UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

NAVEGAÇÃO EXPLORATÓRIA E MAPEAMENTO EM UM SISTEMA MULTIAGENTE ROBÓTICO

JOÃO PAULO GONÇALVES

BLUMENAU
2011

2011/1-21

JOÃO PAULO GONÇALVES

NAVEGAÇÃO EXPLORATÓRIA E MAPEAMENTO EM UM SISTEMA MULTIAGENTE ROBÓTICO

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Fernando dos Santos, Mestre – Orientador

BLUMENAU
2011

2011/1-21

NAVEGAÇÃO EXPLORATÓRIA E MAPEAMENTO EM UM SISTEMA MULTIAGENTE ROBÓTICO

Por

JOÃO PAULO GONÇALVES

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Fernando dos Santos Orientador, Mestre – Orientador, FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB
Membro:	Prof. Alex Sandro da Silva, Mestre – FURB

Blumenau, 29 de junho de 2011

Dedico este trabalho a todas as gerações humanas anteriores, cujo sacrifício me proporcionou a capacidade de adicionar uma pequena pitada de sal no grande caldeirão de conhecimento humano.

AGRADECIMENTOS

Primeiramente, devo agradecer a minha família, que me suportou por mais tempo que eu diria imaginável.

A minha namorada Denise, que com a perseverante frase “E então, está pronto?” torturou-me até a conclusão. Duvido que este trabalho teria sido completado sem a cobrança dela.

Ao meu orientador, com o apoio que foi fundamental para o sucesso deste trabalho.

Ao coordenador do curso de Ciência da Computação, José Roque, que nas palavras dele: “olha, mas felizmente isso nunca aconteceu...” foram de grande valia para evitar problemas.

Ao monitor da matéria de Robótica, Matheus Luan Krueger, que sempre se mostrou disposto em fornecer a sua ajuda com os robôs.

A empresa onde trabalho, a Senior Sistemas, cuja experiência adquirida e o tempo disponibilizado por ela me foi de alto valor na confecção do trabalho, e onde eu aprendi a sempre buscar a melhor implementação dentro do tangível.

Aos meus amigos e colegas, cujas personalidades, ações e opiniões ajudaram a formar aquele quem eu sou.

E por fim, meus mais sinceros agradecimentos a algumas estrelas anônimas, que há milhões ou bilhões de anos explodiram e possibilitaram a existência de toda a raça humana, e por conseqüência, a formação do meu ser com a poeira das estrelas.

Ao infinito e além!

Buzz Lightyear

RESUMO

Este trabalho apresenta a implementação de um sistema multiagente para realização de navegação exploratória e mapeamento utilizando robôs LEGO Mindstorms. Este sistema através de comunicação sem fio coordena um conjunto de robôs para exploração de um ambiente desconhecido. Os robôs se movem utilizando campos de força e representam áreas mapeadas como polígonos. Também é disponibilizada uma interface com usuário, para visualização do mapeamento.

Palavras-chave: Robótica. Inteligência artificial. Navegação exploratória. Campos de força.

ABSTRACT

This work presents the implementation of a multi-agent system to conduct exploratory navigation and mapping using LEGO Mindstorms robots. The system coordinates a set of robots using wireless communication. The robots does explore an unknown environment moving using force fields and they represent mapped areas as polygons. An user interface is also available to visualize the enviroment mapped.

Key-words: Robotics. Artificial intelligence. Exploratory navigation. Potential fields.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

A robótica evoluiu rapidamente nas últimas décadas, e desde o seu surgimento como braços robóticos, ela nunca esteve tão presente e acessível no mundo, quanto atualmente. Impulsionada pela miniaturização dos componentes eletrônicos, a robótica começa a concretizar os sonhos de escritores de ficção-cientifica. Hoje já existem veículos voadores capazes de decolar e pousar autonomamente em navios, assim como realizar ataques bélicos, pesquisas, reconhecimento, desarme de minas e bombas. Esse grande crescimento da robótica, fez com que surgissem novas demandas para a área de Inteligência Artificial (IA). Sem a IA não seriam possíveis muitas das missões realizadas no espaço. A tele-operação não é uma escolha na exploração do espaço quando o tempo de comunicação com o robô leva dezenas de minutos, ou simplesmente não é possível, como no lado não visível da Lua(MURPHY, 2000, p. 31).

Essa relação entre a robótica e a IA atualmente é uma via de mão dupla. Com a alta acessibilidade e o aumento computacional dos robôs foi possível para a IA testar as abordagens que inicialmente ficavam confinadas em simulações. Isso permitiu a definição de estratégias melhores, que consideram a incerteza e os problemas que podem, e provavelmente irão, ocorrer no mundo real. Entre os muitos assuntos beneficiados com isso, está o de exploração e mapeamento de ambientes desconhecidos. Por exemplo, um robô que desarme minas tem que ser capaz de uma vez iniciado o seu funcionamento, mapear a região onde está localizado, encontrando e desarmando as minas espalhadas pelo ambiente, tudo isso evitando obstáculos e no menor tempo possível.

O mapeamento de ambientes desconhecidos também obtém ganhos na utilização de múltiplos robôs em relação à utilização de somente um robô. Segundo Dudek et al.(1996, p. 375), vários robôs podem ser mais simples fisicamente que um maior e mais complexo robô, tornando o sistema mais econômico, escalável e menos suscetível a falhas globais.

Diante do exposto, este trabalho implementa uma abordagem para a exploração e mapeamento de ambientes desconhecidos utilizando vários robôs físicos, formando um sistema multiagente robótico. Os robôs são construídos utilizando o *kit* LEGO Mindstorms. A abordagem utiliza-se de polígonos para representação do ambiente mapeado, usando o algoritmo de Bentley-Ottmann para a rápida detecção de intersecções entre os vários segmentos de reta. Para realizar a navegação dos robôs, escolheu-se campos de força por sua simplicidade e baixo requerimento em relação às capacidades computacionais do robô. A

partir de experimentos realizados, verificou-se que a abordagem proposta é capaz de realizar a exploração e o mapeamento.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é realizar o mapeamento de ambientes desconhecidos com barreiras.

Os objetivos específicos do trabalho são:

a) disponibilizar robôs capazes de se movimentar e realizar o mapeamento do ambiente;
b) disponibilizar um aplicativo para visualizar em um computador pessoal, o mapeamento realizado pelos robôs;
c) disponibilizar a comunicação entre os robôs e o aplicativo de visualização do mapeamento.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está dividido em quatro capítulos, sendo que o segundo capítulo apresenta a fundamentação teórica com os conceitos de sistemas multiagente, robótica, exploração e mapeamento. Além disso, o capítulo também expõe detalhes de trabalhos correlatos.

O terceiro capítulo trata do desenvolvimento do sistema multiagente, iniciando com os requisitos e especificação da aplicação. Fazem parte desta especificação os diagramas de caso de uso, de classes, de atividades e de sequência. Ainda no terceiro capítulos são comentados os resultados e os problemas encontrados durante a implementação do sistema.

Por fim, no quarto capítulo são apresentadas as conclusões finais sobre o trabalho e sugestões para extensões.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta inicialmente o conceito de sistema multiagente. Em seguida, o conceito de robótica é abordado, seguido pela apresentação do LEGO Mindstorms. Técnicas para realizar navegação exploratória e mapeamento são apresentadas em seguida. Por fim, em trabalhos correlatos, são mencionados quatro trabalhos com assuntos relacionados a este trabalho.

2.1 SISTEMAS MULTIAGENTE

Segundo Russel e Norvig(2004, p. 33), um agente é qualquer coisa que possa perceber o ambiente através de sensores e atuar neste ambiente através de atuadores. Por exemplo, tem-se o agente humano que utiliza olhos e ouvidos (sensores) para perceber o ambiente e mãos, braços e pernas (atuadores) para atuar nele. Há também o agente de software que recebe entradas de teclado, arquivos e pacotes da rede como percepção e atua no ambiente apresentando algo em uma tela, escrevendo arquivos, etc.. Wooldridge (2000, p. 28-29) afirma que há consenso que um agente deve ser autônomo, definindo um agente como: "[...] um sistema de computador que é situado em algum ambiente, e é capaz de ações autônomas neste ambiente a fim de cumprir os seus objetivos".

As propriedades do ambiente no qual o agente atuará também afetam na complexidade deste agente(WOOLDRIDGE, 2000, p. 30). Russel e Norvig(2004, p. 41-44) definem uma série de características para os ambientes, entre as quais: completamente observável *versus* parcialmente observável, onde se verifica a capacidade de percepção do agente em relação ao tempo; determinístico *versus* estocásticos, sendo determinístico o ambiente em que o próximo estado é definido unicamente em função do estado e ação atuais do agente; estático *versus* dinâmico, onde se o ambiente puder se alterar enquanto o agente está decidindo, então ele é dinâmico.

Um sistema multiagente é uma rede fracamente acoplada de agentes que trabalham juntos, interagindo, por exemplo, através de troca de mensagens, para resolver problemas que estão além da capacidade individual ou conhecimento de cada um dos agentes (JENNINGS; SYCARA; WOOLDRIDGE, 1998, p. 285). Os sistemas multiagente têm como

características:

a) informações distribuídas: as informações sobre o problema estão distribuídas entre os agentes;
b) percepção limitada: como as informações do sistema multiagente estão distribuídas, a percepção de cada agente é limitada. Isso força os agentes a interagirem buscando compartilhar informações;
c) controle do sistema distribuído: os agentes devem interagir para determinar o seu comportamento. Não há uma entidade central com controle total do sistema para determinar o que cada agente deve realizar;
d) processamento assíncrono: cada agente executa independente dos outros. Se o sistema multiagente requerer sincronismo, os agentes devem se coordenar para obtê-lo.

2.2 ROBÓTICA

Robótica é o estudo da construção de máquinas para substituir os seres humanos na execução de alguma tarefa, tanto fisicamente (ações), quanto intelectualmente (decisões). Para Russel e Norvig(2004, p. 870) "os robôs são os agentes físicos que executam tarefas manipulando o mundo físico".

Muitos robôs não apresentam comportamento inteligente, como por exemplo, braços robóticos utilizados em linhas de montagem. A ênfase deste tipo de robô está na repetição e precisão (MURPHY, 2000, p. 21). Um robô deste tipo não foi preparado para tratar de situações inesperadas.

De acordo com Murphy (2000, p. 3), um robô com comportamento inteligente é uma criatura mecânica que funciona autonomamente. Isso vem de encontro com a definição de agente, tornando-se assim, o robô com comportamento inteligente um exemplo de agente de sistemas multiagente. Isso possibilita a construção de um sistema multiagente robótico.

A utilização de robôs inteligentes deve-se a três motivos. O primeiro é quando a tarefa a ser realizada traz um grande risco para um humano, podendo ser citados a manipulação de objetos altamente radioativos ou tóxicos e a exploração espacial. O segundo motivo é quando o uso de humanos é economicamente ineficiente, como nas linhas de montagens. O último motivo é o quando o uso de humanos traz riscos desnecessários, como por exemplo, a limpeza

de áreas minadas.

2.2.1 LEGO Mindstorms

LEGO Mindstorms é um *Robotics Invention System* (RIS), ou seja, conjunto para construção de robôs, fabricado pela LEGO Group. Encontra-se na terceira versão, nomeada LEGO Mindstorms NXT 2.0 (LEGO GROUP, 2010).

Nesta versão, o conjunto é composto por 619 peças, incluindo três servo motores elétricos e quatro sensores. Os quatro sensores disponíveis são: um sensor de cor e luz, capaz de distinguir entre oito cores; dois sensores de toque e um sensor ultrassônico, para medição de distâncias e detecção de presença.

Como unidade de processamento, existe o bloco programável denominado NXT. Este bloco possui um processador Atmel 32-bit ARM[1], rodando a 48 MHz, podendo acessar diretamente 64 KB[2] de RAM[3] e 256 KB de memória flash. Existem sete portas de comunicação, com quatro de entrada para os dados dos sensores e três de saída para comando dos motores. A comunicação entre os motores e sensores para com o bloco programável é feito através de cabos com conectores RJ12, semelhantes com os conectores utilizados em aparelhos telefônicos. Além dessas entradas, há uma entrada *Universal Serial Bus* (USB) 2.0, por onde se pode realizar a transferência de código e informações de um computador para o NXT. O NXT também possui comunicação através da tecnologia de redes pessoais sem fio *Bluetooth*, capaz de se comunicar com computadores ou outros dispositivos NXT. No bloco, é disponibilizado um pequeno *Liquid Crystal Display* (LCD) monocromático e 4 botões para manipulação. O conjunto todo é alimentado por seis pilhas AA, sendo possível utilizar baterias recarregáveis.

Junto ao *kit* de peças é também fornecido um software para programação do NXT, criado pela própria LEGO Group. Este software é adequado para programação básica e voltado basicamente para crianças ou iniciantes na programação. Entretanto, existem numerosas adaptações de variadas linguagens de programação para o NXT. Entre elas, tem se a LeJOS NXJ (LEJOS, 2010), uma pequena máquina virtual Java. Além disso, a biblioteca do

---

[1] *Advanced RISC Machine* (ARM).

[2] *Kilobyte.* Equivalente a 1024 bytes.

[3] *Random Access Memory*. Memória de acesso aleatório que permite a leitura e escrita, sendo utilizada como memória principal.

LeJOS fornece todas as classes necessárias para programação do NXT, incluindo-se as características da linguagem Java como, por exemplo, segmentos de execução, vetores multidimensionais, recursão, sincronização e exceções.

A Figura 1 apresenta o bloco programável NXT (centro da imagem), com os botões e tela LCD visíveis na parte superior, os motores conectados ao bloco (parte superior da imagem) e os sensores disponíveis, também conectados ao bloco (parte inferior da imagem).

Fonte: LEGO Group (2010a).

Figura 1 - Bloco programável NXT conectado a todos os motores e sensores

2.3 NAVEGAÇÃO EXPLORATÓRIA E MAPEAMENTO

Exploração é a descoberta de informações relevantes a partir do ambiente parcial ou completamente desconhecido (ZLOT et al., 2002, p. 3016). Bugard et al. (2000, p. 476) afirmam que a exploração de um ambiente pertence aos problemas fundamentais dos robôs móveis. A navegação busca direcionar o curso de um robô móvel quando ele se desloca em um ambiente (MCKERROW, 1995 apud SILVA JÚNIOR, 2003, p. 18). Logo a navegação exploratória busca definir a direção ou curso do robô visando minimizar o tempo necessário para realizar a exploração para completar o mapeamento (BURGARD et al., 2000, p. 476).

O mapeamento tem como objetivo a construção de um mapa, que é a representação do ambiente. Segundo Murphy(2000, p. 321) o mapa para o robô é chamado de memória espacial, e deve ser capaz de quatro funções: atenção, que é a capacidade de determinar qual é

o próximo marco a ser percebido; raciocínio, a capacidade de determinar se o ambiente suporta o robô; planejamento de caminho, para descobrir quais os caminhos mais curtos; e por fim, coleta de informações, para determinar se o ambiente sofreu modificações. Silva Júnior (2003, p. 26) diz que "Um robô capaz de realizar o mapeamento de um ambiente desconhecido e se auto localizar a partir uma versão parcial do mapa é capaz de construir um mapa representativo e confiável de qualquer ambiente somente a partir de observações relativas ao ambiente". Para realizar o mapeamento, os robôs móveis devem possuir a habilidade de explorá-lo eficazmente (BURGARD et al., 2000, p. 476).

Uma das estratégias de navegação disponível na literatura é a de campos de força. Nesta estratégia os obstáculos descobertos pelos robôs determinam a movimentação. A área mapeada é representada através de um polígono. Cada vez que um robô explora uma nova área, o polígono que representa a nova área é unido com o polígono que representa a área anteriormente mapeada. A seguir, é apresentada a técnica de campos de força, e também um algoritmo para determinar as intersecções entre segmentos de reta para realização da união dos polígonos.

2.3.1 Campos de Força

Campos de força é uma arquitetura reativa, onde vários vetores representam distintos comportamentos do robô, como por exemplo, o comportamento de desviar de obstáculos e o comportamento de atingir um ponto objetivo (MURPHY, 2000, p. 122-123). A soma dos vetores combina os diferentes comportamentos para determinar o comportamento resultante.

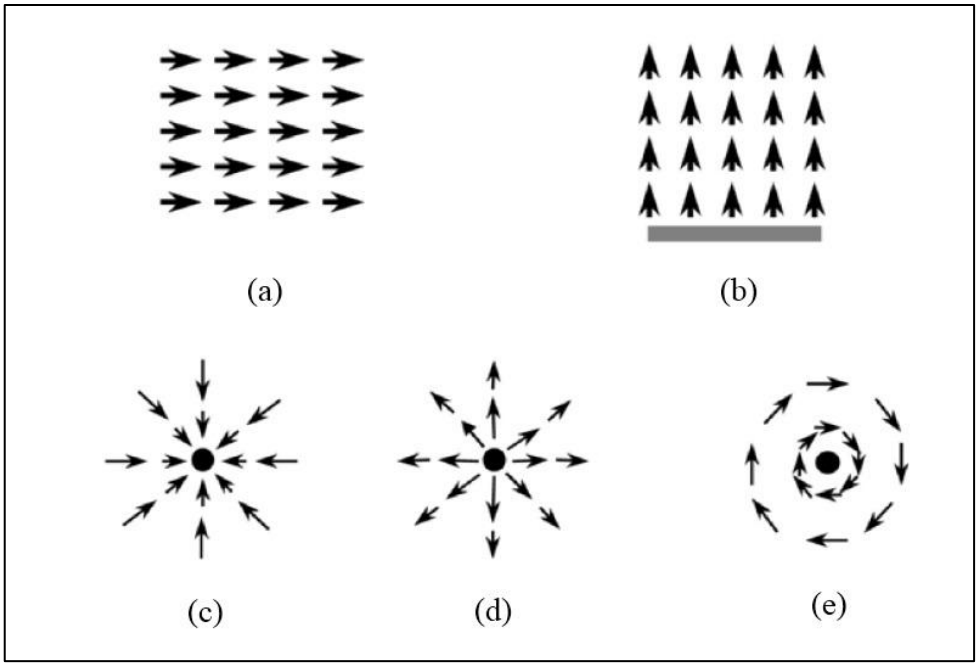
Segundo Murphy (2000, p. 123), um vetor é uma representação de uma força, onde sua construção matemática é formada pela magnitude (intensidade da força) *m* e a direção (ângulo) *d*, escritas como uma tupla *(m, d)* e representados por uma letra em maiúsculo, normalmente *V*. Por convenção, a magnitude é comumente um número real entre 0,0 e 1,0. O valor de magnitude pode ser um valor constante, ou pode ser associada a uma função, permitindo assim um controle maior sobre os comportamentos adotados pelo robô.

Um campo de força é um conjunto de vetores que age sobre o robô, sendo determinados cinco tipos básicos de campos de força (MURPHY, 2000, p. 124):

a) uniforme: campo em que a mesma força é aplicada não importando a posição do robô. Tipo de campo utilizado para forçar o robô a seguir uma determinada direção;

b) perpendicular: campo em que a força é aplicada perpendicularmente a algum objeto, parede ou borda, podendo apontar para o objeto ou não.
c) atrativo: campo cujo centro representa um objeto ou ponto que atrai o robô independente da sua posição. Geralmente este tipo de campo é utilizado para representar objetivos ou metas;
d) repulsivo: campo cujo centro representa um objeto ou ponto que repele o robô. Quando mais próximo o robô está do centro, mais forte é a força repulsiva. Geralmente este tipo de campo é utilizado para representar obstáculos;
e) tangencial: campo que forma uma tangente ao redor do objeto. Campos tangenciais podem tem sentido horário ou anti-horário. Este tipo de campo é utilizado para orientar um robô ao redor de um obstáculo, ou determinar que o robô investigue algo.

A Figura 2 apresenta exemplos de cada um dos tipos de força: o item (a) é um campo de força uniforme; o item (b), um campo de força perpendicular; o item (c) um campo atrativo; o item (d) o repulsivo e o item (e) um campo tangencial horário.

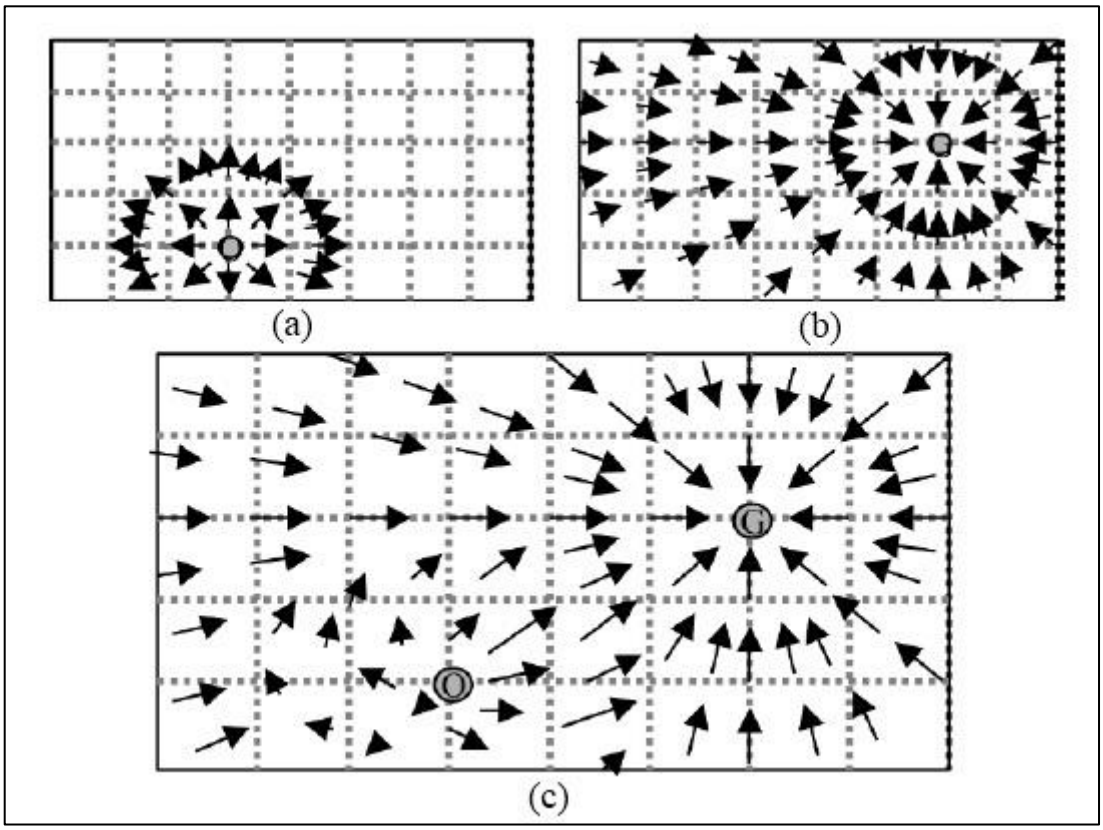


Fonte: Murphy (2000, p. 125).

Figura 2 - Tipos básicos de campos de força

Conforme descrito, cada campo de força determina um comportamento para o robô, e a soma de todos os campos de força determina o comportamento emergente. A soma de campos de força é o vetor resultante da soma dos vetores de cada campo de força atuando no robô. Por exemplo, supondo os vetores (x_1, y_1) e (x_2, y_2), o vetor resultante será: $(x_R, y_R) = (x_1 + x_2, y_1 + y_2)$.

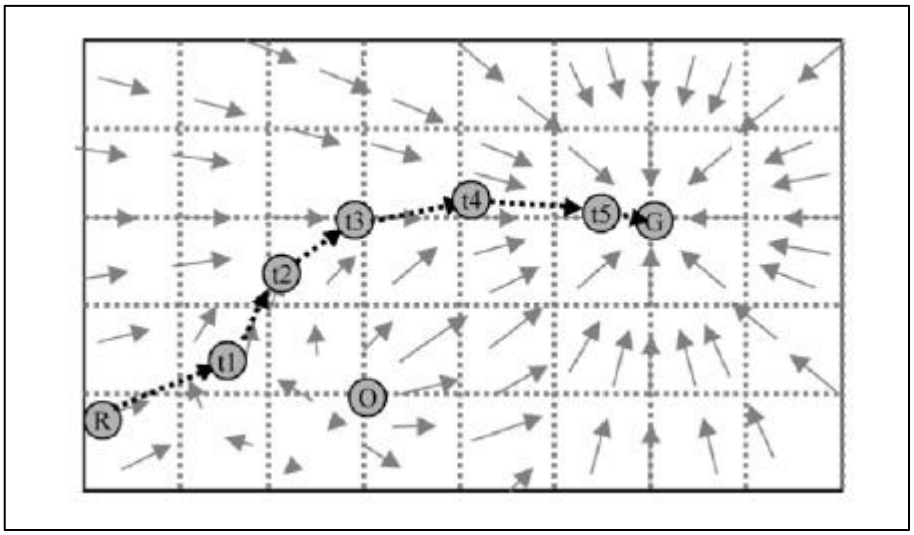
A Figura 3 apresenta este comportamento. No item (a) da figura está um campo de força do tipo repulsivo, criado devido a um obstáculo. No item (b), um campo de força atrativo, representando um ponto objetivo do robô. A soma de todos os campos resulta no comportamento definido pelo item (c). Na região próxima do obstáculo *O*, o robô é atraído pelo objetivo *G* mais repelido pelo obstáculo O ao mesmo tempo. Por outro lado, na região próxima do objetivo *G*, a magnitude do obstáculo *O* não permite que seu campo de força influencie o robô, que é atraído pelo objetivo *G*.



Fonte: Murphy (2000, p. 132).

Figura 3 - Soma de campos de força

A Figura 4 apresenta o caminho percorrido pelo robô, considerando os campos de força do item (c) da Figura 3. Inicialmente o robô, posicionado no ponto *R* move-se considerando apenas o campo de força atrativo do objetivo *G*. Do ponto *t1* ao *t3* o campo de força repulsivo do obstáculo *O* é combinado com o campo atrativo, afastando o robô do obstáculo. Após *t4*, o campo repulsivo do obstáculo não possui mais efeito, fazendo com que o robô avance em linha reta para o objetivo.



Fonte: Murphy (2000, p. 133).

Figura 4 - Caminho percorrido pelo robô

2.3.2 Algoritmo de Intersecção e União

O algoritmo de Bentley-Ottmann tem como objetivo determinar as intersecções entre vários segmentos de reta em um custo $O(n \log n + k)$, sendo n o número de segmentos de entrada e k o número de intersecções encontradas (BENTLEY e OTTMANN, 1979). O algoritmo trabalha com o conceito de *sweep line* (linha de varredura), onde uma linha imaginária se desloca pelo plano cartesiano, tratando os pontos de interesse presentes nos segmentos (BENTLEY e OTTMANN, 1979, p. 644). Estes pontos de interesse são denominados eventos.

A entrada do algoritmo é uma coleção de segmentos de reta *S*. Sua saída é a coleção dos pontos de interseção detectados em *S*.

O algoritmo utiliza-se de duas estruturas de dados:

a) uma árvore binária de segmentos de reta *SL*. Os segmentos são ordenados de acordo com a coordenada *y* do ponto de intersecção do segmento de reta com a linha de varredura. Esta estrutura é utilizada para determinar as relações de acima e abaixo entre segmentos de reta;
b) uma fila de eventos *Q*. Esta fila é ordenada de acordo com a prioridade dos eventos, que é estabelecida de forma crescente pela coordenada *x* do evento. A fila não permite eventos duplicados, sendo utilizada para determinar o próximo evento que linha de varredura deve tratar ao avançar.

Os eventos são divididos em três tipos:

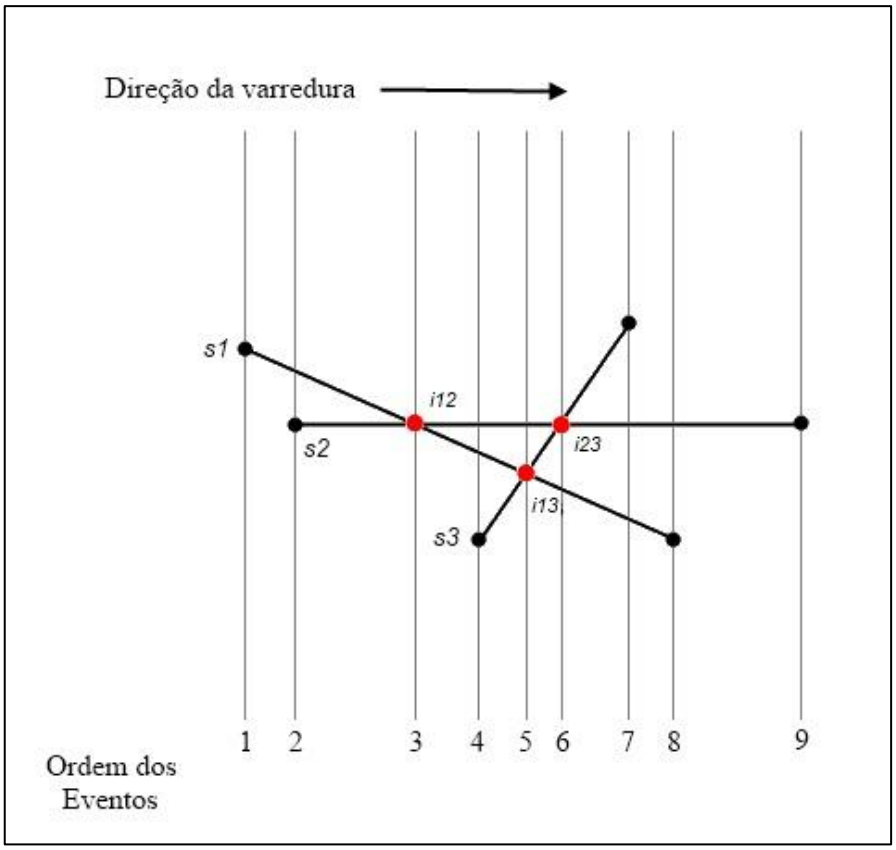
a) ponto de início de um segmento: quando a linha de varredura chega ao início de um segmento e passa a considerá-lo nas avaliações de intersecções (incluindo o segmento em *SL*);
b) ponto de término de um segmento: quando a linha de varredura chega ao final de um segmento e passa a desconsiderá-lo nas avaliações de intersecções (removendo o segmento em *SL*);
c) ponto de intersecção: quando a linha de varredura chega a uma intersecção entre dois segmentos.

A linha de varredura avança sempre para o próximo evento da fila *Q*. De acordo com cada o tipo do evento, determina-se quais são os segmentos de reta que devem ser verificados para se encontrar intersecções, que são adicionados a Q e tratados posteriormente. O pseudo-código do algoritmo Bentley-Ottmann é apresentado no Quadro 1.

Um exemplo do funcionamento do algoritmo pode ser observado na Figura 5. A ordem em que os eventos na fila *Q* serão tratados será:

a) evento 1: evento de início do segmento de reta *s1*. Adiciona o segmento s1 a árvore SL;
b) evento 2: evento de início do segmento de reta s2. Adiciona o segmento s2 a árvore SL. O segmento *s1* está acima do segmento *s2* na árvore *SL*, então é verificado se ambos geram uma intersecção. Como a intersecção é encontrada (representada pelo ponto *i12*), um evento para ela é adicionado na fila *Q*;
c) evento 3: evento de intersecção *i12*. A intersecção *i12* é adicionada aos pontos encontrados *IP*. Troca-se a posição dos segmentos *s1* e *s2* na árvore *SL*. Agora a árvore *SL* coloca o segmento *s2* acima de *s1*. Nenhuma nova intersecção é encontrada;
d) evento 4: evento de início do segmento de reta *s3*. Adiciona o segmento *s3* a árvore *SL*. O segmento *s3* está abaixo do segmento *s1* na árvore *SL*, então é verificado se ambos geram uma intersecção. Como a intersecção é encontrada (representada pelo ponto *i13*), um evento para ela é adicionada na fila Q;
e) evento 5: evento de intersecção *i13*. A intersecção *i13* é adicionada aos pontos encontrados *IP*. Troca-se a posição dos segmentos *s1* e *s3* na árvore *SL*. Nesse momento a árvore *SL* tem os segmentos na ordem (do mais alto para o mais baixo): *s2*, *s3*, *s1*. Como o segmento *s2* está acima do segmento *s3*, é verificado se ambos geram uma intersecção. Como a intersecção é encontrada (representada pelo ponto *i23*), um evento de intersecção é adicionado na fila *Q*;

f) evento 6: evento de intersecção *i23*. A intersecção *i23* é adicionada aos pontos encontrados *IP*. Troca-se a posição dos segmentos *s2* e *s3* na árvore *SL*, ficando o segmento *s1* abaixo do segmento *s3*. Nenhuma nova intersecção é encontrada;
g) evento 7: evento de término do segmento de reta *s3*. Remove o segmento *s3* na árvore *SL*;
h) evento 8: evento de término do segmento de reta *s1*. Remove o segmento *s1* na árvore *SL*;
i) evento 9: evento de término do segmento de reta *s2*. Remove o segmento *s2* na árvore *SL*;
j) No término da execução, os pontos representados por *i12*, *i13* e *i23* estão em *IP* e são retornados como pontos de intersecção encontrados.



Fonte: adaptado de Softsurfer (2006).

Figura 5 - Exemplo da execução do algoritmo de Bentley-Ottmann

```
Defina Q como uma fila de eventos ordenados por x de forma crescente.
Defina SL como a árvore binária vazia.
Defina IP como os pontos de intersecção encontrados.
Adicione todos os pontos dos segmentos de reta S em Q.
Enquanto Q não for vazia faça
   Defina E como o elemento removido do início de Q.
   Se E for o ponto esquerdo de um segmento de reta então
       Defina segE como o segmento de E.
       Adicione segE a SL
       Defina segUp como o segmento acima de segE em SL.
       Defina segDown como o segmento abaixo de segE em SL.
       Se existe intersecção entre segE com segUp então
          Adicione a intersecção em Q.
       Se existe intersecção entre segE com segDown então
          Adicione a intersecção em Q.
   Senão se E for o ponto direito de um segmento de reta então
       Defina segE como o segmento de E.
       Defina segUp como o segmento acima de segE em SL.
       Defina segDown como o segmento abaixo de segE em SL.
       Remova segE de SL
       Se existe intersecção entre segUp e segDown então
          Adicione a intersecção em Q.
   Senão // E é uma intersecção
       Adicione o ponto de E a lista IP.
       Obtenha os dois segmentos que formam a intersecção.
       Defina segE1 como o segmento mais alto em SL da intersecção.
       Defina segE2 como o segmento mais baixo em SL da intersecção.
       Troque as posições de segE1 e segE2 em SL.
       Defina segUp como o segmento acima de segE2 em SL.
       Defina segDown como o segmento abaixo de segE1 em SL.
       Se existe intersecção entre segE2 e segUp então
          Adicione a intersecção em Q.
       Se existe intersecção entre segE1 e segDown então
          Adicione a intersecção em Q.

Retorne todos os pontos de IP.
```

Fonte: Softsurfer (2006).

Quadro 1 - Pseudo-código do algoritmo de Bentley-Ottmann

Para realizar a união dos polígonos de área mapeada, basta varrer os pontos de interseção encontrados pelo algoritmo de Bentley-Ottmann, determinando quais são os pontos vizinhos aos de intersecção que devem ser eliminados. Desta forma é possível realizar a união de dois polígonos com o custo *O(2n)* sendo *n* o número de intersecções entre os dois polígonos. O algoritmo varre os pontos de interseção encontrados pelo algoritmo de Bentley-Ottmann, determinando quais são os pontos vizinhos aos de intersecção que devem ser eliminados. A entrada do algoritmo são dois polígonos *A* e *B*, e os pontos de interseção *IP* entre estes dois polígonos. Os polígonos devem ter a capacidade de fornecer o ponto posterior e antecessor de um determinando ponto *p*, seguindo a ordem horária dos pontos. Sua saída é o polígono resultante da união dos dois polígonos de entrada.

O pseudo-código do algoritmo de união é apresentado no Quadro 2.

```
Defina IP como os pontos de intersecção recebidos.
Defina A como o primeiro polígono recebido.
Define B como o segundo polígono recebido.
Defina U como uma coleção de segmentos, que inclua todos os segmentos de A
e B.
VERIFICA_PONTOS_INTERNOS(I, A, B, U);
VERIFICA_PONTOS_INTERNOS(I, B, A, U);
Retorna o polígono formado pelos segmentos presentes em U.

VERIFICA_PONTOS_INTERNOS(Intersecções, Polígono, Polígono_Oposto, União)
Para cada ponto I em Intersecções faça
   Defina S1 como o ponto antecessor de I em Polígono_Oposto
   Defina S2 como o ponto posterior de I em Polígono_Oposto

   Defina R como o ponto, S1 ou S2, o qual está contido dentro de
Polígono.
   Remove de União o segmento de I para R.

   Enquanto R não for uma intersecção faça
      Defina P como o ponto antecessor/posterior de R em Polígono_Oposto.
      Remove de União o segmento de R para P.
      Defina R como P
```

Quadro 2 - Pseudo-código do algoritmo de união

Na Figura 6 tem-se o exemplo de funcionamento do algoritmo de união. No item (a) o início do algoritmo, com os pontos de intersecção detectados.

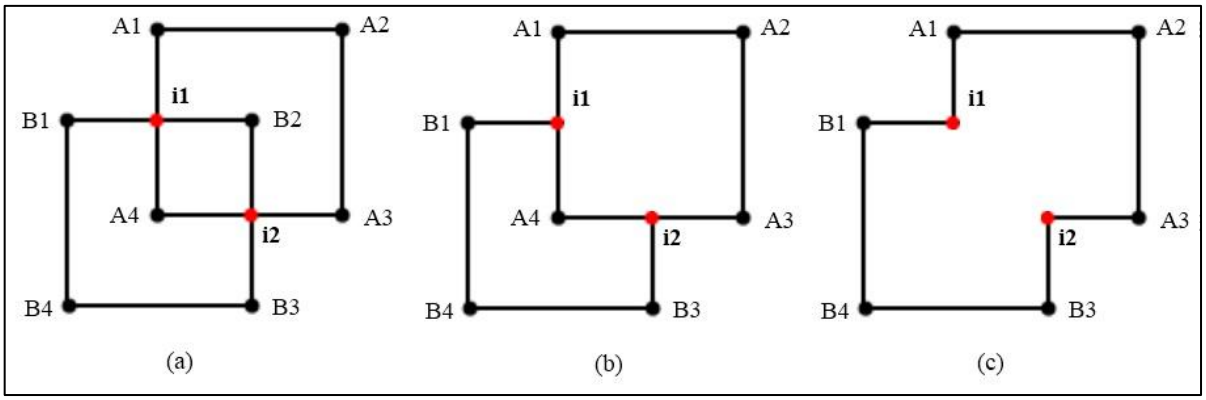


Figura 6 - Exemplo de execução do algoritmo de união

Primeiramente avaliam-se os pontos de intersecção pelo polígono *A*. A primeira interseção é *i1*, e determinam-se os dois pontos vizinhos no polígono *B*, os pontos *B1* e *B2*. Somente o ponto *B2* está dentro do polígono *A*, então o segmento de *i1* até *B2* é removido. O ponto *B2* não é um ponto de interseção, então remove o segmento de *B2* ao seu próximo ponto, que é *i2*. Sendo *i2* uma intersecção, então a avaliação do ponto *i1* está completa. A próxima intersecção é *i2*, sendo o seu ponto vizinho somente *B3* (*B2* já foi descartado). O ponto *B3* não está dentro do polígono *A*, então a avaliação de *i2* está completa. O estado atual do algoritmo pode ser visto no item (b) da Figura 6.

Passa-se a avaliar os pontos de intersecção pelo polígono *B*. A primeira interseção é *i1*, e determinam-se os dois pontos vizinhos no polígono *A*, os pontos *A1* e *A4*. Somente o ponto *A4* está dentro do polígono *B*, então o segmento de *i1* até *A4* é removido. O ponto *A4* não é um ponto de interseção, então remove o segmento de *A4* ao seu ponto anterior, que é *i2*. Sendo *i2* uma intersecção, então a avaliação do ponto *i1* está completa. A próxima intersecção é *i2*, sendo o seu ponto vizinho somente *A3* (*A4* já foi descartado). O ponto *A3* não está dentro do polígono *B*, então a avaliação de *i2* está completa. Feita as avaliações de ambos os polígonos, a união está completa conforme o item (c) da Figura 6.

2.4 TRABALHOS CORRELATOS

As seções a seguir descrevem os trabalhos que tenham relação com a navegação exploratória. No âmbito nacional, pode-se citar a tese de navegação exploratória utilizando Problema de Valores de Contorno (PVC) de Silva Júnior (2003) e o artigo sobre a exploração de ambientes com multiagente nos jogos eletrônicos de Soares e Campos (2006). Internacionalmente, têm-se os artigos de Burgard et al. (2000), que descreve uma abordagem

para robôs heterogêneos e o de Cohen (1996), que define a divisão do time de robôs de acordo com funções específicas.

2.4.1 Navegação exploratória baseada em PVC

Descreve uma abordagem para construção de um algoritmo de exploração que funcione também para planejamento de caminhos. Essa necessidade surgiu da tentativa de evitar o chaveamento entre duas estratégias, uma para descobrir o ambiente e montar o mapa, e outra para caminhar no mapa já construído(SILVA JÚNIOR, 2003, p. 16). Segundo Silva Júnior (2003, p. 17), "o objetivo desta tese consiste em desenvolver uma estratégia que integre a capacidade exploratória e o planejamento de caminhos através de um princípio único: campos potenciais oriundos da solução de problemas de valores de contorno (PVC).". A estratégia foi testada utilizando somente um robô.

Neste trabalho, o ambiente é representado por uma matriz bidimensional cujas células correspondem a posições específicas do ambiente e armazenam valores computados a partir das informações oriundas dos sensores. Os valores seriam, por exemplo, posição ocupada, livre ou não explorada. A cada uma dessas posições é definido um campo de força, sendo repulsivo para obstáculos e atrativo para as posições não exploradas. Estes campos de força são representados de forma conjunta por uma equação matemática. O PVC então é definido como sendo encontrar valores para a equação matemática que permitam contornar os obstáculos (que a equação representa).

O trabalho foi validado com experimentos utilizando um único robô NOMAD 200[4] em diversos ambientes, desde labirintos formados por paredes ortogonais até ambientes esparsos. A partir dos resultados obtidos, o autor conclui que a abordagem proposta foi robusta e eficiente, permitindo o robô explorar e mapear todas as regiões acessíveis do ambiente.

2.4.1.1.1.1 Multiagent exploration task in games through negociation

O objetivo do trabalho é a exploração de ambientes desconhecidos em jogos

[4] O NOMAD 200 é um robô comercial, voltado para pesquisas, fabricado pela Nomadic Technologies, Inc. Possui cinco sensores: infravermelho, ultrassom, visão, bússola e tátil. Construído em formato cilíndrico, possuindo uma base com 3 rodas alinhas, permitindo mover o robô em qualquer direção.(SILVA JÚNIOR, 2003, p. 60)

eletrônicos, especialmente jogos de estratégia em tempo real. A exploração em jogos possui certas diferenças com a exploração de robôs. Algumas destas diferenças seriam, por exemplo, que os agentes de exploração podem ser removidos do jogo, com a unidade sendo morta ou destruída; e que os agentes devem evitar áreas inimigas, sendo que estas áreas são inicialmente desconhecidas.

No trabalho são implementadas e avaliadas quatro estratégias simples e uma baseada em negociação, em um simulador de exploração multiagente. As estratégias simples são basicamente funções que alteram o vetor de velocidade dos agentes, sendo elas: em linha reta, parábola, espiral e senoidal. A estratégia baseada em negociação, definidas por Soares e Campos (2006), determina que cada agente defina um conjunto de posições alvos, planejadas para serem exploradas em sequência. Estas posições poderão ser negociadas com outro agente, caso mostre-se mais vantajoso para o outro agente. As rotas dos agentes são constantemente negociadas e, como consequência, o desempenho da exploração da área é melhorado.

2.4.2 Collaborative multi-robot exploration

Define uma estratégia para controlar múltiplos robôs heterogêneos que colaboram para a exploração do ambiente. Neste trabalho o ambiente também é representado como uma matriz bidimensional, mantendo as regiões já mapeadas. A idéia da abordagem de Burgard et al. (2000, p. 481) é considerar o custo para chegar a uma região inexplorada e sua utilidade. Burgard et al. (2000, p. 418) definem a utilidade de uma localidade alvo como a probabilidade que esta localidade seja visível a partir de uma localidade alvo atribuída a outro robô. Dessa forma, é sempre atribuída a localidade alvo a um robô que tenha melhor relação entre a utilidade da localização e o custo do robô para chegar à localidade.

A abordagem proposta foi testada em ambientes reais utilizando dois robôs, com times coordenados entre si ou não. Também se realizou várias simulações com o objetivo de obter dados mais quantitativos. Como resultado, mostrou-se que times coordenados são mais rápidos na resolução do que times de robôs não coordenados. Os autores afirmam que dois robôs coordenados obtiveram o mesmo tempo no mapeamento do ambiente que três robôs descoordenados. A abordagem foi considerada mais capaz de coordenar os múltiplos robôs dos que as técnicas anteriores, melhorando o tempo total do mapeamento.

2.4.3 Adaptive mapping and navigation by teams of simple robots

Desenvolve uma abordagem utilizando um time de robôs para mapear um ambiente desconhecido em busca de um local objetivo. Cohem (1996, p. 412) divide o time em um navegador, responsável por determinar o caminho ao objetivo, e em cartógrafos, responsáveis pela exploração aleatória do ambiente, para montar o mapa. O mapa é construído para ser uma estrutura que permita ao navegador planejar um caminho mais curto (ou o possivelmente o mais curto) entre os pontos do ambiente. Os cartógrafos podem explorar o ambiente de maneira aleatória e caso algum deles encontre o objetivo, este repassa a informação aos robôs em seu campo de visão, que repassam sucessivamente até que ela chegue ao navegador. Com base no caminho feito pela mensagem, o navegador consegue determinar o local do objetivo.

Todos os experimentos deste trabalho foram realizados em quatro ambientes simulados em computadores. Isso permitiu que os experimentos trouxessem dados sobre a qualidade dos caminhos realizados, os efeitos do alcance dos sensores dos robôs e os efeitos do erro nas movimentações (quando há erros de posicionamento que se acumulam). O autor conclui que o tempo de mapeamento diminui rapidamente quando o número de robôs cresce.

3 DESENVOLVIMENTO

Neste capítulo são abordadas as etapas do desenvolvimento do projeto. São apresentadas os principais requisitos, a especificação, a implementação e por fim são listados os resultados e discussão.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Um sistema multiagente robótico capaz de realizar navegação exploratória e mapeamento deve ser formado por certa quantidade de robôs autônomos. Estes robôs devem ser capazes de perceber o ambiente e seus obstáculos, bem como se movimentar pelo ambiente para explorá-lo mapeando-o.

Para que o usuário do sistema multiagente possa solicitar o mapeamento de um ambiente, o sistema deve dispor de um gerenciador. Este gerenciador tem como objetivo coordenar os robôs, permitindo iniciar, parar, e visualizar a exploração e mapeamento.

Portanto, o sistema multiagente deverá atender aos seguintes requisitos:

a) gerenciar os robôs para obter o mapeamento completo do ambiente (requisito funcional – RF 01);
b) suportar o mapeamento de ambientes estáticos, possuindo somente barreiras que se elevam do solo, no mínimo, 0,5 metros (RF 02);
c) disponibilizar uma interface para gerenciar o sistema multiagente e apresentar o mapa do ambiente, que deve ser atualizado durante a realização do mapeamento (RF 03);
d) utilizar robôs criados pelo *kit* LEGO Mindstorms (requisito não-funcional - RNF 01);
e) suportar no mínimo 2 robôs (RNF 02);
f) realizar a comunicação dos robôs com o computador utilizando a tecnologia *Bluetooth* (RNF 03);
g) ser implementado utilizando a linguagem de programação Java (RNF 04);
h) utilizar o ambiente Eclipse para o desenvolvimento (RNF 05);
i) utilizar as bibliotecas da máquina virtual LeJOS para construção do controlador de

cada robô (RNF 06);

j) utilizar as bibliotecas do Java, Swing e Java2D para construção da interface de apresentação do mapa obtido pela exploração (RNF 07).

3.2 ESPECIFICAÇÃO

A especificação deste trabalho foi desenvolvida na ferramenta Enterprise Architect 9.0.905 utilizando os conceitos da *Unified Modeling Language* (UML) para a elaboração dos diagramas de casos de uso, de estados, de classes e de sequência.

3.2.1 Diagrama de casos de uso

O sistema multiagente desenvolvido possui os casos de uso apresentados no diagrama de casos de uso da Figura 7.

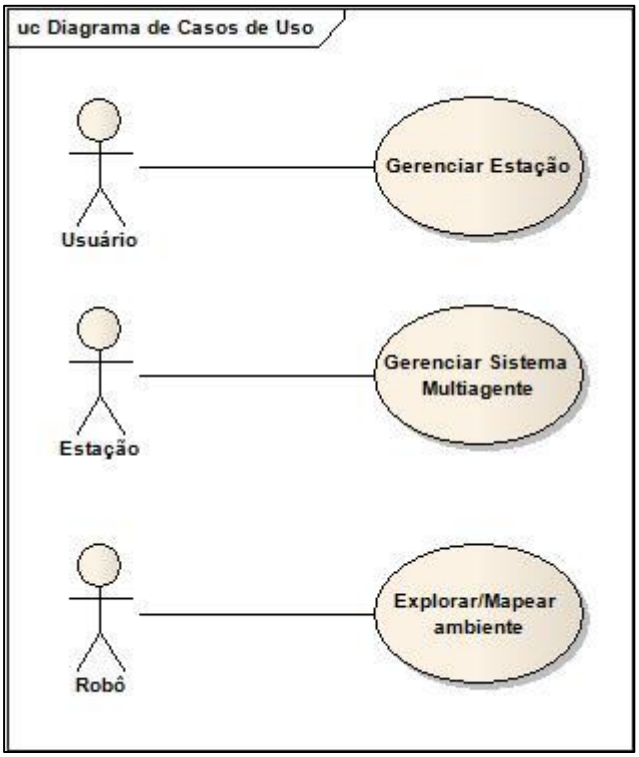


Figura 7 - Diagrama de casos de uso

O primeiro caso de uso é `Gerenciar Estação` (Quadro 3). A estação é o componente que faz a interface com o usuário e gerencia o sistema multiagente. Ela permite que o usuário controle o início e fim do mapeamento, assim como determinar movimentação do robô para pontos determinados. Este caso de possui um cenário principal e atende ao RF 03.

Gerenciar Estação: permite ao usuário gerenciar a estação de mapeamento.	
Pré-Condição	Estação inicializada.
Cenário Principal	1. Usuário determina o início da exploração do ambiente. 2. A estação apresenta a representação visual da área mapeada e das informações de cada robô, atualizando esta representação à medida que os robôs realizam o mapeamento. 3. Usuário acompanha a execução da exploração. 4. Usuário determina o fim da exploração do ambiente.
Pós-Condição	Nenhuma.

Quadro 3 - Caso de uso `Gerenciar Estação`

O segundo caso de uso é `Gerenciar Sistema Multiagente` (Quadro 4), e permite que a estação gerencie os robôs para que seja feita a exploração do ambiente. Este caso de uso também possui somente um cenário principal e atende aos requisitos RF 01 e RF 02.

O terceiro caso de uso é `Mapear/Explorar Ambiente` (Quadro 5), e permite que o robô comporte-se para realizar a exploração. Este caso de uso também possui um cenário principal e atende aos RF 01 e RF 02.

Gerenciar Sistema Multiagente: permite que a estação gerencie a exploração do ambiente pelos robôs.	
Pré-Condição	Estação inicializada. Robôs inicializados.
Cenário Principal	1. Estação realiza busca por robôs disponíveis para conexão. 2. Estação conecta em todos os robôs disponíveis. 3. Estação gerencia os robôs, solicitando explorações até que o mapeamento seja concluído. As solicitações são pontos objetivo onde o robô deve deslocar-se até o ponto e a partir disso realizar exploração. As solicitações são feitas para que os robôs de forma cooperativa aumentem a área mapeada. 4. Estação atualiza mapeamento com as explorações realizadas pelos robôs. 5. Estação desconecta os robôs utilizados.
Pós-Condição	Mapeamento realizado.

Quadro 4 - Caso de uso `Gerenciar Sistema Multiagente`.

Mapear/Explorar Ambiente: permite que o robô realize a exploração do ambiente.	
Pré-Condição	Robô iniciado.
Cenário Principal	1. Robô aguarda a conexão da estação. 2. Robô aguarda comando para iniciar a execução. 3. Robô recebe solicitações de exploração da estação. 4. Robô toma decisões de forma autônoma para atender as solicitações, utilizando a representação de mundo e campos de força.
Pós-Condição	Robô completa as solicitações.

Quadro 5 - Caso de uso `Mapear/Explorar Ambiente`

3.2.2 Diagrama de estados

Durante a execução do robô, o mesmo deve modificar seu comportamento inúmeras vezes para atender às solicitações de exploração enviadas pela estação, como determinado pelo caso de uso `Gerenciar Sistema Multiagente`. Os estados que devem ser suportados são:

a) `CONECTADO`: é o estado inicial do robô. O robô está conectado a estação aguardando o comando para iniciar o mapeamento;
b) `EM ESPERA`: o robô permanece parado enquanto aguarda uma nova solicitação da estação;
c) `EXPLORADOR`: o robô se desloca para o ponto objetivo da solicitação utilizando os campos de força, com o sonar desligado. Ao chegar ao objetivo, o robô realiza a exploração à frente com o sonar, permanecendo parado para evitar colisões. O robô notifica a estação qualquer obstáculo detectado;
d) `DESCONECTADO`: o robô foi desconectado da estação e terá sua execução encerrada.

Estes estados são alterados de acordo com as solicitações enviadas pela estação. A realização da solicitação é feito de forma autônoma, utilizando a representação de mundo e campos de força presentes no robô. Na Figura 8 é possível visualizar os estados suportados e as condições em que um estado é trocado por outro.

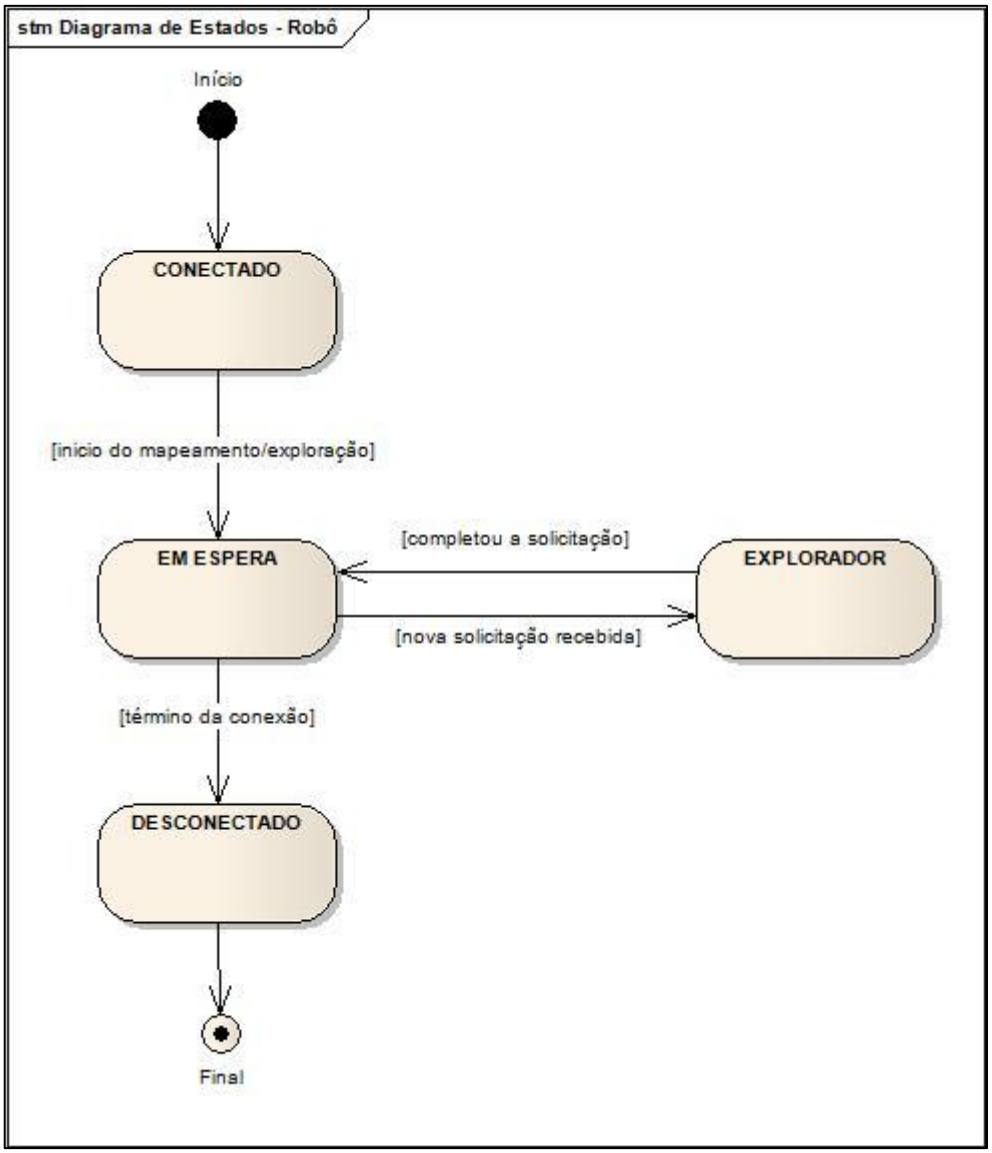


Figura 8 - Diagrama de estados do robô

3.2.3 Diagramas de classes

Nesta seção é apresentada a especificação das classes desenvolvida para o sistema e feita uma análise sobre a funcionalidade de cada um delas. Algumas classes meramente auxiliares foram omitidas para uma melhor visualização do diagrama, não comprometendo o entendimento do mesmo.

3.2.3.1 Classes do robô

Neste trabalho o robô representa o agente do sistema multiagente. Através do seu sensor de sonar e da comunicação com a estação ele obtém a percepção para através dos motores (seus atuadores) ele possa se deslocar no ambiente. Devido à complexidade do sistema embarcado no robô, como boa prática, as funcionalidades do robô foram divididas em componentes, separando-as da lógica de controle do robô. A Figura 9 apresenta a especificação das classes do programa embarcado do robô em detalhes.

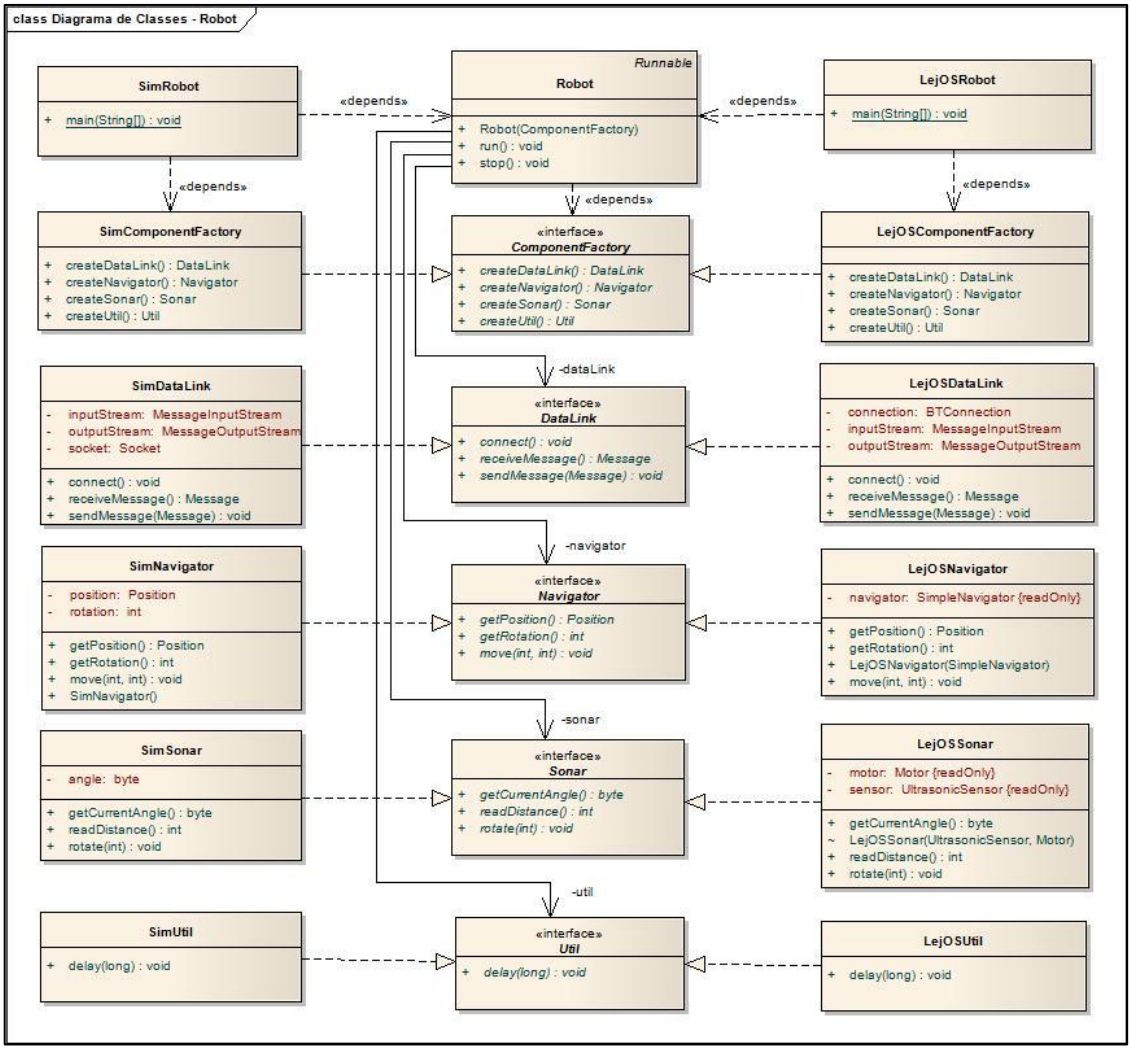


Figura 9 - Diagrama de classes de `Robot`

A classe `Robot` é a classe responsável pelo comportamento do robô. Ela é responsável por interpretar as solicitações da estação e executá-los utilizando-se dos componentes disponíveis e dos campos de força. A Interface `ComponentFactory` implementa o padrão *Abstract Factory*, como uma fábrica de componentes para `Robot`. Este padrão permite que `Robot` obtenha os componentes sem a necessidade de conhecer a implementação concreta dos

mesmos, diminuindo o acoplamento e o que possibilita a execução do robô em um ambiente simulado.

Como componentes tem-se: a interface `DataLink`, responsável por estabelecer a comunicação entre o robô e a estação, fornecendo funcionalidades de enviar e receber mensagens; a interface `Navigator`, componente responsável por manter a posição e rotação correntes do robô e de realizar a rotação do robô em qualquer ângulo e movimentá-lo em linha reta; a interface `Sonar`, responsável por realizar a exploração do ambiente através de ondas ultra-sônicas, determinando se foi detectado um obstáculo; e por último o a interface `Utils`, que disponibiliza funções utilitárias para o funcionamento do robô.

As classes que possuem prefixos `LejOS` são as implementações para a máquina virtual do LeJOS rodando no LEGO Mindstorms. `LejOSDataLink`, `LejOSNavigator`, `LejOSSonar` e `LejOSUtils` são as implementação disponibilizadas para cada um dos componentes utilizados por `Robot`. `LejOSComponentFactory` é a implementação de `ComponentFactory` que fornece a criação dos componentes `LejOS`. `LejOSRobot` é a classe executada pelo LEGO `Mindstorms` que cria uma instância de `Robot` e a executa.

As classes que possuem prefixos `Sim` são as implementações que buscam simular o comportamento do robô verdadeiro. `SimDataLink`, `SimNavigator`, `SimSonar` e `SimUtils` são as implementações disponibilizadas para cada um dos componentes utilizados por `Robot`. `SimComponentFactory` é a implementação de `ComponentFactory` que fornece a criação dos componentes `Sim`. `SimRobot` é a classe executada pela máquina virtual Java, cria uma instância de `Robot` e a executa.

3.2.3.2 Classes da comunicação

A Figura 10 apresenta as especificações das classes utilizadas para troca de mensagens entre a estação e o robô. A classe `Message` representa a mensagem trocada entre o robô e a estação. Todo o conteúdo da mensagem é armazenado em um vetor de *bytes*. A classe suporta a escrita e leitura dos tipos primitivos *int* e *byte* do Java.

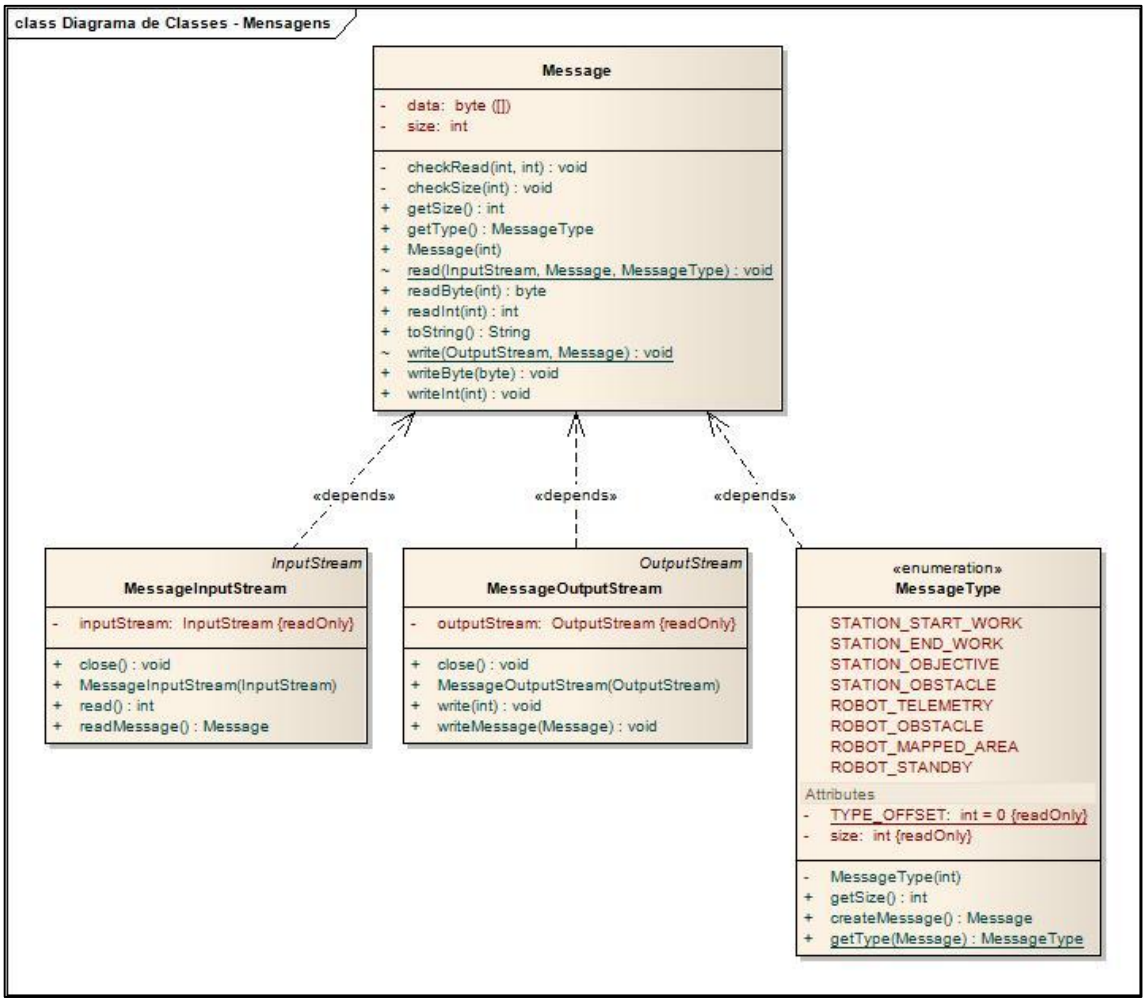


Figura 10 - Diagrama de classes de `Message`

A enumeração `MessageType` determina os tipos conhecidos de mensagens. Cada tipo é representado pelo primeiro *byte*. Cada tipo também define a quantidade de *bytes* necessários para armazenar as informações necessárias. O tamanho da mensagem sempre pode ser determinado pela fórmula: 1 (tipo da mensagem) + tamanho definido pelo tipo. Na sequência são apresentados os tipos de mensagens possíveis:

a) `STATION_START_WORK`: determina que o robô deva iniciar a execução da exploração, aguardando a definição do seu objetivo. Essa mensagem não possui nenhuma informação (0 *byte*);

b) `STATION_END_WORK`: determina que o robô encerre a execução da exploração. Essa mensagem não possui nenhuma informação (0 *byte*);

c) `STATION_OBJECTIVE`: determina um novo objetivo para o robô. Constituída de 9 *bytes*, sendo 4 utilizados para coordenada *x* e 4 para coordenada *y* do ponto objetivo e 1 para indicar se o robô deve utilizar o sonar ou não durante o deslocamento;

d) `STATION_OBSTACLE`: informa ao robô a detecção de um obstáculo. Constituída de 8

bytes, 4 utilizados para coordenada *x* e 4 para coordenada *y* do obstáculo;

e) `ROBOT_TELEMETRY`: mensagem enviada do robô para a estação contendo informações sobre o estado do robô. Constituída de 21 *bytes*, sendo 4 utilizados para a coordenada *x* e 4 para coordenada *y* da posição corrente do robô, 1 *byte* para indicar o ângulo do sonar em relação à proa[5] do robô, 4 *bytes* para indicar o ângulo do robô em relação ao eixo das abscissas e mais 4 utilizados para a coordenada *x* e 4 para coordenada *y* da posição do objetivo do robô;

f) `ROBOT_MAPPED_AREA`: mensagem enviada do robô para estação informando o triângulo representando a área mapeada pelo sonar. Constituída de 24 *bytes*, sendo os 8 primeiros *bytes* para informar as coordenadas *x* e *y* do primeiro ponto, os próximos 8 *bytes* para informar as coordenas *x* e *y* do segundo ponto e os últimos 8 *bytes* para informar as coordenas *x* e *y* do terceiro ponto;

g) `ROBOT_OBSTACLE`: mensagem enviada do robô para estação informando sobre a detecção de um obstáculo pelo robô. Este obstáculo é processado pela estação a fim de determinar a redundância do obstáculo. Constituída de 8 *bytes*, 4 utilizados para coordenada *x* e 4 para coordenada *y* do obstáculo;

h) `ROBOT_STAND_BY`: significa que o robô está parado e aguardando novas solicitações da estação. Não possui nenhuma informação (0 *byte*).

Os tipos que tenham o prefixo `STATION` são mensagens enviadas da estação para o robô. Tipos que tenham o prefixo `ROBOT` são as mensagens enviadas do robô para estação.

3.2.3.3 Classes da estação

A Figura 11 apresenta a especificação das classes da estação, com os métodos e atributos suprimidos para melhor visualização. A classe `Chartis`[6] é a classe principal, onde a execução é iniciada e são criados os componentes necessários para realizar o mapeamento.

As classes restantes da estação podem ser divididas em cinco grandes grupos: interface, gerenciamento dos agentes, comunicação, algoritmos e geometria. Estes grupos são explicados nas seções seguintes. As classes em roxo pertencem a biblioteca CompGeom (KIERS, 2011).

---

[5] Proa é um termo náutico para se referir à parte da frente da embarcação. Sua utilização é comumente estendida para qualquer veículo.

[6] O termo *chartis* vem do grego, cujo significado é mapa.

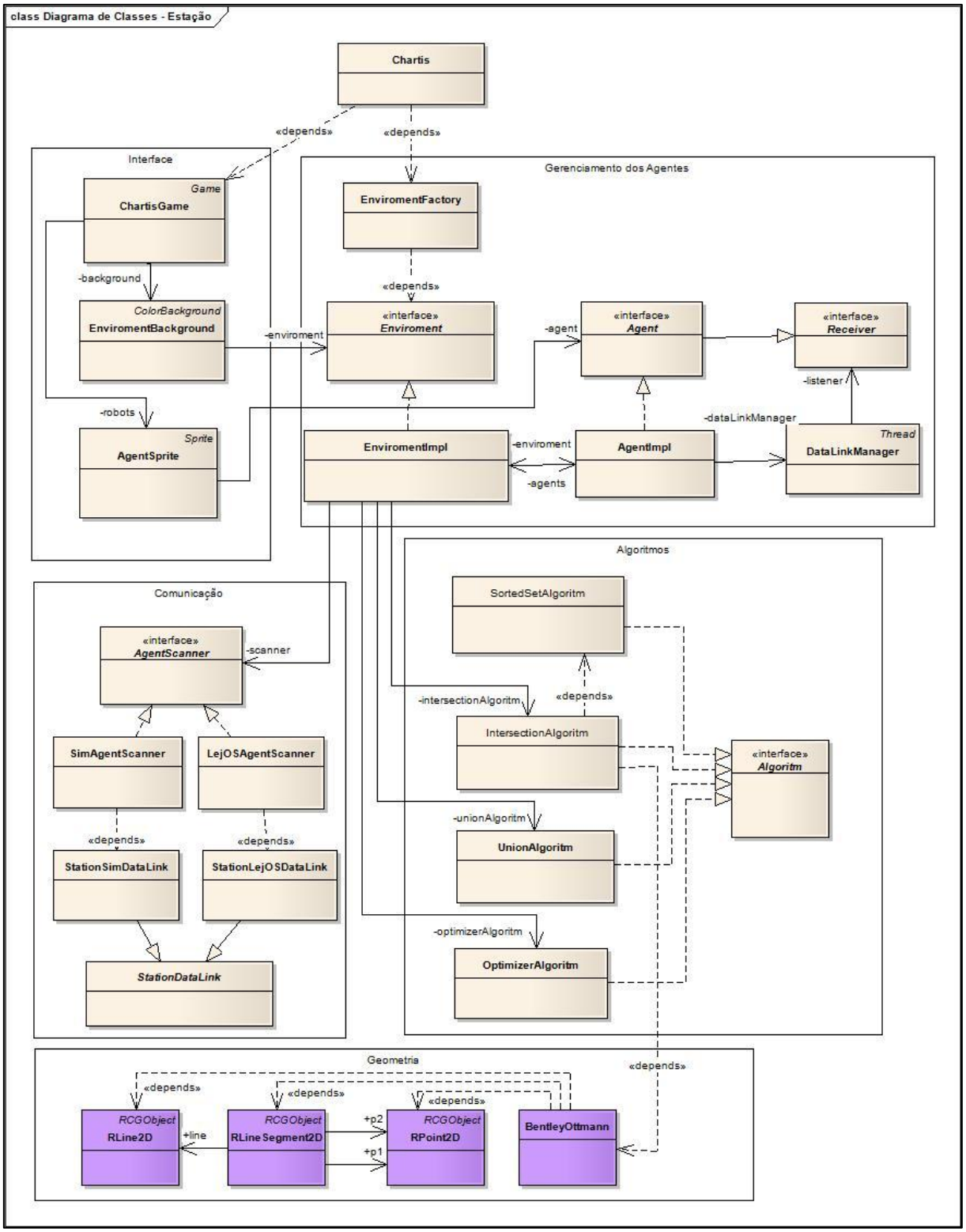


Figura 11 - Diagrama de classes da estação

3.2.3.3.1 Grupo gerenciamento dos agentes

Este grupo representa o núcleo da estação. A interface `Enviroment` fornece acesso às

informações do mapeamento para visualização. A implementação desta interface, `EnviromentImpl`, gerencia o sistema multiagente, recebendo notificações dos agentes referentes a novas áreas mapeadas, e a descoberta de obstáculos. A partir destas notificações, determina quais as solicitações que devem ser atendidas pelos agentes.

A interface `Agent` fornece acesso às informações do agente robô, como posição corrente, ângulo do sonar, a área mapeada pelo sonar etc. Sua implementação, `AgentImpl`, é responsável gerenciar o estado do robô, interpretando as mensagens enviadas pelo robô para tratá-las junto a `EnviromentImpl` e responde-las. A classe `DataLinkManager` é a classe que gerencia a conexão de `DataLink` com o agente, criando uma *thread* separada para cada agente, para que a comunicação de um agente não atrapalhe a comunicação de outro agente ou a atualização da interface com o usuário.

Por fim, a classe `EnviromentFactory` utilizando-se do padrão *Factory Method*, permite desacoplar a implementação de `Enviroment` e `Agent` das classes utilizadas para desenhar a interface com o usuário.

3.2.3.3.2 Grupo interface

A classe `ChartisGame` é a classe que coordena o ciclo de atualização da interface para refletir as alterações no mapeamento. A classe `EnviromentBackground` é responsável por desenhar as informações provenientes de `Enviroment`, como áreas mapeadas e obstáculos. Para cada agente conectado na estação, é criado um `AgentSprite`, que tem objetivo de desenhar as informações do agente, como por exemplo, a posição corrente, sonar e objetivo.

3.2.3.3.3 Grupo comunicação

A interface `AgentScanner` é a responsável por descobrir para a estação todos os agentes disponíveis para conexão. Esta interface possui duas implementações: a `SimAgentScanner` é utilizada para descobrir agentes simulados e a `LejOSAgentScanner`, que descobre robôs NXT. Para cada agente disponível, o `AgentScanner` retorna um `StationDataLink` correspondente daquela conexão, o qual também possuía duas derivações: `StationSimDataLink` e `StationLejOSDataLink`, para conexões com agentes simulados ou

robôs NXT, respectivamente.

3.2.3.3.4 Grupo algoritmos

A interface `Algoritm` determina a funcionalidade padrão de um algoritmo, que recebe uma entrada, processa e devolve um resultado. Utiliza-se do padrão de projeto *Decorator* para encadear uma sequência de algoritmos, onde a saída de um algoritmo é à entrada de outro. A classe `SortedSetAlgoritm` recebe como entrada uma coleção de segmentos de reta (`RLineSegment2D`) que forma um polígono e retorna uma coleção de pontos (`RPoint2D`) ordenada no sentido horário. Com a coleção de pontos e segmentos dos dois polígonos, a classe `IntersectionAlgoritm` utiliza o algoritmo de `BentleyOttmann` para determinar as intersecções entre os polígonos. Determinado as intersecções, a classe `UnionAlgoritm` realiza-se a união conforme a seção 2.3.2. O resultado da união (caso ocorra uma união) é otimizado pela classe `OptimizerAlgoritm`, que descarta pontos próximos uns dos outros e pontos colineares.

3.2.3.3.5 Grupo geometria

A classe `BentleyOttmann` utiliza-se das classe `RLine2D`, `RLineSegment2D` e `RPoint2D` para implementação do algoritmo de Bentley-Ottmann. Todas estas quatro classes (assim como outras não citadas) são disponibilizadas na biblioteca CompGeom (KIERS, 2011).

3.2.4 Diagramas de sequência

O diagrama de sequência da Figura 12 detalha a interação entre as classes para atender ao caso de uso `Gerenciar Sistema Multiagente` (Quadro 4), apresentando o fluxo de mensagens entre a estação e o robô para realizar a exploração.

Inicialmente a estação estabelece contato com o robô, que passa a aguardar solicitações da estação. Em seguida, a estação e o robô permanecem em um laço de repetição, onde o robô recebe uma nova solicitação da estação. O robô passa a executar buscando atender a

solicitação, notificando a estação a cada nova área mapeada e obstáculo detectados. O robô também é notificado de quais são os obstáculos relevantes a serem armazenados. Ao completar a solicitação, o robô notifica estação, que pode requerer uma nova solicitação, continuando o laço.

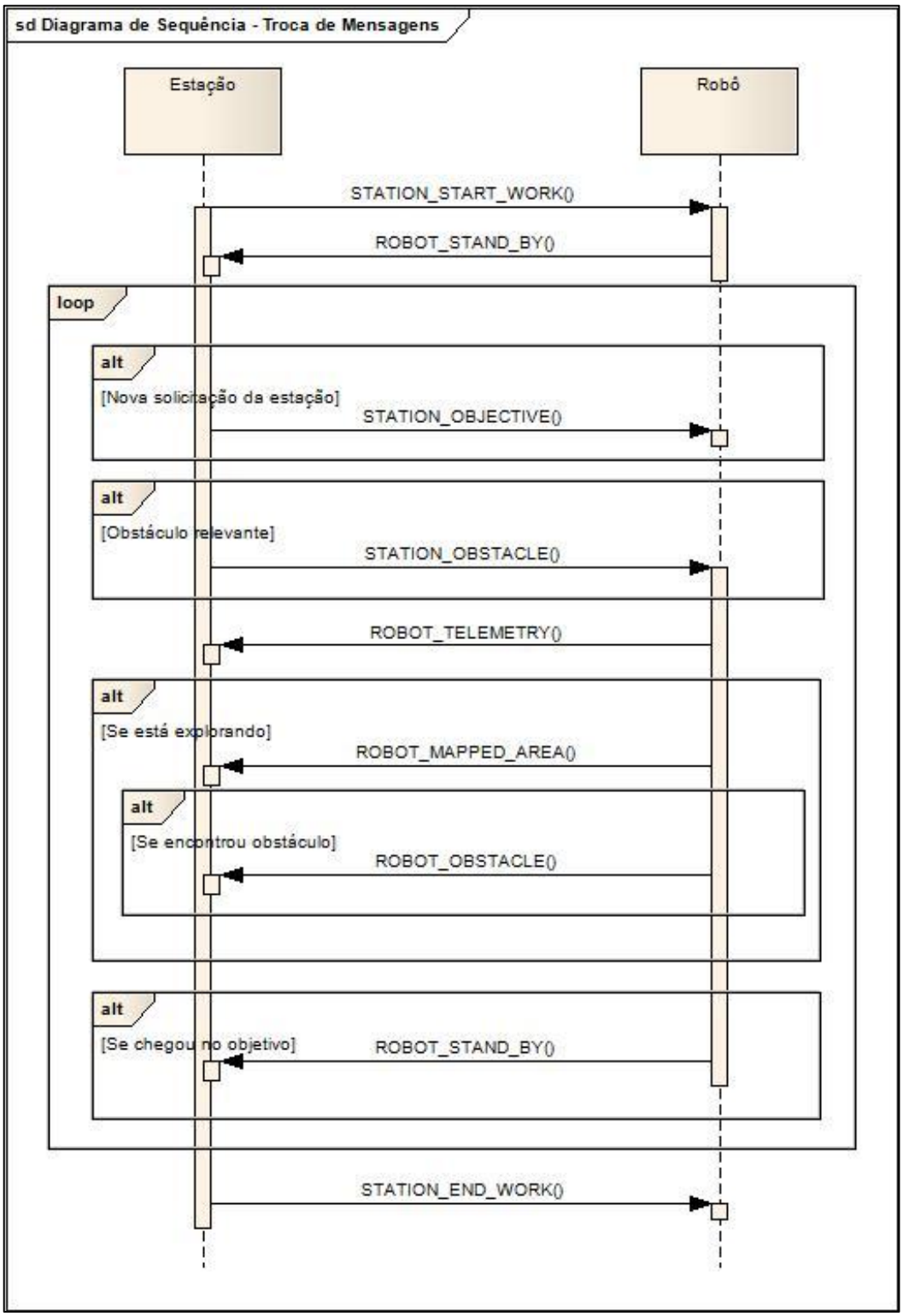


Figura 12 - Diagrama de sequência das trocas de mensagens entre a estação e o robô

A Figura 13 apresenta a iteração entre as classes do robô para atender o caso de uso `Mapear/Explorar Ambiente` (Quadro 5), sendo demonstrada a interação da classe `Robot` com seus componentes.

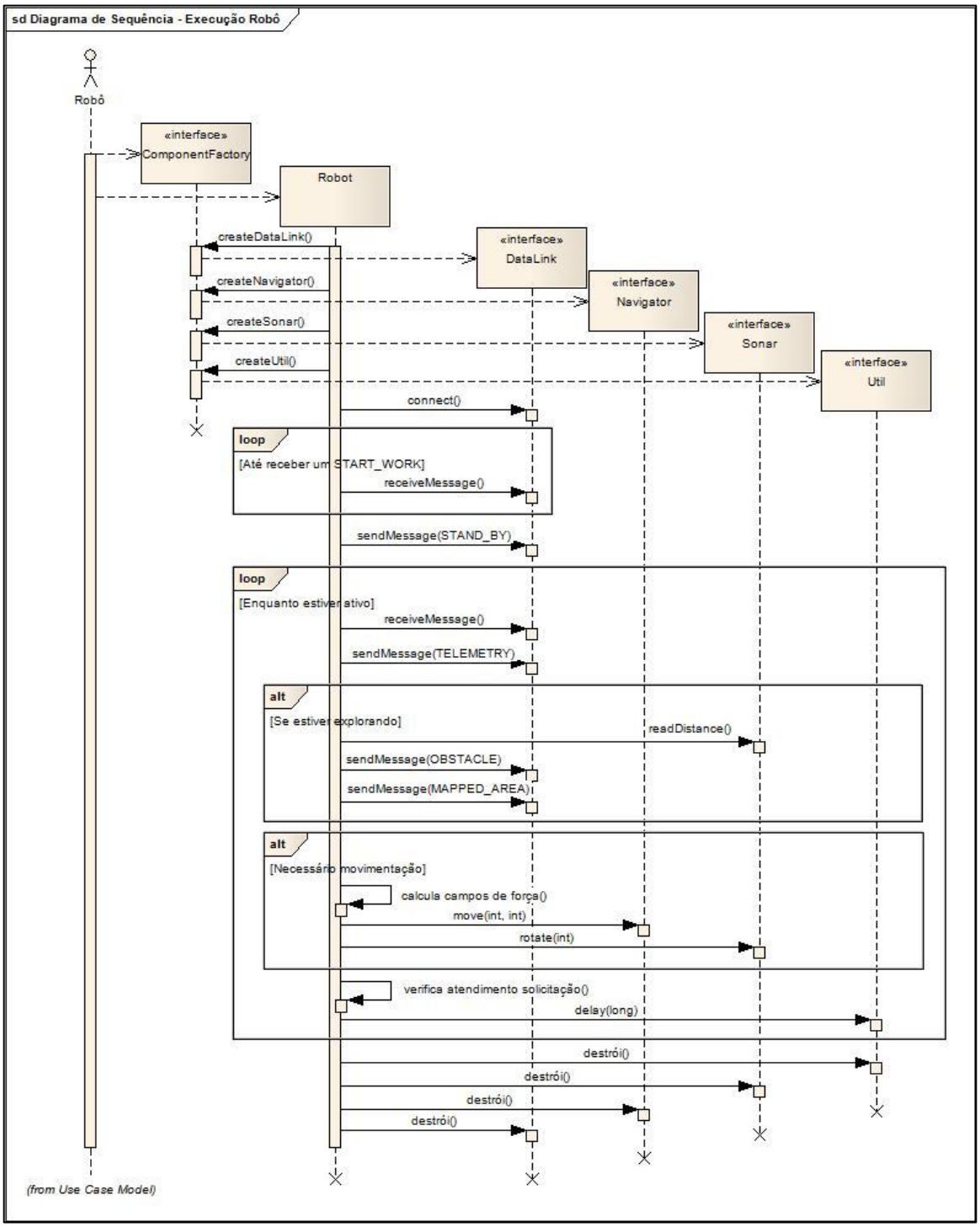


Figura 13 - Diagrama de sequência da execução do robô

Primeiramente, a classe `Robot` é criada, que por sua vez, requisita a `ComponentFactory` a criação de todos os componentes necessários. A classe `Robot` então determina que `DataLink` conecte-se a estação, e aguarda até que o sinal da estação para

iniciar a execução. Entrando no ciclo principal, `Robot` verifica se recebeu mensagens da estação com uma nova solicitação a ser realizada ou obstáculos relevantes, enviando em seguida uma mensagem contendo informações de telemetria. Após isso, caso esteja explorando a frente, verifica a distância obtida pelo sonar para determinar se existe um obstáculo e notifica a estação da nova região mapeada. Em seguida, caso esteja movimentando-se para o ponto objetivo, calcula dos campos de força. Então verifica se a solicitação foi atendida (notificando a estação se sim) e interrompe por alguns milissegundos a execução, para reiniciar o ciclo principal até receber uma mensagem para interromper a execução.

3.2.5 Especificação física do robô

A Figura 14 apresenta o robô construído para exploração. A tração é fornecida por dois motores independentes, um para cada roda traseira. As rodas dianteiras são livres para realizar a mudança de direção, permitindo que o robô realize rotações sobre o próprio eixo sem cair. O sonar é posicionado no topo, movimentado por um motor que permite a rotação de -90 a 90 graus em relação à proa do robô.

Figura 14 - Robô explorador

3.3 IMPLEMENTAÇÃO

A seguir são apresentadas as técnicas, ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

Para a implementação do sistema foi utilizado a linguagem de programação Java junto ao ambiente de desenvolvimento Eclipse.

3.3.1.1 Implementação do robô

A implementação do robô segue o diagrama de sequência apresentado na Figura 13. Nesta seção são detalhados trechos de códigos relevantes, como a leitura do sonar e detecção de obstáculos, e também a utilização dos campos de força.

A implementação da leitura do sonar, realizado pela classe `Robot`, é demonstrada no Quadro 6. Se o sonar detectou um obstáculo (linha 12), determina-se a posição dele (linha 12 até 20) e notifica-se a estação (linha 21). Depois é determinada a área que o sonar mapeou (linha 24 até 48) sendo enviada as suas coordenadas para estação (linha 51).

O Quadro 7 apresenta um trecho de código da classe `Robot` onde são utilizados os campos de força, descritos na seção 2.3.1. Primeiramente é somado o campo de força do objetivo (linha 7 e 8). Depois são somados os campos de força dos obstáculos (linha 10 até 15) que estejam pertos o suficiente para exercer influência (linha 43 até 45). Os campos dos obstáculos são do tipo tangencial, para que o robô contorne o obstáculo (linha 44 até 50). A magnitude do campo é influenciada pela proximidade com o obstáculo (linha 52 até 58).

```
/* ************************************* */
/* Detecção dos Obstáculos               */
/* ************************************* */

// DETECÇÃO DO SONAR.
int distance = sonar.readDistance();
// O ângulo correto é calculado com base no ângulo do robô com a
rotação do sonar
double sonarAngle = Math.toRadians(navigator.getRotation() +
sonar.getCurrentAngle());

if (distance < SONAR_MAX_RANGE) {
    // Determina onde está o obstáculo.
    int x = (int) (Math.cos(sonarAngle) * distance) +
            currentPosition.x;
    int y = (int) (Math.sin(sonarAngle) * distance) +
            currentPosition.y;

    Position position = new Position(x, y);
    tempObstacles.add(position);
    sendObstacleMessage(position);
}

/* ************************************* */
/* Área Mapeada                          */
/* ************************************* */
// ÁREA ATUAL
// Define qual é a área atualmente mapeada pelo sonar.
// É um triângulo onde um dos vértices está na posição corrente do robô
// Os outros dois vértices são na distância lida, considerando a
abertura do sonar.
distance = Math.min(distance, SONAR_MAX_RANGE);

double angleVertex02 = sonarAngle +
           Math.toRadians(SONAR_ANGLE_RANGE / 2);
int xVertex02 = (int) (Math.cos(angleVertex02) * distance) +
            currentPosition.x;
int yVertex02 = (int) (Math.sin(angleVertex02) * distance) +
           currentPosition.y;
Position vertex02 = new Position(xVertex02, yVertex02);

double angleVertex03 = sonarAngle -
           Math.toRadians(SONAR_ANGLE_RANGE / 2);
int xVertex03 = (int) (Math.cos(angleVertex03) * distance) +
           currentPosition.x;
int yVertex03 = (int) (Math.sin(angleVertex03) * distance) +
           currentPosition.y;
Position vertex03 = new Position(xVertex03, yVertex03);

// Envia mensagem com a área mapeada.
sendMappedAreaMessage(vertex02, vertex03);
```

Quadro 6 - Implementação da leitura do sonar

```
1  // Trecho de código omitido.
2  /* ************************************ */
3  /* Processamento dos Campos de Força    */
4  /* ************************************ */
5  // CÁLCULO DO CAMPO DE FORÇA RESULTANTE
6  Position resultForceField = new Position(0, 0);
7  resultForceField.x += (objective.x - currentPosition.x);
8  resultForceField.y += (objective.y - currentPosition.y);
9
10 for (Position obstacle : obstacles) {
11    addObstacle(currentPosition, resultForceField, obstacle);
12 }
13 for (Position obstacle : tempObstacles) {
14    addObstacle(currentPosition, resultForceField, obstacle);
15 }
16 /* ************************************ */
17 /* Alterações ambientais                */
18 /* ************************************ */
19 // ROTAÇÃO E MOVIMENTAÇÃO DO ROBO.
20 int resultAngle = (int) Math.toDegrees(calculateAngle(resultForceField.x,
21 resultForceField.y));
22 navigator.move(resultAngle, ROBOT_MOVEMENT_SPEED);
23 //Trecho de código omitido
24 /**
25  * Adiciona o obstáculo ao campo de força resultante, caso necessário.
26  *
27  * @param currentPosition a posição corrente do robô
28  * @param resultForceField o campo de força resultante
29  * @param obstacle a posição do obstáculo
30  */
31 private void addObstacle(Position currentPosition, Position resultForceField,
32 Position obstacle) {
33     // Verifica se o obstáculo está perto o suficiente para afetar o campo de
34 força.
35     double obstacleDistance = calculateDistance(currentPosition, obstacle);
36     if (obstacleDistance < OBSTACLE_RADIUS) {
37
38         // Calculo o ângulo que o obstáculo se encontra em relação a posição
39 corrente do robô.
40         int obstacleX = obstacle.x - currentPosition.x;
41         int obstacleY = obstacle.y - currentPosition.y;
42         double obstacleAngle = calculateAngle(obstacleX, obstacleY);
43
44         // Ajuste no ângulo de influência do campo do obstáculo.
45         // Caso utilizemos o valor normal, o campo do obstáculo iria repelir o
46 robô em uma direção contrária ao do objetivo.
47         // Anulando o vetor resultante.
48         // Assim, ajustá-se o ângulo em 45 graus, para fazer com que o
49 obstáculo repele o robô para o lado e não para trás.
50         obstacleAngle += Math.PI / 4;
51
52         // Calculando a intensidade do campo de repulsão.
53         // Quanto mais perto do obstáculo, mais forte será o a repulsão.
54         double repulsionIntensity = (OBSTACLE_RADIUS - obstacleDistance) *
55 OBSTACLE_REPULSE_ ;
56         // Calculando os ajustes no campo
57         resultForceField.x -= Math.cos(obstacleAngle) * repulsionIntensity;
58         resultForceField.y -= Math.sin(obstacleAngle) * repulsionIntensity;
59     }
60 }
```

Quadro 7 - Implementação dos campos de força

3.3.1.1.1 Utilização do LeJOS

A integração com o LeJOS é feita na classe `LejOSComponentFactory`, que é a fábrica de componentes LeJOS para o Robô, cujo código a apresentado no Quadro 8. São definidas constantes para a utilização das portas de motores e sensores (linha 7 até 18). O `DataLink` é criado no método `createDataLink` (linha 26 até 29). O `Navigator` é criado no método `createNavigator` (linha 31 até 36). O componente `Sonar` é criado pelo método `createSonar` (linha 38 até 48), onde também são realizados algumas leituras inicias, um correção para um problema conhecido do LeJOS em que as primeiras leituras retornam valores incorretos. Por fim, o componente `Utils` é criado no método `createUtils` (linha 50 até 53).

3.3.1.2 Implementação do gerenciamento do sistema multiagente

A classe `AgentImpl` controla a mudança de estados do agente robô como definido na seção 3.2.2. O tratamento das mensagens recebidas do agente é demonstrado no Quadro 9. O tratamento das mensagens informativas de telemetria, área mapeada e obstáculos (linha 5 até 24) não altera o estado do agente. Contudo, a representação do ambiente é alterada ao receber mensagens da área mapeada (linha 17) e obstáculo detectado (linha 23). Quando é recebida uma mensagem de aguardando solicitação, é verificado junto ao ambiente um novo ponto objetivo (linha 36), sendo o agente então notificado da nova solicitação (linha 43 até 53).

A classe `EnviromentImpl` controla o mapeamento realizado e define os pontos objetivos para as solicitações enviadas aos agentes, tendo trechos do seu código demonstrados a seguir. O método `notifyMappedArea` (Quadro 10), ao ser chamado por `AgentImpl` (Quadro 9, linha 17) primeiramente faz a união da nova área mapeada do agente com a área previamente mapeada (linha 25). Depois é verificado se essa área do agente pode ser unida com o uma área de outro agente, e em caso positivo, é realizada a união pois ambos os agentes estarão mapeando a mesma área (linha 34 até 38). O método `doUnion` realiza a invocação dos algoritmos de intersecção, união e otimização (linha 41 até 55), de modo encadeado, onde a saída de um algoritmo é passado como entrada para o outro. Caso não existe a possibilidade de união, ele retorna nulo.

```java
/**
 * Construtor de components específicos para o LejOS.
 * @author João Paulo Gonçalves
 */
public class LejOSComponentFactory implements ComponentFactory {
    /** O motor utilizado pela roda esquerda. */
    private static final Motor MOTOR_LEFT_WHEEL = Motor.A;
    /** O motor utilizado pela roda direita. */
    private static final Motor MOTOR_RIGHT_WHEEL = Motor.B;
    /** O motor utilizado pelo sonar. */
    private static final Motor MOTOR_SONAR = Motor.C;
    /** A porta de sensor utilizada pelo sonar. */
    private static final SensorPort SENSOR_PORT_SONAR = SensorPort.S1;
    /** O diâmetro da roda utilizada (em milimetros). */
    private static final int WHEEL_DIAMETER = 42;
    /** O comprimento do eixo, contando a partir do centro da roda
esquerda até o centro da roda direita (em milimetros). */
    private static final int TRACK_WIDTH = 184;

    static {
        MOTOR_LEFT_WHEEL.smoothAcceleration(true);
        MOTOR_RIGHT_WHEEL.smoothAcceleration(true);
        MOTOR_SONAR.smoothAcceleration(true);
    }

    @Override
    public DataLink createDataLink() {
        return new LejOSDataLink();
    }

    @Override
    public Navigator createNavigator() {
        TachoPilot pilot = new TachoPilot(WHEEL_DIAMETER, TRACK_WIDTH,
           MOTOR_LEFT_WHEEL, MOTOR_RIGHT_WHEEL);
        return new LejOSNavigator(new SimpleNavigator(pilot));
    }

    @Override
    public Sonar createSonar() {
        LejOSSonar sonar = new LejOSSonar(
            new UltrasonicSensor(SENSOR_PORT_SONAR), MOTOR_SONAR);
        // Realiza a leitura afim de ajudar o Sonar. Problema do Lejos
        for (int i = 0; i < 40; i++) {
            sonar.readDistance();
            Delay.msDelay(200);
        }
        return sonar;
    }

    @Override
    public Util createUtil() {
        return new LejOSUtil();
    }
}
```

Quadro 8 – Integração com o LeJOS

```
1  @Override
2  public void notifyReceive(Message message) {
3      // Atualizações dos dados de telemetria do robô.
4      switch (message.getType()) {
5          case ROBOT_TELEMETRY:
6              int x = message.readInt(1); // Deslocamento para leitura
7              int y = message.readInt(5); // do dados.
8              position = toGlobal(new Position(x, y));
9              sonarAngle = message.readByte(9);
10             robotAngle = message.readInt(10);
11             x = message.readInt(14);
12             y = message.readInt(18);
13             objective = toGlobal(new Position(x, y));
14             break;
15         case ROBOT_MAPPED_AREA:
16             sonarArea = getMappedArea(message);
17             enviroment.notifyMappedArea(this, sonarArea);
18             break;
19         case ROBOT_OBSTACLE:
20             x = message.readInt(1);
21             y = message.readInt(5);
22             Position obstacle = toGlobal(new Position(x, y));
23             enviroment.notifyObstacle(obstacle);
24             break;
25         case ROBOT_STANDBY:
26             Position newObjective = null;
27             // Mudança de Estados!!!!
28             switch (state) {
29                 case CONNECTED:
30                     newObjective = initialPosition;
31                     break;
32                 case EXPLORER:
33                     objective = null;
34                     sonarArea = null;
35                     newObjective = enviroment.getNewObjective(this);
36                     break;
37                 default:
38                     throw new UnsupportedOperationException("Invalid
39 state:" + state);
40             }
41
42             if (newObjective == null) {
43                 state = State.STAND_BY;
44             } else {
45                 newObjective = toLocal(newObjective);
46                 state = State.EXPLORER;
47                 message =
48 MessageType.STATION_OBJECTIVE.createMessage();
49                 message.writeInt(newObjective.x);
50                 message.writeInt(newObjective.y);
51                 dataLinkManager.sendMessage(message);
52             }
53
54             break;
55         default:
56             throw new UnsupportedOperationException("Invalid message:"
57 + message.getType());
58     }
59 }
```

Quadro 9 - Tratamento das mensagens do agente

```
synchronized void notifyMappedArea(Agent agent, Position[] points) {
    Set<RLineSegment2D> newMappedArea = new HashSet<RLineSegment2D>(3);

    RPoint2D point0 = new RPoint2D(points[0].x, points[0].y);
    RPoint2D point1 = new RPoint2D(points[1].x, points[1].y);
    RPoint2D point2 = new RPoint2D(points[2].x, points[2].y);

    newMappedArea.add(new RLineSegment2D(point0, point1));
    newMappedArea.add(new RLineSegment2D(point1, point2));
    newMappedArea.add(new RLineSegment2D(point2, point0));

    /* ********************************** */
    /* União da nova área mapeada         */
    /* ********************************** */
    // Verifica se este agente já tem uma área mapeada.
    // Se não tiver, esse já é a primeira área.
    if (!mappedAreas.containsKey(agent)) {
        // Primeira mapeada não deveria ser feita união
        // Otimização... talvez não seja assim.
        mappedAreas.put(agent, newMappedArea);
        return;
    }

    Set<RLineSegment2D> mappedArea = mappedAreas.get(agent);
    Set<RLineSegment2D> union = doUnion(mappedArea, newMappedArea);

    // Não houve união... descartando área mapeada
    if (union == null) {
        return;
    }

    mappedAreas.put(agent, union);

    /* ********************************** */
    /* União com os outros agente         */
    /* ********************************** */
    // Verifica se houve união com á área dos outros agentes.
    checkAreas(agent);
}

private Set<RLineSegment2D> doUnion(Set<RLineSegment2D> mappedArea,
Set<RLineSegment2D> newMappedArea) {
    Set<RLineSegment2D> union = null;

    Intersections intersections = intersectionAlgoritm.process(
          new Pair<Set<RLineSegment2D>>(mappedArea, newMappedArea));

    if (intersections != null) {
       union = unionAlgoritm.process(intersections);
       if (union != null) {
           union = optimizerAlgoritm.process(union);
       }
    }
    return union;
}
```

Quadro 10 - Tratamento área mapeada da classe `EnviromentImpl`

Já o método `getNewObjective`, apresentado no Quadro 11, ao ser chamado por `AgentImpl` (Quadro 9, linha 36), define o objetivo da solicitação de exploração a ser feita selecionando um ângulo a frente do agente em arco de 60 graus (linha 12 até 14). O ponto da solicitação também não deve estar próximo ou ocupado por um obstáculo, não ser objetivo de outro agente e estar dentro da área mapeada (linha 29 até 31).

```
1  synchronized Position getNewObjective(AgentImpl agent) {
2      objectives.remove(agent.getObjective());
3      int distance = 400;
4
5      // Cria um polígono com  área mapeada.
6      SortedPointSet pointSet =
7           sortedSetAlgoritm.process(mappedAreas.get(agent));
8      RPolygon2D polygon = new RPolygon2D(pointSet.toList());
9
10     // Determina um ângulo randômico em um cone de 60 graus a frente do
11     // robô.
12     Position currentPosition = agent.getPosition();
13     int minimum = agent.getRobotAngle() - 30;
14     int maximum = minimum + 60;
15
16     int initial = random.nextInt(maximum) + minimum;
17     int angle = initial;
18
19     /* ************************************ */
20     /* Verifica se o ângulo é viável        */
21     /* ************************************ */
22     do {
23         Position objective = new Position();
24         objective.x = (int) (Math.cos(Math.toRadians(angle)) *
25              distance) + currentPosition.x;
26         objective.y = (int) (Math.sin(Math.toRadians(angle)) *
27              distance) + currentPosition.y;
28
29         if (!objectives.contains(objective) &&
30         /**/checkObstacles(objective) &&
31         /**/polygon.contains(new RPoint2D(objective.x, objective.y))) {
32             objectives.add(objective);
33             return objective;
34         }
35
36         angle++;
37         if (angle > 180) {
38             angle = 0;
39         }
40
41     } while (angle != initial);
42
43     return currentPosition;
44 }
```

Quadro 11 - Método `getNewObjective` da classe `EnviromentImpl`

Por fim, o método `notifyObstacle` (Quadro 12), ao ser chamado por `AgentImpl` (Quadro 9, linha 23), faz a verificação se já não existe um obstáculo naquele espaço (representado por uma célula). Caso não exista, ele o adiciona (linha 11) e notifica todos os

agentes daquele obstáculo (linha 13 até 16).

```
1  synchronized void notifyObstacle(Position position) {
2      // Só permite um obstáculo em cada região de uma célula.
3      int xCell = position.x / CELL_SIZE;
4      int yCell = position.y / CELL_SIZE;
5
6      Position obstacle = new Position(
7              xCell * CELL_SIZE + (CELL_SIZE / 2),
8              yCell * CELL_SIZE + (CELL_SIZE / 2));
9
10     if (!obstacles.contains(obstacle)) {
11         obstacles.add(obstacle);
12
13         // Notifica todos agentes para adicionarem esse obstáculo.
14         for (AgentImpl agent : agents) {
15             agent.validObstacle(obstacle);
16         }
17     }
18 }
```

Quadro 12 – Método `notifyObstacle` da classe `EnviromentImpl`

3.3.1.3 Implementação da interseção e união

O algoritmo de Bentley-Ottmann é implementado na classe `BentleyOttmann` da biblioteca CompGeom. O método `process` da classe `IntersectionAlgoritm` utiliza-se desta classe, sendo demonstrado no Quadro 13. São definidas as informações que devem ser a resposta do algoritmo, como as intersecções no polígono A, intersecções do polígono B, pontos do polígono A e pontos do polígono B (linha 6 até 15). Realiza-se a intersecção, tendo o cuidado de diferenciar os segmentos de um polígono de outro (linha 52 até 63), já que o algoritmo de Bentley-Ottmann considera um segmento igual a outro somente de acordo com os seus pontos, mesmo sendo de polígonos diferentes (um segmento pode estar tanto no polígono A quanto no polígono B). Então é iterado sobre os pontos de intersecção detectados para ajustar a resposta do algoritmo, ignorando as intersecções que ocorreram entre segmentos do próprio polígono (linha 234 até 46). Encontrando uma intersecção válida, adiciona a informação dela a resposta do algoritmo (linha 46 até 54).

```
 1  public Intersections process(Pair<Set<RLineSegment2D>> input) {
 2      // Não há intersecção.
 3      if (input.m.isEmpty() || input.n.isEmpty()) {
 4          return null;
 5      }
 6      /* ********************************** */
 7      /* Criando OUTPUT                     */
 8      /* ********************************** */
 9      Map<RPoint2D, RLineSegment2D> intersectionsM = new HashMap<RPoint2D,
10  RLineSegment2D>();
11      Map<RPoint2D, RLineSegment2D> intersectionsN = new HashMap<RPoint2D,
12  RLineSegment2D>();
13      Set<RPoint2D> intersections = new HashSet<RPoint2D>();
14      SortedPointSet pointsM = converter.process(input.m);
15      SortedPointSet pointsN = converter.process(input.n);
16      /* ********************************** */
17      /* Realizando intersecção             */
18      /* ********************************** */
19      Map<RPoint2D, Set<PolygonLineSegment>> intersectionMap = intersect(input);
20      /* ********************************** */
21      /* Preenchendo OUTPUT                 */
22      /* ********************************** */
23      for (Entry<RPoint2D, Set<PolygonLineSegment>> entry :
24  intersectionMap.entrySet()) {
25          PolygonLineSegment segmentM = null;
26          PolygonLineSegment segmentN = null;
27          findSegments: for (PolygonLineSegment segment : entry.getValue()) {
28              if (segment.polygon == POLYGON_M) {
29                  segmentM = segment;
30              } else if (segment.polygon == POLYGON_N) {
31                  segmentN = segment;
32              }
33              if (segmentM != null && segmentN != null) {
34                  break findSegments;
35              }
36          }
37          if (segmentM != null && segmentN != null) {
38              RPoint2D point = entry.getKey();
39              pointsM.addPoint(point, segmentM.p1, segmentM.p2);
40              pointsN.addPoint(point, segmentN.p1, segmentN.p2);
41
42              intersections.add(point);
43              intersectionsM.put(point, segmentM.originalSegment);
44              intersectionsN.put(point, segmentN.originalSegment);
45          }
46      }
47      RPolygon2D polygonM = new RPolygon2D(pointsM.toList());
48      RPolygon2D polygonN = new RPolygon2D(pointsN.toList());
49      return new Intersections(intersections, polygonM, polygonN, intersectionsM,
50              intersectionsN);
51  }
52  private Map<RPoint2D, Set<PolygonLineSegment>>
53                      intersect(Pair<Set<RLineSegment2D>> input) {
54
55      HashSet<RLineSegment2D> allSegments = new HashSet<RLineSegment2D>();
56      for (RLineSegment2D segmentM : input.m) {
57          allSegments.add(new PolygonLineSegment(POLYGON_M, segmentM));
58      }
59      for (RLineSegment2D segmentN : input.n) {
60          allSegments.add(new PolygonLineSegment(POLYGON_N, segmentN));
61      }
62      return (Map) BentleyOttmann.intersectionsMap(allSegments);
63  }
```

Quadro 13 - Implementação de `IntersectionAlgoritm`

O método `process` da classe `UnionAlgoritm` é demonstrado no Quadro 14. Verifica-se se é possível existir união (linha 3 até 9). Tendo intersecções, junta-se todos os segmentos dos dois polígonos (linha 11 até 16) e para cada ponto de intersecção remove os segmentos desnecessários do polígono A (linha 23 até 27) e do polígono B (linha 29 até 33).

```
1  @Override
2  public Set<RLineSegment2D> process(Intersections input) {
3      /* ********************************** */
4      /* Verifição de Não União.            */
5      /* ********************************** */
6      if (input.intersections.size() < 2) {
7          // Faz-se necessário duas intersecções para existir uma união.
8          return null;
9      }
10
11     /* ********************************** */
12     /* Criando OUTPUT                     */
13     /* ********************************** */
14     HashSet<RLineSegment2D> output = new HashSet<RLineSegment2D>();
15     output.addAll(input.polygonA.getSegments());
16     output.addAll(input.polygonB.getSegments());
17
18     /* ********************************** */
19     /* Removendo Segmentos Internos       */
20     /* ********************************** */
21     for (RPoint2D intersectionPoint : input.intersections) {
22
23         // Verificando Polígono A para determinar os segmentos
24 removidos.
25         Set<RLineSegment2D> segmentsA =
26 removeSegments(input.intersections, intersectionPoint, input.polygonA,
27 input.polygonB);
28
29         // Verificando Polígono B para determinar os segmentos
30 removidos.
31         Set<RLineSegment2D> segmentsB =
32 removeSegments(input.intersections, intersectionPoint, input.polygonB,
33 input.polygonA);
34
35         output.removeAll(segmentsA);
36         output.removeAll(segmentsB);
37     }
38
39     assert checkUnion(input, output);
40     return output;
41 }
```

Quadro 14 - Implementação de `UnionAlgoritm`

3.3.1.4 Implementação da interface

Como biblioteca gráfica foi utilizada a GTGE (GOLDEN STUDIOS, 2006), uma biblioteca avançada para programação de jogos. A sua escolha é devida a facilidade de uso,

com suporte a carregamento de imagens, gerenciamento gráfico e aceleração por hardware, permitindo que a implementação da interface fosse focada nas funcionalidades, evitando esforço nos temas não relacionados a este trabalho.

A classe `ChartisGame` estende a classe `Game` da biblioteca, sendo necessário somente a implementação de três métodos para o GTGE. O primeiro é `initResources` (Quadro 15), onde é feita a carga dos recursos necessários. Para ilustrar a facilidade de utilização da biblioteca, o Quadro 15 possui o método `loadImages` da classe `AgentSprite`, (linha 9 até 13, invocado na linha 4), onde é demonstrado a simplicidade do carregamento de imagens.

```
    @Override
1   public void initResources() {
2       bsGraphics.setWindowTitle("Chartis");
3       AgentSprite.loadImages(bsLoader);
4       drawer = new Drawer(getWidth(), getHeight(), 10);
5       background = new EnviromentBackground(enviroment, drawer);
6   }
7
8   // Trecho de código da classe AgentSprite
9   public static void loadImages(BaseLoader loader) {
10      robotImage = loader.getImage("/br/furb/chartis/robot.png");
11      sonarImage = loader.getImage("/br/furb/chartis/sonar.png");
12  }
13
```

Quadro 15 - Métodos `initResoures` e `loadImages` de `ChartisGame`

O segundo método necessário para o GTGE é o `update` (Quadro 16), onde é feita a atualização dos objetos visuais. O GTGE cuida do laço de repetição para atualização da tela, invocando o método a cada período de tempo. Neste método é feita a atualização das posições de imagens para refletirem as posições dos agentes (linha 4 até 8).

```
1   @Override
2   public void update(long elapsedTime) {
3       background.update(elapsedTime);
4       if (agents != null) {
5           for (AgentSprite sprite : agents) {
6               sprite.update(elapsedTime);
7           }
8       }
9   }
```

Quadro 16 - Método `update` de `ChartisGame`

Por último, o método `render` (Quadro 17) é invocado toda vez que o GTGE requerer que o desenho da tela seja refeito. Utilizando Java2D, são desenhadas as informações de todos os agentes, tais como posição, sonar e objetivo (linha 7 até 11). Também são desenhadas as informações do ambiente (área mapeada e obstáculos) (linha 6).

```
1  @Override
2  public void render(Graphics2D g) {
3      g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
4  RenderingHints.VALUE_ANTIALIAS_ON);
5
6      background.render(g);
7      if (agents!= null) {
8          for (AgentSprite sprite : agents) {
9              sprite.render(g);
10         }
11     }
12 }
```

Quadro 17 - Método `render` de `ChartisGame`

3.3.2 Operacionalidade da implementação

Esta seção tem por objetivo mostrar a operacionalidade da implementação, demonstrando a interação do usuário com a estação conforme descrito no caso de uso `Gerenciar Estação` (Quadro 3).

A Figura 15 apresenta o início da execução da estação. O item (a) demonstra a tela de comando, onde a ação atual disponível para o usuário é apresentada. No item (b) pode-se verificar a área onde são desenhadas as informações do mapeamento, bem como as informações do robô, sendo cada pixel correspondendo a um centímetro e uma cruz desenhada indica a origem do plano. O item (c) apresenta a quantidade de quadros por segundo, que é atualizado a cada instante.

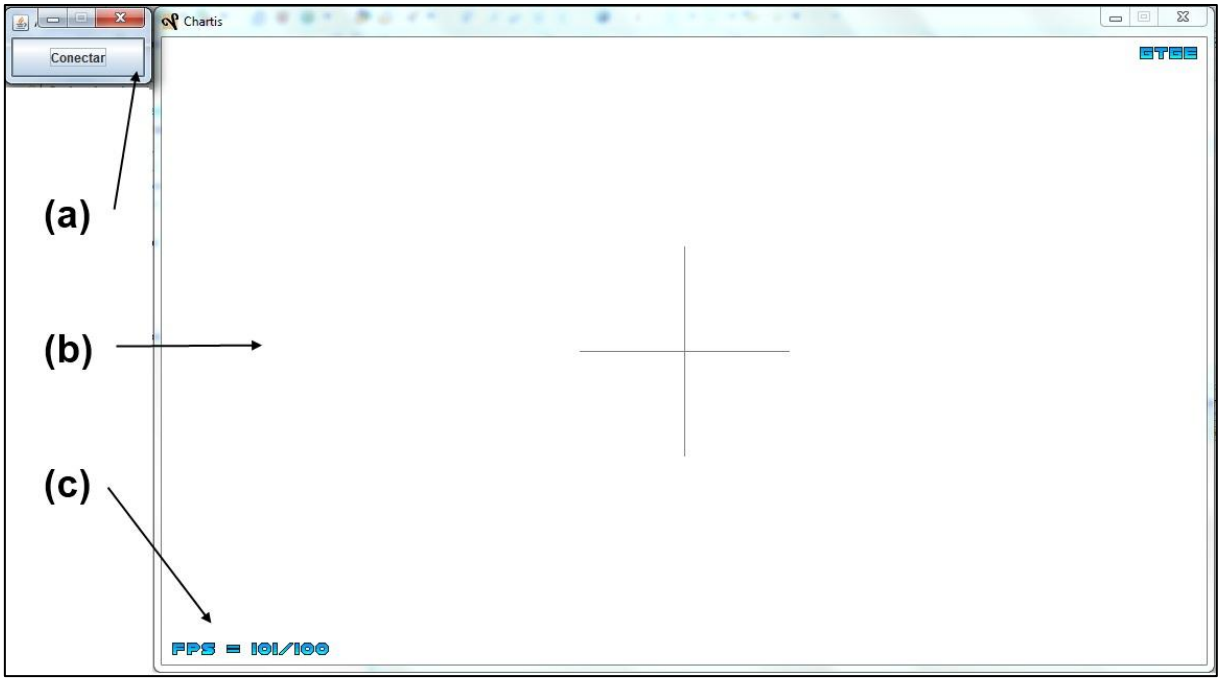


Figura 15 - Tela do gerenciador da estação no início da execução

Ao pressionar o botão `Conectar`, o usuário determina a estação para que se conecte com os agentes disponíveis, mudando a apresentação da tela para Figura 16. O item (a) demonstra a ação atual. No item (b) pode-se verificar que os agentes (em azul, no centro do plano) são desenhados conforme as suas respectivas posições correntes.

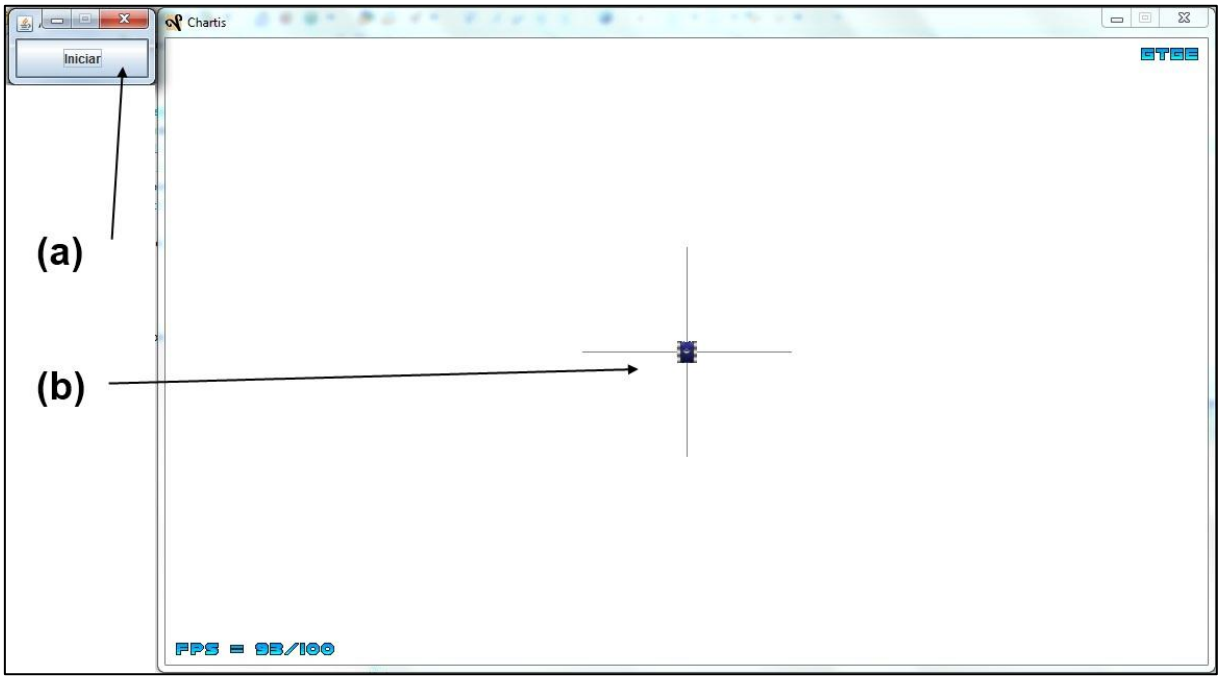


Figura 16 - Tela do gerenciador da estação com agentes conectados

A Figura 17 apresenta o comportamento da estação após o pressionamento do botão `Iniciar`. A estação realiza o gerenciamento dos agentes buscando realizar o mapeamento do ambiente. O item (a) apresenta a ação atual disponível. O item (b) demonstra a representação de obstáculos (pontos em vermelho) detectados pelos agentes. O item (c) aponta a área sendo atualmente explorada com o sonar do agente. O item (d) indica o objetivo de um agente (ponto verde), o ponto para o qual ele deve se deslocar. O item (e) representa o agente explorador; e por fim, o item (f) demonstra a área já mapeada por todos os agentes.

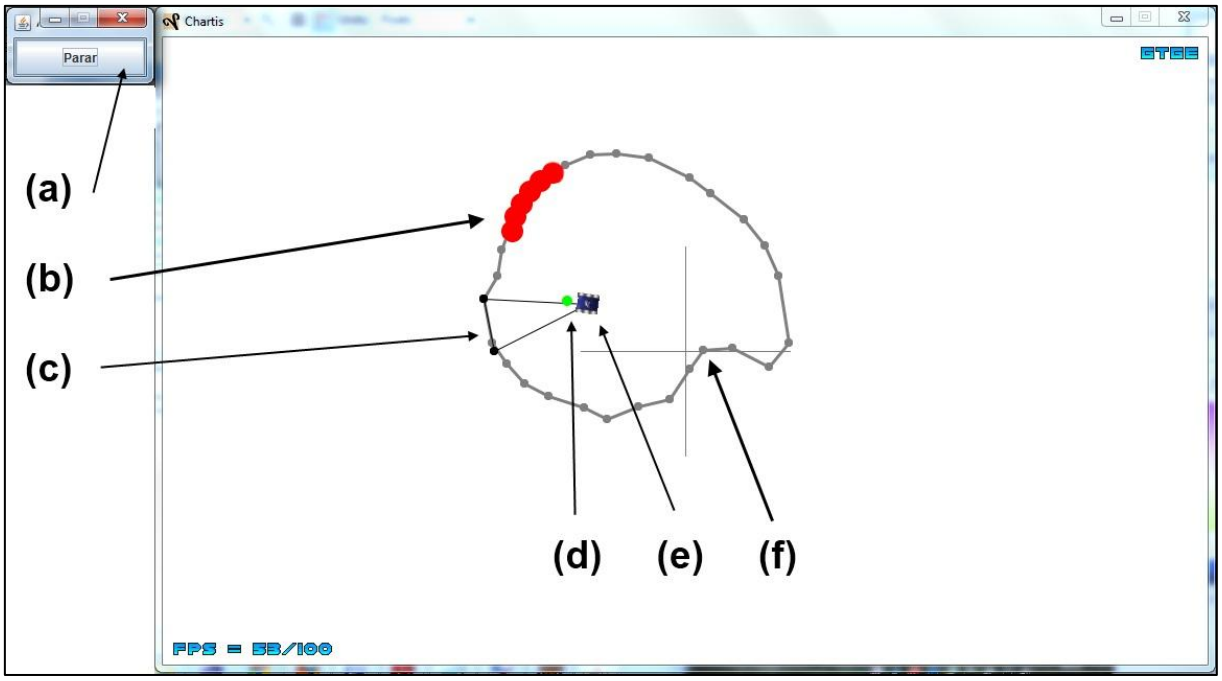


Figura 17 – Tela do gerenciador da estação com o mapeamento sendo realizado.

Pressionando o botão `Parar`, a exploração é interrompida, permanecendo desenhadas todas as informações presentes na Figura 17. A ação atual é substituída por `Fechar`, que termina a execução da estação, fechando todas as telas.

3.4 RESULTADOS E DISCUSSÃO

Para avaliar a implementação realizada foram efetuados testes de exploração e mapeamento, tanto em um ambiente simulado como em um ambiente físico.

O primeiro teste utiliza um robô simulado e cinco obstáculos em um ambiente simulado. O robô rapidamente realizou o mapeamento de três obstáculos e o sendo os dois restantes mapeados em algumas dezenas de segundos depois. A Figura 18 apresenta o resultado obtido. No item (a) tem-se o ambiente definido com os obstáculos. No item (b) o resultado obtido pelo mapeamento do robô.

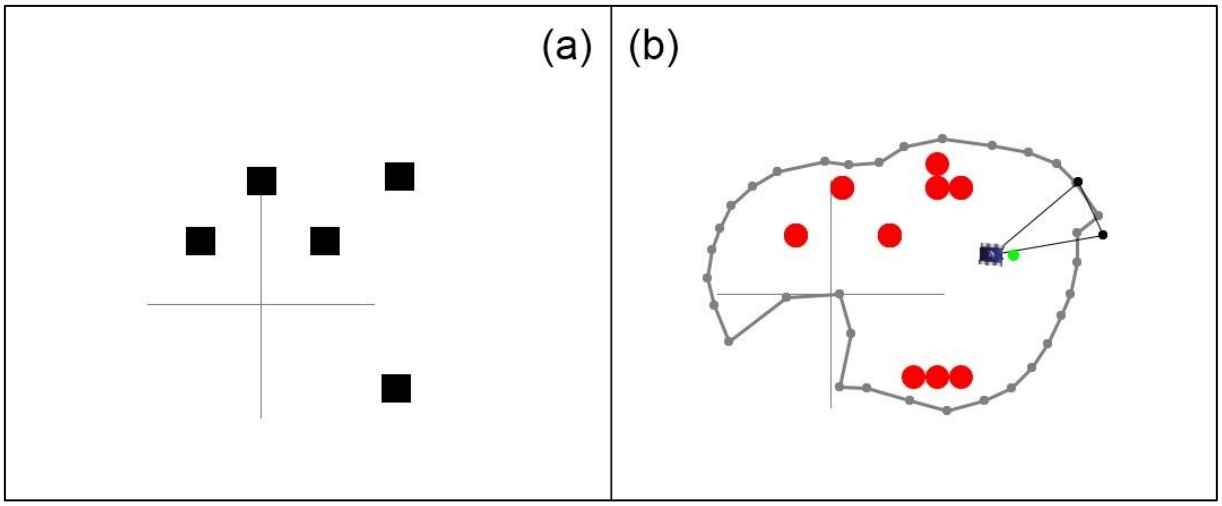


Figura 18 - Teste com um robô simulado

O segundo teste envolveu dois robôs simulados e 8 obstáculos, também utilizando um ambiente simulado, cujo resultado é demonstrado na Figura 19. Novamente, o item (a) demonstra o ambiente definido com os obstáculos, enquanto o item (b) o resultado obtido pelo mapeamento dos robôs. Pode-se notar que o obstáculo `o1` ainda não foi mapeado pelos robôs.

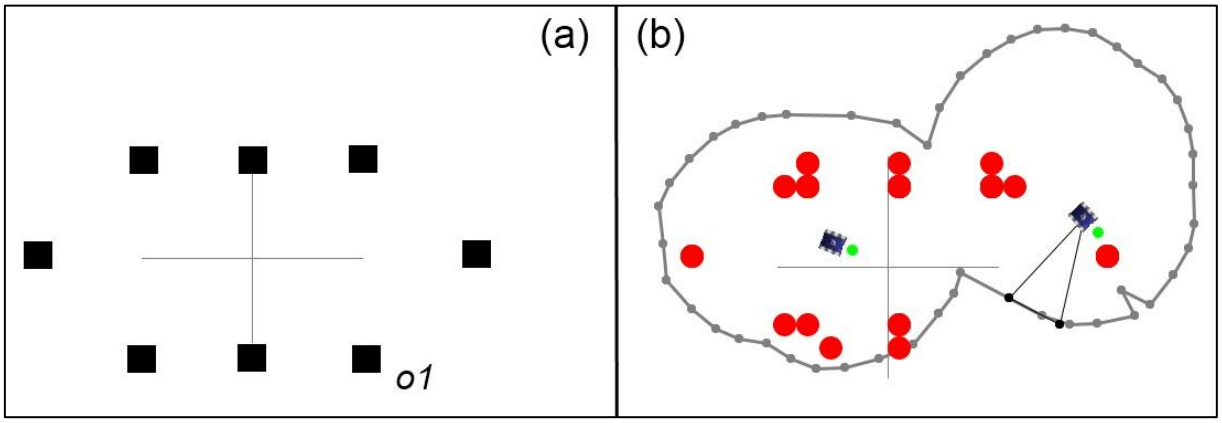


Figura 19 - Teste com dois robôs simulados

O terceiro teste utilizou um robô LEGO Mindstorms com um obstáculo, demonstrando que a implementação realizada funciona tanto em um ambiente simulado quanto real. Na Figura 20 pode ser visualizado o ambiente testado, com um livro fazendo o papel de obstáculo. O mapeamento realizado pelo robô é apresentado na Figura 21.

Figura 20 – Ambiente de teste com um robô físico

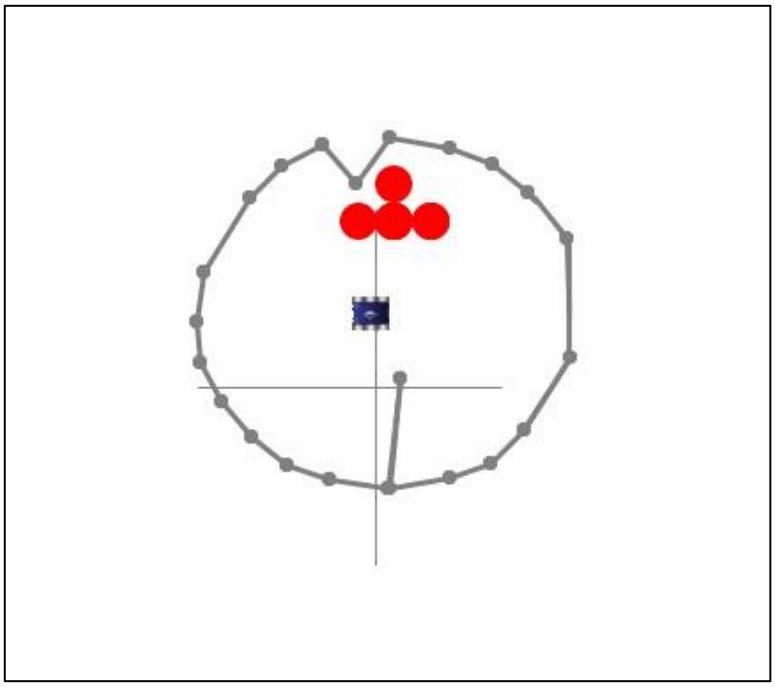


Figura 21 - Mapeamento realizado pelo robô físico

Não foi possível realizar testes com mais de um robô real, devido a uma limitação do *Bluetooth* do LeJOS, que não consegue estabelecer múltiplas conexões com vários NXT ao mesmo tempo. Entretanto, em função do modo como o sistema foi especificado e implementado (permitindo que um mesmo código de robô seja executado tanto em um ambiente simulado quando em um robô físico), uma vez vencida esta limitação, a utilização de vários robôs físicos é plenamente possível.

Em relação aos trabalhos correlatos, pode se notar que Silva Júnior(2003) utilizou

apenas um robô (agente), enquanto este trabalho permite utilizar vários, organizados em um sistema multiagente. De semelhanças, tanto esse trabalho quando o de Silva Júnior(2003) utilizaram campos de força para navegação, sendo somente diferenciando a forma de cálculo. Em relação ao trabalho Burgard et al. (2000), nota-se semelhanças de abordagem ao utilizar tanto simulação quanto robôs reais. Já os trabalhos de Cohem (1996) e Soares e Campos (2006) limitam-se a simulações, não utilizando robôs reais.

4 CONCLUSÕES

O presente trabalho possibilitou desenvolver um sistema multiagente capaz de realizar a exploração e o mapeamento de um ambiente desconhecido. De modo geral os objetivos deste trabalho foram atendidos. Foi construído um sistema multiagente robótico, com robôs criados a partir do *kit* LEGO Mindstorms, capazes de explorar um ambiente movimentando-se através de campos de força. Uma aplicação estação foi construída para gerenciar o sistema multiagente e possibilitar ao usuário visualizar o mapeamento realizado na forma de polígonos.

Os campos de força se demonstraram adequados para a movimentação dos robôs, sendo o seu desenvolvimento feito de forma clara. O algoritmo de Bentley-Ottmann apresentou-se efetivo na detecção de intersecções, assim como a utilização de polígonos simplificou a representação da área mapeada.

As operações geométricas inicialmente sofreram com problemas de exatidão, mas foram totalmente sanados com a utilização da biblioteca CompGeom. A biblioteca LeJOS demonstrou-se problemática, tanto a máquina virtual para o LEGO quanto para a integração com o computador. Métodos da biblioteca não suportados, erros de implementação e falta de documentação atrasaram o desenvolvimento da integração com o robô, tornando demorado o diagnóstico de problemas.

A disponibilidade do *kit* LEGO Mindstorms também se mostrou um problema, não estando sempre disponível para desenvolvimento e testes. Isto ocorreu em função da quantidade limitada de *kits* (8 unidades) e de sua utilização pela disciplina de Robótica. Esta limitação forçou a implementação dos agentes simulados, assim como o desenvolvimento da implementação desacoplada dos robôs, o que se mostrou posteriormente um ponto positivo, visto que é possível realizar o mapeamento também em um ambiente simulado.

A utilização da biblioteca GTGE foi aproveitada de trabalhos anteriores desenvolvidos pelo autor deste trabalho, facilitando o desenvolvimento. A aplicação serviu bem ao seu propósito, permitindo visualizar o progresso do mapeamento bem como as ações do robô, sendo também útil na validação e depuração durante a construção do trabalho.

Por fim, este trabalho possui algumas limitações como a inabilidade da rotina da união de polígonos tratar polígonos com buracos, a definição dos pontos objetivos não ser otimizada e a incapacidade de determinar que o mapeamento do ambiente foi concluído através da exploração. Também se fez necessário um controle central representado pela estação, devido

a limitações do LEGO Mindstorms em tratar rotinas complexas com a intersecção e união de polígonos. Todas essas limitações são sugeridas como extensão para teste trabalho.

4.1 EXTENSÕES

Sugerem-se as seguintes extensões para a continuidade do trabalho:

a) adicionar a capacidade para após o fim da exploração, o usuário possa determinar pontos no ambiente para onde os robôs devem se mover. Neste caso, a estação enviaria a solicitação aos robôs, que se deslocariam pelo ambiente utilizando somente dados do mapeamento realizado;
b) melhorar a definição do ponto objetivo de uma solicitação, buscando otimizar a exploração. Um exemplo seria a utilização de uma função para determinar qual o ponto da área mapeada está mais próximo ou possui o maior valor agregado a ser explorado, como na abordagem de Burgard et al. (2000, p. 481);
c) refinar o algoritmo de união de polígonos, que não está plenamente estável e não trato o caso de união de polígonos que resultam em um polígono com buracos internos;
d) avaliar uma representação da área mapeada alternativa, como por exemplo, uma *grid* de ocupação; buscando determinar as vantagens e desvantagens em relação a representação atual;
e) transportar o cálculo da área mapeada para os agentes. Isto eliminaria a necessidade de uma estação. Entretanto, requereria a implementação adicional da capacidade comunicação entre eles, para que consigam compartilhar o mapeamento;
f) utilizar um algoritmo de sincronia em sistemas distribuídos, como o *token-ring*, para controlar o uso do sonar entre os robôs, já que o uso do sonar por parte de um robô pode causar interferência no sonar de outro;
g) ajustar a rotina de avaliação de obstáculos, para notificar a posição de cada robô como um obstáculo para os outros, modificando este obstáculo a cada alteração de posição do robô;
h) adicionar maiores capacidades a interface do usuário, como *zoom*, deslocamento da área mapeada, definição do ponto inicial de um robô, como também apresentar

mais detalhes do robô como carga da bateria, qualidade do sinal, caminho percorrido etc.;

i) criar um interface para que o usuário possa criar um ambiente com obstáculos, para ser utilizado junto ao agente simulado;

j) investigar com uma maior profundidade a biblioteca de comunicação *Bluetooth* do LeJOS para verificar se existe uma alternativa de estabelecer comunicação com mais de um robô ao mesmo tempo.

REFERÊNCIAS BIBLIOGRÁFICAS

BENTLEY, Jon L.; OTTMANN, Thomas A. Algorithms for reporting and counting geometric intersections. **IEEE Transactions on Computers**, Washington, v. 28, n. 9, p. 643-647, Sept. 1979. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1675432>. Acesso em: 08 jun. 2011.

BURGARD, Wolfram et al. Collaborative multi-robot exploration. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, [16th], 2000, São Franscisco. **Proceedings**… Piscatway: IEEE, 2000. p. 476-481. Disponível em: <http://www.dtic.ua.es/asignaturas/STF/art6.pdf>. Acesso em: 21 set. 2010.

COHEN, Wìlliam W. Adaptive mapping and navigation by teams of simple robots. **Robotics and Autonomous Systems**, [S.l.], v. 18, n. 4, p. 411-434, Oct. 1996. Disponível em: <http://reference.kfupm.edu.sa/content/a/d/adaptive_mapping_and_navigation_by_teams_94504.pdf>. Acesso em: 21 set. 2010.

DUDEK, Gregory et al. A taxonomy for multi-agent robotics. **Autonomous Robot**, Boston, v. 3, n. 4, p. 375-397, Dec. 1996. Disponível em: <http://users.cs.dal.ca/~eem/cvWeb/pubs/kswarm.pdf>. Acesso em: 21 set. 2010.

GOLDEN STUDIOS. **Golden T game engine**. [S.l.], 2006. Disponível em <http://goldenstudios.or.id/products/GTGE/>. Acesso em: 08 jun. 2011.

JENNINGS, Nicholas R.; SYCARA, Katia; WOOLDRIDGE, Michael. A roadmap of agent research and development. **Autonomous Agents and Multi-Agent Systems**, Boston, v. 1, n. 1, p. 275-306, Mar. 1998. Disponível em: <http://userpages.umbc.edu/~amrish1/TimeLine/roadmap%20of%20agent%20research.pdf>. Acesso em: 21 set. 2010.

KIERS, Bart. **CompGeom**. [S.l.], 2011. Disponível em: <http://big-o.nl/apps/compgeom/>. Acesso em: 08 jun. 2011.

LEGO GROUP. **Mindstorms**. [S.l.], 2010. Disponível em: <http://mindstorms.lego.com/en-us/Default.aspx>. Acesso em: 08 jun. 2011.

LEJOS. **LeJOS, Java for lego mindstorms**. [S.l.], 2010. Disponível em: <http://lejos.sourceforge.net/>. Acesso em: 08 jun. 2011.

MURPHY, Robin R. **Introduction to AI robotics**. Cambridge: Mit Press, 2000.

SOFTSURFER. **The Intersections for a set of 2D segments, and Testing simple polygons**. [S.l.], 2006. Disponível em: <http://softsurfer.com/Archive/algorithm_0108/algorithm_0108.htm>. Acesso em: 08 jun. 2011.

RUSSELL, Stuart J.; NORVIG, Peter. **Inteligência artificial**. 2. ed. Rio de Janeiro: Elsevier, 2004.

SILVA JÚNIOR, Edson P. **Navegação exploratória baseada em problemas de valores de contorno**. 2003. 109 f. Tese (Doutorado em Ciência da Computação) - Pós-Graduação em Ciência da Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre. Disponível em: <http://www.lume.ufrgs.br/bitstream/handle/10183/3819/000393490.pdf?sequence=1>. Acesso em: 09 nov. 2010.

SOARES, Rodrigo G. F.; CAMPOS, André M. C. Multiagent exploration task in games through negotiation. In: SIMPÓSIO BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL. 5., 2006, Recife. **Anais eletrônicos...** Recife: Sociedade Brasileira de Computação, 2006. Não paginado. Disponível em: <http://www.sbgames.org/papers/sbgames06/25.pdf>. Acesso em: 21 set. 2010.

WOOLDRIDGE, Michael. Intelligent agents. In: WEISS, Gerhard. **Multiagent systems:** a modern approach to distributed artificial intelligence. 2. ed. Cambridge: The MIT Press, 2000. p. 27-77.

ZLOT, Robert et al. Multi-robot explorationcontrolled by a market. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, [18th], 2002, Washington. **Proceeding**… Piscataway: IEEE, 2002. p. 3016-3023. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.16.2667&rep=rep1&type=pdf>. Acesso em: 21 set. 2010.