. Caso precise trabalhar com outra versão do *bluetooth*, basta trocar a biblioteca do Bluecove por uma compatível com a versão desejada.

5.1.3 Main

Este módulo tem como finalidade iniciar o sistema e fazer o intermédio entre o controlador e servidor. O módulo possui a função *main* que inicia o sistema e permanece em *loop* infinito, sempre a espera de uma nova conexão. Ele instancia a classe RobotCommunication e, ao iniciar uma conexão, chama os métodos de leitura na ordem para guardar as informações recebidas. Desse modo, é nesta classe que fica definido a ordem de leitura dos dados. Ao término, ele também repassa ao RobotCommunication o objeto *Path* que será enviado ao robô.

Com estes deveres, é perceptível que este módulo segue o princípio GRASP de indireção, servindo de intermediário entre o leitor do *bluetooth* e o *framework* em si. Com isso, o RobotCommunication apenas se preocupa em fazer a comunicação com o robô, o *Main* fica responsável por instanciá-lo e definir a ordem de leitura dos dados e repassá-los ao controlador, que executará a lógica do sistema.

5.1.4 Controller

Uma vez recebidos os dados do robô, a classe PathPlannerController tem como propósito executar o *framework* em si. Essa classe controla o fluxo de trabalho da aplicação, chamando os métodos das demais classes na ordem, e verificando seu correto funcionamento.

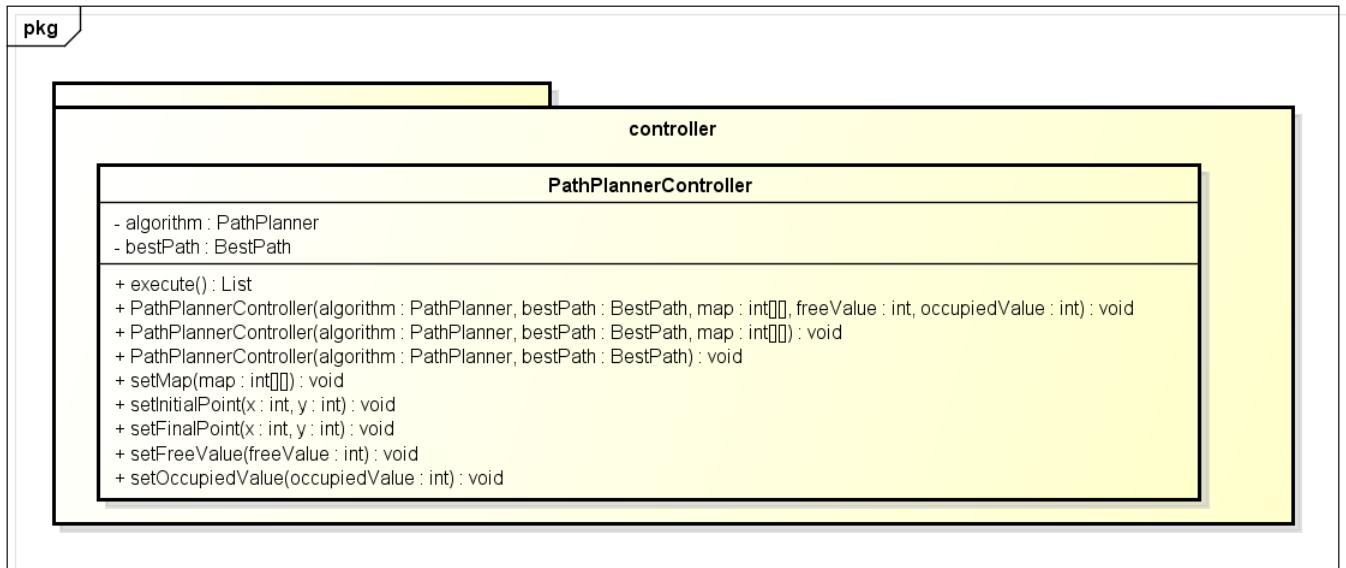


Figura 25 – Componente Controller

O componente segue os princípios Controlador e Inversão de Controle. Em vez do usuário chamar todas as funções do componente e fazer seu controle diretamente toda vez que quiser definir uma trajetória, este componente realiza esta tarefa pelo usuário, diminuindo acoplamento entre o usuário e o subsistema. A Inversão de Controle está presente pelo usuário ser o responsável por definir quais as classes dos outros componentes. Embora o *framework* as instancie, cabe ao usuário escolher quais de suas variações serão implementadas. No construtor da classe, são passados os objetos `PathPlanner` e `BestPath` a serem usados. Esses objetos são armazenados e utilizados no método `execute` para calcular o percurso. Isso permite ao módulo não conhecer as variações de cada interface, desacoplando-o dos demais módulos.

Os padrões utilizados são o Fachada e a Injeção de Dependência.

É também atribuído a esse módulo a responsabilidade por criar a estrutura local do mapa. O controlador inicializa o mapa, definindo os pontos iniciais e finais. Adicionalmente, limpa a memória ao final do procedimento, delegando ao componente `PathPlanner` apenas a tarefa de utilizar o mapa.

O diagrama de sequência apresentado na seção 2.4 mostrou de forma bastante superficial o funcionamento do sistema para dar uma visão geral das funcionalidades e

sequência de ações. O diagrama na Figura 26 demonstra de forma mais aprofundada como realmente é feito o trabalho do framework, com foco no componente controller. Nele é visto a sequência de atividades que o componente Controller executa, começando no topo e conforme vai terminando vai executando as tarefas abaixo. O diagrama também evidencia as classes envolvidas, listadas lado a lado e mostrando a sequência de ações ao longo do tempo de cada uma, também de cima para baixo.

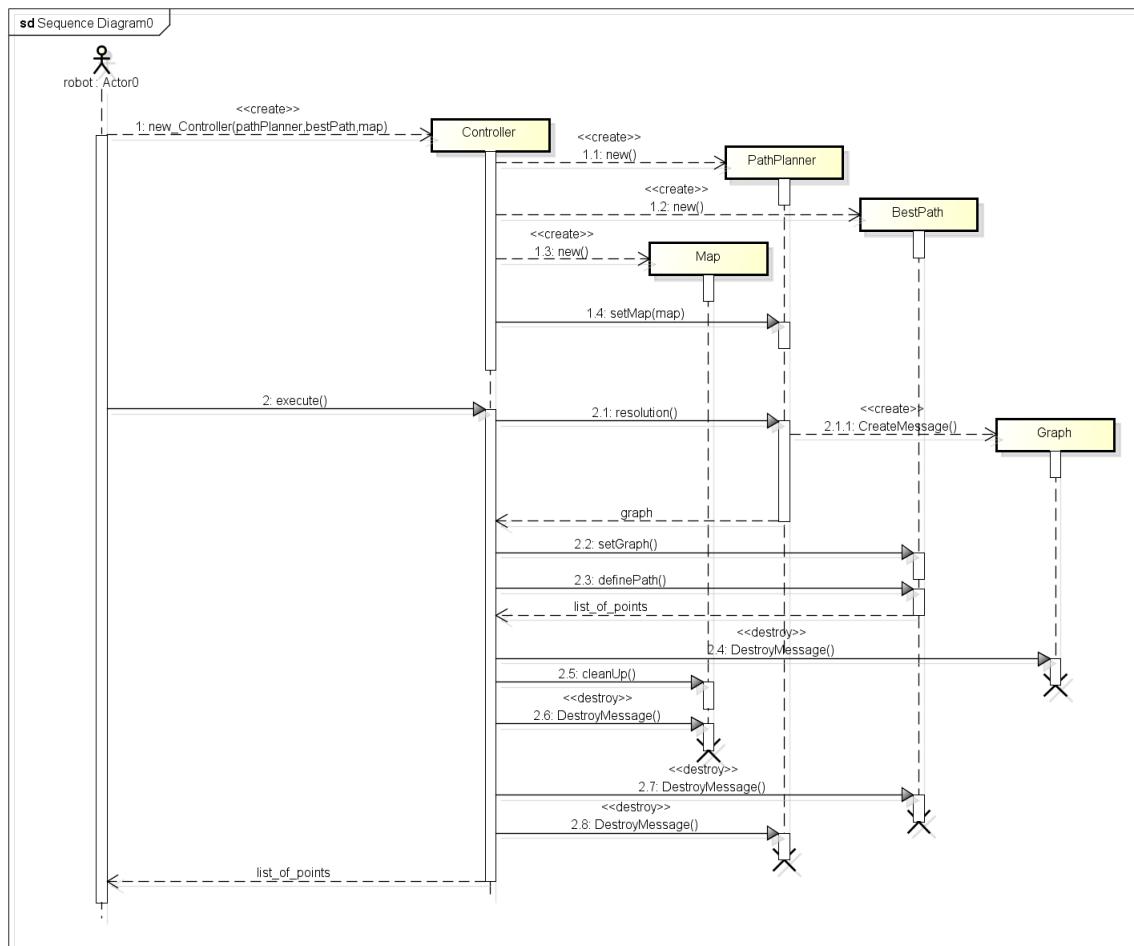


Figura 26 – Diagrama de sequência do componente controller

5.1.5 Map

O componente Map armazena a estrutura do mapa em seu interior. O componente recebe do controlador os dados das posições e guarda essa referência dentro de si para fazer as alterações necessárias. O objeto Map centraliza os dados que a camada de sensoriamento fornece e funções específicas da definição de trajetória que agem sobre eles, como: definir pontos inicial e final; conferir pesos às posições (células); expandir obstáculos, e diferenciar espaços livres de obstáculos.

Internamente, o mapa trabalhado é considerado uma matriz de inteiros, sendo convertido para esse formato logo no construtor da classe. Essa alteração de booleano

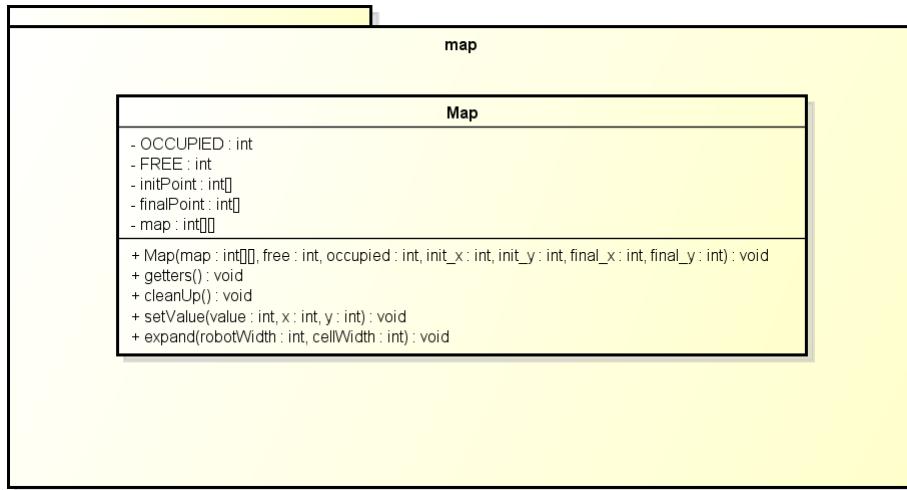


Figura 27 – Componente Map

no mapa interno do robô para inteiro no servidor se deve a necessidade do algoritmo Wavefront de trabalhar com valores inteiros em cada célula. Como esse módulo fica fora do robô, nenhuma alteração feita nos dados recebidos comprometerá os dados reais no robô. A Figura 27 mostra os principais atributos e métodos da classe.

5.1.6 Path Planner

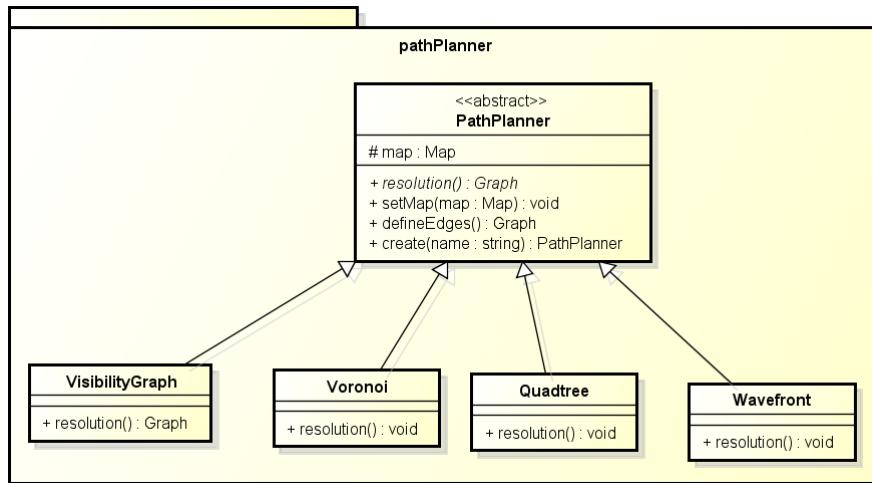


Figura 28 – Componente PathPlanner

O PathPlanner é o componente responsável por implementar os algoritmos de definição de trajetória. A classe abstrata PathPlanner carrega funções úteis às classes concretas e fornece um método abstrato que deve realizar o algoritmo. Cada subclasse sobrescreve *resolution* (responsável pela execução do algoritmo) e métodos específicos e cria um grafo diferente. A classe abstrata também possui um método Fábrica para definir qual subclasse dele será utilizada. A Figura 28 mostra a classe abstrata e todas as classes concretas implementadas.

Este componente é um dos *hot-spots* do Traveller, graças à interface comum aos algoritmos. Esta interface possibilita tratá-los de forma igual em todo o resto do sistema. O comportamento diferente entre eles (a criação do grafo) é abstraído para as subclasses, enquanto a superclasse se preocupa com o que for igual, definindo as entradas e saídas comuns. Isso segue o princípio de Variações Protegidas apresentado por ([LARMAN, 2005](#)), implementado através do padrão Strategy. O padrão Injeção de Dependência diminui seu acoplamento com o módulo Map, recebendo o mapa já pronto para trabalhar do controlador.

Para inserir novos algoritmos ao *framework* ou versões otimizadas do mesmo, basta herdar da classe abstrata e implementar a lógica no método *resolution*.

5.1.7 Graph

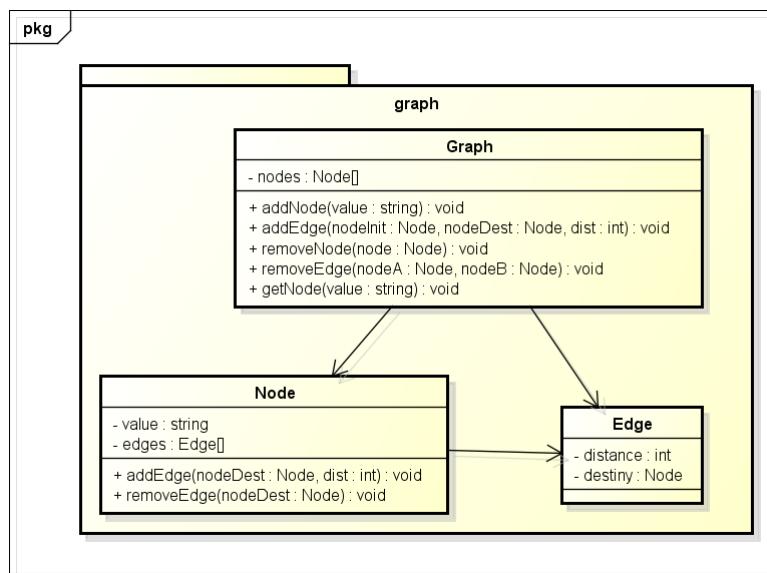


Figura 29 – Componente Graph

O módulo Graph é responsável por manter a estrutura do grafo. O grafo mantém uma lista de nós que representam pontos navegáveis no espaço. Cada nó possui uma lista de arestas (*edges*) que o liga a outros nós.

A classe grafo possui métodos para a construção, busca e eliminação de nós e arestas, podendo realizar qualquer operação CRUD através do próprio objeto grafo ao invés de trabalhar diretamente em cada nó. A Figura 29 demonstra os métodos principais e a relação entre os mesmos.

5.1.8 Best Path

O Best Path é o componente que define o melhor caminho a ser executado pelo robô. Recebendo o grafo com os pontos navegáveis, este módulo define um trajeto a

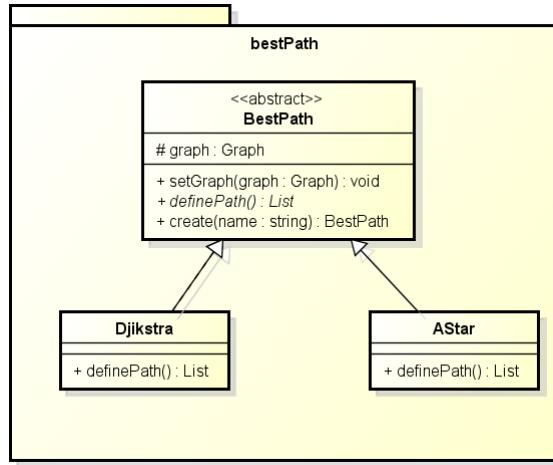


Figura 30 – Componente BestPath

partir de alguma característica desejada (ex. menor caminho, menor curvatura, menor gasto de energia e outras). Esta característica é definida pela subclasse escolhida pelo usuário. No caso, as duas subclasses mostradas no diagrama definem o percurso por menor caminho, porém novas extensões podem ser criadas para formar o trajeto a partir de outra característica. A Figura 30 mostra as três classes desenvolvidas.

Este é outro *hot-spot* do *framework*, permitindo a fácil alteração pelo usuário e a troca dos algoritmos em tempo de execução. Assim como no pathPlanner, isto se deve à classe abstrata BestPath, que define as características comuns a todos os algoritmos, determinando as entradas e saídas que receberão. A implementação de como trabalhar com o grafo e criar a lista de pontos pelos quais o robô passará dependerá da implementação escolhida, que poderá seguir qualquer objetivo que o desenvolvedor da subclasse desejar.

Assim como no componente pathPlanner, o princípio de Variações Protegidas é realizado através do padrão Strategy e a criação da classe é generalizada pelo padrão Fábrica. Como também acontece no pathPlanner, para inserir novos algoritmos ao *framework* ou versões otimizadas do mesmo, basta herdar da classe abstrata e implementar a lógica no método *definePath*.

5.2 O Protótipo

Para validação da ideia, foi desenvolvido um protótipo funcional que provasse a viabilidade do projeto. Nos primeiros meses de trabalho, foi modelada e implementada a arquitetura do *framework* e desenvolvido o algoritmo Quadtree e Djikstra para testar todo o fluxo de execução da ferramenta. Além disto, foram também realizados os testes para as classes produzidas.

No protótipo, o *framework* todo seria executado dentro do robô, assim não existia

os módulos de comunicação *bluetooth* e o código do robô instanciava diretamente as classes concretas dos *hot-spots* desejados, sem necessidade de um método Fábrica. A figura 31 mostra um exemplo de como o Lejos chamava o Traveller.

```

public static LinkedList<Point> executeFramework(float largura_robo) {
    boolean[][] mapa = {{true, true, false, false, false, false, false, false},
                        {false, false, false, false, false, false, false, false},
                        {false, false, false, false, false, true, true, false},
                        {false, false, false, false, false, false, false, false},
                        {false, false, true, true, true, true, false, false},
                        {false, false, true, true, true, false, false, false},
                        {false, false, true, true, false, false, false, false},
                        {false, false, false, false, false, false, false, false}   };
    PathPlanner path_planner = new Quadtree(largura_robo, largura_robo);
    BestPath best_path = new Djikstra();

    PathPlannerController framework = new PathPlannerController(path_planner, best_path);
    framework.defineMap(mapa, false, 1, 6, 5, 6);
    Path path = framework.execute(largura_robo, largura_robo);
    return path.getRoute();
}

```

Figura 31 – Código do *framework* sendo executado na versão protótipo

A arquitetura do protótipo era próxima da atual e já implementava a maioria das interfaces e padrões presentes na versão final. As maiores diferenças estavam no número de implementações de cada *hot-spot* e na inexistência dos módulos web. De início, foi realizada uma análise sobre todo o projeto e foram levantados os *hot-spots* *PathPlanner* e *BestPath*.

A classe *PathPlannerController* tinha inicialmente como propósito servir de intermediário entre o usuário do *framework* e as funções implementadas por ele. Essa era a classe principal do sistema em sua primeira versão, encapsulando todo o funcionamento e sendo chamada pelo usuário para realizar tudo. Isso mudou com o repasse ao servidor, mas a classe manteve sua função de gerenciar o fluxo de atividades.

A suíte de testes realizadas nessa primeira fase realizava testes unitários simples sobre o código, verificando se o grafo montado pelo *Quadtree* estava como esperado, verificando se todos os nós que deveriam existir foram criados. Os testes também verificaram o caminho gerado pelo *Djikstra*, ou seja, se era o mesmo que o esperado para aquele mapa. Sete mapas pequenos foram desenhados e, seus grafos e caminho a ser percorrido, calculados à mão para validar os algoritmos. Os mapas possuíam entre 4 e 9 casas de largura e comprimento, podendo ser perfeitamente quadrados ou retangulares. Ao final todos os testes foram bem sucedidos e as falhas encontradas pelos mesmos foram corrigidas.

5.3 A Versão Final

O código final trabalha como descrito na seção 5.1, permitindo a livre escolha entre algoritmo de melhor caminho e definição da trajetória. A interface no robô é a única

camada que o usuário precisa conhecer e, caso ainda não haja uma implementação para sua plataforma, desenvolvê-la faz-se necessário.

O usuário precisará ainda baixar o código do lado servidor e rodá-lo em uma máquina próxima ao ambiente em que o robô trabalhará. Considerando que o Traveller trabalha com algoritmos globais e ambientes estáticos, o mesmo é recomendado apenas para ambientes fechados e bem conhecidos, como residências, escritórios ou fábricas. Nestes ambientes, um computador precisará ser instalado e com o Java e o *framework* configurados. Será ainda preciso possuir comunicação *bluetooth* e que a mesma esteja habilitada.

A figura 32 mostra um exemplo de código embarcado no robô Lego NXT usando o Traveller.

```
private static Path executeFramework(float robot_width, boolean[][] map) {
    String path_planner = "Quadtree";
    String best_path = "Djikstra";
    float cell_width = 2;
    int init_pos_x = 2, init_pos_y = 2;
    int goal_pos_x = 40, goal_pos_y = 50;
    boolean free_value_map = false;
    boolean expad_obstacles = true;

    Path path = null;
    EmbeddedCommunication framework = new NXTCommunication("LENOVO-PC");
    try{
        framework.connect();
        framework.sendData(map,path_planner,best_path,robot_width,cell_width,init_pos_x,
                           init_pos_y,goal_pos_x,goal_pos_y,free_value_map,expad_obstacles);
        path = framework.receivePath();
        framework.close();
    }catch(CommunicationException e){
        LCD.drawString(e.getMessage(), 0, 0);
    }
    return path;
}
```

Figura 32 – Código do *framework* sendo executado na versão final

A decisão de portá-lo para um servidor externo foi devido aos testes de memória e processamento realizados no mesmo, os quais serão apresentados em detalhes na seção 6.2. Outro resultado mostrado pelos testes de desempenho foi alto gasto de tempo com funções de acesso às listas presentes nas classes (*get*). Em diversas partes do código, foram utilizadas listas encadeadas para armazenar dados. Uma das funções que mais consumia tempo no processador era a de recuperar um dado destas listas. Assim, outra alteração foi o tipo de estrutura utilizada para armazenar estes dados, trocando do *LinkedList* para o *ArrayList* (ambos fornecidos pelo Java).

Caso queira, o usuário pode alterar o comportamento dos algoritmos ou implementar novos que sejam compatíveis com o *framework*, uma vez que possui o código fonte instalado em seu computador e o mesmo foi estruturado para isso. Foi escolhido realizar a implementação básica de cada algoritmo, por serem soluções clássicas em seus meios.

Assim, versões otimizadas dos algoritmos como os propostos por (SOUZA, 2008) e (MEDEIROS; SILVA; YANASSE, 2011) não foram implementadas. Entretanto, podem ser incluídas como outras classes filhas da classe abstrata.

Dos quatro algoritmos desenvolvidos, o algoritmo de Voronoi foi o único a apresentar falhas em seu resultado, realizando um percurso fora do esperado para o comportamento descrito no capítulo 2.4.2 e, na maioria das vezes, não conseguindo encontrar um percurso entre os pontos inicial e final. Todos os demais geram um grafo e devolvem uma lista de pontos compatível com o comportamento descrito no capítulo 2.4.

Algumas observações referentes à relação dos algoritmos com a expansão dos obstáculos precisam ser feitas. O Grafo de Visibilidade precisa ter os obstáculos expandidos, obrigatoriamente, ou haverá a colisão. Portanto, o *framework* força a expansão, mesmo que o usuário não a especifique. Para o algoritmo Wavefront, ele monta o grafo de forma diferente se os obstáculos forem ou não expandidos. Caso haja expansão, o Wavefront considera todos os oito vizinhos para montar o grafo. Caso não haja, o algoritmo considera apenas quatro vizinhos (excluindo as células na diagonal).

A execução da controladora PathPlannerController, antes chamada diretamente pelo usuário, é agora feita pelo módulo *Main*. A ordem de execução não foi alterado, porém algumas variáveis novas foram inclusas para tornar o sistema mais flexível. Hoje, as entradas recebidas pelo Traveller são:

- O algoritmo de definição de trajetória (em forma de String);
- O algoritmo de melhor caminho (em forma de String);
- O mapa (em forma de matriz de booleanos);
- O ponto inicial;
- O ponto final;
- A largura do robô;
- A largura que cada célula da matriz representa no ambiente (na mesma medida que a largura do robô);
- O valor a ser considerado como livre no mapa; e,
- Se os obstáculos devem ser expandidos ou não.

A Figura 33 mostra a ordem em que os mesmos são lidos no servidor (e, consequentemente, a ordem em que são enviados pelo robô) e seu recebimento pela controladora.

```

try {
    init_point = communicator.getPoint();
    goal_point = communicator.getPoint();
    path_planner = communicator.getString();
    best_path = communicator.getString();
    map = communicator.getMap();
    free_value = communicator.getBoolean();
    carWidth = communicator.getFloat();
    cellWidth = communicator.getFloat();
    expand_obstacles = communicator.getBoolean();
} catch (IOException e) {
    System.out.println("data receivement failed: "+e.getMessage());
    return;
}
PathPlannerController controller = new PathPlannerController(path_planner, best_path, carWidth, cellWidth);
controller.defineMap(map, free_value, init_point[0], init_point[1], goal_point[0], goal_point[1]);
controller.expandObstacles(expand_obstacles);
Path path = controller.execute(carWidth, cellWidth);
try {
    communicator.sendPath(path);
    communicator.sendPathSize(path.getSize());
} catch (IOException e) {
    System.out.println(e.getMessage());
}

```

Figura 33 – Ordem em que os dados devem ser enviados

Além dos algoritmos implementados, outros foram estudados, bem como analisadas suas visibilidades. Um deles foi o algoritmo D* (Ferguson ; STENTZ, 2005) para definição de trajetória, porém não incluso por seu uso não se encaixar no cenário de algoritmos globais e do Traveller.

5.4 *Hot-spots* do Framework

Como definido pelo processo de produção de *framework* de (FAYAD; JOHNSON; SHMIDT, 1999), a análise de *hot-spots* é algo iterativo, levantado ao longo da produção com apoio de um especialista. O Traveller possui ao todo três *hot-spots*: *PathPlanner*, *BestPath* e *EmbeddedCommunicator*. O usuário pode incrementar e configurar estes componentes, escolhendo entre as soluções já implementadas e inserindo novas. Seus *hot-spots* servem de gancho para mudanças.

Para dar a variabilidade necessária nesses pontos, para a variância destes pontos foram utilizados os padrões de projeto descritos no tópico 5.1. Eles oferecem a flexibilidade para mudanças sem gerar impactos nos demais módulos, mantendo a coesão alta e o acoplamento baixo entre os módulos.

5.4.1 PathPlanner

Incluso desde a primeira versão do código, este é o principal *hot-spot* do *framework*, dando a liberdade de trocar entre os algoritmos de definição de trajetória. O padrão *Strategy* fornece a variação sem impactos de manutenção no código, e o padrão Fábrica

permite a instanciação da classe especificada pelo usuário. Como o núcleo do Traveller foi retirado de dentro do robô, o usuário não pode instanciar diretamente mais a versão desejada. Assim sendo, o padrão Fábrica permite a verificação do texto recebido pela rede, iniciando o objeto correto.

O *hot-spot card* do *PathPlanner* encontra-se no apêndice A.

5.4.2 BestPath

Também incluso desde a primeira versão, este módulo foi projetado de forma igual ao *PathPlanner*. Ele recebe a saída do módulo *PathPlanner* para gerar o resultado final do *framework*, sendo assim parte do fluxo principal do mesmo. Suas classes concretas são escolhidas via Método Fábrica também e isolam a implementação na classe filha, permitindo que cada classe tenha seu próprio algoritmo. Isso permite que soluções bastante diferentes sejam chamadas com a mesma interface, como um algoritmo que retorne o menor caminho e outro que retorne o caminho mais suave.

O *hot-spot card* do *BestPath* encontra-se no apêndice A.

5.4.3 EmbeddedCommunicator

Este *hot-spot* permite que robôs diferentes consigam conversar da mesma forma com o *framework* no servidor. Isso é feito implementando a sequência de passos para a troca de mensagens com o servidor, passando os dados e as classes a serem instanciadas, e abstraindo a função de comunicação para as classes concretas. Cada classe filha fornece os métodos para enviar e receber dados para um *hardware* diferente.

O *hot-spot card* do *EmbeddedCommunicator* encontra-se no apêndice A.

5.4.4 Hot-spot Descartados: O Mapa

Durante boa parte do desenvolvimento do *framework* imaginou-se que o mapa poderia vir a se tornar um *hot-spot*. Permitir que o usuário use qualquer tipo de representação de mapa mostrada no capítulo 2.3 aumentaria o número de interessados no uso da ferramenta e a tornaria mais flexível. Essa questão foi profundamente analisada e chegou-se a construir seu *hot-spot card*. Entretanto, a inclusão do *hot-spot* *EmbeddedCommunicator* e a transferência do módulo *Map* para o servidor tornou o desenvolvimento desse *hot-spot* mais complexo.

Antes, uma solução semelhante a do *BestPath* e *PathPlanner* poderia ser utilizada, porém ao ter que passar os dados para o servidor se perderia sua estrutura atual, desconhecida até então pelo *EmbeddedCommunicator*. Seria preciso uma estrutura genérica capaz de armazenar as informações de qualquer tipo de mapa sem perder seu sentido

para então convertê-lo na matriz de inteiros. Outra solução seria converter o mapa para uma matriz de inteiros ainda no robô e já enviar ao servidor da forma desejada, porém isto aumentaria o acoplamento entre as partes e consumiria muita memória do robô, que precisaria por alguns instantes guardar dois mapas na memória. Sendo assim, o *hot-spot Map* foi descartado desta versão final.

5.5 Os Obstáculos

Os obstáculos são representados por células de valores diferentes no mapa, marcando-as como ocupadas. Contudo, algumas considerações sobre os obstáculos precisam ser feitas independente da forma que são implementados esses obstáculos.

O robô é considerado como um ponto (seu centro), o que pode gerar problemas se ele se aproximar demais dos obstáculos. Um algoritmo considerando o robô um ponto pode fazê-lo passar muito perto do obstáculo achando que nada acontecerá. Entretanto, quando executado o algoritmo, a colisão ocorre. (GUZMÁN et al., 2008) diz que os algoritmos Grafo de Visibilidade e Quadtree passam o mais perto possível dos obstáculos, gerando um menor percurso, porém mais arriscado. Para evitar que esses algoritmos causem a colisão do robô com o obstáculo uma solução dada por (SOUZA, 2008), (GUZMÁN et al., 2008), (SIEGWART; NOURBAKHSH, 2004) e (HOLDGAARD-THOMSEN, 2010) é a expansão dos obstáculos. O algoritmo considera os obstáculos maiores do que realmente são, aumentando o obstáculo em, no mínimo, metade da largura do robô para que o centro passe tangente ao novo vértice sem haver real contato. Para isso, é preciso receber no controlador dois parâmetros a mais, o tamanho do robô e quanto mede cada célula do mapa. O controlador, então, determina quantas células o robô mede de largura e incrementa a metade mais um em todos os obstáculos.

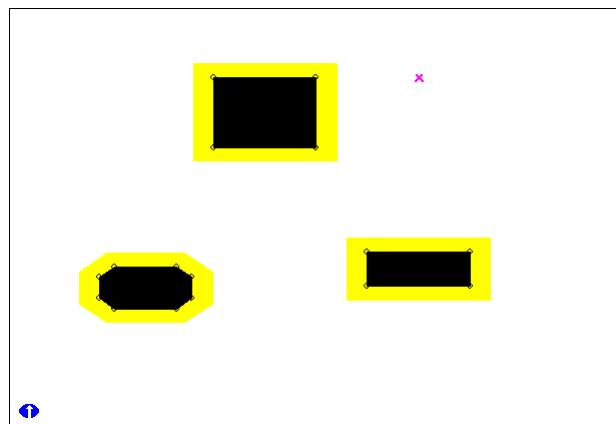


Figura 34 – Obstáculo real em preto e a expansão incrementada em amarelo ([Site do MRIT, 2014](#))

Vale lembrar que esta expansão pode mudar o percurso do robô. Um percurso

passando entre dois obstáculos pode ser fechado por este incremento. Isso impede que o robô entre em corredores muito estreitos ou entre em regiões côncavas de um obstáculo.

O Traveller permite a escolha da expansão ou não pelo usuário. A classe Map possui um atributo booleano que guarda se os obstáculos devem ou não ser expandidos. Por padrão, o *framework* não expande. Ao expandir, o Traveller marca as células expandidas como ocupadas, sendo portanto, realizada a expansão apenas ao executar fluxo completo do *framework*.

Outra consideração são os obstáculos côncavos. ([SIEGWART; NOURBAKHSH, 2004](#)) e ([GUZMÁN et al., 2008](#)) abordam a eliminação de pontos que formem áreas côncavas nos obstáculos. O simulador MRIT, citado também, segue esta abordagem. Outros autores como ([HOLDGAARD-THOMSEN, 2010](#)) e ([CHOSET et al., 2005](#)) não fazem esta consideração, pois nem sempre o espaço côncavo de um polígono é pequeno a ponto de ser perigoso e por vezes pode ser o único caminho até o objetivo.

Computar os vértices côncavos, além de consumir maior tempo e aumentar a complexidade do algoritmo de detecção de vértices, torna o grafo muito maior. Com o maior número de pontos a serem considerados, os algoritmos que dependem dessa informação (grafo de visibilidade e Voronoi) tornam-se mais demorados e consomem mais memória. Desconsiderar estes pontos só impactam de forma decisiva sobre o percurso final se o mesmo for o menor caminho entre os pontos inicial e final e se a passagem que passa pela região côncava for estreita (próxima ao tamanho do robô). Considerando isso os pontos côncavos são desconsiderados pelo Traveller.

5.6 Boas Práticas de Programação

Não apenas escrever um código que funcione, é preocupação da engenharia de software que o código seja manutenível. Isso inclui, além da modularidade e uso de padrões conhecidos, escrever um código legível e simples. ([GOODLIFFE, 2007](#)) e ([MCCONNEL, 2004](#)) descrevem características de um bom código e servem como um guia para boa programação.

Essas técnicas dão uma maior qualidade ao código, melhorando manutenibilidade, e aumentando a coesão e a reutilização. Tais temas são abordados pelos dois autores. As práticas sugeridas por eles não resolvem os problemas do sistema ou implementam suas funcionalidades, mas estão voltadas para a qualidade a longo prazo. Futuramente, a alteração e inserção de novas funcionalidades ao Traveller será facilitada com estas práticas, permitindo o crescimento do mesmo com menor esforço para sua compreensão.

Algumas dessas características, seguidas no desenvolvimento deste projeto, são:

- **Testes inteligentes:** ([GOODLIFFE, 2007](#)) e ([MCCONNEL, 2004](#)) enfatizam que

testes devem ser sempre realizados. Sempre teste o código após fazê-lo, não deixando para outro momento. Eles também enfatizam que os testes devem ser focados nos principais fluxos do sistema. Tentar testar todas as possibilidades do código é inviável e muitas delas raramente virão a ocorrer. Por isso, testes devem ser focados nas partes principais, rodando todos os fluxos das funcionalidades centrais do sistema, testando o *design* do sistema para validá-lo como robusto e testando só os fluxos principais das características menos vitais do sistema. ([MCCONNEL, 2004](#)) sugere o uso de testes automatizados e de cobertura de código.

- **Manter o mesmo estilo de escrita:** o código deve manter o mesmo padrão de escrita por todo o sistema. Se as chaves usadas ao abrir um bloco de instrução estiverão na mesma linha ou na de baixo, a identação, comentários, quebras de linha e outras variações devem ser padronizadas para facilitar a leitura e torná-lo mais agradável de ser lido.
- **Nomes significativos:** o leitor deve ser capaz de saber do que se trata a variável, método ou classe apenas lendo seu nome. O nome deve se referir ao comportamento, identidade ou padrão ao qual está relacionado. ([GOODLiffe, 2007](#)) afirma que se o programador não sabe qual nome dar a uma variável ou função é porque não sabe exatamente o que ela faz.
- **Testar as entradas do sistema:** Não confiar que os valores recebidos estão sempre corretos é indispensável para um código seguro. Deve-se sempre checar se os valores estão dentro da margem esperada e se objetos não são nulos. Isso vale não só para os dados vindos de fora do *framework*, valores vindo de outras classes e módulos devem ser verificados para garantir que não está sendo recebido um objeto nulo e evitar falhas de segmentação.
- **Clareza sobre código curto:** é preferível que o código fique maior do que difícil de entender. Dividir equações em várias linhas, simplificar as linhas de algorítmos e deixar apenas uma operação por linha são algumas das ações para tornar o código limpo e legível.
- **Considerar casos excepcionais:** mesmo que uma possibilidade de valor seja rara o código deve estar preparado para tratá-la. Um conjunto de *if-else-if* em sequência deve sempre ter um bloco *else* ao final para casos fora do esperado, assim como todo bloco *switch* deve possuir uma cláusula *default*. O programa deve estar pronto para criar o objeto incompleto ou lançar algum tipo de exceção nesses casos.
- **Cuidados com variáveis:** sempre inicializar uma variável ao criá-la para evitar ler lixo de memória é uma das ações que tornam o código mais seguro, bem como criar a variável apenas quando ela for útil torna o código mais legível.

- **Evidencie o fluxo padrão do sistema:** O fluxo principal deve ser sempre o primeiro em estruturas condicionais como *if-else* e *switch*. O leitor do código deve conseguir acompanhar o fluxo comum do código, sem procurar em uma série de opções nas estruturas condicionais qual é a correta.
- **Simplicidade sobre velocidade:** um código otimizado é mais complexo. Inserir camadas, indireção e linhas extras diminuem a velocidade do sistema, mas o tornam organizado. Serão otimizados apenas blocos de código que necessitem de velocidade. A não ser que seja essencial, deve ser dada prioridade à simplicidade em todo o sistema. Para saber se o código necessita de otimização será preciso testá-lo com uma entrada complexa para analisar o tempo de processamento. Espera-se que os algoritmos Grafo de Visibilidade e Voronoi possam se tornar lentos e, caso isto ocorra de fato, suas implementações serão revistas.
- **Funções atômicas:** cada método deve realizar apenas uma função. Funções pequenas e simples tornam o código maior, porém o deixam fácil de compreender.
- **Retorne apenas uma vez:** cada função deve ter apenas uma cláusula *return*. Sempre que possível evitar retornar uma função antes de seu fim.
- **Restrinja valores possíveis:** não permitir que uma variável assuma valores que ela não deveria ter. Definir uma variável como *unsigned*, *const* ou *enum* quando necessário.
- **Comentários que façam diferença:** um código deve ser o mais auto-explicativo possível. Quando um comportamento ou operação for complexo demais um comentário pode ajudar a entender o código. Comentários óbvios ou descrevendo o que o código faz por alto serão evitados. O comentário deve explicar o por que o código age daquele modo, não como.

5.7 Resumo do Capítulo

Foi desenvolvido um *framework* de definição de trajetória implementando algoritmos globais. Como não se comunicam diretamente com sensores e atuadores, seu desenvolvimento se torna mais independente dos aspectos físicos do robô. O *framework* será externo ao robô, sendo executado em um servidor e receberá os dados do robô para a execução e devolverá ao mesmo a lista de pontos a serem percorridos.

O Traveller Framework possui três *hot-spots* em que podem ser escolhidos os algoritmos pelo usuário e que podem ser extendidos por outros programadores. Esses *hot-spots* são o algoritmo de definição de trajetória (PathPlanner), o algoritmo de menor caminho (BestPath) e a comunicação do robô com o servidor (EmbeddedCommunicator). Estes *hot-spots* podem ser herdados e configurados como desejar.

Internamente, o Traveller trabalha com um mapa do tipo malha de ocupação com valores inteiros. A comunicação do robô com o *framework* se dará por uma classe dentro do robô, herdada de *EmbeddedCommunicator*. Serão passados os pontos inicial e final para esta classe, além do mapa e de quais algoritmos deseja executar. O *EmbeddedCommunicator* repassa as informações ao servidor que instanciará os objetos e executará o fluxo da resolução, presente na classe Controller.

O *framework* tem oito componentes: *Graph* (responsável pelo grafo), *Controller* (controlador do sistema), *Map* (responsável pelo mapa), *PathPlanner* (algoritmo global), *BestPath* (algoritmo de melhor caminho), *Main* (responsável por iniciar o sistema e desacoplar a comunicação do *framework*) *EmbeddedCommunicator* (responsável pela comunicação no lado do robô) e *RobotCommunication* (responsável pela comunicação no lado do servidor). O desenvolvimento de cada um deles é embasado em boas práticas de programação e voltado à clareza e à simplicidade do código.

6 Resultados Obtidos

O desenvolvimento do Traveller foi, desde seu início, fortemente testado por uma suíte de testes que garantiam a integridade do código em um ambiente de mudanças constantes. Assim, foi possível garantir que mesmo inserindo novos códigos, módulos e alterando as classes já existentes, as funções continuavam em funcionamento. Quando alguma alteração mudava os resultados das saídas, os testes acusavam imediatamente.

Porém, para uma suíte de testes garantir essa integridade, os testes realizados precisam cobrir as áreas passíveis de falhas e críticas do sistema. Um teste que verifica uma função muito simples ou que nunca muda não nos confere tanta integridade ao sistema como uma que verifica se o sistema responde corretamente ao entrar com um mapa, o qual não permite chegar ao destino final. Para isso, a suíte de testes foi constantemente avaliada e revisada, buscando uma maior qualidade dos testes realizados.

A seguir serão descritos os testes realizados durante o desenvolvimento do Traveller *Framework* e os impactos dos mesmos no desenvolvimento do sistema.

6.1 Testes iniciais

Ao iniciar o desenvolvimento do *framework*, foram desenhados sete mapas simples apenas para testar se os algoritmos estavam funcionando. Os mapas desenhados eram pequenos, permitindo calcular o grafo gerado e o menor caminho manualmente. Os mapas variavam de 4 a 9 células de largura, podendo ser quadradas ou retangulares. A mudança de tamanho e formato permitiu testar as divisões em áreas do Quadtree e verificar se o mesmo repartia como planejado. Para esses exemplos iniciais não foram considerados: (i) a expansão dos obstáculos, e (ii) que o robô e a célula possuíssem o mesmo tamanho. As saídas de cada classe (o grafo e a lista de pontos) foram testadas para cada caso. A classe *controller* não foi usada, permitindo acesso ao grafo e a verificação e validação do mesmo. A figura 35 mostra um destes testes.

6.2 Testes de desempenho

Após a validação do tema e verificado o funcionamento correto do fluxo de trabalho do *framework*, o próximo passo foi o teste de desempenho do mesmo. Foram analizados o consumo de memória e o tempo de processamento para a avaliação do mesmo.

Os testes foram realizados na IDE Netbeans, usando sua ferramenta de perfis. Esta ferramenta permite avaliar a memória que a JVM (*Java Virtual Machine*) está consumindo

```

public class TestQuadtree {

    Map map;
    PathPlanner quadtree;
    Graph graph;

    @Before
    public void setUp() throws Exception {
        boolean[][] mapa = {{true,true,false,false,false,false,false}, 
            {false,false,false,false,false,false,false}, 
            {false,false,false,false,false,true,true,false}, 
            {false,false,false,false,false,false,false}, 
            {false,false,true,true,true,false}, 
            {false,false,true,true,true,true,false}, 
            {false,false,true,true,false,false}, 
            {false,false,false,false,false,false}};

        map = new Map(mapa, false);
        map.setInitialPoint(1, 6);
        map.setFinalPoint(5, 6);
        quadtree = new Quadtree(1,1);
        quadtree.setMap(map);
        graph = quadtree.resolution();
    }

    @After
    public void tearDown() throws Exception {
        graph.remove();
        graph = null;
        quadtree = null;
        map = null;
    }
}

@Test
public void testPath() {
    BestPath djikstra = new Djikstra();
    djikstra.setGraph(graph);
    List<Point> list = djikstra.definePath("6,1", "6,5").getRoute();
    assertEquals(list.size(), 5);
    assertEquals((int)list.get(0).getX(), 1);
    assertEquals((int)list.get(0).getY(), 6);
    assertEquals((int)list.get(1).getX(), 2);
    assertEquals((int)list.get(1).getY(), 7);
    assertEquals((int)list.get(2).getX(), 3);
    assertEquals((int)list.get(2).getY(), 7);
    assertEquals((int)list.get(3).getX(), 5);
    assertEquals((int)list.get(3).getY(), 7);
    assertEquals((int)list.get(4).getX(), 5);
    assertEquals((int)list.get(4).getY(), 6);
}

@Test
public void test() {
    assertEquals(graph.getNumNodes(), 22);
    assertTrue(graph.nodeAlreadyExists("1,3"));
    assertTrue(graph.nodeAlreadyExists("3,3"));
    assertTrue(graph.nodeAlreadyExists("3,1"));
    assertTrue(graph.nodeAlreadyExists("1,0"));
    assertTrue(graph.nodeAlreadyExists("1,1"));
    assertTrue(graph.nodeAlreadyExists("1,5"));
    assertTrue(graph.nodeAlreadyExists("1,7"));
    assertTrue(graph.nodeAlreadyExists("2,4"));
    assertTrue(graph.nodeAlreadyExists("3,4"));
    assertTrue(graph.nodeAlreadyExists("3,5"));
    assertTrue(graph.nodeAlreadyExists("2,7"));
    assertTrue(graph.nodeAlreadyExists("3,6"));
    assertTrue(graph.nodeAlreadyExists("3,7"));
    assertTrue(graph.nodeAlreadyExists("5,1"));
}

```

Figura 35 – Teste inicial do *framework*

e o tempo de processamento de um programa, tomando como base cada método e classe. Ela fornece o número de vezes que um método é chamado, o tempo total para execução dos mesmos, o tempo requerido para execução de toda a classe e a porcentagem de cada um.

O algoritmo utilizado nestes testes foi o Quadtree. Este algoritmo permitiu uma análise mais rigorosa devido a sua implementação exigir maior memória para sua execução, através de sua recursão e de aumentar muito o número de objetos de acordo com o número e disposição dos obstáculos. Assim, foi analisado o *framework* em seu pior caso, com mapas grandes, muitos obstáculos e com um algoritmo que cresce seu consumo junto com estas variantes.

6.2.1 Testes de Stress - Simulação

Os primeiros testes foram para testar os limites de processamento do Lego NXT, inserindo um mapa de 100 por 100 células e com uma enorme quantidade de obstáculos. Foram inseridos 1089 obstáculos (um a cada três casas) de 2 por 2 células. Dificilmente um ambiente assim seria utilizado para uso real. Entretanto, esse teste visava apenas avaliar o quanto o *hardware* do robô utilizado suportava. As Figuras 36 e 37 mostram o resultado da análise de processamento deste teste.

O teste levou ao todo quase 4 minutos para ser executado no computador, sendo este tempo repartido entre todos os processos da máquina e entre os vários processos

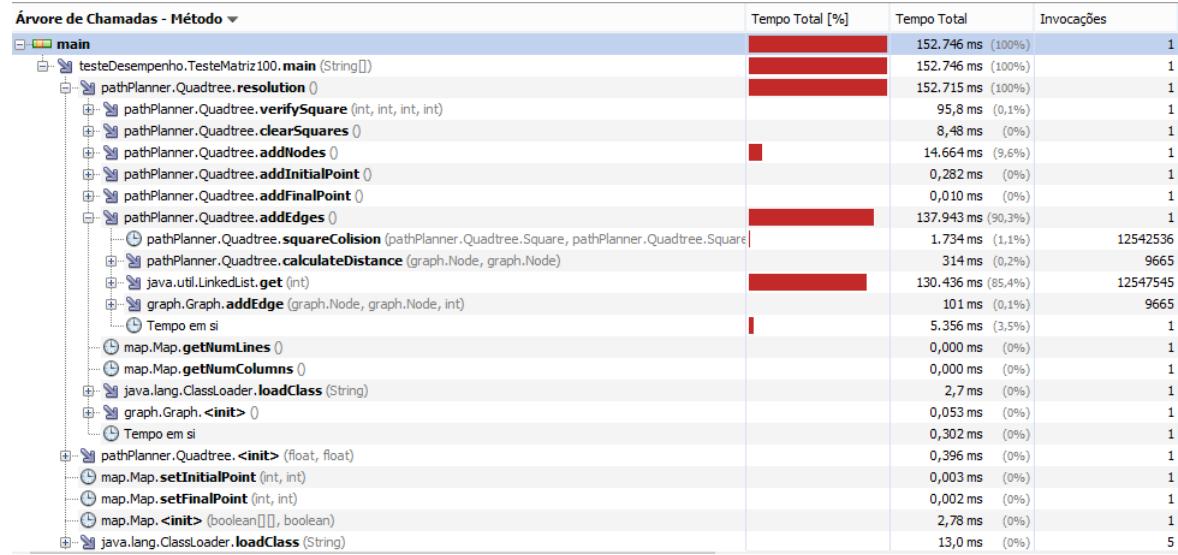


Figura 36 – teste de stress, tempo de execução

Hot Spots - Método	Tempo Decorrido	Tempo Total	Invocações
java.util.LinkedList.node (int)	121.215 ms (79,4%)	121.215 ms	12.557.563
pathPlanner.Quadtree.addEdge()	5.356 ms (3,5%)	137.943 ms	1
java.util.ArrayList.get (int)	5.194 ms (3,4%)	8.701 ms	12.607.681
java.util.LinkedList.get (int)	4.837 ms (3,2%)	130.645 ms	12.552.554
graph.Graph.nodeAlreadyExists (graph.Node)	3.851 ms (2,5%)	14.425 ms	5.009
java.util.LinkedList.checkElementIndex (int)	3.078 ms (2%)	4.595 ms	12.557.563
java.lang.String.equals (Object)	1.914 ms (1,3%)	1.914 ms	12.542.562
java.util.ArrayList.elementData (int)	1.767 ms (1,2%)	1.767 ms	12.607.681
java.util.ArrayList.rangeCheck (int)	1.738 ms (1,1%)	1.738 ms	12.607.681
pathPlanner.Quadtree.squareCollision (pathPlanner.Quadtree.Square, pathPlanner.Quadtree.Square)	1.734 ms (1,1%)	1.734 ms	12.542.536
java.util.LinkedList.isElementIndex (int)	1.516 ms (1%)	1.516 ms	12.557.563
java.lang.Integer.parseInt (String, int)	36,9 ms (0%)	147 ms	38.660
java.lang.Character.digit (int, int)	30,2 ms (0%)	71,7 ms	73.538
graph.Node.addEdge (graph.Node, int)	27,9 ms (0%)	96,2 ms	28.995
pathPlanner.Quadtree.calculateDistance (graph.Node, graph.Node)	26,2 ms (0%)	314 ms	9.665
java.lang.CharacterDataLatin1.digit (int, int)	21,1 ms (0%)	31,3 ms	73.538
java.lang.Character.digit (char, int)	19,4 ms (0%)	91,1 ms	73.538
graph.Node.getCoordinates ()	17,8 ms (0%)	88,1 ms	19.330
pathPlanner.Quadtree.verifySquare (int, int, int, int)	17,8 ms (0%)	95,8 ms	14.677
pathPlanner.Quadtree.addNode()	16,8 ms (0%)	14.664 ms	1
java.lang.String.substring (int)	16,5 ms (0%)	29,6 ms	19.330
java.util.Arrays.copyOfRange (char[], int, int)	15,6 ms (0%)	21,5 ms	43.718
graph.Node.edgeAlreadyExists (graph.Node)	15,2 ms (0%)	53,7 ms	28.995
java.lang.String.charAt (int)	14,5 ms (0%)	14,5 ms	112.577
java.lang.String.<init> (char[], int, int)	13,5 ms (0%)	35,1 ms	43.718

Figura 37 – teste de stress, métodos que mais foram utilizados

dentro da própria máquina virtual Java. A análise retornou que o tempo gasto pelo framework foi pouco mais de 152 milissegundos. A quantidade de tempo massiva gasta foi no método *addEdges* (responsável por definir as ligações entre os nós do grafo) do Quadtree (90,3%), aonde 85,4% do tempo total foi com o método *get* do LinkedList e apenas 3,5% do tempo foi com o método *em si*.

A figura 37 nos mostra que, neste mesmo teste, a classe LinkedList consumiu muito mais tempo de processador que as classes do próprio framework. No total, a classe foi responsável por 85,5% do tempo de processamento.

Com relação ao gasto de memória, a função que mais consumiu foi a classe Square, do Quadtree. Justamente pela imensa quantidade de obstáculos e o pouco espaço entre

eles, o algoritmo gerou uma imensa quantidade de quadrados para a geração do grafo, consumindo boa parte da memória. Não foi possível determinar a quantidade exata de memória gasta, pois a ferramenta indicava o gasto total da máquina virtual Java e não apenas o do processo estudado.

Um segundo teste foi realizado com o mesmo ambiente e os resultados foram os mesmos, com poucas mudanças em relação ao tempo.

6.2.2 Testes de Stress no Robô

Ao embarcar este código no robô, foi identificado um erro, devido à falta de memória. O tamanho do mapa foi diminuído para 30 por 30, mantendo a mesma proporção de obstáculos e o espaçamento entre eles. Mesmo com esta diminuição, ainda faltou memória no robô, precisando diminuir para 20 o tamanho do mapa. Foi medido com cronômetro digital que o tempo levado pelo *framework* para calcular a trajetória com o mapa de 20x20 no robô foi de 15 segundos (tempo elevado devido ao baixo poder de processamento do robô). Este mapa gerou um grafo com 500 nós e 1889 arestas. Este valor sozinho não pode definir o tamanho máximo de um grafo que o robô suporta (em um teste futuro, um mapa com grafo menor não executou), pois o tamanho do mapa e outras estruturas também ocupam a memória e afetam o limite da máquina.

Apesar de dificilmente haver um ambiente onde haja tantos obstáculos ou que o robô só possa se mover sobre uma grade, foi visto que mesmo com mapas pequenos a memória do Lego NXT (64KB) não suporta o *framework*.

Outro teste feito no robô confirmou que o robô não suportaria o *framework* para a maioria dos mapas. Nele, um mapa de 60x60 foi criado e inseridos apenas três obstáculos. Este mapa foi retirado do simulador MRIT, que trabalha com um mapa de 60x60 e três obstáculos inicialmente.

Ao embarcá-lo no Lego, houve novamente estouro de memória. No grafo gerado por este mapa haviam 208 nós e 1060 arestas. O robô só conseguiu rodar o sistema ao retirar um dos obstáculos, gerando um grafo de 133 nós e 666 arestas. O tamanho do mapa foi mantido 60x60.

6.2.3 Decisões tomadas

Considerando que mesmo com mapas pequenos de 30x30 o Traveller pode não funcionar em alguns robôs e, principalmente, pelo teste com o mapa retirado do MRIT de 60x60 e 3 obstáculos não ter funcionado no kit da Lego, foi verificado que o consumo de memória do algoritmo Quadtree é muito alto e o *framework* precisaria ser movido para fora do robô. Outros algoritmos, como o Wavefront e Grafo de Visibilidade, podem

consumir bem menos memória, porém o *framework* deve funcionar mesmo no pior caso. Dessa forma, a decisão de colocá-lo em um servidor foi tomada.

Vale lembrar que um mapa de 60x60 pode ainda ser pequeno dependendo da área de atuação do robô e do seu tamanho, já que a proporção entre robô e a área influencia no tamanho da matriz que o mesmo gerará.

Outra decisão foi a troca pela estrutura `LinkedList` pelo `ArrayList`. A grande maioria do tempo era gasto nesta classe, portanto, ela foi substituída por outra semelhante, também fornecida pelo Java. O `ArrayList` acessa um elemento com o método `get` com ordem de complexidade $O(1)$, enquanto o `LinkedList` faz o mesmo em ordem $O(n)$. Mesmo o `LinkedList` apresentando vantagens quanto a gasto de memória e inserção e remoção de dados, essas funções eram pouco expressivas na totalidade do código, nem sequer chegando a aparecer nas estatísticas geradas. Enquanto a função `get` estava sempre entre as que mais consumiam tempo.

6.3 Testes unitários e de Integração

Para a realização dos testes foi utilizada a ferramenta JUnit, que fornece classes para a realização de testes unitários. Estas classes foram aproveitadas para realizar outros tipos de testes, como os testes de integração explicados na seção 6.3.2.

A cada módulo construído e cada algoritmo implementado, uma nova suíte de testes era desenvolvida para validá-la. Outros detalhes sobre suas implementações são descritos nas seções a seguir.

6.3.1 Testes unitários e cobertura de código

Os testes unitários realizados buscavam a validação dos algoritmos, dos componentes e suas comunicações. Para tanto, foram realizados testes que chamavam as classes dos *hot-spots* separadamente (sem se utilizar da controladora) para testar a saída de cada algoritmo.

Para os testes foram usados os mesmos mapas utilizados nos testes iniciais, que por seu tamanho reduzido permitiam calcular seus resultados manualmente para os quatro algoritmos e verificar-lhos com as funções do JUnit. Caso o algoritmo funcionasse nos casos testados, era considerado correto.

Para o diagrama de Voronoi houve uma maior dificuldade de implementar os testes unitários, mesmo em mapas menores e com menos obstáculos, devido a complexidade de seu algoritmo. Como o algoritmo fazia uso de diversos métodos privados, a maior parte do algoritmo não pôde ser testada diretamente pelos testes automatizados. Entretanto, testes manuais mostraram o correto funcionamento da maioria das funções internas, e

que a falha estava presente na criação das triangulações de Delauney, o ponto central do algoritmo.

Como a maioria dos métodos que executam a lógica é privado, foi verificado diretamente pelas funções do JUnit apenas as saídas dos algoritmos e se as mesmas eram totalmente compatíveis com os valores esperados. Caso as saídas fossem compatíveis em todos os casos, então os métodos internos estariam corretos.

Algumas funções da classe *Map* de maior complexidade e relevância também foram diretamente testadas. Os métodos de expansão dos obstáculos e de localizar e retornar os vértices dos obstáculos também foram verificados pela suíte de testes pela sua importância nos algoritmos por possuírem uma lógica mais elaborada que as demais da mesma classe. Quanto à expansão, foi expandido um obstáculo aonde o robô possuía o mesmo tamanho da célula da matriz (expansão de 1 célula), aonde o robô era menor (expansão também de 1 célula por arredondar para cima) e aonde o robô era 1,5 vezes maior (expansão de duas casas). Para a localização de vértices foram testados obstáculos retangulares, triangulares e hexagonais. Considerando que o mapa é uma matriz, todo polígono pentagonal ou maior à vizinhança do ponto seria a mesma.

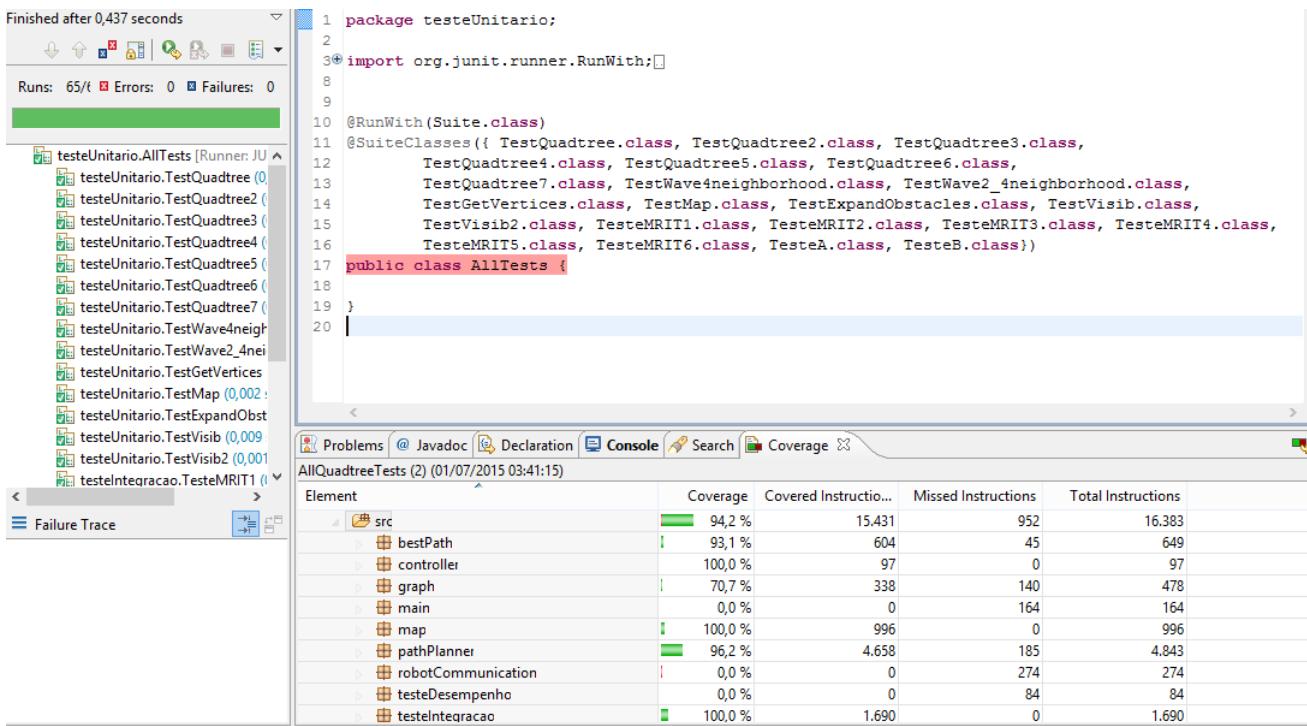
Além do JUnit, foi utilizado também o plugin Eclemma para a cobertura de código (Figura 38). Este plugin é executado juntamente com os testes unitários e marca as linhas executadas e qual a porcentagem do código foi verificada naquele teste. Como a execução do algoritmo executava todos os métodos internos, as classes eram testadas e cobertas em sua maioria. Ao final de todos os testes, foi alcançada uma cobertura de código de 94,2%.

6.3.2 Testes de Integração: Comparações Com o Simulador MRIT

Além dos testes unitários, testando os métodos e os algoritmos de forma direta e atômica, foram realizados testes de integração, que testaram a comunicação entre os módulos do sistema. Esse grupo de testes é importante, pois após garantir que cada módulo funciona de forma correta, é preciso garantir que eles se comunicam de forma correta também.

Para tanto, foram montados seis cenários a serem testados. Para a comparação dos algoritmos e validação de seu funcionamento com o simulador MRIT, foram criados seis mapas no simulador e definida suas trajetórias para cada algoritmo. Estes mapas foram também criados no *framework*, com o mesmo tamanho e os obstáculos nas mesmas posições para a comparação dos resultados.

Uma vez montados os seis mapas, foi definido o algoritmo de menor caminho como Djikstra e executado o *framework* para cada um dos algoritmos de definição de trajetória. A execução foi feita a partir da controladora PathPlannerController, do mesmo modo que o módulo *Main* o executa. Assim, foi testado o módulo controlador e sua integração

Figura 38 – Cobertura de código do *framework*

com os módulos PathPlanner, Map e BestPath. Todo o fluxo de trabalho foi executado em cada teste, criando o mapa, instanciando os algoritmos, expandido os obstáculos caso fosse definido pelo teste e, por fim, executado ambos os algoritmos, gerando o grafo, passando-o para o BestPath e gerando a lista de pontos no final. Com estas verificações, foi possível validar que o Traveller funcionava como um todo e que respondia corretamente às entradas dos dados e gerava a saída de forma correta.

As figuras 39 à 44 mostram os mapas montados no simulador MRIT. As trajetórias em vermelho foram geradas pelo simulador, enquanto as azuis foram geradas pelo Traveller. Uma observação importante a ser feita é que tanto o simulador MRIT quanto o Traveller podem não retornar uma trajetória caso não consigam encontrar um caminho. Para o mapa 3, o MRIT não conseguiu encontrar uma solução para o Wavefront, para o mapa 5 não encontrou um caminho para o Voronoi e para o mapa 6 o único algoritmo a encontrar um percurso foi o Grafo de Visibilidade. Uma curiosidade interessante é que, quando o simulador não consegue gerar um percurso, os obstáculos somem, ficando apenas os pontos inicial e final no mapa (como visível nas Figuras 41, 43 e 44).

Quanto ao Traveller, para os seis mapas criados, apenas o Voronoi apresentou problemas, sendo encontrado um resultado apenas para o segundo mapa. No sexto mapa, além do Voronoi, o Quadtree também não encontrou um caminho. Isso se deve a ambos recortarem os ambientes livres em áreas e nenhum dos dois encontrar um espaço grande o suficiente para criar as áreas necessárias para ligar os pontos inicial e final.

Para os testes, a orientação do robô foi considerada a mesma para todos, 0 graus.

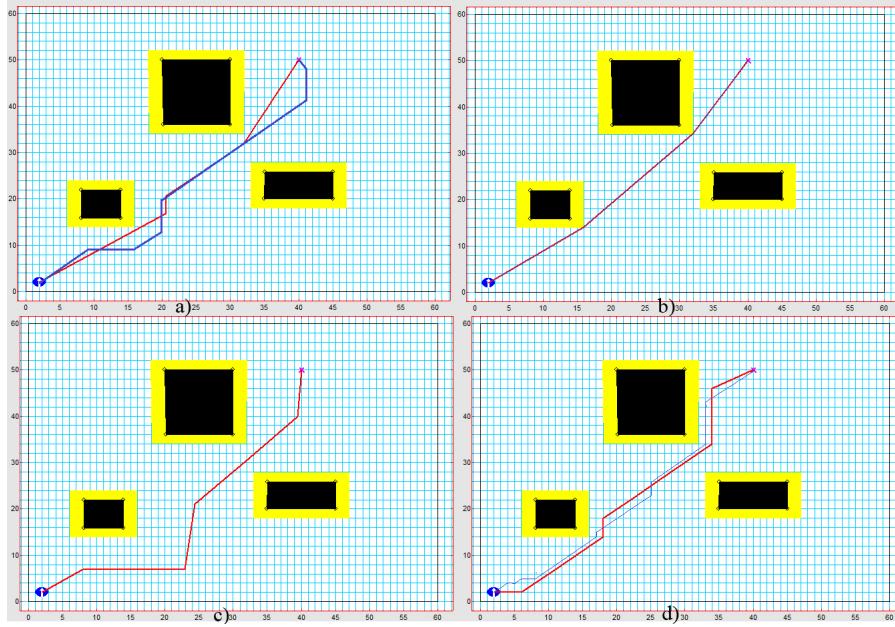


Figura 39 – Trajetórias para o mapa 1 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront

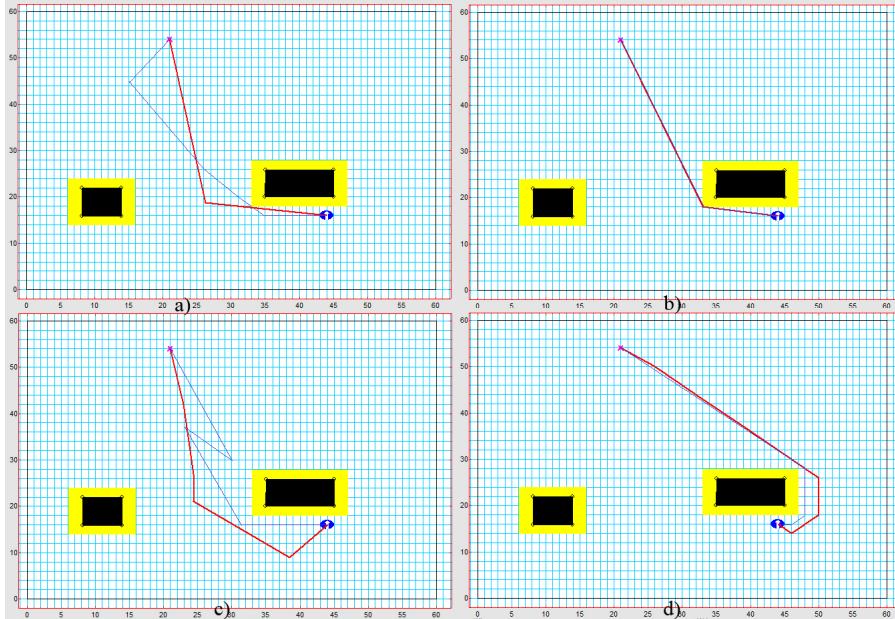


Figura 40 – Trajetórias para o mapa 2 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront

Considerando as seis amostras, é possível analisar algumas semelhanças e diferenças entre as implementações dos algoritmos do simulador MRIT e do Traveller.

Nos resultados do *framework*, o Quadtree nem sempre parece estar se dirigindo ao ponto final. Ele pode, em alguns casos, seguir para outras direções ao longo do percurso,

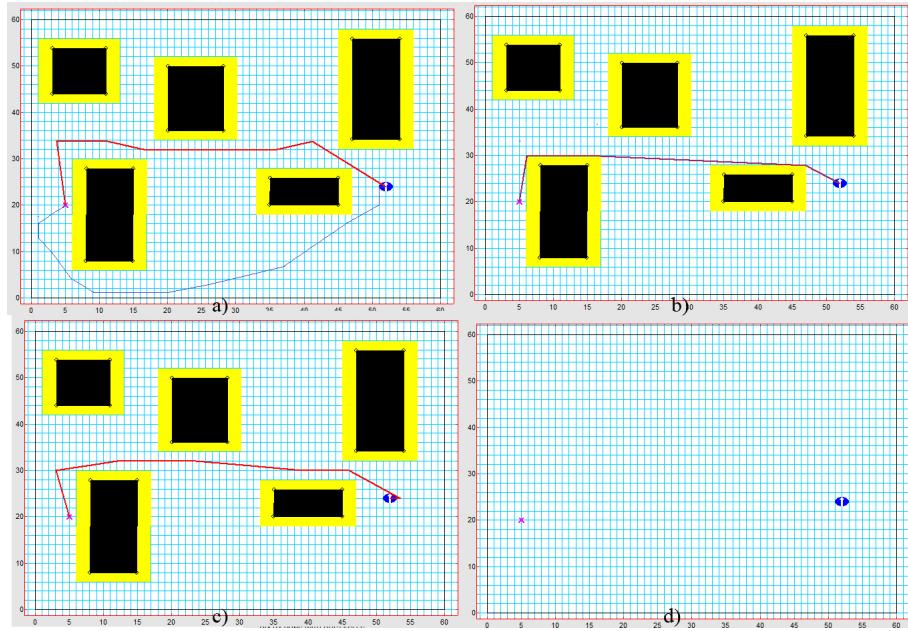


Figura 41 – Trajetórias para o mapa 3 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront

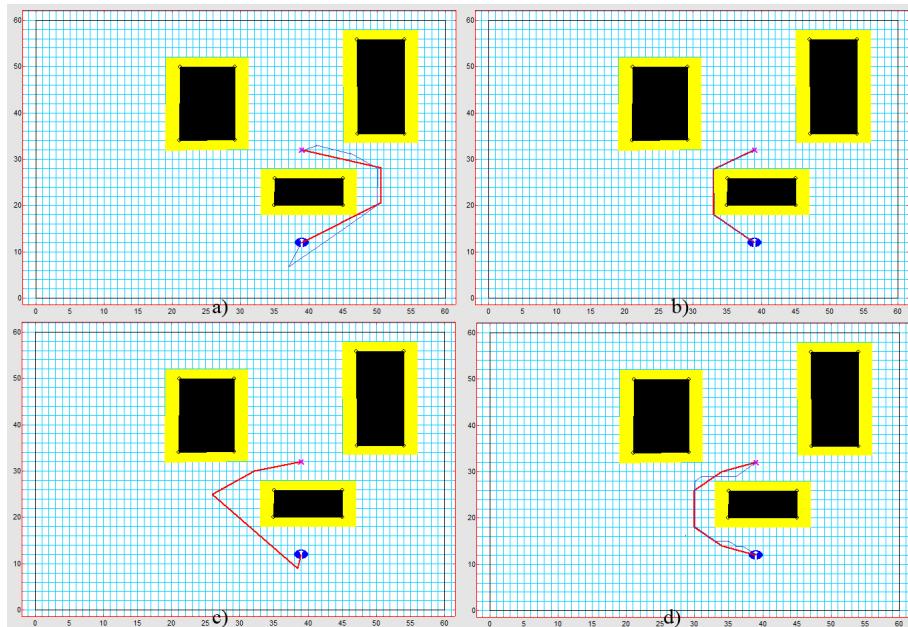


Figura 42 – Trajetórias para o mapa 4 (MRIT em vermelho e Traveller em azul) a) Quadtree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront

para depois virar em direção ao objetivo. Esse comportamento fez com que o Quadtree gerasse um caminho maior. Foi observado também que para os mapas 3 e 5 o algoritmo do *framework* contornou o obstáculo por um lado diferente do simulador, o que indica que a divisão do mapa em retângulos foi implementada de forma diferente do simulador. O percurso gerado pelo Traveller também gerou um caminho menos retilíneo.

Para o algoritmo Grafo de Visibilidade, o *framework* gerou o mesmo caminho do

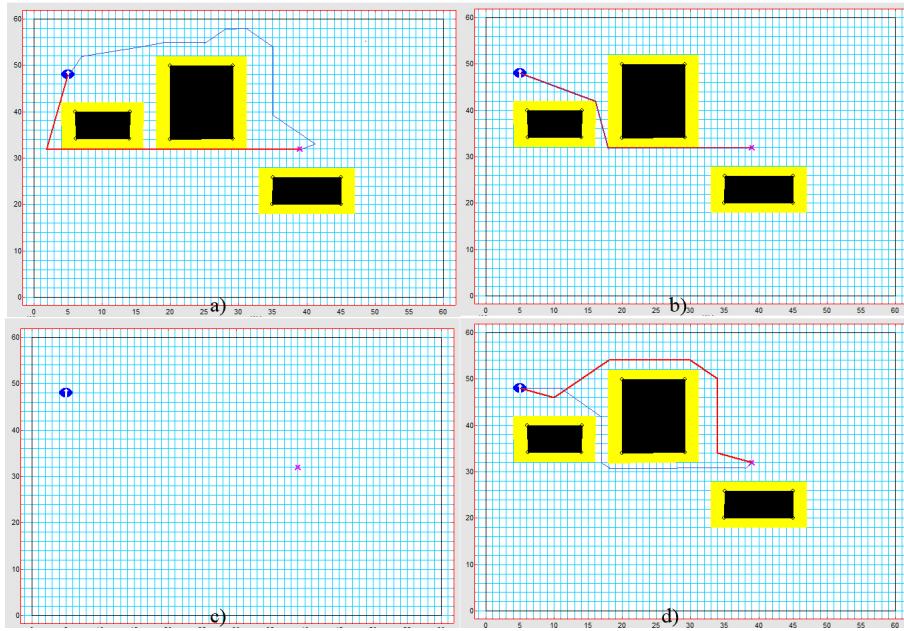


Figura 43 – Trajetórias para o mapa 5 (MRIT em vermelho e Traveller em azul) a) Quad-tree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront

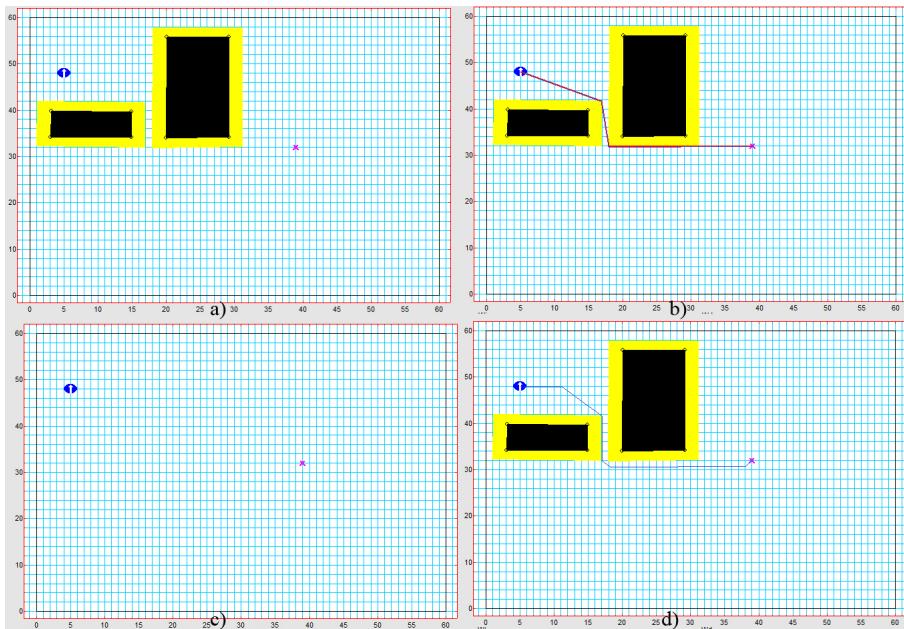


Figura 44 – Trajetórias para o mapa 6 (MRIT em vermelho e Traveller em azul) a) Quad-tree, b) Grafo de Visibilidade, c) Voronoi, d) Wavefront

simulador. O Grafo de Visibilidade implementado pelo Traveller seguiu exatamente o que se esperava do algoritmo.

Como já comentado, o algoritmo Voronoi não conseguiu chegar a um resultado satisfatório. Para os seis mapas, apenas para o segundo o algoritmo conseguiu gerar um caminho. Pelos testes realizados durante sua implementação, a falha no algoritmo está presente na criação dos triângulos de Delauney e validação das arestas destes triângulos.

Quanto ao algoritmo Wavefront, para os mapas 3 e 6 o simulador não conseguiu gerar um resultado, enquanto o *framework* conseguiu. Quanto aos percursos gerados, estes não mudaram tanto, seguindo mais ou menos a mesma trajetória. A única exceção foi o mapa 5, em que o percurso do *framework* contornou um obstáculo por um lado, diminuindo o caminho, enquanto o simulador contornou por outro lado. O *framework* gerou um caminho significativamente menor para a maioria dos casos, obtendo um resultado pior apenas para o mapa 4, onde a diferença foi mínima (1,9 unidades).

O tamanho dos percursos gerados por cada algoritmo em cada mapa pode ser visto nas Tabela 1 à 6.

Tabela 1 – Tamanho dos percursos gerados pelo simulador e o *framework* para o mapa 1

Algoritmo	Traveller	MRIT
Grafo de Visibilidade	61.94	61.94
Voronoi	X	60.79
Quadtree	68.49	63.78
Wavefront	65.00	67.88

Tabela 2 – Tamanho dos percursos gerados pelo simulador e o *framework* para o mapa 2

Algoritmo	Traveller	MRIT
Grafo de Visibilidade	49.13	49.13
Voronoi	70.69	87
Quadtree	55.24	54.46
Wavefront	52.20	56.8

Tabela 3 – Tamanho dos percursos gerados pelo simulador e o *framework* para o mapa 3

Algoritmo	Traveller	MRIT
Grafo de Visibilidade	57.52	57.52
Voronoi	X	77.88
Quadtree	77.38	66.57
Wavefront	60.20	X

Tabela 4 – Tamanho dos percursos gerados pelo simulador e o *framework* para o mapa 4

Algoritmo	Traveller	MRIT
Grafo de Visibilidade	28.94	28.94
Voronoi	X	39.14
Quadtree	44.39	33.98
Wavefront	32	30.1

Tabela 5 – Tamanho dos percursos gerados pelo simulador e o *framework* para o mapa 5

Algoritmo	Traveller	MRIT
Grafo de Visibilidade	43.73	43.73
Voronoi	X	X
Quadtree	61.47	53.28
Wavefront	47.2	55.75

Tabela 6 – Tamanho dos percursos gerados pelo simulador e o *framework* para o mapa 6

Algoritmo	Traveller	MRIT
Grafo de Visibilidade	43.73	43.73
Voronoi	X	X
Quadtree	X	X
Wavefront	47.2	X

6.3.3 Testes de Integração: Desempenho

Foi também verificado o desempenho do *framework* diante da execução de todo seu fluxo de dados. O primeiro teste feito foi uma comparação entre os métodos `LinkedList` e `ArrayList`. Foi executado o mapa 1 com `LinkedList` em seus métodos e, após isso, substituído todos por `ArrayList` e executado o teste novamente. As diferenças de tempo podem ser vistas nas Figuras 45 e 46.

Com o `LinkedList`, o *framework* levou 39,2 milisegundos para ser executado e a função `get` levou, ao total de todas as suas chamadas, 1,72 milisegundos de processamento. Ao substituir para `ArrayList`, o tempo total decaiu para 35,47 milisegundos e o tempo da função `get` caiu para 1,41 milisegundos. A diferença de 3,73 milisegundos foi o suficiente para levar a mudança de todas as estruturas para `ArrayList`.

Uma vez que todo o *framework* usava `ArrayList`, foi testado o desempenho de todos os algoritmos nos seis mapas. Nas figuras 47 à 50 é mostrado o resultado do processamento

Hot Spots - Método	Tempo Decorrido	Tempo Total
java.io.File.exists()	3,20 ms (8,2%)	3,20 ms
pathPlanner.Quadtree.addEdge()	1,77 ms (4,5%)	16,4 ms
java.util.LinkedList.get(int)	1,72 ms (4,4%)	4,55 ms
java.io.FileInputStream.<init>(java.io.File)	1,31 ms (3,4%)	1,39 ms
pathPlanner.Quadtree.isAllFree(int, int, int)	1,28 ms (3,3%)	1,60 ms
java.util.LinkedList.checkElementIndex(int)	1,19 ms (3,1%)	1,78 ms
pathPlanner.Quadtree.squareCollision(pathPlanner.Quadtree.Square, pathPlanner.Quadtree.Square)	1,19 ms (3,1%)	1,19 ms
bestPath.Dijkstra.definePath(graph.Node, graph.Node)	1,16 ms (3%)	7,37 ms
java.io.File.length()	1,8 ms (2,8%)	1,9 ms
java.util.LinkedList.node(int)	1,4 ms (2,7%)	1,4 ms
java.lang.Integer.parseInt(String, int)	1,1 ms (2,6%)	3,43 ms
pathPlanner.Quadtree.verifySquare(int, int, int, int)	0,993 ms (2,5%)	5,57 ms
java.lang.ClassLoader.defineClass(String, byte[], int, int, java.security.ProtectionDomain)	0,976 ms (2,5%)	1,30 ms
graph.Node.getCoordinates()	0,759 ms (1,9%)	3,20 ms
graph.Graph.nodeAlreadyExists(graph.Node)	0,744 ms (1,9%)	2,61 ms
java.util.Arrays.copyOfRange(char[], int, int)	0,743 ms (1,9%)	0,864 ms
java.lang.Character.digit(int, int)	0,722 ms (1,8%)	1,75 ms
graph.Node.getIntCoordinates()	0,653 ms (1,7%)	7,59 ms
map.Map.expandObstacles(float, float)	0,627 ms (1,6%)	0,632 ms
graph.Node.addEdge(graph.Node, float)	0,615 ms (1,6%)	3,7 ms
java.lang.CharacterDataLatin1.digit(int, int)	0,592 ms (1,5%)	0,823 ms
java.util.LinkedList.isElementIndex(int)	0,584 ms (1,5%)	0,584 ms
java.lang.String.<init>(char[], int, int)	0,533 ms (1,4%)	1,42 ms
graph.Node.edgeAlreadyExists(graph.Node)	0,446 ms (1,1%)	1,17 ms
pathPlanner.Quadtree.calculateDistance(graph.Node, graph.Node)	0,427 ms (1,1%)	8,64 ms

Figura 45 – tempo de processamento do mapa 1 com LinkedList

Hot Spots - Método	Tempo Decorrido	Tempo Total
java.io.File.exists()	2,98 ms (8,4%)	2,98 ms
pathPlanner.Quadtree.addEdge()	1,77 ms (5%)	14,7 ms
java.io.FileInputStream.<init>(java.io.File)	1,50 ms (4,2%)	1,58 ms
java.util.ArrayList.get(int)	1,41 ms (4%)	2,52 ms
pathPlanner.Quadtree.squareCollision(pathPlanner.Quadtree.Square, pathPlanner.Quadtree.Square)	1,20 ms (3,4%)	1,20 ms
pathPlanner.Quadtree.isAllFree(int, int, int)	1,16 ms (3,3%)	1,44 ms
bestPath.Dijkstra.definePath(graph.Node, graph.Node)	1,5 ms (3%)	6,64 ms
java.io.File.length()	1,5 ms (2,9%)	1,5 ms
pathPlanner.Quadtree.verifySquare(int, int, int, int)	0,958 ms (2,7%)	5,44 ms
java.lang.ClassLoader.defineClass(String, byte[], int, int, java.security.ProtectionDomain)	0,954 ms (2,7%)	1,28 ms
java.lang.Integer.parseInt(String, int)	0,940 ms (2,6%)	3,3 ms
java.util.Arrays.copyOfRange(char[], int, int)	0,778 ms (2,2%)	0,886 ms
graph.Node.getCoordinates()	0,771 ms (2,2%)	3,15 ms
graph.Graph.nodeAlreadyExists(graph.Node)	0,701 ms (2%)	1,98 ms
graph.Node.getIntCoordinates()	0,698 ms (2%)	7,15 ms
java.lang.Character.digit(int, int)	0,616 ms (1,7%)	1,53 ms
graph.Node.addEdge(graph.Node, float)	0,590 ms (1,7%)	2,62 ms
map.Map.expandObstacles(float, float)	0,589 ms (1,7%)	0,594 ms
java.util.ArrayList.elementAt(int)	0,560 ms (1,6%)	0,560 ms
java.util.ArrayList.rangeCheck(int)	0,539 ms (1,5%)	0,539 ms
java.lang.CharacterDataLatin1.digit(int, int)	0,532 ms (1,5%)	0,736 ms
java.lang.String.<init>(char[], int, int)	0,518 ms (1,5%)	1,43 ms
graph.Node.edgeAlreadyExists(graph.Node)	0,452 ms (1,3%)	0,855 ms
pathPlanner.Quadtree.calculateDistance(graph.Node, graph.Node)	0,408 ms (1,1%)	8,18 ms
map.Map.<init>(boolean[], boolean)	0,399 ms (1,1%)	0,399 ms

Figura 46 – tempo de processamento do mapa 1 com ArrayList

para o mapa 3. Ao avaliar os 24 resultados, foi analisado o desempenho de cada algoritmo para variados tipos de ambiente. O tempo médio das simulações foi de: 32.1 milisegundos para o Quadtree, 18.4 milisegundos para o Grafo de Visibilidade, 70 milisegundos para o Voronoi e 167 milisegundos para o Wavefront. Portanto, pela simulação, o Wavefront levou muito mais tempo para ser calculado que os demais algoritmos e o Voronoi consumiu um tempo considerável a mais que o Quadtree e o Grafo de Visibilidade. Também foi observado que, nas simulações, o Grafo de Visibilidade teve um desempenho mais uniforme, com pouca variação entre seus tempos.

Em todos os casos, o tempo do algoritmo de definição de trajetória foi muito maior que o de definição de menor caminho e, de todo o fluxo, a definição de trajetória

e os métodos de outras classes chamadas dentro dele ocuparam sempre mais de 50% do processador.

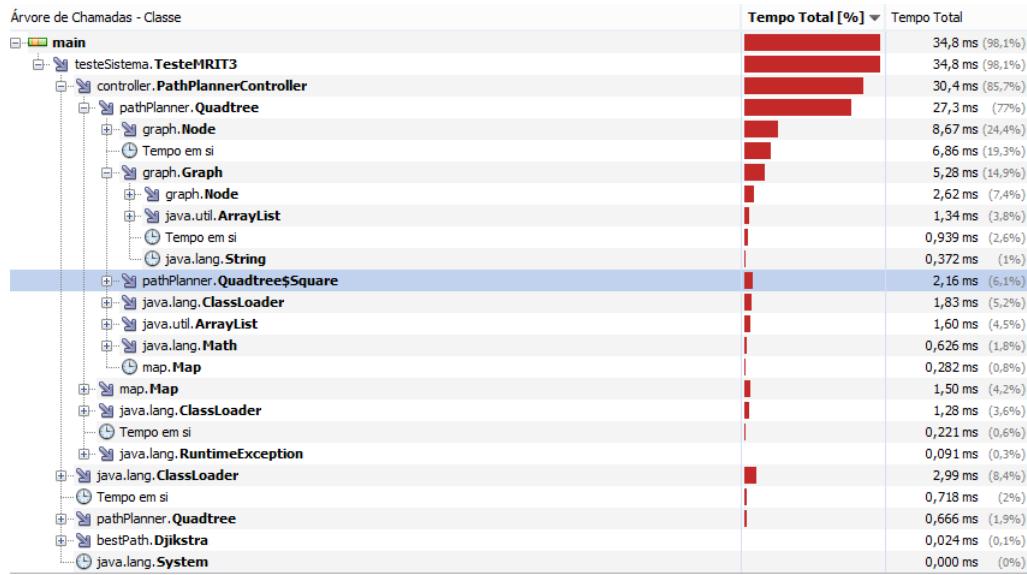


Figura 47 – tempo de processamento do mapa 3 com Quadtree

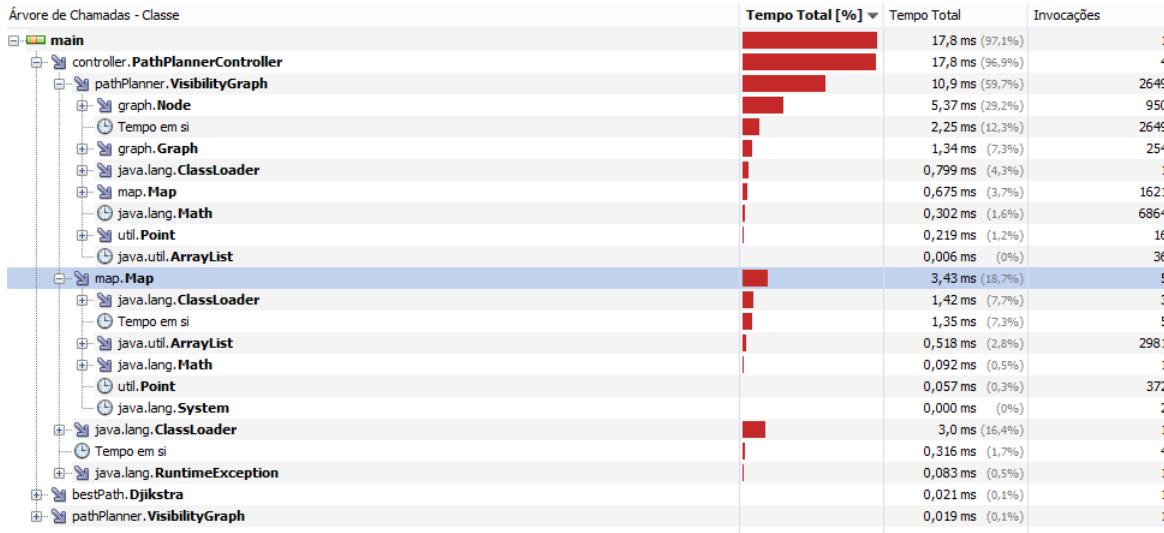


Figura 48 – tempo de processamento do mapa 3 com Grafo de Visibilidade

6.4 Testes Finais Com o Robô

Com o código-fonte do Traveller concluída e de posse dos resultados das simulações, foi realizado um teste final com o robô, verificando os quatro algoritmos. Foi inserido no robô o módulo EmbeddedCommunication e executado a classe concreta NXTCommunication. O mapa inserido na memória foi o mapa 1. Como servidor foi usado um notebook comum, rodando o código do Traveller e com o Bluecove inserido para a comunicação bluetooth.

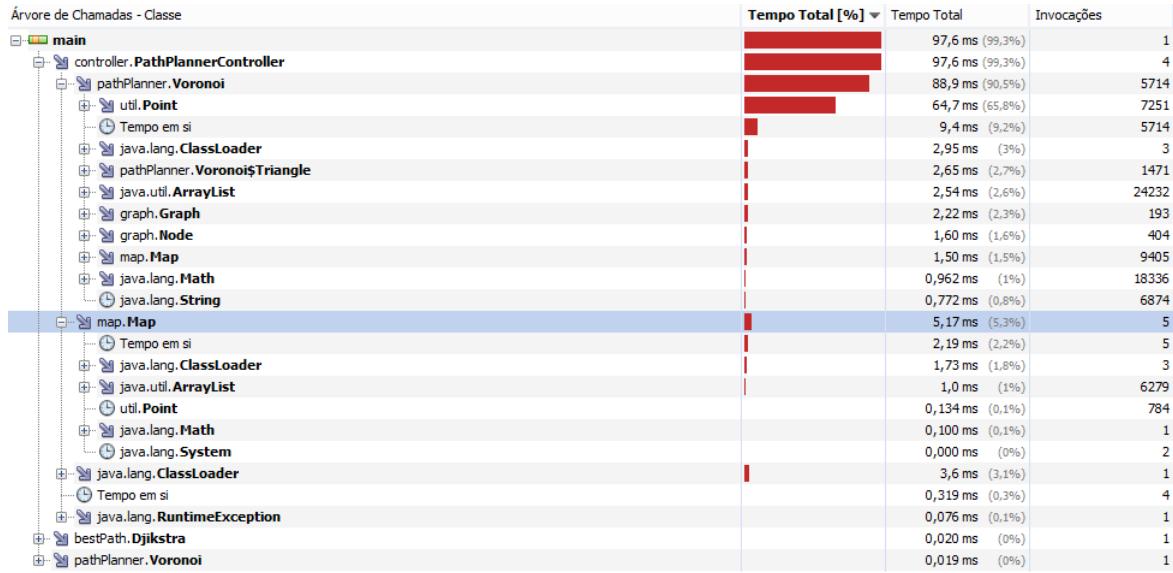


Figura 49 – tempo de processamento do mapa 3 com Voronoi

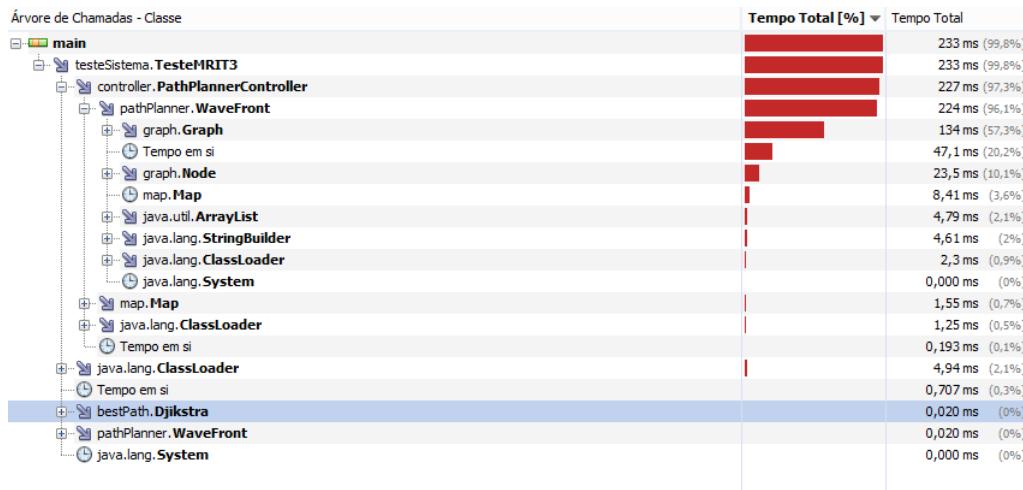


Figura 50 – tempo de processamento do mapa 3 com Wavefront

O robô executou toda a comunicação de forma correta, enviando e recebendo os dados sem perdas nem alterações. O robô recebeu a lista de pontos e executou a movimentação entre os mesmos para os quatro algoritmos.

Por fim, foi medido o tempo da comunicação entre o robô e o servidor. Para tanto, foi utilizado a biblioteca do próprio Java implementado no robô. Foram medidos dois tempos: o tempo para abrir a comunicação entre o robô e o servidor e o tempo para enviar os dados, o Traveller executar todo o fluxo de trabalho e enviar a resposta de volta para o robô. Para cada algoritmo foi medido três vezes estes dois valores.

A abertura da comunicação entre as partes, em todas as 12 medidas, levou entre 4.6 segundos e 7.8 segundos. A média de tempo foi de 5.5 segundos. Para os algoritmos, o tempo de processamento real foi de: i) Quadtree teve média de 4.981 segundos ii)

Wavefront teve média de 7.078 segundos iii) Grafo de Visibilidade teve média de 4.389 segundos e iv) Voronoi teve média de 4.393 segundos.

Os tempos reais foram bastante discrepantes com os valores simulados, porém uma proporção se manteve. Em ambos os casos, o algoritmo Wavefront levou muito mais tempo que os demais. Enquanto os outros três algoritmos levaram entre 4 e 5 segundos, o Wavefront levou, em todas as medições, pouco mais de 7 segundos. O Quadtree consumiu um pouco de tempo a mais que o Voronoi e o Grafo de Visibilidade, sendo o único dos três a passar de 5 segundos em alguma medida e ficando com uma média de mais de meio segundo acima dos outros dois. Não houve uma diferença considerável entre o Voronoi e o Grafo de Visibilidade. Os valores das medidas podem ser vistos na Tabela 7.

Tabela 7 – Tempos reais gasto pelos algoritmos

Algoritmo	Medida 1	Medida 2	Medida 3
Grafo de Visibilidade	4,965s	5,022s	4,956s
Voronoi	7,011s	7,216s	7,008s
Quadtree	4,462s	4,366s	4,340s
Wavefront	4,393s	4,365s	4,422s

6.5 Observações Sobre o Funcionamento dos Algoritmos

Uma observação importante a se fazer é em relação ao mapa 6. Ele foi desenhado propositalmente para que nenhum algoritmo conseguisse gerar um percurso nele. O ponto inicial foi cercado por obstáculos que, após expandidos, deixariam um espaço menor que a largura do robô e, assim, impedindo a geração da trajetória. No simulador MRIT, o Grafo de Visibilidade conseguiu gerar uma solução mesmo com a distância entre os obstáculos sendo de uma célula e o robô ocupando duas.

No caso do Traveller, tanto o Grafo de Visibilidade quanto o Wavefront geraram uma solução. Este mapa permite avaliar algumas características importantes sobre os quatro algoritmos.

O Quadtree e o Voronoi não são capazes de gerar uma solução em um ambiente como o do mapa 6 por trabalharem com divisão espacial. Ambos os algoritmos dividem o mapa em regiões e as conecta, o que os torna inviáveis em ambientes muito apertados. Em um cenário como o do mapa 6, os algoritmos não são capazes de criar uma região tão pequena e acabam por desconsiderar corredores estreitos.

Já os algoritmos Grafo de Visibilidade e Wavefront trabalham de forma linear. Ambos os algoritmos geram uma linha reta entre dois pontos quaisquer e, se essa linha reta for toda livre de obstáculos, então o robô poderá trafegar por ela.

No caso do Traveller, em específico, o Grafo de Visibilidade expande os obstáculos obrigatoriamente. Assim, mesmo que haja um corredor de apenas uma célula entre dois obstáculos expandidos, o algoritmo garantiu uma área de segurança de metade da largura de robô para cada lado. Desse modo, existe na verdade um percurso igual a largura do robô mais uma célula entre os obstáculos. O robô irá invadir a área de risco, porém não colidirá com os obstáculos e seu centro passará no centro entre os obstáculos.

Quanto ao Wavefront, este também considera que, se há ao menos uma célula livre entre os obstáculos expandidos, então há espaço suficiente para o robô se locomover entre eles. Porém, diferente do Grafo de Visibilidade, que força a expansão, o Wavefront não garante que essa expansão realmente ocorra. Se os obstáculos não forem expandidos e houver um percurso menor que a largura do robô, o algoritmo tentará passar por este corredor e a colisão ocorrerá.

6.6 Resumo do Capítulo

Os primeiros testes realizados visavam testar a capacidade de memória do kit utilizado, criando mapas grandes e com muitos obstáculos. Foi então diminuídos esses valores gradativamente até que o robô fosse capaz de armazená-los. Com esses testes foi verificado que mesmo com um mapa não muito grande e com poucos obstáculos a memória não suportava esse volume de dados. Portanto, foi definido que o núcleo do sistema seria migrado para um servidor e comunicado com o robô via *bluetooth*.

Foram realizados testes unitários e de integração do *framework* desenvolvido, visando garantir a qualidade do código. Cada algoritmo desenvolvido tinha um conjunto de classes de teste que verificavam o correto funcionamento do mesmo. Métodos mais complexos da classe Map também foram testados diretamente por testes unitários.

Os testes de integração realizados envolveram seis ambientes diferentes, modelados e simulados no simulador MRIT. Foi medido o percurso para cada algoritmo em cada um dos seis mapas, tanto no simulador quanto no *framework*. O Grafo de Visibilidade teve os mesmos resultados ou muito próximos, o Quadtree teve resultados um pouco maiores e menos retilíneos, o Wavefront teve resultados consideravelmente menores e o Voronoi conseguiu solucionar apenas um dos mapas.

Foram realizadas simulações do processamento do Traveller, visando mensurar seu tempo de processamento. Porém, quando comparado aos valores reais medidos ao implantá-lo no robô, foram bastante discrepantes e apenas uma das proporções foi mantida.

7 Conclusão

Neste trabalho foram estudados métodos de definição de trajetória e como estes métodos poderiam ser desacoplados dos detalhes físicos do robô em que são implementados. Foram estudados também os conceitos e técnicas para a construção de *frameworks* e a construção de uma arquitetura de software facilmente comprehensível e flexível. Com estas duas pesquisas, foi desenvolvido um *framework* para a definição de trajetórias para robôs, buscando gerar um sistema o mais independente da plataforma ou kit construído. Para a simplificação do escopo e permitir o desacoplamento entre hardware e software, o *framework* trabalha apenas com algoritmos globais.

O desenvolvimento foi todo baseado na implementação de *hot-spots*, que permitem a fácil troca entre os algoritmos e a mudança de comportamento sem impactar no restante do código. Foram levantados quatro *hot-spots* e desenvolvidos três deles.

O *framework* implementado funciona em um servidor remoto e se comunica com o robô via *bluetooth*, recebendo o mapa da região, os pontos inicial, final e outras variáveis de controle e é devolvida à máquina uma lista de coordenadas que deve ser seguida para chegar ao destino sem colidir com obstáculos. O usuário pode ainda escolher entre quatro algoritmos diferentes, além de poder desenvolver seus próprios algoritmos e usar a plataforma para o controle do fluxo de dados e validação dos mesmos.

Para o desenvolvimento desta aplicação, foi seguido a visão da engenharia de software sobre a produção de sistemas complexos e como mantê-los simples, facilitando a compreensão e manutenção, além de ser projetado para a reutilização. Seguindo a visão da engenharia de software, foram realizados testes sobre o sistema durante o seu desenvolvimento, buscando garantir a qualidade do sistema final.

Visando a continuidade do sistema, o mesmo será mantido com licença aberta, para o livre uso desde que referenciada a fonte. O *link* para o código-fonte encontra-se no apêndice B.

7.1 Sugestão de Trabalhos Futuros

Imaginando a perpetuidade do estudo e a constante melhoria do *framework*, algumas sugestões de melhorias são indicadas para os interessados:

1. Evoluir a implementação do algoritmo de Voronoi e melhorar a implementação do Quadtree para que se compare aos resultados retornados pelo simulador MRIT;

2. Desenvolver outros algoritmos de definição de trajetória e melhor caminho, ou mesmo versões otimizadas dos algoritmos já implementados, como os referenciados neste trabalho;
3. Realizar uma seleção de pontos da lista de coordenadas retornadas, descartando pontos intermediários aonde a orientação do robô não é alterada;
4. Implementar um algoritmo para suavização da trajetória; e,
5. Desenvolver um solução para que o robô possa enviar um mapa de qualquer tipo (matricial, topológico e geométrico) para o servidor e o mesmo consiga re-estruturá-lo de forma correta.

Referências

- ABREU, P. *robótica industrial*. Portugal: [s.n.], 2001. Visualizado em 23/09/2014. Disponível em: <http://paginas.fe.up.pt/~aml/maic_files/introd.pdf>. Citado 2 vezes nas páginas 19 e 20.
- BAGNALL, B. *Intelligence Unleashed, Creating LEGO NXT Robots with Java*. 1. ed. EUA: Variant Express, 2011. 519 p. Citado 2 vezes nas páginas 22 e 26.
- BORENSTEIN, J.; EVERETT, H. R.; FENG, L. *Where i am? Sensors and Methods for Mobile Robot Positioning*. 1. ed. Michigan, EUA: Universidade de Michigan, 1996. 282 p. Citado 4 vezes nas páginas 21, 22, 23 e 28.
- CHOSET, H. et al. *Principles of Robot Motion. Theory, Algorithms, and Implementations*. 1. ed. Massachusetts, EUA: The MIT Press, 2005. 626 p. Citado 6 vezes nas páginas 24, 28, 29, 31, 32 e 73.
- FAYAD, M. E.; JOHNSON, R. E.; SHMIDT, D. C. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. 1. ed. EUA: Wiley Computer Publishing, 1999. 664 p. Citado 12 vezes nas páginas 9, 35, 36, 39, 40, 41, 42, 43, 44, 53, 70 e 103.
- Ferguson , D.; STENTZ, A. T. *The Field D* Algorithm for Improved Path Planning and Replanning in Uniform and Non-Uniform Cost Environments*. Pittsburgh, PA, 2005. Citado na página 70.
- GAMMA, E. et al. *Padrões de projeto*. 1. ed. Brasil: Bookman, 1995. 370 p. Citado 2 vezes nas páginas 38 e 39.
- GOODLIFFE, P. *Code Craft*. 1. ed. San Francisco, EUA: No Starch Press, 2007. 580 p. Citado 4 vezes nas páginas 16, 38, 73 e 74.
- GUZMÁN, J. L. et al. Interactive tool for mobile robot motion planning. *Robotics and Autonomous Systems*, Elsevier, n. 56, p. 396–409, 2008. Citado 11 vezes nas páginas 16, 24, 25, 28, 30, 31, 32, 49, 53, 72 e 73.
- HOLDGAARD-THOMSEN, J. *Path Planning of Robots*. 90 p. Dissertação (Mestrado) — Technical University of Denmark, Dinamarca, 2010. Visualizado em 23/10/2014. Disponível em: <http://etd.dtu.dk/thesis/258988/ep10_05.pdf>. Citado 9 vezes nas páginas 9, 24, 31, 32, 33, 45, 53, 72 e 73.
- http://sicmscj.com.br/links.imagem_de_braço_robótico. 2014. Visualizado em 01/10/2014. Disponível em: <<http://sicmscj.com.br/links/roca1024.jpg>>. Citado 2 vezes nas páginas 9 e 20.
- <http://www.enigmaindustries.com/configurations.htm>. *exemplos de estruturas de direção diferencial*. 2014. Visualizado em 09/10/2014. Disponível em: <<http://www.enigmaindustries.com/configurations.htm>>. Citado 2 vezes nas páginas 9 e 51.
- LARMAN, C. *Utilizando UML e Padrões*. 3. ed. Brasil: Bookman, 2005. 695 p. Citado 7 vezes nas páginas 9, 16, 35, 36, 37, 39 e 65.

MATARIC, M. J. *The Robotics Primer*. 1. ed. Cambridge, EUA: Massachusetts Institute of Technology, 2007. 323 p. Citado 8 vezes nas páginas 9, 19, 22, 24, 25, 27, 28 e 50.

MCCONNEL, S. *Code Complete*. 2. ed. Washington, EUA: Microsoft Press, 2004. 914 p. Citado 4 vezes nas páginas 16, 38, 73 e 74.

MEDEIROS, F. L. L.; SILVA, J. D. S. da; YANASSE, H. H. Simplificação de grafos de visibilidade construídos através de modelos digitais de elevação. In: . Instituto Nacional de Pesquisas Espaciais, 2011. p. 4. Visualizado em 15/09/2014. Disponível em: <http://mtc-m18.sid.inpe.br/col/sid.inpe.br/mtc-m18/2011/10.18.01.32/doc/worcap2011_submission_13.pdf?ibiurl.language=en>. Citado 2 vezes nas páginas 32 e 69.

NEHMZOW, U. *Mobile Robotics, A Pratical Introduction*. 2. ed. Inglaterra: Springer, 2003. 280 p. Citado 3 vezes nas páginas 9, 57 e 58.

PEREIRA, J. *Avaliação e correção do modelo cinemático de robôs móveis visando a redução de erros no seguimento de trajetórias*. 141 p. Dissertação (Mestrado) — Universidade Do Estado De Santa Catarina, Joinville, Brasil, 2003. Visualizado em 24/09/2014. Disponível em: <http://www.tede.udesc.br/tde_arquivos/8/TDE-2006-05-19T054914Z-190/Publico/Jonas%20Pereira.pdf>. Citado 2 vezes nas páginas 20 e 23.

SECCHI, H. A. *Uma Introdução a Robôs Móveis*. Argentina, 2008. 91 p. Visualizado em 08/10/2014. Disponível em: <http://www.obr.org.br/wp-content/uploads/2013/04/Uma_Introducao_aos_Robos_Moveis.pdf>. Citado 6 vezes nas páginas 9, 19, 21, 22, 24 e 28.

SIEGWART, R.; NOURBAKHSH, I. R. *Introduction to autonomous mobile robots*. 1. ed. Massachusetts, EUA: Massachusetts Institute of Technology, 2004. 336 p. Citado 13 vezes nas páginas 9, 20, 21, 23, 24, 28, 29, 31, 32, 33, 34, 72 e 73.

Site do Bluecove. *Bluecove Site*. 2015. Visualizado em 20/06/2015. Disponível em: <<http://bluecove.org/>>. Citado na página 49.

Site do Carmen. *Carmen site do framework*. 2014. Visualizado em 23/10/2014. Disponível em: <<http://carmen.sourceforge.net/intro.html>>. Citado na página 45.

Site do Eclemma. *Eclemma, plugin para cobertura de código*. 2014. Visualizado em 01/10/2014. Disponível em: <<http://www.eclemma.org/installation.html>>. Citado na página 49.

Site do Eclipse. *Eclipse, ferramenta de desenvolvimento na versão utilizada*. 2014. Visualizado em 01/10/2014. Disponível em: <<http://www.eclipse.org/downloads/packages/release/Indigo/SR2>>. Citado na página 49.

Site do JUnit. *plugin para ferramenta de teste sunitários*. 2014. Visualizado em 01/10/2014. Disponível em: <<http://junit.org/>>. Citado na página 49.

Site do Lejos. *Lejos, plataforma Java para robôs da Lego*. 2014. Visualizado em 01/10/2014. Disponível em: <<http://www.lejos.org/nxj.php>>. Citado na página 50.

Site do MRIT. *MRIT, ferramenta de simulação de algoritmos de definição de trajetória.* 2014. Visualizado em 01/10/2014. Disponível em: <<http://aer.ual.es/mrit/>>. Citado 6 vezes nas páginas 9, 10, 32, 35, 49 e 72.

Site do plugin NXT. *plugin para programação Java para o kit Lego NXT.* 2014. Visualizado em 01/10/2014. Disponível em: <<http://lejos.sourceforge.net/tools/eclipse/plugin/nxj>>. Citado na página 49.

Site do RIA. *Site do RIA.* 2015. Visualizado em 12/06/2015. Disponível em: <<http://www.robotics.org/company-profile-detail.cfm/Internal/Robotic-Industries-Association/company/319>>. Citado na página 19.

Site do ROS. *Site do framework ROS.* 2014. Visualizado em 23/10/2014. Disponível em: <<http://wiki.ros.org/ROS>>. Citado na página 46.

Site do Voronoi. *Voronoi site.* 2014. Visualizado em 29/10/2014. Disponível em: <<http://www.Voronoi.com>>. Citado na página 32.

SOUZA, S. C. B. de. *Planejamento de trajetória para um robô móvel com duas rodas utilizando um algoritmo A-estrela modificado.* 110 p. Dissertação (Mestrado) — Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil, 2008. Visualizado em 02/10/2014. Disponível em: <<http://www.pee.ufrj.br/teses/textocompleto/2008121701.pdf>>. Citado 10 vezes nas páginas 22, 23, 24, 25, 30, 31, 33, 53, 69 e 72.

STRANDBERG, M. *Robot Path Planning: An Object-Oriented Approach.* 254 p. Tese (Doutorado) — Royal Institute of Technology, Estocolmo, Suécia, 2004. Visualizado em 06/11/2014. Disponível em: <<http://www.diva-portal.org/smash/get/diva2:7803/FULLTEXT01.pdf>>. Citado 3 vezes nas páginas 25, 46 e 53.

SZYPERSKI, C. *Component Software.* 2. ed. New York, EUA: acm press, 2002. 589 p. Citado 5 vezes nas páginas 35, 37, 38, 39 e 44.

VIEIRA, F. C. *Controle Dinâmico de robôs móveis com acionamento diferencial.* 106 p. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte, Natal, Brasil, 2005. Visualizado em 11/10/2014. Disponível em: <<ftp://ftp.ufrn.br/pub/biblioteca/ext/bdtd/FredericoCV.pdf>>. Citado 5 vezes nas páginas 9, 20, 25, 26 e 27.

Apêndices

APÊNDICE A – *Host-spot Cards*

Nesse apêndice, serão apresentados os *hot-spot cards* de cada *hot-spot*, conforme apresentado por (FAYAD; JOHNSON; SHMIDT, 1999).

A.1 PathPlanner

Nome do hot-spot: Algoritmo de Path Plannig

Graus de flexibilidade especificados:

adaptação sem reinício: sim

adaptação pelo usuário final: não

Descrição geral

Definição da trajetória em um mapa já conhecido a partir de um algoritmo estabelecido. Recebe o mapa com os obstáculos e espaços livres, o algoritmo a ser executado e os pontos iniciais e finais e retorna um grafo com os caminhos possíveis

Exemplo 1

Recebe-se o seguinte mapa, aonde a célula preta significa ocupado, branca significa livre e os pontos iniciais e finais marcados.

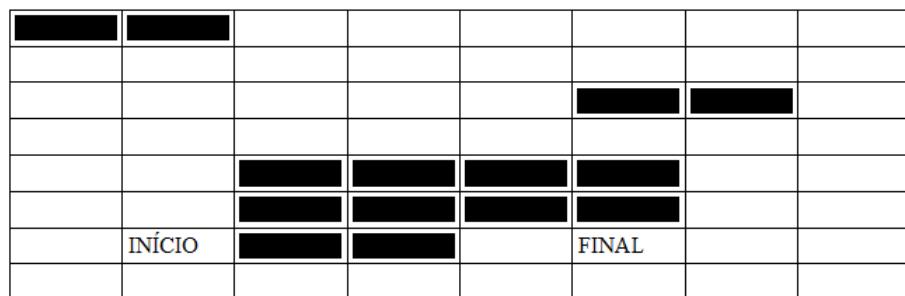


Figura 51 – mapa exemplo do *hot-spot card* PathPlanner

Para o algorítimo Quadtree será gerado o grafo apresentado na Figura 52.

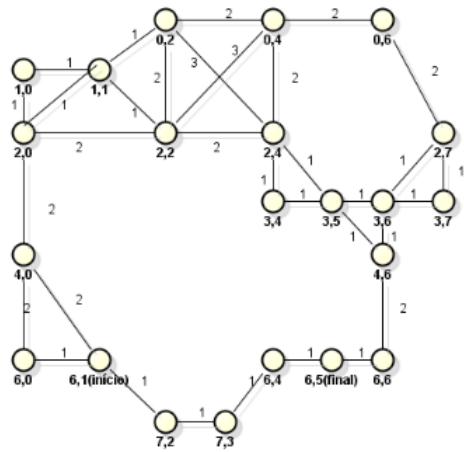


Figura 52 – resultado do exemplo 1 do *hot-spot card* PathPlanner

Exemplo 2

Recebe-se o seguinte mapa, aonde a célula preta significa ocupado, branca significa livre e os pontos iniciais e finais marcados.

		p		p			p
	p			p		p	p
	INÍCIO				FINAL		
	p			p			

Figura 53 – mapa exemplo 2 do *hot-spot card* PathPlanner

Para o algorítimo Grafo de Visibilidade gerará o grafo:

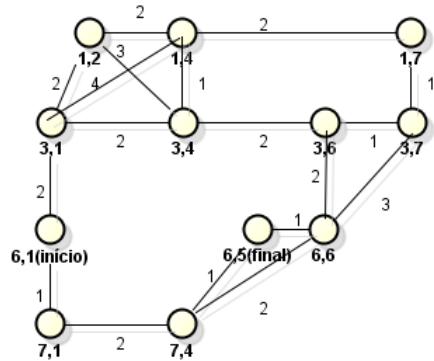


Figura 54 – resultado do exemplo 2 do *hot-spot card* PathPlanner

A.2 BestPath

Nome do hot-spot: Algoritmo de Menor caminho

Graus de flexibilidade especificados:

adaptação sem reinício: sim

adaptação pelo usuário final: não

Descrição geral

Definição do menor caminho entre um nó e outro em um grafo, retornando a lista de nós com menor custo possível. O grafo e os nós de início e fim são recebidos via construtor e é retornada uma lista de nós.

Exemplo 1

Recebe-se o grafo abaixo, aonde o inicial é “6,1” e o final é “6,5”.

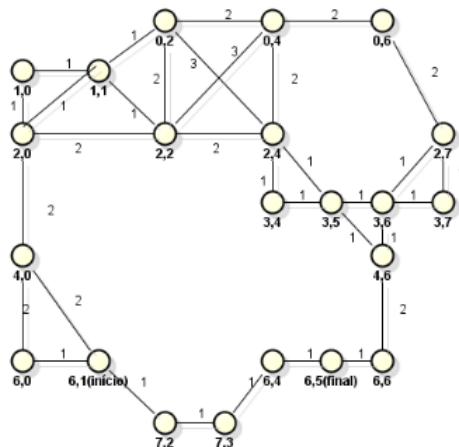


Figura 55 – grafo exemplo do *hot-spot card* BestPath

Para o algoritmo Djikstra, será gerado o grafo: 6,1 -> 7,2 -> 7,3 -> 6,4 -> 6,5.

Exemplo 2

Recebe-se o grafo abaixo, aonde o inicial é “6,1” e o final é “6,5”.

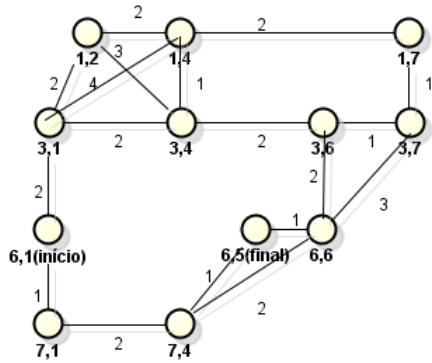


Figura 56 – grafo do exemplo 2 do *hot-spot card* BestPath

Para o algorítimo A*, será gerado o grafo: 6,1 -> 7,1 -> 7,4 -> 6,6.

A.3 EmbeddedCommunicator

Nome do hot-spot: Embedded Communication

Graus de flexibilidade especificados:

adaptação sem reinício: não

adaptação pelo usuário final: não

Descrição geral

Implementa a comunicação via *bluetooth* em hardwares diferentes, utilizando as bibliotecas disponíveis para cada kit.

APÊNDICE B – Código fonte

O código fonte encontra-se disponível no seguinte endereço de repositório:

<https://github.com/rodrigorincon/path-planner-framework/tree/master/codigo/src>

Colaborando com a filosofia *OpenSource*, o *framework* pode ser utilizado por terceiros, sendo apenas requisitada a menção desse TCC como fonte inicial dos primeiros esforços.

Nesse caso, segue a referência ao TCC:

Rincon, Rodrigo Lopes. "Traveller: Um Framework de Definição de Trajetórias para Robôs Móveis". Trabalho de Conclusão de Curso. Universidade de Brasília (UnB), Faculdade do Gama (FGA), Curso de Engenharia de Software, Maurício Serrano (prof. orientador) e Milene Serrano (profa. coorientadora). Junho de 2014 a Julho de 2015.