Simulador extensível para navegação de agentes baseado em inteligência de enxames

Danilo Nogueira Costa

Simulador extensível para navegação de agentes baseado em inteligência de enxames[1]

Danilo Nogueira Costa

Orientador: ***Prof. Dr. Eduardo do Valle Simões***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional.

USP – São Carlos
Fevereiro/2007

[1]Trabalho realizado com auxílio financeiro do CNPq.

àquela que não saiu do meu lado um só minuto durante toda essa jornada, acreditando, apostando, incentivando e não me deixando desistir, abrindo mão de muitas coisas, e se dedicando a esse trabalho tanto quanto eu, algo imprescindível para a conclusão desse projeto.
ao meu Amor, Tanecy.

Agradecimentos

Inicialmente Àquela que faz e Se regozija, fonte incomensurável de poder, *Shri Adi Shakti Nirmala Devi*

à minha querida esposa por tudo, tudo mesmo! Tanecy.

ao meu inicialmente orientador, e atualmente amigo, Eduardo Simões, por tudo aquilo que me ensinou. Lições que levarei para a vida inteira.

aos meus amados pais, Fernando e Eliana, e queridos irmãos, Bruno e Carolina, que mesmo à distância sempre fizeram o impossível para investir na minha educação.

ao meu tio Clóvis e sua bela família que sempre apoiaram minhas decisões.

aos meus queridos sogros, Cleide e Tadeu, por todo o apoio e encorajamento.

ao meu colega de laboratório e amigo, Mauro Miazaki, por toda amizade e ajuda, desde não me deixar perder os prazos para entrega de documentos até valiosas críticas e sugestões ao trabalho.

por fim, mas não menos importante a minha querida cunhada e irmã, Analuiza, pelo carinho, incentivo e ajuda na pesquisa encontrando e me trazendo um livro enorme desde Lavras até São Paulo.

Resumo

A visão de muitas pessoas sobre uma colônia de formigas, em geral, é de que estes pequenos e inofensivos insetos somente se movem aleatoriamente para coletar alimento e conservá-los em seus ninhos. Um olhar destreinado não conseguiria notar o nível de complexidade e organização que é requerido por uma colônia de formigas para sua sobrevivência. Uma formiga simples é parte de um grande grupo que coopera entre si para criar um superorganismo. Sem uma autoridade central ou indivíduos com habilidade de um pensamento cognitivo complexo, a colônia se auto-organiza, e, de fato, ajusta seus recursos de uma maneira muito eficiente.

Essa dissertação investiga o papel da comunicação indireta nas tarefas de exploração e forrageamento, e como isso afeta as decisões de um agente simples e traz um comportamento emergente útil à toda colônia. Por fim, este trabalho implementa uma plataforma de simulação multi-agente inspirado em formigas.

Abstract

Most people's view of an ant colony and ants in general is that they simply pose harmless little insects that move randomly and gather food in their underground nests. The untrained eye would have never guessed the level of complexity and organisation that is required in order for an ant colony to survive. The simple ant is a part of a huge group, which cooperate one superorganism. Without any central authority or the ability of complex cognitive thought from the individuals, the colony seems to self organise and in fact adjust its resources in a quite efficient way.

This dissertation investigates the role of indirect communication in the exploration and forage task and how it affects the decisions of the single agent and brings an emergent behaviour that is useful to all the colony. Finally this work implements an ant inspired multi-agent simulation plataform.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Recentemente a biociência tem causado um impacto significativo na matemática e na computação, criando uma nova área, a computação bioinspirada, a qual usa princípios biológicos para melhorar os processos tecnológicos e de informação (Shtovba, 2005). Desse modo, cientistas da computação iniciaram seus estudos sobre o comportamento de grupos de animais sociais (pássaros, peixes, abelhas, formigas, cupins, entre outros), observando como tais animais interagem, atingem objetivos e evoluem, com o intuito de modelar computacionalmente os enxames biológicos e com isso extrair modelos matemáticos aplicáveis nas diferentes áreas do conhecimento. Este comportamento de enxame tem sido utilizado como uma solução de problemas de otimização (Shtovba, 2005), com aplicações em sistemas de telecomunicação (Bonabeau et al., 1999) (Vittori, 2005), robótica (Arkin, 1998) (Beni e Wang, 1989), padrões de tráfego em sistemas de transporte e aplicações militares (Pachter, 1998).

Um olhar mais atento sobre um enxame, de formiga ou cupim, por exemplo, sugere que as atividades exercidas pelos diversos indivíduos deste grupo são atos cooperativos para atingir um determinado objetivo, ainda que cada indivíduo não seja consciente desse objetivo global. Esta inteligência coletiva surge de grandes grupos de indivíduos relativamente simples que usam regras simples para governar suas ações e, via interações do grupo, o enxame alcança sua meta (Gordon, 1996). Assim, um tipo de auto-organização surge do total de ações desempenhadas individualmente.

Sendo a Inteligência de Enxames (IE) (*Swarm Intelligence*) definida como a inteligência coletiva emergente de um grupo de agentes (Bonabeau et al., 1999), em que cada agente é um subsistema que interage com o meio ambiente, independente da ação dos outros indivíduos do grupo. Porém, numa escala maior, eles acabam por realizar uma tarefa coordenada de objetivo comum (Martinoli, 1999). O agente autônomo não segue comandos de um líder, ou um plano global (Flake, 1999). Por exemplo, para um pássaro participar de um bando em vôo, ele somente ajusta o seu movimento em coordenação com o movimento do grupo, e permanece próximo de outros do bando mas evita uma colisão com eles. Nenhum pássaro recebe ordens de um pássaro líder uma vez que tal figura não existe. Qualquer pássaro pode voar na frente, no centro ou atrás do bando; esse comportamento coletivo ajuda os pássaros a terem vantagem em muitas tarefas, incluindo proteção de predadores (especialmente para os pássaros no meio do grupo) e procura por comida (Kennedy et al., 2001).

Dentre os diversos grupos de animais sociais, as colônias de formigas têm recebido grande atenção da comunidade científica, particularmente no método pelo qual uma colônia de formigas determina a fonte de comida que seja mais vantajosa (Beckers et al., 1993), a que possua o tipo de alimento mais requerido pela colônia Portha et al. (2002) e a mais próxima do ninho, dentre diversas disponíveis (Aron et al., 1990), (Goss et al., 1989). Outra característica notável é a capacidade de realizarem tarefas complexas, tais como a construção de ninhos e formação de pontes. O sistema das formigas como um todo se mostra robusto e adaptativo às mudanças do ambiente, quando por exemplo, se por algum motivo parte de seu ninho é destruído, este é restaurado em pouco tempo e o equilíbrio é novamente estabelecido (Hölldobler e Wilson, 1990). Todas estas atividades surgem da interação de indivíduos simples, sem controle centralizado, sem mapeamento prévio do ambiente e comunicação global, e ainda assim, esta e outras colônias de insetos são capazes de solucionar problemas complexos (Bonabeau et al., 1997), (Bonabeau et al., 1999), (Camazine et al., 2001).

Proposto no começo dos anos 90, o método de Otimização Baseada em Colônias de Formigas, *Ant Colony Optimization* (ACO), apresenta algoritmos de formigas (*ant*

algorithms) como técnica para solucionar problemas possíveis de serem decompostos e explorados por agentes, que tenham uma natureza dinâmica e não requerem tempo limite ou soluções ótimas (Dorigo et al., 1991), (Dorigo, 1992). Este algoritmo de formiga pode ser atribuído à computação bioinspirada, pois se fundamenta em mecanismos de auto-organização de insetos sociais e se tornou um importante campo da teoria da otimização (Shtovba, 2005), inicialmente aplicado ao problema do caixeiro viajante, que é baseado em grafos (Dorigo e Gambardella, 1997), (Dorigo et al., 1991).

Um dos modos de se estudar os diversos sistemas de enxames tem sido através da simulação (Camazine et al., 2001), (Reynolds, 1987). Tais simulações têm utilizado mecanismos importantes de sistemas distribuídos, incluindo retroalimentação positiva, estimergia, gradientes de sinais, informação de posição, divisão de tarefas, estabilidade, adaptabilidade, entre outros (Yamins, 2005). No âmbito da inteligência coletiva, as principais contribuições da engenharia para os biólogos também foram o desenvolvimento de simuladores (Langton, 1989), (Deneubourg et al., 1990), (Epstein e Axtell, 1996).

Uma simulação bastante conhecida de um bando de pássaros foi publicada por Reynolds (1987), no qual ele supôs que um bando era guiado por três forças locais: colisão evitável, velocidade comum e movimento em direção ao centro. Implementando somente essas três condições, seus programas mostraram, através dos agentes simulados denominados *boids*, comportamentos muito realísticos de pássaros, girando no espaço tridimensional simulado, separando-se do grupo frente a um obstáculo e juntando-se novamente.

De acordo com Kennedy et al. (2001) em simulações de grupos como um bando de pássaros ou um cardume, o mais importante a ser simulado é o movimento coordenado dos organismos. O desejo de se estudar os aspectos sociais do comportamento e a criação de efeitos gráficos atraentes e realistas, configuram algumas das razões para se dedicar a tal tópico. Um motivo mais pragmático é o de aprender sobre os movimentos coordenados com regras de descentralização simples, para então, por exemplo, modelar e desenvolver sociedades de robôs que implementarão tarefas de exploração de ambi-

entes. Assim, o sistema descentralizado tem as vantagens: (i) de que a quantidade de tarefa pode ser distribuída entre um certo número de trabalhadores, (ii) a modelagem de cada indivíduo pode ser mais simples que a modelagem de um único agente autônomo mais complexo, e (iii) a capacidade de processamento necessária poderá ser menor e de menor custo.

Também no campo da robótica alguns estudos sobre auto-organização de robôs são primeiramente implementados em *software* para que através da simulação apropriada do robô real os pesquisadores possam, previamente, encontrar e corrigir falhas e assim evitar maiores gastos com execuções em sistemas reais (Fukuda, 1998), (Hodgins e Brogan, 1994). Adicionalmente, para propósitos de pesquisa é sabido que modelar e desenvolver um simulador computacional de agentes simples geralmente é menos dispendioso do que implementá-lo fisicamente (por exemplo, o caso de robôs reais, especialmente quando estes são em grande número).

Com o crescente interesse de diversos pesquisadores em estudar os comportamentos de insetos sociais e aplicá-los a problemas de otimização e navegação, várias plataformas de simulação baseadas em agentes tem sido propostas (Railsback et al., 2006), entre elas o Swarm, o Repast, o Manson e o NetLogo. Analisando-se detalhadamente essas plataformas, pode-se constatar algumas deficiências específicas em cada uma delas: (i) o *Swarm* é o mais antigo, escrito em Objective-C, de difícil domínio e, consequentemente, difícil alteração de funcionalidades; (ii) o *Java Swarm* é um reprojeto do Swarm para permitir acesso a sua biblioteca através de interfaces em Java, comprometendo com isso seu desempenho devido ao encapsulamento, e aumentando a dificuldade de alteração devido à necessidade do domínio de duas linguagens de programação; (iii) o *Repast* é voltado ao domínio específico das ciências sociais; (iv) o *Manson* é uma alternativa mais compacta e de maior desenpenho, contudo ainda se encontra em fase de desenvolvimento, não possuindo todas as funcionalidades e nem estando inteiramente documentado; (v) o *NetLogo* que é uma solução madura e profissional, com ampla funcionalidade, apresenta o problema de ser de código proprietário, dificultando sua extensão. Sendo assim, o pesquisador da área de inteligência computacional encontra

dificuldades na utilização dessas ferramentas para simular seus modelos de IE.

Desta maneira, este trabalho tem como objetivo implementar um simulador de sistema multi-agente baseado em IE, mais especificamente em Colônias de Formigas, voltado à realização de experimentos nas áreas de inteligência computacional e robótica. Para isso, esse simulador deve apresentar as seguintes características: implementação em linguagem Java, o que facilita sua futura extensão; ser voltado para simulação de colônias de formigas em tarefas de exploração e forrageamento; disponibilizado com código aberto; e disponibilização de uma interface para parametrização do modelo de IE, permitindo a execução de experimentos com o modelo pré-implementado, sem a necessidade de programação por parte do usuário.

Esta ferramenta visa a realização de experimentos simulados para análise de eficiência de algoritmos que deverão ser embarcados, em pesquisas futuras, em robôs reais que se utilizam de IE entre outros métodos, com o propósito de realizar tarefas de navegação e exploração de ambientes. Como objetivos específicos desse trabalho pretende-se: implementar e parametrizar algoritmos de agentes móveis inspirados em enxames de formigas, explorando, principalmente, a tarefa de forrageamento; e disponibilizar a ferramenta na forma de código aberto a outros pesquisadores para a realização de experimentos com IE.

A estrutura geral da dissertação é dada a seguir, com um breve resumo do que é discutido em cada capítulo:

- O Capítulo 2 - Inteligência de Enxames apresenta a pesquisa que foi conduzida a fim de prover a base teórica para essa dissertação. As principais características do comportamento de formigas reais e artificiais, e uma revisão das aplicações da Inteligência de Enxames no mundo real.

- O Capítulo 3 - Plataformas para Simulações Baseadas em Agentes discorre sobre os simuladores existentes, suas principais características e problemas, para apresentar o estado da arte na área de implementação de simuladores aplicados à experimentos com IE, destacando as principais idéias que darão origem à ferramenta proposta.

- O Capítulo 4 - Projeto e Desenvolvimento apresenta a modelagem do simulador desenvolvido nessa pesquisa, sua parametrização e interface com o uso de diagramas em UML.

- O Capítulo 5 - Testes e Resultados apresenta os experimentos realizados e os resultados obtidos. O raciocínio por trás dos experimentos escolhidos é apresentado e discutido com alguns resultados esperados, inspirados em outros autores.

- O Capítulo 6 - Conclusão apresenta as conclusões oriundas dos trabalhos realizados nesta dissertação e algumas recomendações de pesquisa futura são mostradas aqui.

Capítulo 2

Inteligência de Enxames

O presente capítulo apresenta a Inteligência de Enxames, tema que vem mobilizando grupos de pesquisas nas mais conceituadas universidades dos Estados Unidos e Europa a modelarem computacionalmente o comportamento de grupos, ou enxames de animais, para o emprego destes em problemas de otimização, robótica e outras áreas. A introdução do capitulo traz exemplos de diversos comportamentos animais que já foram estudados e permanecem como fonte de inspiração para a ciência da computação. Visto que este trabalho pretende simular comportamentos de colônias de formigas, no que tange à busca por alimentos. É imperativo apresentar os fundamentos relativos às propriedades biológicas das formigas como indivíduos ou colônia, os algoritmos de formigas inspirados em colônias reais, bem como suas aplicações.

A formação e as interações que ocorrem em um enxame, simultaneamente trazem algum benefício ao indivíduo e ao grupo, mesmo que não se tenha consciência disso (Deneubourg et al., 1987). Na vida gregária de muitas espécies, os animais mais velhos ajudam a cuidar dos mais jovens coletivamente, esse comportamento além de prover segurança quando ocorre o ataque de um predador, também facilita o compartilhamento de comida e os sinais de advertência entre o grupo. Da mesma maneira, são exemplos de um benefício à sobrevivência um peixe nadando com o cardume, ou pássaros voando em bandos, pois estes estão mais seguros dos predadores do que nadando ou voando sozinhos; e, quando atacados, tanto o cardume quanto o bando podem se dividir e

evitar o predador.

O potencial para otimização de um comportamento simples tem sido bastante notado em estudos de insetos, em particular no comportamento dos insetos sociais. Análises como a de Hoskins (1995) mostraram como comportamentos simples dos organismos mais singelos do reino animal podem funcionar como algoritmos de otimização. Seu estudo aponta que a bactéria *E. coli* talvez tenha o comportamento inteligente mais simples que se possa imaginar (Hoskins define inteligência como a habilidade do organismo de controlar sua distribuição no ambiente). Essa bactéria é capaz de dois tipos de locomoção, um chamado "correr" e o outro "rolar". O "correr" é um movimento para frente por meio da rotação anti-horária da flagela enquanto que o "rolar" ocorre quando a flagela é girada em sentido horário. A mudança faz o grupo separar, então a próxima "corrida" se inicia numa direção aleatória. A célula "rola" mais freqüentemente na presença de uma química adversa, e a mudança completa de direção é suficiente para aumentar as chances de sobrevivência da bactéria, dando-lhe a capacidade de escapar das toxinas. Outro exemplo de comportamento simples, porém adaptativo (explicado adiante na seção 2.2.2), é o mecanismo exclusivo de mobilidade de uma ameba, que está baseado em sua fluidez, efetivada por alternar a maciez e rigidez do protoplasma, esta forma simples de locomoção consiste em evitar um estímulo nocivo e se aproximar de um estímulo positivo (Lorenz, 1973).

No caso de um inseto, mesmo com poucas centenas de células cerebrais, suas organizações são capazes de admiráveis arquiteturas, um sistema de comunicação elaborado e uma espantosa resistência às ameaças da natureza. Wilson (1978) iniciou seus estudos sistemáticos sobre o comportamento de formigas em 1953 e teorizou que o brilhante sucesso das sociedades de formigas pode ser explicado e entendido em termos de um elementar padrão fixo de ação - uma resposta ao feromônio - uma substância química que possui um odor possível de ser detectado por outras formigas. Na tarefa de forrageamento cada formiga procura por alimento, e assim que o encontra ela retorna ao ninho depositando ao longo do caminho uma trilha de feromônio. Outras formigas detectam a trilha e seguem-na até a fonte de alimento. Ao retornar, elas também emi-

tem o feromônio até chegarem à colônia, reforçando assim a trilha. Como resultado, a concentração de feromônio é maior próxima à fonte de alimento, dessa maneira sem uma supervisão central a colônia de formigas localiza a fonte de comida mais próxima, otimizando sua busca (Beckers et al., 1992).

2.1 Formigas Naturais

As formigas são insetos sociais da família *Formicidae* da ordem *Hymenoptera*, que também inclui as abelhas e vespas. O corpo de uma formiga é composto de três partes principais: cabeça, com um par de antenas (que é o órgão sensorial fundamental), olhos e mandíbulas; tórax, com três pares de pernas articuladas; e abdome, com o sistema digestivo e o ferrão, o qual em muitas espécies é carregado de ácido fórmico. Com suas antenas elas podem cheirar o feromônio, e também hidrocarbonetos presentes na camada mais externa de seu exosqueleto, que ajudam a distinguir companheiras de não companheiras ou até saber qual trabalho cada uma está fazendo (Gordon, 1999a). As formigas operárias são todas fêmeas e estéreis, os machos voadores também existem mas têm sua vida restrita apenas à época da reprodução. Geralmente existe uma rainha, que se distingue pelo seu grande porte, e fornece novas formigas à colônia durante toda a sua vida, que dura de quinze a vinte anos, após sua morte a colônia diminui gradualmente e é extinta após um ano (Gordon, 1999b).

O estudo intensivo da comunicação entre as formigas ao longo dos últimos anos rendeu a profusão de resultados que afetam profundamente o entendimento sobre a sua organização social. Seu modo de comunicação é extremamente diverso, utilizando-se de um toque físico ligeiro, de rangidos, das antenas, do paladar, e dos químicos que servem para o reconhecimento e até mesmo para o recrutamento e aviso de perigo (Hölldobler e Wilson, 1990). A principal forma de comunicação entre as formigas ocorre através da quimio-recepção, ou seja, na resposta fisiológica de um órgão do sentido a um estímulo químico, denominado semioquímico, usado tanto em membros de uma mesma espécie quanto em espécies diferentes (Law e Regnier, 1971). Os feromônios são semioquímicos depositados pelas formigas, geralmente uma secreção glandular, sentidos

pelos demais membros da colônia pelo paladar ou olfato (Karlson e Lüshcer, 1959) e são utilizados para fins diversos como: recrutamento para uma fonte de alimento ou para o ninho, sinal de alarme, atividade sexual, defesa do ninho e do alimento. Isso explica a existência de mais de um tipo de feromônio, cuja especificidade é obtida através de diferentes combinações de seus componentes. Além destas atividades em comum, cada espécie e cada colônia de formigas produz um feromônio específico, de forma que indivíduos pertencentes a outras colônias ou espécies possam reconhecer companheiros ou competidores (Hölldobler e Wilson, 1990).

2.1.1 Forrageamento

A maior parte das espécies de formigas não explora o ambiente de forma solitária, mas recruta companheiras para as fontes de alimentos descobertas, a fim de recolherem coletivamente o alimento para a colônia e, ao mesmo tempo, protegerem os recursos obtidos. O recrutamento em massa, exemplificado pelas espécies *Solenopis, Monomorium, Pheidole, Pheidologeton*, constitui a forma de comunicação mais avançada e mais utilizada pelas formigas na captura de alimento (Hölldobler e Wilson, 1990). Neste tipo de recrutamento em massa a formiga que descobriu a fonte de alimento não está presente para guiar suas companheiras pela trilha de feromônio que foi deixada em sua volta ao ninho. Este fenômeno pode ocorrer enquanto outros indivíduos buscam alimento de forma independente, e as demais formigas que atingem a fonte de comida comportam-se como as primeiras, construindo a trilha no retorno ao ninho (Verhaeghe et al., 1980). Outras formas mais primitivas de recrutamento, das quais este modo mais sofisticado provavelmente se desenvolveu são (Hölldobler e Wilson, 1990):

- Recrutamento *tandem*: através do contato das antenas, uma formiga orienta uma companheira à fonte de alimento, e, se este contato for perdido, a formiga-guia interrompe seu movimento e deposita feromônio sobre o ambiente, para que sua companheira volte a seguí-la (Hölldobler et al., 1974).

- Recrutamento em grupo: uma formiga conduz um grupo de companheiras do ninho até a fonte de alimento descoberta, através da trilha de feromônio que

foi depositada em seu trajeto alimento-ninho, porém neste caso, se um dos indivíduos perde o sinal químico emitido, a formiga líder não interrompe seu percurso (Verhaeghe, 1982).

Quanto maior for o tamanho da colônia maior é a confiança depositada nas trilhas de feromônio em oposição ao contato feito no recrutamento *tandem* (Hölldobler e Wilson, 1990). O recrutamento pode ser descrito pela interação entre uma retroalimentação positiva, que ocorre quando as formigas reforçam uma trilha com feromônio, e uma retroalimentação negativa, representada pela evaporação do feromônio (Camazine et al., 2001). Assim, como observou Vittori (2005) a decisão das formigas sobre qual fonte de alimento explorar, ou qual caminho tomar, "não se baseia em uma comparação direta entre as diferentes alternativas utilizadas pelos indivíduos", mas ocorre por meio de um processo de auto-organização, baseado na construção das trilhas de feromônio e movimentação através delas.

Na espécie *Solenopsis invicta*, as formigas ajustam a quantidade de feromônio de acordo com a qualidade do alimento encontrado (Hangartner, 1969), porém existe outro sistema, empregado pela espécie *Myrmica*, habitante de florestas européias, como mostra a figura 2.1. Quando as formigas *Myrmica sabuleti* encontram água ou sentem o cheiro de uma presa, elas coletam o material individualmente, porém, se o ambiente onde estão estiver iluminado, elas não aplicam o feromônio no trajeto do alimento ao ninho pois podem se orientar visualmente. Ao descobrir água com açúcar ou uma presa em outra condição, ou seja, se o ambiente estiver escuro, elas não somente usam a orientação pelo feromônio como também recrutam companheiras através de contato (Cammaerts e Cammaerts, 1980).

A escolha sobre a qualidade do alimento encontrado permite que as formigas decidam construir ou não a trilha de feromônio após a inspeção da área encontrada, ou seja, se o alimento for de baixa qualidade, se estiver distante do ninho ou a colônia estiver satisfeita com a quantidade de alimento, então a área descoberta não será totalmente explorada pelas formigas (Vittori, 2005). De maneira geral, as formigas tomam suas decisões de forma probabilística (Wilson, 1971), baseado na quantidade de feromônio

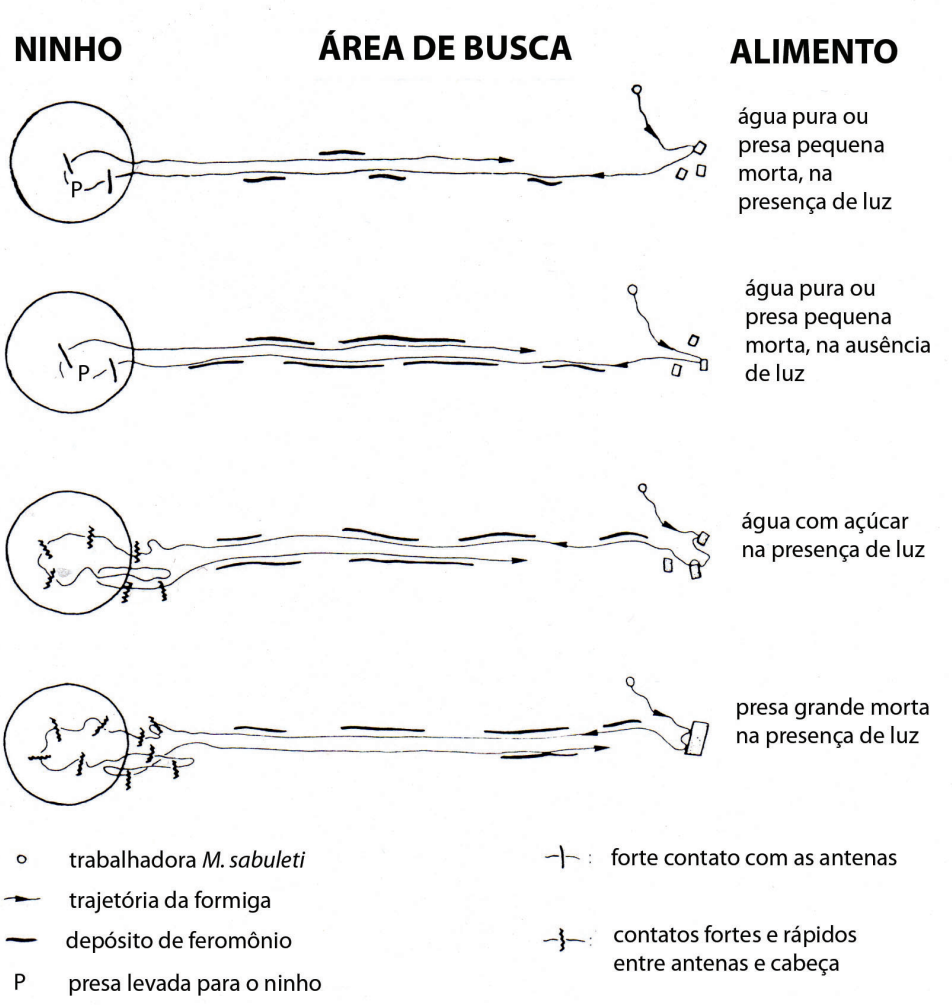

Figura 2.1: Sistema de recrutamento da formiga Myrmica sabuleti (baseado em Hölldobler e Wilson (1990)).

presente nas diversas trilhas a partir do ninho, as quais podem conduzir a uma mesma fonte ou a fontes diferentes de alimento. Estudos de Van Vorhis Key e Baker (1986) sobre a espécie *Iridomyrmex humilis* ou *Linepithema humile*, conhecida com formiga argentina, e de Beckers et al. (1992) sobre a espécie *Lasius niger* mostraram que ambas espécies de formigas depositam feromônio tanto quando se movimentam em direção à fonte como quando retornam ao ninho com o alimento, sendo que, além disso, a espécie *Lasius niger* também deposita feromônio em direção ao alimento após a sua descoberta. No entanto, a detecção dessa trilha de feromônio pelas outras formigas só ocorre quando sua concentração se encontra acima de um dado limiar (Bossert e Wilson, 1963).

Hölldobler e Wilson (1990) estudaram que as trilhas de feromônio, depositadas pelas formigas em busca de alimento, apresentam tipicamente a forma de dendritos,

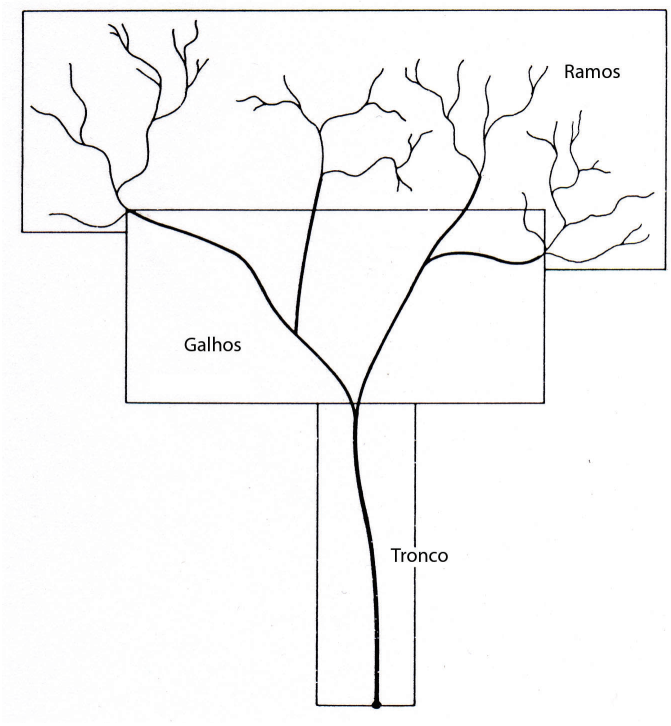

Figura 2.2: Trilha de feromônio em forma de dendrito (retirada de Hölldobler e Wilson (1990)).

a partir daí representaram esquematicamente uma rota de forrageamento completa, como mostra figura 2.2, em que cada um destes dendritos se inicia na vizinhança do ninho como um único caminho que se divide inicialmente em galhos, e posteriormente em ramos. Esse padrão distribui rapidamente um grande número de formigas nas áreas de forrageamento. Neste modelo, quando pequenas quantidades de comida são encontradas, as formigas levam-nas até os galhos e depois ao ninho. Já quando se trata de grandes depósitos de alimento, as formigas recrutam outras para finalizar a tarefa. Também, algumas espécies que ocupam vários ninhos, usam essas trilhas em forma de dendrito como rotas de conexão.

2.1.2 Escolhas Binárias

Alguns estudos mostram que as formigas se utilizam tanto das trilhas de feromônios quanto de suas capacidades de orientação e memória para retornarem ao ninho pelo caminho mais curto. Este é o caso da espécie *Formica rufa*, que pode memorizar simultaneamente a posição de quatro pontos de referência separados e utilizá-los para

alcançar uma fonte de alimento por até uma semana. Além disto, ela consegue se movimentar na direção do ninho por uma rota direta, enquanto percorre um caminho tortuoso pois acompanha o movimento do sol e ajusta o ângulo de sua trilha de retorno ao ninho comparando-o a este movimento (Vittori, 2005). A seguir são descritos experimentos desenvolvidos para mostrar o comportamento de colônias de formigas em busca de alimento, considerando escolhas binárias dos insetos sobre o meio.

Escolha entre dois caminhos de mesmo comprimento

Como experimento em condições controladas, Deneubourg et al. (1990) estudaram através de uma ponte binária o comportamento de forrageamento das formigas, na espécie *Linepithema humile*. O ninho e a fonte de alimento foram separados por uma ponte com duas ramificações de mesmo comprimento, figura (2.3). Deixou-se as formigas livres para se moverem entre a comida e o ninho, e a porcentagem das que escolheram entre um caminho ou outro foi observada continuamente. Após uma fase transitória inicial, as formigas tenderam a convergir para um mesmo caminho.

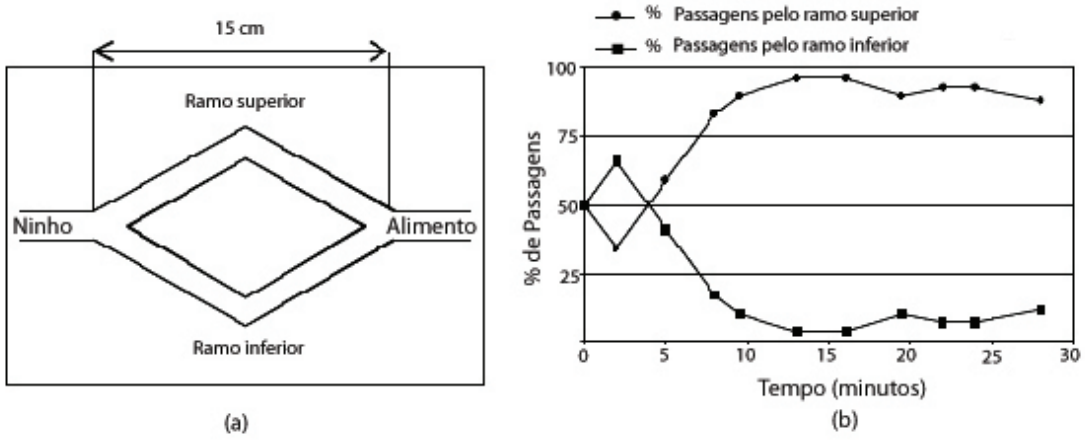

Figura 2.3: (a) Experimento da ponte binária e (b) porcentagem das formigas nos dois caminhos em função do tempo (adaptado de Deneubourg et al. (1990)).

Inicialmente, como é de se esperar, não existia feromônio em nenhuma das ramificações e, assim, qualquer uma delas poderia ser escolhida com a mesma probabilidade. No entanto, a escolha é aleatória, e quanto mais formigas passam por um mesmo caminho, mais feromônio elas depositam nele, isso estimula mais companheiras a escolherem o mesmo rumo. O experimento foi realizado assumindo a hipótese de que a quanti-

dade de feromônio numa ramificação é proporcional ao número de formigas que por ela passa. No modelo assumido, a probabilidade de escolha por um caminho, em um certo tempo, depende da quantidade total de feromônio na ramificação, que é proporcional ao número de formigas que usaram aquele caminho até aquele momento.

Após a observação do comportamento, Deneubourg et al. (1990) chegam ao modelo descrito a seguir, onde U_n e L_n correspondem ao número de formigas que usaram o ramo superior e o ramo inferior, após n formigas terem passado pela ponte, sendo $U_n + L_n = n$. A probabilidade $P_s(n)$ com a qual a formiga $(n+1)$ escolhe o ramo superior é:

$$P_s(n) = \frac{(U_n + k)^h}{(U_n + k)^h + (L_n + k)^h} \tag{2.1}$$

Enquanto a probabilidade $P_i(n)$ das que escolhem o ramo inferior é $P_i(n) = 1 - P_s(m)$. Os parâmetros h e k foram necessários para o ajuste do modelo aos dados experimentais. Simulações de *Monte Carlo* foram realizadas para testar a correspondência entre o modelo adotado e os dados reais. Os resultados dessas simulações estiveram em concordância com dados experimentais de formigas reais quando os parâmetros foram configurados para $k \approx 20$ e $h \approx 2$ (Pasteels et al., 1987).

Escolha entre dois caminhos de comprimentos diferentes

Goss et al. (1989), realizaram experimentos semelhantes ao descrito anteriormente, porém em suas pontes os caminhos tinham comprimentos diferentes, como visto na figura (2.4). Neste caso, devido ao mesmo mecanismo de utilizar o feromônio para marcar o caminho, o ramo mais curto é escolhido com maior freqüência. As primeiras formigas a chegarem à fonte de alimento são aquelas que, por um acaso, seguiram o menor trajeto, da mesma forma elas conseguem retornar ao ninho mais rapidamente do que aquelas que começaram seguindo o caminho mais longo. Desta maneira elas depositam mais feromônio no caminho mais curto evidenciando-o, o que estimula outras formigas a utilizarem o mesmo.

Estes experimentos mostram que este processo é um tipo de mecanismo de oti-

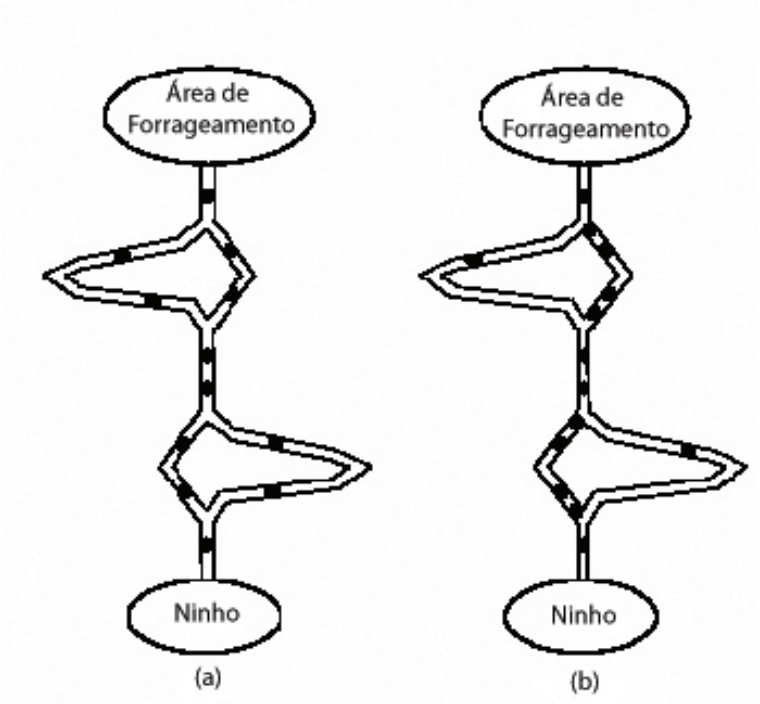

Figura 2.4: Início da exploração da ponte pelas formigas (a) Concentração das formigas no caminho mais curto (b). (adaptado de Goss et al. (1989))

mização distribuído no qual cada formiga tem uma contribuição pequena. Nota-se que mesmo uma formiga sendo capaz de descobrir sozinha uma solução (um caminho entre o ninho e a fonte de alimento), apenas o trabalho da colônia (um conjunto de formigas) é apto de encontrar o menor caminho, utilizando para isso um modo de comunicação indireto, conhecido como estimergia (Grassé, 1959).

2.1.3 Comunicação e Estimergia

Numa definição ampla, a palavra sinergia, do grego *synergos*, se refere a efeitos combinados ou cooperativos produzidos por dois ou mais agentes. A definição é freqüentemente associada à citação de Aristóteles em Metafísica: "o todo é melhor que a soma das partes", ou mais precisamente, ao agirem juntos estes agentes criam um efeito melhor do que aquele produzido por agentes sozinhos (Ramos e Merelo, 2002). A sinergia é um fenômeno ubíquo na natureza e igualmente em sociedades humanas. Um bom exemplo é a auto-organização que emerge da interação entre indivíduos nos insetos sociais, via interação direta (mandibular, antenas, contato químico ou visual) ou indi-

reta, definida por Grassé (1959) como estimergia para poder explicar a coordenação de tarefa e a regulação na reconstrução de ninho dos cupins *Macrotermes*. É um exemplo de comunicação estimergética a interação indireta de dois indivíduos, quando um modifica o ambiente e o outro, em seguida, responde ao novo ambiente. Em outras palavras, a estimergia poderia ser definida como um caso típico de sinergia ambiental, ou ainda, conforme Dorigo et al. (1999), a comunicação estimergética é uma comunicação indireta mediada pela modificação física dos estados do ambiente, o qual é acessado apenas localmente pelos agentes da comunicação.

Grassé (1946) observou que os insetos são capazes de responder ao então chamado "estímulo significante", o qual ativa uma reação codificada geneticamente. Em insetos sociais, tais como formigas e cupins, o efeito dessa reação pode agir como um novo estímulo, tanto para o inseto que o produziu como para outros insetos da colônia. A produção de novos estímulos significantes como resposta a um estímulo determina uma forma de coordenação de atividades, e pode ser interpretada como uma forma de comunicação indireta. Segundo Dorigo et al. (1999), os fatores que diferenciam a estimergia de outros meios de comunicação são:

- a natureza física da informação que modifica os estados do ambiente visitado pelo inseto, e
- a natureza local da informação, a qual pode ser acessada apenas por insetos que visitam o mesmo local no qual a informação foi lançada.

A maior parte da comunicação entre os insetos sociais é desempenhada indiretamente pela estimergia, que biologicamente é realizada pelo feromônio. Se, por exemplo, membros de uma espécie têm uma responsabilidade inata de encontrar feromônio, então qualquer inseto que deposite tal substância está desintencionalmente enviando uma mensagem para qualquer outro que porventura passe por ali. Com o tempo o feromônio evapora, o que permite às formigas adaptarem seu comportamento ao ambiente modificado (Shtovba, 2005). E, assim, através da alteração do estado em que o ambiente se encontra o comportamento para aqueles que têm o ambiente como um estímulo será afetado (Kennedy et al., 2001). Novamente a estimergia está presente no

modo como as formigas formam pilhas de itens tais como presas, larvas e alimentos, inicialmente elas depositam itens em locais aleatórios, e quando outras formigas percebem os depósitos, elas são estimuladas a depositarem os mesmos itens próximos a eles, como ocorre na formação de cemitérios e na separação da ninhada. Em cada um destes exemplos o ambiente serve como meio de comunicação. Todavia, o que todos estes exemplos têm em comum é que eles mostram como a estimergia pode facilmente ser operacional (Robinson, 1992).

2.1.4 Auto-Organização

A teoria da auto-organização foi originalmente desenvolvida no quadro da física e química no intuito de descrever padrões macroscópicos que emergem de interações definidas no nível microscópico (Haken, 1997)(Nicolis e Prigogine, 1977). No caso das formigas, ela é a base para o comportamento social, formada por um conjunto de mecanismos dinâmicos que garante ao sistema atingir seu objetivo global através de interações simples entre seus agentes. A característica chave desta interação é que o sistema usa somente informações locais. Neste caso, qualquer controle centralizado e referência ao padrão global representando o sistema no mundo externo são rejeitados (Ramos e Merelo, 2002). Bonabeau et al. (1997) aponta os seguintes mecanismos que caracterizam a auto-organização nos insetos:

- retroalimentação positiva (do inglês *positive feedback*) ou auto-catálise, por exemplo, recrutamento de outros agentes (Pasteels et al., 1987) ou reforço de um comportamento individual;
- retroalimentação negativa (*negative feedback*) sob a forma de saturação, exaustão ou competição;
- amplificação de flutuações tais como caminhadas aleatórias, erros ou tarefas de troca também aleatórias;
- interações múltiplas: uma densidade mínima de indivíduos mutuamente tolerantes é necessária; estes devem ser capazes de explorar os resultados das suas

próprias atividades, assim como a de seus companheiros, quer eles percebam uma diferença entre eles ou não.

A formação de cemitérios pelas formigas tem sido descrita como uma atividade estimergética (Kennedy et al., 2001), liderada pelo *feedback* positivo e auto-catalítico (Ramos e Merelo, 2002). Quando uma formiga morre no ninho ela é ignorada pelas outras formigas nos primeiros dias, até o corpo começar a se decompor. A eliminação de substâncias relativas ao ácido oléico estimulam uma formiga que esteja passando pelo cadáver a pegá-lo e carregá-lo para fora do ninho. Se os restos mortais dessas espécies estão aleatoriamente dispersos numa área, os sobreviventes irão se apressar em pegar esses corpos, movendo-os, até que todos eles estejam empilhados em pequenos grupos. Estas pilhas podem ser formadas nos cantos de uma área ou numa proeminência ou em qualquer outro lugar heterogêneo no ambiente (Kennedy et al., 2001). Deneubourg et al. (1991) mostraram que a formação de um cemitério pode ser explicado com regras simples, cuja essência é que itens isolados devem ser pegos e jogados em outro lugar onde mais itens desse tipo estão presentes.

A avaliação implícita de soluções é outro comportamento que apresenta conexão direta com o mecanismo de auto-catálise e está presente no forrageamento de formigas, apresentado por Dorigo et al. (1999), ou seja, o fato de que caminhos mais curtos são completados anteriormente aos caminhos mais longos, recebendo, dessa forma, um reforço de feromônio mais rapidamente. Uma ilustração de como a estimergia e a auto-organização podem ser combinadas em comportamentos adaptativos ainda mais sutis é o recrutamento em insetos sociais para a limpeza do ninho por alguns trabalhadores (Bonabeau et al., 1999). A autocatálise é um mecanismo poderoso que já vem sendo usado em algoritmo de otimização baseado em população. Um bom exemplo disso são os algoritmos de computação evolutiva em que a autocatálise é implementada pelo mecanismo de seleção e reprodução (Fogel, 1995) (Holland, 1975).

2.2 Formigas Artificiais

Assim como nas colônias reais, algoritmos de formigas são compostos por uma população de entidades concorrentes e assíncronas cooperando globalmente para encontrar uma boa solução para a tarefa em consideração (Dorigo, 1992). A complexidade de uma formiga artificial, assim como a de uma formiga real que encontra um caminho entre o ninho e a comida, deve ser suficiente para que ela possa construir uma solução viável, não exatamente a melhor. As soluções com grande qualidade devem ser o resultado da cooperação entre os indivíduos de toda a colônia. Sua cooperação deve surgir da leitura e escrita da informação realizada de maneira concorrente nos estados por elas visitados.

Logo, do mesmo modo como fazem as formigas reais, as formigas artificiais também modificam alguns aspectos do ambiente, como por exemplo, ao invés de depositar o feromônio, elas alteram algumas informações numéricas que estão armazenadas no local. Por analogia, esta informação numérica é chamada de trilha de feromônio artificial. No algoritmo ACO as trilhas de feromônio são o único meio de comunicação das formigas, cujo efeito principal é mudar a maneira como o ambiente é percebido por elas, atuando como uma função de todo o histórico da colônia (Dorigo, 1992). Não obstante, é comum nesse tipo de algoritmo um mecanismo de evaporação, similar à evaporação dos feromônios reais, para modificar a informação do feromônio artificial através do tempo. Esta evaporação permite que a colônia de formigas lentamente esqueça seu histórico. Com isso, a colônia pode direcionar sua busca para novas direções sem uma sobrecarga de feromônio artificial que poderia se tornar um empecilho para esta nova empreitada.

2.2.1 *Ant Colony Optimization* e suas aplicações

Tendo como modelo o comportamento cooperativo das formigas naturais, mais especificamente a maneira como elas localizam o caminho mais curto da fonte de comida até o ninho, no começo dos anos 90 na Itália, Dorigo (1992) em sua tese de doutorado propôs os Algoritmos de Otimização como uma técnica multi-agente para resolução de

problemas de otimização combinatorial. No ACO uma colônia de formigas artificiais coopera para encontrar boas soluções em problemas de otimização. Esta cooperação é tida como chave fundamental para tais algoritmos, nas quais as formigas artificiais modeladas adquirem uma natureza dupla, pois ao passo em que elas são uma abstração dos traços comportamentais, observados em colônias de formigas reais, elas também têm sido enriquecidas com algumas capacidades que não são encontradas na natureza. Deve-se ter em mente que os algoritmos de enxame, que aqui encontram um foco maior nos enxames de formiga, consistem numa nova técnica de modelagem e implementação de sistemas para a solução de problemas. Desta maneira, é razoável prover as formigas artificiais destas novas capacidades, permitindo-as serem mais eficazes e eficientes na tarefa atribuída.

Os algoritmos de otimização de formigas têm sido aplicados com sucesso para solucionar diversos problemas de otimização combinatória, como por exemplo: o do caixeiro viajante (Dorigo et al., 1996), de rotas de veículos, ordenação seqüencial, coloração de grafos, rotas em rede de comunicação, o planejamento de compras (*job-shop schedule planning*) (Colorni et al., 1994), entre outros. Para resolver estes problemas com os algoritmos de formiga é requerido: (i) reduzí-los à busca pelo caminho mais curto em algum gráfico; (ii) definir mecanismos de inicialização e atualização do feromônio; e (iii) determinar regras heurísticas para a seleção de rota (Shtovba, 2005). Entre as principais características do ACO advindas de formigas reais pode-se citar:

- um enxame de indivíduos cooperativos;
- uma trilha de feromônio (artificial), para comunicação estimergética;
- uma política de decisão estocástica que faz uso de informação local.

O algoritmo busca um bom equilíbrio entre a solução precisa e o tempo de otimização e também pode ser aplicado a problemas de otimização combinatória estocástica (Shtovba, 2005). A convergência de algoritmo de formiga para uma condição global ótima tem sido demonstrada em (Gutiahr, 2003). As seguintes aplicações no sistema de formigas são enfatizadas por Shtovba (2005):

- na engenharia: múltiplos modelos de grades de irrigação, otimização da distribuição de água, otimização de grades geodésicas de GPS, otimização da confiabilidade com a ajuda de redundância, desenho ergonômico nos teclados de computadores, distribuição de dados na memória de supercomputadores, e otimização dinâmica de processos químicos;
- no gerenciamento: esquema de atividades de curso universitário, otimização de distribuição de pontos de ônibus;
- na biologia: previsão da dobra de proteína por sua corrente de aminoácidos;
- nas artes: composição musical e pintura.

Além destes, bons resultados foram obtidos usando o algoritmo de formigas para aprendizado de redes Bayesan (Campos et al., 2002), regras lógicas clássicas (Raspinelli et al., 2002), conhecimento em base fuzzy (Cassillas et al., 2000), bem como para extração de regras fuzzy de experimentação de dados (Cassillas et al., 2005). Baseada na otimização de formigas e lógica fuzzy, a Corporação Siemens desenvolveu um método híbrido para controle logístico. Um uso piloto deste método em armazéns de Munique reduziu os atrasos na entrega de produtos em 44 por cento (Shtovba, 2005).

Política de Transição Estocástica e o Uso de Informações Locais

As soluções construídas pelas formigas artificiais e reais, baseiam-se numa política de decisão probabilística que determina o movimento através de estados adjacentes. As suas políticas fazem uso apenas de informação local sem a capacidade de predizer os estados futuros. Com isso, a política aplicada é completamente local, em espaço e tempo, atuando como uma função das informações a priori representada pelas especificidades do problema e das modificações locais do ambiente, introduzidas por formigas passadas (Dorigo et al., 1999). As características apresentadas até esse ponto, mostram a semelhança entre as formigas naturais e as artificiais, contudo, como já foi dito, algumas características das formigas artificiais, listadas a seguir, não são inspiradas na

natureza, mas são modeladas com o intuito de torná-las aptas a resolverem o problema ao qual serão submetidas:

- As formigas artificiais vivem em um mundo discreto e seus movimentos consistem de transições entre estes estados discretos;
- Elas têm um estado interno que contém a memória das ações passadas;
- Elas depositam uma quantidade de feromônio que é uma função da qualidade da solução encontrada;
- O tempo em que o feromônio é posto no solo pelas formigas artificiais é um problema dependente; em muitos casos estas formigas atualizam as trilhas de feromônio somente após ter gerado uma solução.

2.2.2 Flexibilidade, Plasticidade e Adaptação

A flexibilidade, plasticidade e adaptação são qualidades comumente empregadas para descrição e definição de sistemas de multi-agentes, dessa forma, uma definição adequada desses conceitos se faz necessária. Baseado nos exemplos dos insetos sociais, Gordon (1992) explica que flexibilidade é a capacidade que uma sociedade tem em mudar o seu comportamento coletivo, e que plasticidade é a capacidade de um controlador adaptar os seus parâmetros. Seguindo os mesmos exemplos, Belew e Mitchell (1996) mostra que adaptação não é apenas a capacidade de mudança, mas a melhora apropriada que esta mudança representa. A palavra "adaptação" tem raiz no latim *adaptare* e significa "acomodar", assim, seu significado sugere dois fatos: existe algo que se adapta, que se acomoda, e existe um local ao qual este algo se adapta. Por exemplo, entre os organismos vivos se diz que estes se adaptam aos seus meios, dessa maneira, é importante notar que algo é adaptativo em relação a um critério (Kennedy et al., 2001).

Experimentalmente, Pasteels et al. (1987), Franks et al. (1991) e Gordon (1996) e teoricamente Pacala et al. (1996) e Bonabeau et al. (1998) mostraram que nas sociedades de insetos, uma colônia pode se adaptar às condições de um novo ambiente

pela modificação do número de trabalhadores envolvidos em uma determinada tarefa, através de mecanismos de recrutamento. Como conseqüência, a flexibilidade no âmbito do grupo não necessariamente requer plasticidade no plano individual. Isto é, ainda que uma sociedade tenha um grau de liberdade adicional no seu número de trabalhadores, as regras de seu comportamento são fixas como em um sistema. No indivíduo a plasticidade pode ser vista como um mecanismo de amplificação, como o recrutamento (Deneubourg et al., 1987). Em trabalho de campo, Aron et al. (1993) demonstrou como esse mecanismo de amplificação pode ocorrer no nível social (como recrutamento de massa) ou no nível individual (como reforço de aprendizado) dependendo das condições do meio ambiente e da espécie de formiga considerada.

2.3 Aplicações da Inteligência de Enxames

Muitos problemas do mundo real são acessíveis a uma abordagem pela Inteligência de Enxames, especificamente as áreas de computação gráfica, escalonamento, e redes têm influenciado as metodologias de enxame para criar soluções robustas, elegantes e escaláveis para problemas complexos. Esta seção apresenta um resumo de algumas aplicações de Inteligência de Enxames no mundo real.

Animação Computadorizada Criado por Reynolds (1987), os *boids* são criaturas que vivem em grupos, simuladas para modelar o comportamento coordenado de pássaros e peixes. O modelo básico de agrupamento consiste em três comportamentos dirigidos, sem complexidade, que descrevem como um indivíduo *boid* se movimenta, baseado nas posições e velocidades dos seus vizinhos, através de: separação, alinhamento e coesão. O comportamento de separação coordena um *boid* a evitar a aglomeração de seus colegas. O comportamento de alinhamento coordena um *boid* a se dirigir na média da direção de seus colegas. O comportamento coesão coordena um *boid* a se mover na média do deslocamento de seus colegas. Estes três comportamentos diretivos, que usam somente informações sentidas de outros *boids* próximos, permitiram Reynolds produzir comportamentos realísticos de vida em bandos. A primeira aplicação comercial de um

modelo *boid* apareceu no filme "O Retorno de Batman", no qual bandos de morcegos e colônias de pinguins foram simulados.

Escalonamento Um dos comportamentos emergentes que podem surgir através da Inteligência de Enxames é a de designação de tarefas. Um dos mais conhecidos exemplos em uma sociedade de insetos é a das abelhas. Tipicamente, as abelhas mais velhas saem a procura de alimento, mas em tempos de fome, as abelhas jovens também serão recrutadas para esta tarefa a fim de aumentar a probabilidade de se encontrar uma fonte de comida viável (Bonabeau et al., 1997). Usando tal sistema biológico como um modelo, Michael Campos da *Northwestern University* arquitetou uma técnica para escalonar cabines de pintura em uma fábrica de caminhões. Cada cabine de pintura é modelada como uma abelha artificial que se especializa em pintar uma cor específica. As cabines têm a habilidade de mudar sua cor, mas o processo é oneroso e demorado (Bonabeau e Theraulaz, 2002). A regra básica usada para administrar a divisão de trabalho é que uma cabine de pintura especializada irá pintar na sua cor até que ela perceba uma necessidade importante de trocá-la. Por exemplo, se ocorre uma pintura urgente para a cor branca e a fila para esta cor é significante, uma cabine, por exemplo verde, irá trocar sua cor para o branco para executar essa tarefa. Isto tem um duplo benefício, pois, primeiro, o trabalho mais urgente é controlado em uma quantidade de tempo aceitável, e segundo, a adição de uma nova cabine de tinta branca pode aliviar as longas filas de outras cabines que estão pintando com esta mesma cor. O sistema de escalonamento de pintura tem sido usado com sucesso, resultando em um baixo número de mudança de cores e um alto nível de eficiência. Também, o método é responsivo às mudanças na demanda e pode facilmente suportar colapsos.

Roteamento de Redes Talvez o mais conhecido e mais amplamente aplicado exemplo de Inteligência de Enxames encontrado na natureza é aquele da capacidade das formigas de forragear. Dados múltiplas fontes de comida ao redor do ninho, através da estimergia e do caminhar aleatório, as formigas localizarão a fonte de alimento mais próxima (de caminho mais curto) sem qualquer coordenação global. Elas se utilizam de trilhas de

feromônio para indicar à outras formigas a localização desta fonte; quanto mais forte o feromônio, mais perto estará da fonte (Bonabeau et al., 1999). Caro e Dorigo (1998) se inspiraram na biologia, mais especificamente no forrageamento de formigas para desenvolverem o *AntNet*, uma colônia de formigas baseada em algoritmo, no qual um grupo de formigas se move em um grafo que representa uma rede de dados. Ao longo do caminho que cada formiga constrói de uma fonte a um destino, ela coleta informação sobre o total de tempo e carregamento de rede. Esta informação é então compartilhada com as outras formigas na mesma área. Estes pesquisadores mostraram que o *AntNet* foi capaz de superar o desempenho estático e adaptativo do vetor-distância (vector-distance) e (link-state) com caminho mais curto de algoritmos de rede, especialmente em relação a carregamentos de rede pesados. O sucesso do roteamento de algoritmos de rede inspirados em enxames tais como o *AntNet* tem encorajado muitos na indústria de comunicações a explorar a abordagem de colônia de formigas para otimização de redes. Por exemplo, a *British Telecom* vem aplicando conceitos do forrageamento de formigas à problemas de balanço de carregamento e roteamento de mensagens nas redes de comunicação (Appleby e Steward, 2000). Seu modelo de rede é povoado por agentes, que neste caso são modelados como formigas artificiais. As formigas depositam feromônio em cada nó atravessado em sua viagem através da rede, dessa maneira povoando uma mesa de roteamento com uma informação atual. O roteamento de chamadas é então decidido baseado nestas mesas de roteamento.

Enxames de Robôs Atualmente uma das áreas de aplicações mais importantes para a Inteligência de Enxames é a dos Enxames de Robôs, que permitem o aprimoramento do desempenho de tarefas conferindo ao sistema características como: alta confiabilidade (tolerância a falhas), baixa complexidade para cada componente do grupo e diminuição do custo tradicional de sistemas robóticos. Segundo Dudek et al. (1993), os enxames podem executar tarefas que seriam impossíveis para apenas um robô realizar e podem ser aplicados em diversas áreas, tais como: sistemas industriais flexíveis, naves espaciais, inspeção/manutenção, construção, agricultura e exploração de ambientes de maneira geral. Muitos modelos têm sido propostos, como por exemplo o sistema

robótico celular introduzido por Beni e Wang (1989), que consiste na reunião de robôs autônomos, não sincronizados, cooperando em um espaço celular n-dimensional sob controle distribuído. Existe apenas comunicação limitada entre robôs adjacentes, estes operam autonomamente e cooperam com os outros para realizar tarefas globais pré-definidas.

Hackwood e Beni (1992) propuseram um modelo no qual os robôs são simples mas agem sob a influência de "robôs sinalizadores", os quais modificam o estado interno das unidades do enxame ao passar por eles. Sob a ação dos sinalizadores, todo o enxame age como uma unidade para concluir comportamentos complexos. A auto-organização é realizada de preferência via um modelo geral cujo princípio é a condição limite cíclica.

Mataric (1992) descreveu experimentos com uma população homogênea de robôs que agem sob diferentes limitações de comunicação, os robôs tanto agem sem o conhecimento um do outro, como são informados um pelo outro, ou agem em cooperação. Assim que a comunicação inter-robôs aumenta, mais comportamentos complexos se tornam possíveis.

Enxames de robôs são mais que apenas redes de agentes independentes, eles são potencialmente redes reconfiguráveis de agentes comunicantes capazes de coordenar percepção e interação com o ambiente. Considerando a variedade de modelos possíveis de grupos de robôs móveis, Dudek et al. (1993) apresentou uma taxonomia para enxames de robôs propondo diferentes maneiras pelas quais eles podem ser caracterizados. As dimensões dos eixos da taxonomia são: o tamanho do enxame, o alcance da comunicação, o alcance à topologia e à largura de banda da comunicação, a reconfigurabilidade do enxame e a habilidade de processamento de cada unidade.

2.4 Discussão

As formigas são insetos sociais que vivem em colônias e cujo comportamento é mais direcionado para a sobrevivência do grupo do que para um único indivíduo, e, sendo assim, tem inspirado modelos de otimização com aplicação prática em problemas de engenharia. Os algoritmos de formiga, inspirados em colônias de formigas reais e

no comportamento destas ao executar tarefas de forrageamento, se mostram bastante adaptativos e flexíveis, por permitir a exploração das possibilidades de maneira concorrente e sem controle centralizado. Adicionalmente, a comunicação indireta, que permite essa descentralização, associada a um conjunto simples de regras baseado em estímulo-resposta, permite a análise implícita de soluções através da auto-catálise trazendo uma convergência do algoritmo para as melhores soluções.

Com o desenvolvimento de pesquisas em sistemas de agentes autônomos e decentralizados, muitas áreas receberam maior atenção incluindo a modelagem de enxames de agentes, planejamento de agente, tomada de decisão e comportamento emergente. Para atender tais necessidades, os estudos em IE usando ambientes simulados vêm crescendo nos últimos anos. Isto resulta em um crescente número de artigos propondo soluções para aplicações reais. Assim, aparece a necessidade de ferramentas de simulação para testar estes algoritmos rapidamente, permitindo um processo realimentado de ajuste de parâmetros, antes de serem testados em *hardware*. Estas ferramentas devem ser flexíveis o bastante para permitir com facilidade o teste de variações nos algoritmos propostos. O próximo capítulo traz uma revisão das plataformas para simulação de agentes móveis mais utilizadas para esse fim.

Capítulo 3

Plataformas para Simulações Baseadas em Agentes

Uma vez que o propósito desse trabalho consiste na modelagem e desenvolvimento de um ambiente para simulação de agentes, mais especificamente, de formigas artificiais, com o objetivo de permitir o estudo de técnicas de navegação e exploração de ambientes baseados em comunicação indireta entre os agentes, uma revisão das plataformas existentes para Simulação Baseada em Agentes (SBA) é apresentada nesse capítulo. Além disso, faz-se necessário a definição de agente utilizada nesta dissertação.

A palavra "agente" recebe inúmeras definições na computação.Franklin e Graesser (1996) estudaram minuciosamente esse tema e criaram uma taxonomia de agentes autônomos. Deste trabalho surgiu uma definição clara e objetiva: "um agente autônomo é um sistema situado dentro e como parte de um ambiente com percepções e ações sobre ele, sendo que sua percepção pode ser influenciada e alterada, ao longo do tempo, conforme os objetivos que o guiam". O estudo das plataformas de SBA descrito a seguir, observou que esta definição se encaixa adequadamente em sua terminologia, desta forma, o agente modelado e implementado neste trabalho segue esta mesma definição.

O uso de simuladores baseados em agentes (SBA) para pesquisa está crescendo rapidamente em diversas áreas do conhecimento, como pode ser visto, por exemplo, no

aumento de publicações sobre ecologia usando SBAs que teve início em 1990 (Angelis e Mooij, 2005). Este crescimento é primeiramente guiado pela habilidade destes modelos endereçarem problemas que modelos computacionais convencionais não eram capazes de lidar adequadamente. Contudo, esse crescimento também se deve à evolução da teoria (Grimm e Railsback, 2005) e estratégias (Grimm et al., 2005) para desenvolvimento de pesquisas baseadas em SBAs, assim como pelo crescimento em número e qualidade das plataformas de software para simulações baseadas em agentes.

Entretanto, o desenvolvimento de software para SBA permanece um obstáculo para muitos pesquisadores. Este problema, se deve, em grande parte, pela falta de treinamento em desenvolvimento de *software* de pesquisadores que aplicam estes SBAs em vários campos como por exemplo, biologia, economia, ciência política, sociologia. Minar et al. (1996) ainda salientam que a programação de computadores não é a única e nem a mais importante habilidade necessária para o desenvolvimento desse tipo de sistema.

As plataformas de SBA mais utilizadas seguem o paradigma "*framework* e biblioteca", fornecendo um *framework* - um conjunto de conceitos padrões para projetar e desenvolver um determinado tipo de sistema, no caso o SBA - juntamente com uma biblioteca de *softwares* como ferramentas para o desenvolvimento de SBAs. A primeira delas foi a Swarm (Minar et al., 1996)[1], cujas bibliotecas foram escritas em Objective-C. Java Swarm é um conjunto de classes Java simples que permite o uso da biblioteca Objective-C do Swarm a partir da plataforma Java [2]. O Repast [3] começou como uma implementação Java do Swarm, acabou por divergir significativamente do Swarm. Mais recentemente, MASON [4] está está sendo desenvolvido como uma nova plataforma SBA em Java (Luke et al., 2005).

Estas plataformas têm obtido sucesso em larga escala pois fornecem projetos de

[1]http://www.swarm.org

[2]Objective-C Swarm e Java Swarm se referem a pontos específicos de uma destas duas implementações, e a Swarm para pontos aplicáveis a ambos.

[3]http://repast.sourceforge.net

[4]`http://cs.gmu.edu/~eclab/projects/mason/`

software padronizado e ferramentas sem limitar o tipo ou complexidade dos modelos que eles podem implementar, porém eles também têm limitações. Uma revisão recente do Java Swarm e Repast (junto à duas plataformas menos utilizadas) classificaram-nos numericamente de acordo com um critério bem definido (Tobias e Hofmann, 2004). Este critério foi avaliado a partir da documentação e de outras informações sobre cada plataforma. A revisão apontou importantes fraquezas: dificuldade de uso; ferramentas insuficientes para construir modelos, especialmente ferramentas para representação do espaço; insuficiência de ferramentas para executar e observar experimentos em simulação, e falta de ferramentas para documentação.

A família Logo de plataformas segue uma evolução diferente, seu maior propósito tem sido fornecer uma plataforma de alto nível que permita aos estudants, ainda no ensino básico, construir e aprender SBAs simples. No entanto, o NetLogo[5] atualmente contém muitas capacidades sofisticadas (comportamentos, listas de agentes, interfaces gráficas, etc.) e é muito parecida com a maioria das plataformas SBAs amplamente utilizadas. O NetLogo inclui sua própria linguagem de programação, que é mais simples do que a linguagem Java ou Objective-C, um painel de animação conectado ao programa, controles visuais opcionais e gráficos. Talvez porque o NetLogo tenha sido claramente projetado prioritariamente para um único tipo de modelo (seção 3.1) e use uma linguagem de programação simples, alguns cientistas tendem a supor que ele seja muito limitado para SBAs sérios, contudo, um considerável número de modelos científicos foi implementado em plataforma Logo, como por exemplo An (2001). Das plataformas aqui revisadas, somente NetLogo não distribui o código fonte, mas seus desenvolvedores garantem que se esforçam ao máximo para acomodar as necessidades dos cientistas de compreender exatamento como essa plataforma funciona.

[5]http://ccl.northwestern.edu/netlogo/

3.1 Objetivos, Filosofia e Terminologia das Plataformas

Essa seção sumariza os objetivos e filosofias das principais plataformas SBAs existentes, baseado em suas documentações e nos estudos de Vrionides (2006) que inclui conversas com os desenvolvedores de cada uma delas. Além destes, é incluído aqui a mesma análise para a plataforma de desenvolvimento de sistemas Java, que apesar de não específica para soluções de SBAs, é de grande adoção como plataforma genérica para desenvolvimento de softwares científicos e de código livre, tendo sido adotado integral ou parcialmente por todas as outras plataformas aqui descritas.

Swarm O *Swarm* foi projetado para prover uma linguagem comum e um grupo de ferramentas (conjunto de rotinas pré-definidas) para SBAs, com abrangência para uso no meio científico. Seus desenvolvedores começaram por planejar uma abordagem conceitual geral para *softwares* de simulação baseados em agentes. O conceito do Swarm é que seu software deve ao mesmo tempo implementar um modelo, e separadamente, fornecer um laboratório virtual para observação e condução dos experimentos com o modelo. Outro conceito chave é projetar um modelo como uma hierarquia de "enxames", sendo um enxame um grupo de objetos e um planejamento de ações que os objetos executam. Um enxame pode conter enxames de baixo nível cujos planejamentos são integrados com aqueles enxames de alto nível. Um dos objetivos do projeto é fornecer um conjunto amplo de ferramentas de utilização geral para modelagem de SBAs e não ferramentas específicas a um domínio em particular. O Swarm foi projetado antes do surgimento do Java como uma linguagem madura. Objective-C foi escolhido por sua popularidade como linguagem orientada à objetos. O Swarm usa sua própria estrutura de dados e gerenciamento de memória para representar objetos de modelos; uma das consequências é que o Swarm implementa completamente o conceito de "teste", isto é, ferramentas que permitem aos usuários monitorar e controlar qualquer objeto durante a execução da simulação garantindo acesso a todos os métodos e variáveis de instância, não importando seu tipo de declaração (público ou privado).

Java Swarm O Java Swarm foi projetado para permitir, com a menor alteração possível, o acesso à biblioteca do Swarm em Objective-C a partir do Java. Essa iniciativa foi motivada pela forte demanda dos usuários do Swarm em escrever modelos em Java, o objetivo não era trazer novas capacidades ou características ao Swarm, mas apenas permitir a passagem de mensagem de objetos Java aos objetos em Objective-C.

Repast O objetivo principal do projeto Repast sofreu significativas modificações ao longo do seu desenvolvimento. O propósito inicial era portar o Swarm para a plataforma Java pura, eliminando a camada de objetos escritos em Objective-C, entretanto o Repast não adotou todos os conceitos de projeto Swarm e passou a ter como objetivo o suporte a um domínio em particular, ciência social, incluindo ferramentas específicas à esse domínio. Um objetivo adicional foi torná-lo mais fácil para usuários inexperientes construírem seus modelos. Essa abordagem inclui modelo próprio simples e uma interface com menus através dos quais é possível inserção de código em Python, que pode ser usada para iniciar a construção de um novo modelo.

MASON O MASON foi projetado como uma alternativa menor e mais rápida ao Repast, com o foco claro nas demandas de modelos computacionais com muitos agentes executando muitas interações. O projeto aparenta ter sido dirigido amplamente pelos objetivos de maximizar a execução de velocidade e assegurar completa reprodutibilidade através de hardware. As habilidades de atar e desatar interfaces gráficas, e parar uma simulação para movê-la entre computadores são consideradas uma prioridade para simulações longas. Os desenvolvedores do MASON optaram por incluir somente ferramentas gerais, nenhuma para domínios específicos. O MASON é a mais imatura dentre essas plataformas, com funcionalidades de interface gráfica e geração de distribuição de número aleatório ainda sendo adicionados.

NetLogo O NetLogo reflete claramente sua herança como uma ferramenta educacional, já que seu principal objetivo de projeto consiste na facilidade de uso. Sua linguagem de programação inclui muitas estruturas e primitivas de alto nível que reduzem o es-

força de programação, ele também disponibiliza documentação extensiva. A linguagem contém muitos recursos, porém não todos os controles, as capacidades e a estrutura de uma linguagem de programação padrão. Além disso, o NetLogo foi claramente projetado com um tipo específico de modelo em mente: agentes móveis agindo concorrentemente em uma grade com comportamento dominado por interações locais em tempos curtos. Enquanto modelos desse tipo são fáceis de serem implementados em NetLogo, a plataforma não é limitada a eles. O NetLogo é considerada a plataforma mais profissional em sua aparência e documentação.

Java Java é uma plataforma de desenvolvimento de sistemas computacionais de propósito geral desenvolvido pela *Sun Microsystems* no começo dos anos 90. Essa plataforma especifica uma linguagem de programação (linguagem Java) orientada à objetos com a sintaxe similar ao C e C++ (Campione et al., 2000). Porém, diferentemente desta última, ela implementa um modelo de objeto mais simples não incluindo características de baixo nível tais como acesso direto à memória e ponteiros. Um dos seus pontos fortes é ser independente de sistema operacional, isso se deve ao fato de os compiladores Java não compilarem o código direto para arquivos binários executáveis nativos, mas para um código intermediário chamado *bytecode*. Para a execução do programa esse código é traduzido e executado por um interpretador nativo chamado de Máquina Virtual Java. Outro ponto forte da linguagem é a vasta quantidade de bibliotecas de suporte que cobrem as mais importantes necessidades de programação e a sua constante atualização e suporte oferecidos pela maior comunidade de desenvolvedores de uma linguagem no mundo. Apesar de algumas críticas recebidas pelo seu baixo desempenho na execução, quando comparado com linguagens compiladas nativamente, esse quadro mudou drasticamente nas últimas versões do Java tornando-o tão competitivo ou mais para alguns tipos de aplicação (Davison, 2005). Outra característica importante é a padronização para documentação de código fonte e a ferramenta *javadoc* para gerar automaticamente esta documentação. A vasta classe de bibliotecas permite o reuso de código fornecendo algoritmos úteis, aumentando a produtividade e trazendo um alto nível de abstração no desenvolvimento.

3.2 Discussão

As plataformas de SBAs apresentadas neste capítulo são de extrema utilidade à comunidade científica permitindo aos pesquisadores das mais diversas áreas tais como psicologia social, inteligência artificial, biologia social, sociologia e economia concentrarem os seus esforços na lógica de aplicação relevante ao seu domínio. Os SBAs se apresentam como ferramentas metodológicas importantes no auxílio da modelagem e desenvolvimento de programas de simulação.

A característica interdisciplinar dos SBAs é um importante desafio enfrentado por todos esses pesquisadores uma vez que demandam um interlaçamento difícil de diferentes metodologias, terminologias e pontos de vista. Para auxiliar nesse processo de integração, as plataformas de simulação com suporte às necessidades educacionais, industriais e de pesquisa científica estão em demanda crescente (Marietto et al., 2002).

As existentes plataformas apresentadas ainda estão em fase imatura e longe de atender as necessidades e especificidades dos mais diferentes domínios. Apesar de plataformas como, por exemplo, o Swarm e o NetLogo serem extremamente abrangentes, podendo ser aplicadas nos mais diferentes domínios, elas exigem um grande esforço de modelagem e programação para a execução de uma simulação específica. Isso dificulta sua utilização por inúmeros pesquisadores, principalmente aqueles que não são do campo de computação ou áreas afins. É justamente nesse ponto que o presente trabalho se diferencia, pois ele não intenciona ser abrangente como estes últimos, mas dedicado ao domínio específico da simulação de agentes que se utilizam da comunicação estimergética para a exploração de ambientes e localização de objetivos. Em contrapartida, tem como objetivo permitir a simulação de diferentes experimentos utilizando diversas configurações dos parâmetros existentes sem a necessidade de qualquer reprogramação, contudo, outro requisito é garantir a possibilidade de extensão do software através da criação de um *framework* específico permitindo, por exemplo, a programação de diferentes agentes.

Capítulo 4

Projeto e Desenvolvimento

Depois de fundamentar os conceitos de inteligência de enxames mais relevantes para esse trabalho, (Capítulo 2), e de apresentar uma revisão das principais plataformas para SBA no Capítulo 3, este capítulo descreve os passos tomados para a realização do projeto proposto, desde a análise, com as considerações iniciais levadas em conta para a modelagem do sistema, até alguns detalhes relevantes de implementação. Essa dissertação se concentra no desenvolvimento de um sistema de simulação multi-agente para navegação de ambientes que se utiliza de técnicas de inteligência de enxames, ou seja, os agentes seguirão regras simples e suas interações com outros agentes dar-se-ão através de estímulos encontrados no ambiente de simulação. Um modelo capaz de parametrização também será proposto para validar a ferramenta.

4.1 Análise

A análise (Sommerville, 2000) é a fase inicial de desenvolvimento de um sistema, pois define os princípios e estratégias básicas que por sua vez servirão de entrada para o próxima etapa do ciclo de desenvolvimento. Ela também permite que o desenvolvedor tenha uma visão completa do problema. Portanto, esta é uma fase importante que guiará o desenvolvimento do simulador.

4.1.1 Modelo Proposto

O modelo proposto se concentra no estudo do desempenho de uma colônia de agentes em relação à tarefa de forrageamento, tendo como principal objetivo estudar a influência da comunicação indireta na exploração de ambientes, através da busca por objetivos e retorno ao ponto de origem. Os agentes não apresentam qualquer tipo de comunicação direta, nem individual e nem em grupo, nem mesmo através de contato direto, não tendo qualquer referência ao número de agentes existente. Cada agente tem acesso apenas a informações locais do ambiente, tais como: quantidade de feromônio em dado local no momento em questão; se entrou em uma região de interesse, ou se colidiu com algum obstáculo ou barreira. O ambiente no qual os agentes interagem abriga além destes, paredes, obstáculos e objetivos, permitindo um número variável de cada um destes elementos, podendo sofrer modificações em tempo de execução, como por exemplo a inserção ou remoção de obstáculos e inclusão de novos agentes. Isso é importante pois é mostrado em vários trabalhos que a tarefa de forrageamento é sensível a perturbações. Esse modelo expõe de maneira explícita parâmetros que influenciam seu desempenho, como tamanho da colônia, limiar de respostas ao estímulo do feromônio, quantidade de feromônio liberada, probabilidade de fazer giros e força de atração dos feromônios.

4.1.2 Plataforma de desenvolvimento

A plataforma de desenvolvimento escolhida é a plataforma Java. O fato de ser adequada para esse tipo de implementação, fortemente orientada à objetos, independente de sistema operacional, amparada por uma vasta quantidade de bibliotecas e ser a plataforma escolhida para o desenvolvimento da maioria dos softwares *open-source* atualmente, faz dela uma escolha bastante atrativa. Java oferece um nível de controle e flexibilidade ao programador muito maior do que qualquer outra plataforma apresentada anteriormente (Capítulo 3). Além disso, o desafio de desenvolver um simulador desde o início é bastante atraente pela facilidade de customização às necessidades tra-

zidas pelo projeto. A curva de aprendizado também deve ser levada em consideração, sendo que o tempo necessário para estudo aprofundado de uma plataforma de propósito geral como o *Swarm* pode ser grande devido a não familiaridade com suas características e comandos específicos. Além disso, a plataforma Java traz consigo uma padronização de documentação que atende a um dos requisitos do projeto.

4.1.3 Metodologia de Projeto

O paradigma orientado a objeto está fortemente acoplado com a linguagem de modelagem unificada UML (do inglês *Unified Modeling Language*) (Fowler, 2004). Todo o processo de projeto do sistema será descrito usando diagramas UML, o que traz uma maior compreensão do sistema para referência futura. Para implementação desse projeto uma técnica mais direta de desenvolvimento se mostrou mais adequada. O modelo *Waterfall* (Ahmed e Umrysh, 2003) foi usado como ciclo de desenvolvimento. Essa técnica consiste em segmentar o desenvolvimento em fases seqüenciais como mostrado na figura 4.1, ela funciona bem para projetos pequenos ou para projetos nos quais os requisitos são bastante estáveis e fixos, o problema é bem entendido e a solução foi provada em projetos similares.

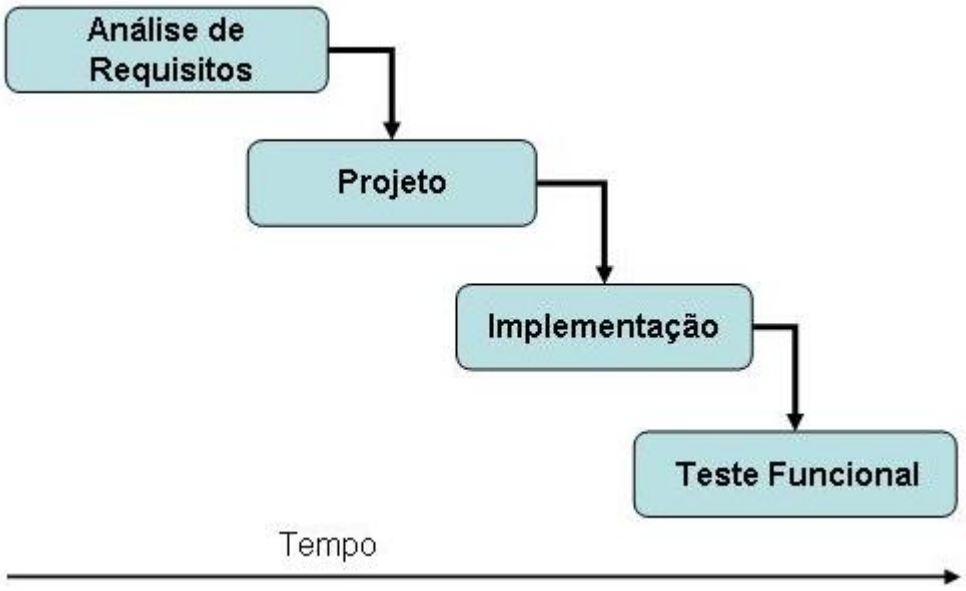

Figura 4.1: Processo de Desenvolvimento Waterfall ((Ahmed e Umrysh, 2003)).

4.1.4 Requisitos

Análise de requisitos é o estudo das características que o sistema deverá contemplar para atender às necessidades e expectativas do projeto. Cada funcionalidade demandada deve ser analisada para verificar os possíveis impactos no desenvolvimento das demais funcionalidades do sistema, e verificar se as necessidades tecnológicas para a sua implementação estão disponíveis. Os seguintes conjuntos de requisitos foram levantados para guiarem o desenvolvimento.

Requisitos funcionais: são aqueles que estão diretamente relacionado com as funcionalidades do sistema.

1. as funcionalidades da ferramenta devem atender ao modelo proposto (4.1.1);
2. aceitação de um conjunto de parâmetros úteis ou importantes;
   (a) número de forrageadores;
   (b) limiar de resposta para o estímulo de ambiente (feromônio):
      i. quando o forrageador não estiver carregando o objetivo (comida).
      ii. quando estiver carregando comida.
   (c) quantidade de estímulo liberado (feromônio):
      i. quando o forrageador não estiver carregando comida.
      ii. quando estiver carregando comida.
3. resposta visual contínua refletindo o estado do modelo;
4. execução contínua e passo-a-passo;
5. ferramenta para criação e distribuição de obstáculos;
6. ferramenta para medição do nível de feromônio em um ponto escolhido;
7. registrar em arquivo as seguintes informações ao final de um experimento:
   (a) número de forrageadores;

(b) número de atualizações do sistema;

(c) quantidade de comida forrageada.

Requisitos não funcionais: são aqueles que não estão relacionados com a funcionalidade do sistema, mas são pertinentes à qualidade do sistema como um todo.

1. interface com o usuário clara e intuitiva;
2. a janela de visualização deve mostrar claramente as interações que ocorrem no ambiente de simulação;
3. a entrada de parâmetros do modelo deve ser a mais intuitiva possível;
4. o sistema deve ser construído de maneira que forneça classes úteis e reutilizáveis;
5. deve ser bem documentado facilitando sua manutenção e expansão futura.

4.1.5 Interface de Usuário

A interface com o usuário é uma questão central para o sucesso e adoção de um sistema, caso ela tenha uma interface ruim, terá menos apelo, ainda que seja excelente em outros requisitos, como rapidez e robustez (Vrionides, 2006). Dessa forma, uma atenção especial foi dada para a construção de uma interface intuitiva e de fácil uso. O ambiente de simulação foi concebido de forma a apresentar uma visualização atraente para manter a atenção do usuário na ferramenta. Como parte da análise inicial de interface, o esboço (figura 4.2) foi criado para nortear o projeto da interface.

4.1.6 Um Breve Cenário do Simulador

Após iniciar o simulador o usuário configura os parâmetros necessários e dá início à simulação, a partir daí o usuário recebe uma resposta visual da movimentação dos agentes, seus estados e decisões. Ao final da simulação existe a opção de salvar os resultados obtidos em um arquivo para análise e interpretação futura. O usuário também pode configurar o tempo de realização da tarefa pré-determinada, ou seja, escolher um

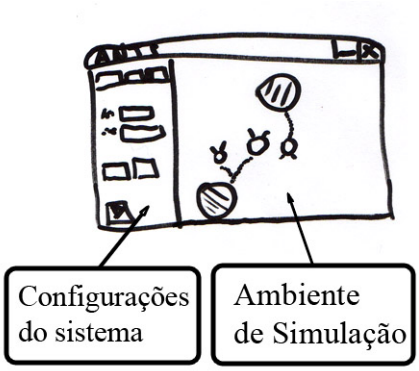

Figura 4.2: Esboço da interface com usuário.

número máximo de atualizações que pode ocorrer no ambiente, ou então, permitir que a simulação rode por tempo indefinido. Para uma melhor compreensão das atividades envolvidas entre o usuário e o simulador, foi criado o diagrama de seqüência, (Ahmed e Umrysh, 2003), mostrado na figura A.1 com o caso de uso geral do simulador.

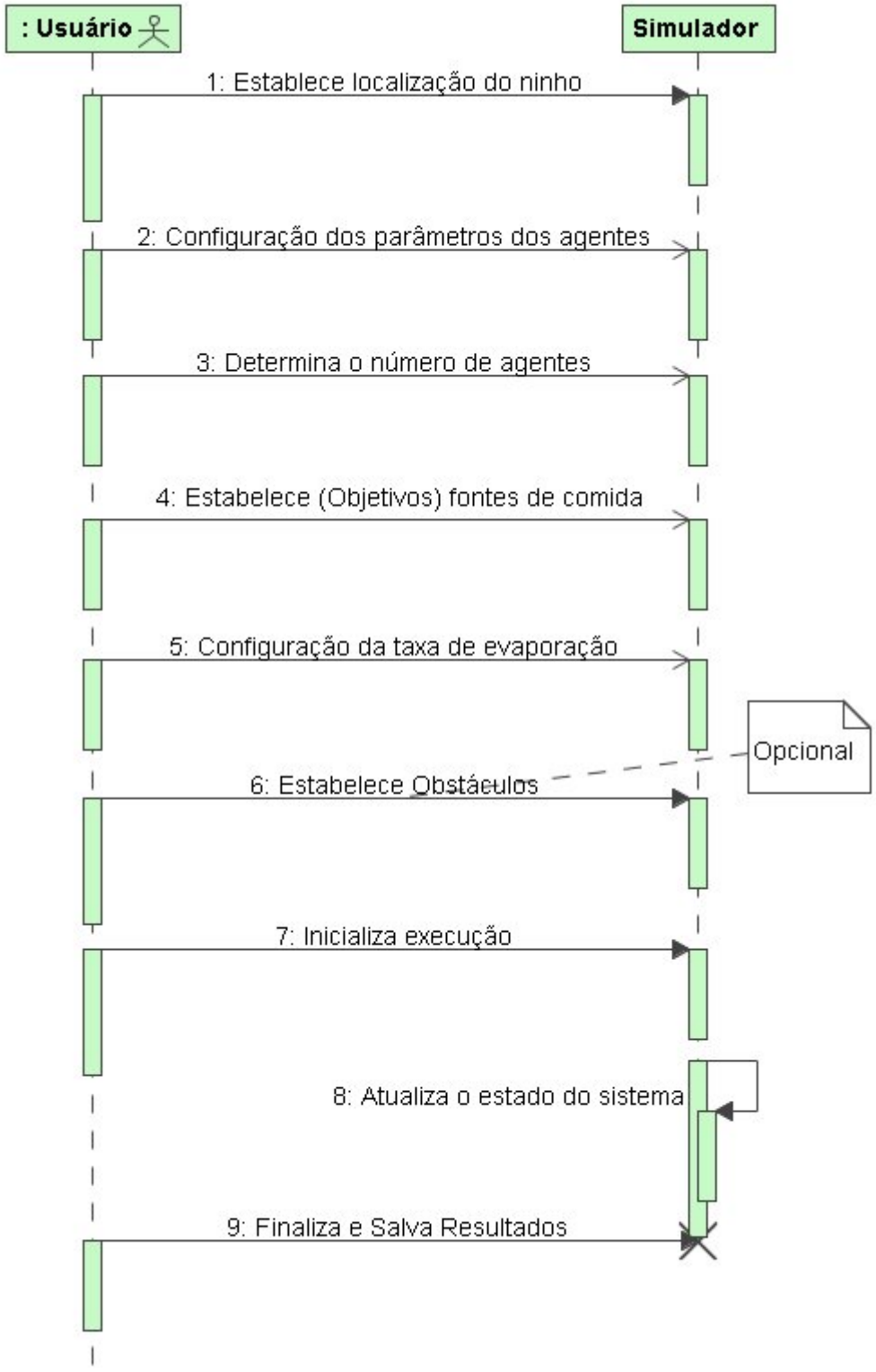

Figura 4.3: Diagrama de seqüência para o caso de uso geral do simulador.

4.2 Projeto

A fase de projeto é onde se realiza um estudo em profundidade permitindo a definição de uma arquitetura apropriada ao sistema desenvolvido, com o uso de diagramas de classe UML para mostrar precisamente as classes definidas e seus relacionamentos (Sommerville, 2000). Nesta seção será apresentado como cada um dos requisitos enumerados anteriormente na seção 4.1.4 foi alcançado.

4.2.1 Arquitetura do Sistema

Um dos pontos chaves para se atingir o requisito não funcional 4 é a adoção de uma arquitetura adequada que permita o desenvolvimento organizado e escalável. Dessa forma, a bem conhecida arquitetura *MVC* (do inglês *Model-View-Controller*) (Buschmann et al., 1996) é a que melhor se encaixa para esse tipo de projeto. *Model View Controller* ou Modelo-Visão-Controlador é um padrão de arquitetura de aplicações que visa separar a lógica da aplicação (Model) da interface do usuário (View) e do fluxo da aplicação (Controller). Permite que a mesma lógica de negócios possa ser acessada e visualizada por várias interfaces. A figura 4.4 mostra o relacionamento entre os camadas de Modelo, Visão e Controle.

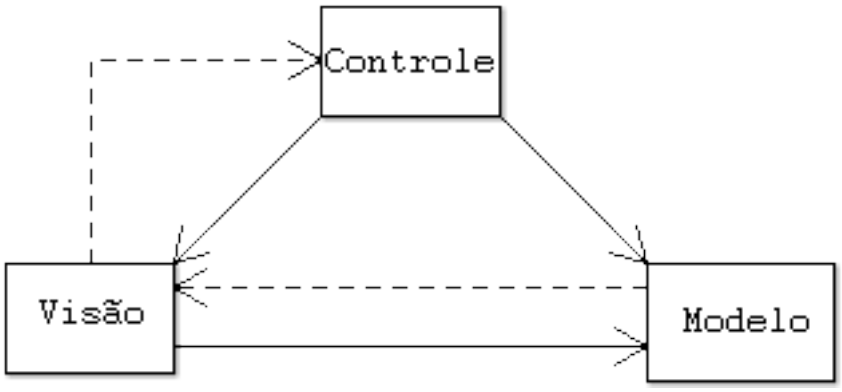

Figura 4.4: Relacionamento entre Modelo, Visão e Controle. As linhas contínuas denotam uma associação direta enquanto as pontilhadas uma associação indireta.

Em um projeto de *software* baseado no padrão MVC, é definida uma arquitetura básica com 3 camadas (Freeman e Freeman, 2004):

- **Modelo**: O modelo é responsável por manter todos os dados, estados e lógica da aplicação.

- **Controle**: Implementa a camada responsável pelo gerenciamento de eventos no projeto, tais como cliques do usuário, chamando a camada Modelo para processar os eventos. Também pode manter informações de estado do usuário na aplicação.

- **Visão**: Fornece a apresentação visual do Modelo. A visão geralmente recupera o estado e dados que necessita diretamente do Modelo.

A figura 4.5 apresenta as classes que compõem o sistema segundo a separação lógica de pacotes, explicados a seguir, atendendo a arquitetura descrita.

- `controller`: contém as classes de controle responsável pelo tratamento de eventos para as classes do pacote `gui`;

- `gui` (*graphical user interface*): contém as classes que representam os painéis de configuração do sistema, bem como o janela principal do simulador `MainFrame`, organizando todos os componentes visuais que compõem a interface gráfica com o usuário, tal como botões, campos de entrada de textos, entre outros, delegando os eventos para a camada de controle;

- `helper`: pacote para classes auxiliadoras, que contém apenas a classe `ActionSuport` que é utilizada pelas classes no pacote `gui` de maneira a encapsular e abstrair as classes de controle evitando a dependência específica entre uma determinada classe de `gui` com uma classe específica de controle;

- `model`: contém as classes principais para implementação do modelo proposto na seção 4.1.1, mantendo o estado do sistema e contendo a lógica necessária de execução dos agentes. É importante observar que a lógica para representação gráfica das classes do modelo, como por exemplo a classe `Nest`, que representa o ninho, é mantida pelas próprias classes do modelo, por questão de simplicidade, uma vez que estas não demandam interação direta com o usuário, não se fez

necessário a separação em classes de visão para fins dessa representação. Contudo, sua separação é facilmente realizada através de procedimentos simples de refatoração de código uma vez que se faça necessário.

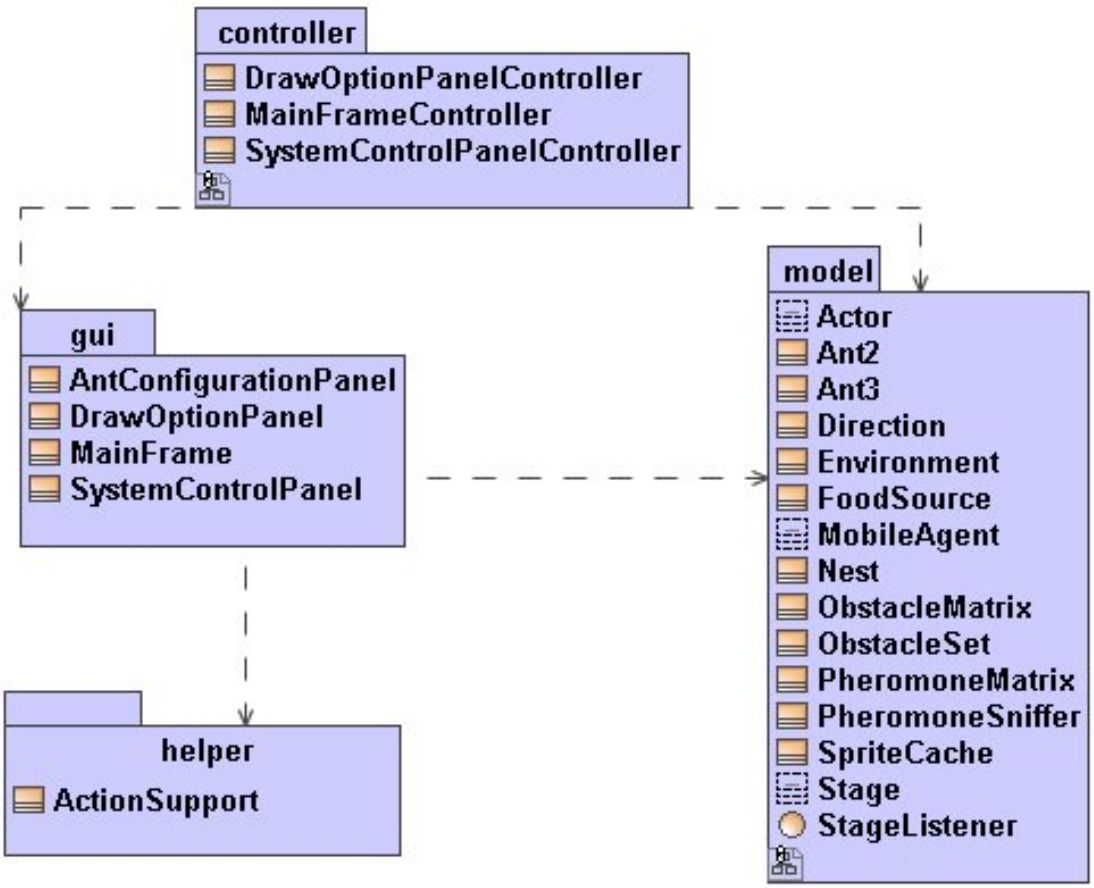

Figura 4.5: Diagrama de Pacotes com suas respectivas classes.

4.2.2 Implementação do Modelo Proposto

Essa seção se propõe a descrever como o requisito funcional 1 (implementação do modelo) foi implementado, enumerando as classes desenvolvidas e seus relacionamentos, ilustrados no diagrama A.5. O agente principal do sistema, o forrageador, é representado pela classe `Ant3`, que é uma especialização da classe `Actor`. Toda sua interação ocorre no ambiente definido pela classe `Environment` que mantém referências para uma matriz de feromônios existentes (`PheromoneMatrix`), um conjuto de obstáculos e barreiras (`ObstacleSet`) e o ponto de origem do forrageador, (o mesmo local onde deve

depositar o alimento forrageado) representado pela classe `Nest`, assim como todos os locais de fontes de alimento, que o forrageador deseja encontrar, descrito pela classe `FoodSource`.

O Andar do Forrageador - Sem Influência do Feromônio

O caminhar do forrageador deve ser aleatório, o menos determinístico possível, respeitando uma certa probabilidade de mudar sua direção ao longo de sua trajetória. Para tanto, um dos atributos da classe `Ant3` é um vetor de 100 posições, `turnProbability` que armazena constantes, relativas aos possíveis tipos de rotação, em quantidade diretamente proporcional à sua probabilidade. Antes de dar um passo, é sorteado um número aleatório entre [0,100[, que é usado como índice para localização da ação a ser tomada. As possíveis rotações são descritas por constantes na classe `Direction`:

- `NO_TURN`: sem rotação, no vetor de probabilidades representa a chance de apenas andar para frente sem virar;
- `CLOCKWISE_45`: rotação de 45° sentido horário;
- `CLOCKWISE_90`: rotação de 90° sentido horário;
- `CLOCKWISE_135`: rotação de 135° sentido horário;
- `CLOCKWISE_180`: rotação de 180° sentido horário;
- `COUNTERCLOCKWISE_45`: rotação de 45° sentido anti-horário;
- `COUNTERCLOCKWISE_90`: rotação de 90° sentido anti-horário;
- `COUNTERCLOCKWISE_135`: rotação de 135° sentido anti-horário;

Por exemplo, considerando o valor para `NO_TURN` = 0, para `CLOCKWISE_45` = 1 e para `COUNTERCLOCKWISE_45` = 2, se for requerido que o forrageador rotacione com as respectivas probabilidades de 96%, 2% e 2%, o vetor de probabilidades será preenchido conforme a figura 4.7.

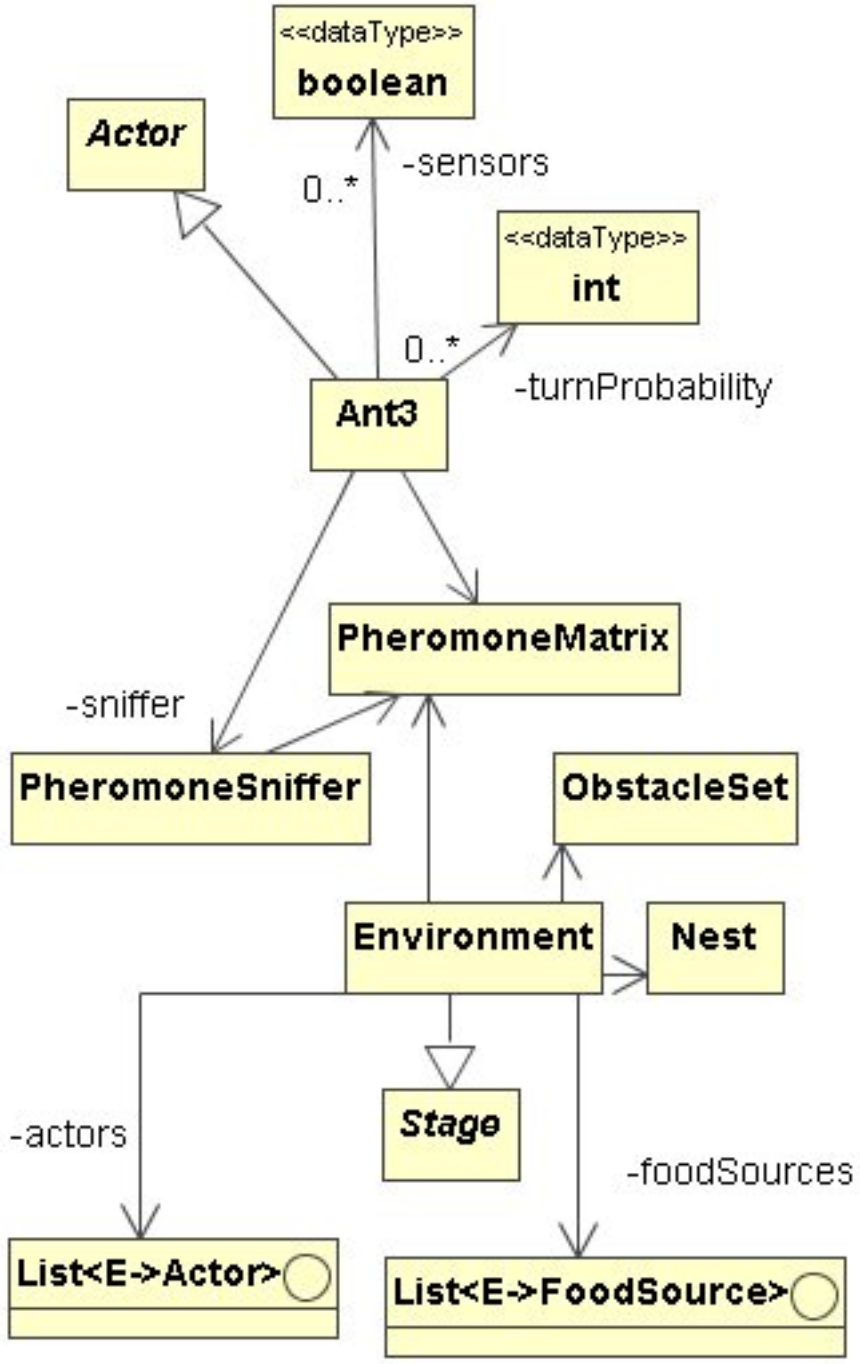

Figura 4.6: Diagrama com as principais classes para implementação do modelo proposto.

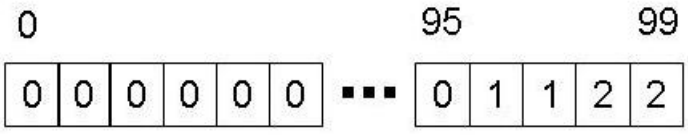

Figura 4.7: Vetor de probabilidades preenchido com 96% de chance de não virar, 2% de rotacionar 45% sentido horário e anti-horário.

O andar do forrageador - com influência do feromônio

A comunicação indireta entre os agentes do sistema ocorre através da existência de uma matriz de feromônios na qual cada célula armazena a quantidade de feromônio existente relativa a certa posição do ambiente. Para uma maior proximidade com o mundo real, não discreto, o ambiente aqui representado pela classe `Environment` é discretizado em uma grade de unidade de menor tamanho possível, no caso cada posição da grade tem correspondência direta aos *pixels* da imagem representada por ele. Ou seja, num ambiente de tamanho 800x600 *pixels*, a matriz de feromônio poderá conter até 480.000 células. Para uma maior eficiência, ela foi programada de modo a ser representada por uma simples matriz esparsa, utilizando-se um mapa `java.util.Map` em que a chave é o par ordenado (x,y) representado por uma classe `Point`, e o valor é a quantidade de feromônio armazenada.

Ao caminhar pelo ambiente, o agente faz a leitura do feromônio existente a sua volta. Através da classe `PheromoneSniffer`, ele pode "cheirar" em uma ou mais direções. Por padrão ele o faz a sua frente, a 45°a sua direita e a 45°a sua esquerda, conforme figura 4.8. Para isso existe um parâmetro interno que diz até qual distância ele consegue "cheirar" em cada uma das direções.

O algoritmo abaixo ilustra os passos de como e quando o feromônio influencia a sua ação.

```
- cheira em todas as direções pré-configuradas
- calcula a quantidade total (Q) de feromônio cheirado
- escolhe o limiar (L) de ativação correto
  # se estiver carregando comida L = k1
  # se não estiver carregando comida L = k2
- se Q > L
  # seleciona o peso P de influência do feromônio
    * se estiver carregando comida P = p1
    * se não estiver carregando comida P = p2
  # calcula vetor de probabilidades Vp
  # sorteia um número aleatório (Idx) entre 0 - 99
  # Ação = Vp[Idx]
  # executa Ação
```

Como mostra o algoritmo, a quantidade de feromônio no entorno do agente pode influenciar o vetor de probabilidades da seguinte forma: do total percebido a sua volta,

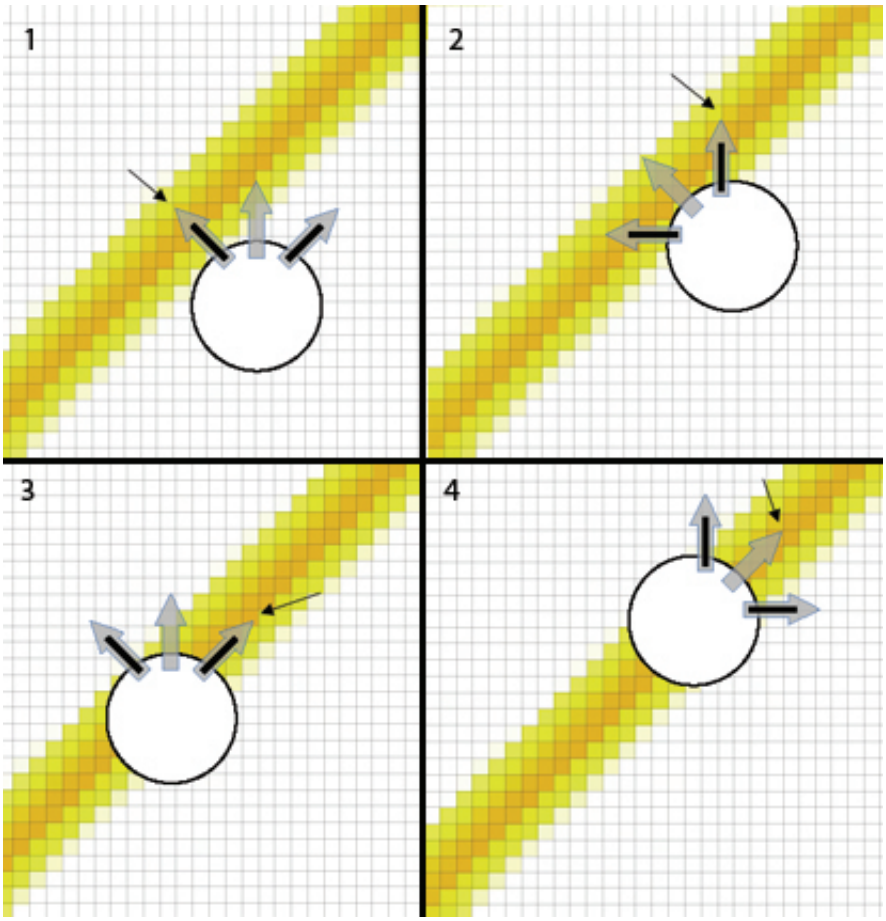

Figura 4.8: Representação da leitura de feromônio pelo forrageador, a seta indica qual das direções foi percebida a maior concentração de feromônio.

calcula-se, para cada uma das direções cheiradas, o quanto a quantidade parcial representa em porcentagem. Depois é realizada uma média ponderada com as probabilidades existentes, considerando-se peso 1 para os valores configurados sem a influência de feromônio e peso P para os valores representando os feromônios. A tabela 4.2.2 exemplifica como é calculado as porcentagens de feromônio em volta de um forrageador. A tabela 4.2.2 mostra as probabilidades de rotação para esse forrageador sem levar em conta a influência do feromônio e também mostra como é feito o cálculo da nova tabela de probabilidades. A tabela 4.2.2 mostra outro valor para o peso. Esse exemplo ilustra bem como a importância atribuída ao feromônio afeta o vetor de probabilidades.

Outro aspecto importante da implementação para atender o modelo especificado é a maneira como o forrageador altera o ambiente. Para isso ele faz uso de uma simples regra: ao caminhar ele sempre deposita feromônio. O valor depositado é afetado

Tabela 4.1: Cálculo de porcentagens de feromônio

Direção	**Feromônio**	**Porcentagem**
Para frente	20	10
45° horário	140	70
45° anti-horário	40	20
Total	200	100

Tabela 4.2: Cálculo do vetor de probabilidades de rotacionamento influenciado pelo feromônio, com peso maior.

Direção	**Probabilidade de rotação(%)**	**Quantidade de feromônio (%)**	**Peso do feromônio**	**Probabilidade de rotação final**
sem rotação	90	10	6	21
45° horário	3	70	6	60
90° horário	2	0	6	0
135° horário	0	0	6	0
180° horário	0	0	6	0
45° anti-horário	3	20	6	18
90° anti-horário	2	0	6	0
135° anti-horário	0	0	6	0

pelo fato dele estar ou não carregando alimento de volta para o ninho. Ao depositar feromônio, uma regra simples de dispersão é usada, o que faz com que células a seu redor sejam afetas, mas em proporção inversa à distancia. Metade do valor depositado inicialmente é depositado nas celulas vizinhas com distância igual a 1. Um quarto do valor inicial é depositado nas células de distância igual a 2 e assim sucessivamente até que não haja mais valor a ser depositado. Todo esse processo pode ser visualizado na figura 4.9

A evaporação do feromônio, por sua vez, é controlada pelo ambiente, classe `Environment`, utilizando-se de duas propriedades parametrizáveis: o intervalo de evaporação, `evaporation Interval`, que corresponde ao número de atualizações que deve ocorrer no sistema antes que ocorra uma etapa de evaporação, e a taxa de evaporação, `evaporationRate`, que diz o quanto será evaporado por turno de evaporação. Por exemplo, se o intervalo de evaporação estiver configurado para 10 e a taxa de evaporação configurada para 3, isso significa que a cada 10 iterações e atualizações nos estados do modelo do sistema, a quantidade de feromônio existente em cada célula da matriz de feromônio será

Tabela 4.3: Cálculo do vetor de probabilidades de rotacionamento influenciado pelo feromônio, com peso menor.

Direção	**Probabilidade de rotação(%)**	**Quantidade de feromônio (%)**	**Peso do feromônio**	**Probabilidade de rotação final**
sem rotação	90	10	3	30
45° horário	3	70	3	53
90º horário	2	0	3	1
135° horário	0	0	3	0
180º horário	0	0	3	0
45º anti-horário	3	20	3	16
90º anti-horário	2	0	3	1
135º anti-horário	0	0	3	0

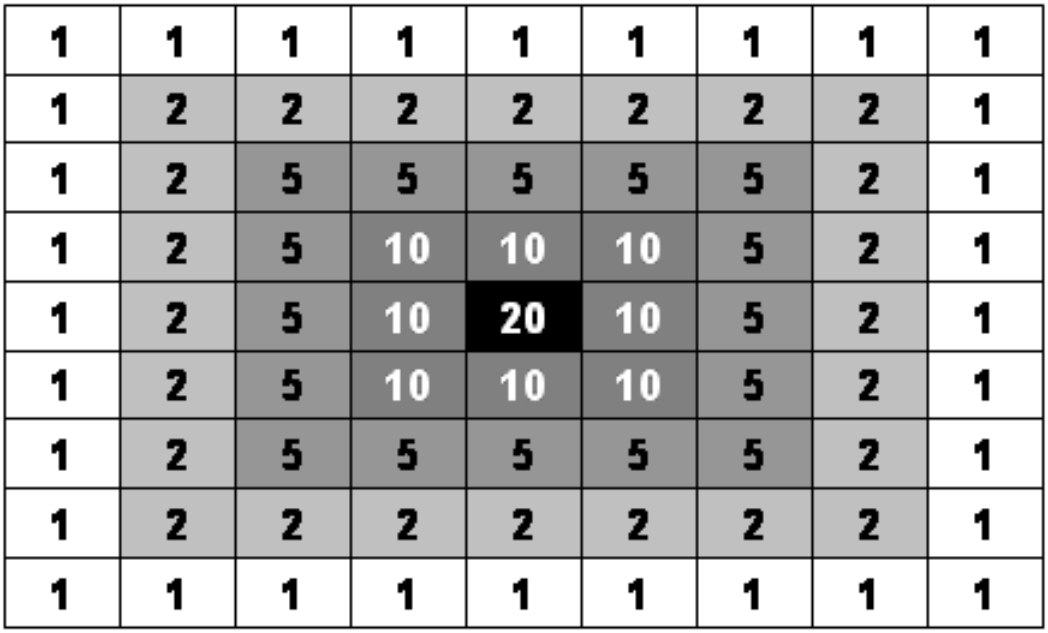

Figura 4.9: Dispersão do feromônio

subtraída de 3. Esses parâmetros são configuráveis através de um painel na interface gráfica (figura 4.10(a)).

4.2.3 Conjunto de Parâmetros Customizáveis

Para atender o requisito funcional 2 (conjunto de parâmetros), a classe `Ant3` tem como propriedades acessíveis por métodos públicos o `pheromoneWeight` e o `pheromoneThreshold`, que respectivamente controlam o peso do feromônio na tomada de decisão com relação às rotações empregadas a cada passo, e o valor limite de feromônio ao redor do for-

rageador para que passe a exercer influência sobre o mesmo. Estes parâmetros são para o caso do agente não estar carregando alimento, para o outro caso (carregando alimento) as propriedades `carryingFood PheromoneDepositAmount` e `carryingFood Pheromone Threshold` exercem funções análogas. Além destes, existe uma propriedade para configuração de cada uma das probabilidades de rotação, (seção 4.2.2) e outra para ajuste de velocidade. Para a fácil identificação do agente, existem duas propriedades `normalColor` e `carryingFoodColor` para escolha das cores que o representará quando estiver sem ou com alimento. Estas propriedades são customizadas através da classe de visão `AntConfigurationPanel`, ilustrada na figura 4.10(b).

4.2.4 Representação Visual do Ambiente

De maneira a atender os requisitos funcionais 3 e 4, o simulador foi construído de maneira a fornecer uma resposta visual dos estados dos agentes, tais como posição e direção, e do ambiente, tal como distribuição de feromônio a cada interação do sistema. Contudo, uma vez que a tarefa de rendereização do ambiente é computacionalmente cara, e as alterações não são muito expressivas entre uma iteração e outra, pode-se controlar o intervalo de atualizações que ocorrem no sistema antes de uma nova renderização do mesmo (figura 4.10(c)) aumentado o desempenho geral da simulação. Outra característica importante é a possibilidade de se executar passo-a-passo a simulação, através de um botão na interface (figura 4.10(d)) permitindo uma melhor compreensão dos acontecimentos em determinados momentos da simulação. A figura 4.11 mostra uma representação visual do ambiente, com trilhas de feromônios, dos agentes e dos obstáculos.

4.2.5 Obstáculos e Medição do Feromônio

Em conformidade aos requisitos funcionais 5 (ferramenta de obstáculos) e 6 (ferramenta para medição de feromônio), foram desenvolvidas ferramentas para desenho, seleção, deslocamento e remoção de obstáculos (figura 4.11(a) e (b)) bem como uma ferramenta para medição da quantidade existente de feromônio existente em dado local

do ambiente. A figura 4.11(c) mostra as coordenadas e o nível de feromônio na posição (figura 4.11(d)).

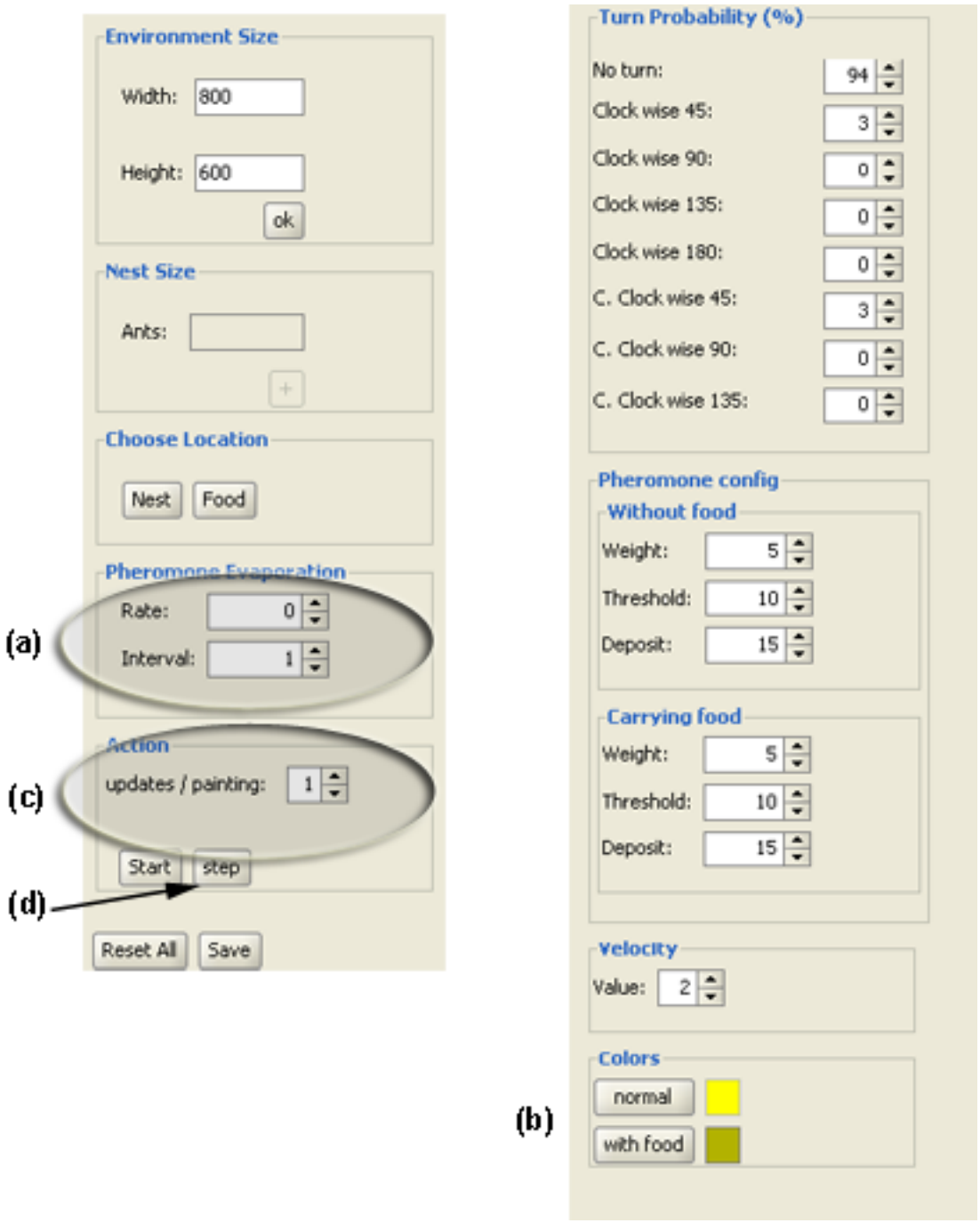

Figura 4.10: Painéis de configurações: (a) evaporação do feromônio, (b) configurações do agente, (c) atualizações/renderizações, (d) botão para execução passo-a-passo.

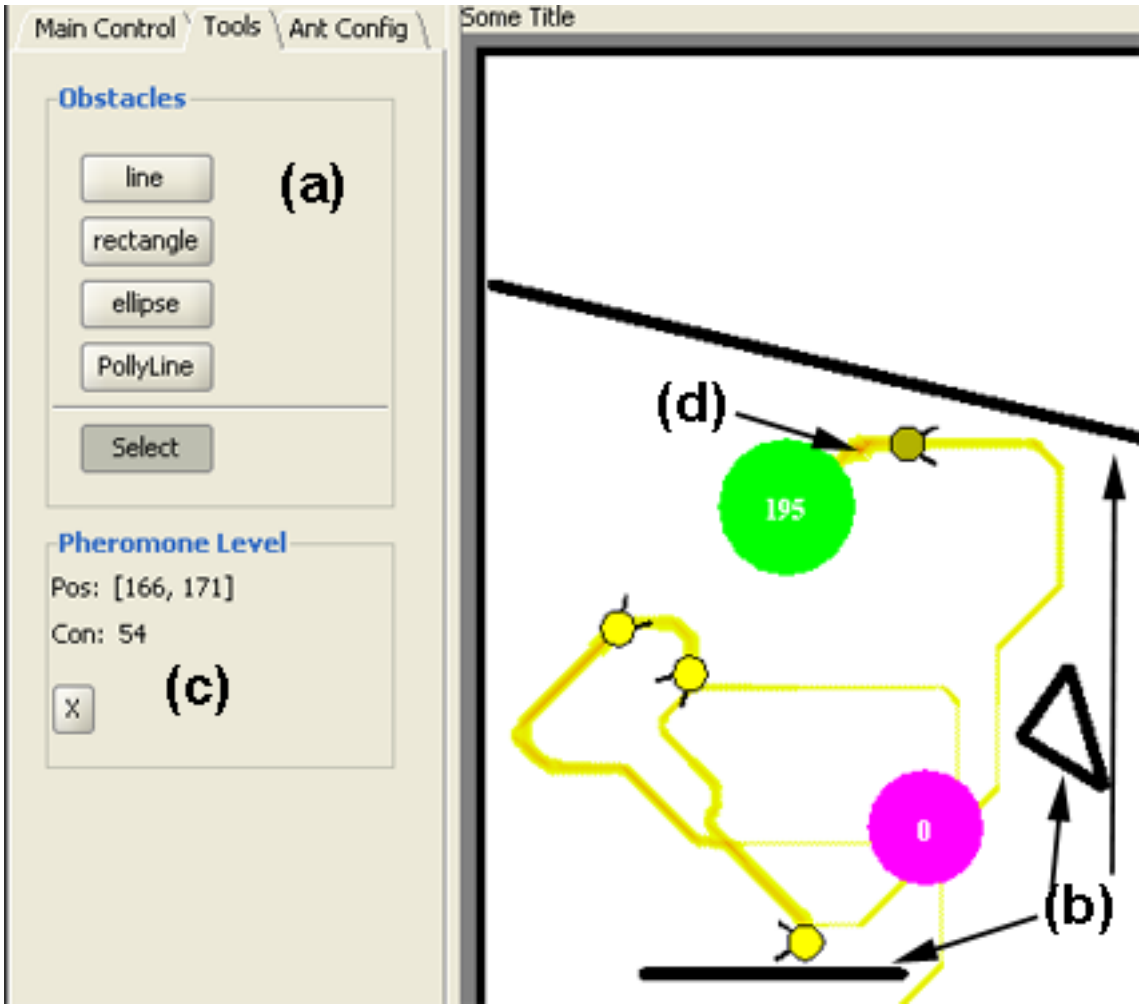

Figura 4.11: Ferramenta para desenho de obstáculos, (a), (b) obstáculos, (c) cálculo do nível de feromônio do ponto (d).

4.2.6 Armazenamento dos Resultados

Como especificado pelo requisito funcional 7 (arquivo com resultados), ao final de cada simulação é possível registar o número de atualizações do ambiente, o número total de agentes bem como a quantidade total de alimento forrageado em um arquivo texto para posterior análise. O arquivo tem formato simples:

```
name:Teste10
number of foragers: 20
collected food:650
```

4.2.7 Requisitos não Funcionais

A presente seção descreve os esforços empreendidos afim de atender os requisitos não funcionais descritos na seção 4.1.4, que são fundamentais para o sucesso da ferramenta desenvolvida, permitindo fácil adoção por parte dos usúarios, e futura expansão em termos de funcionalidades e abrangência através das classes implementadas por outros programadores.

Atendendo aos 3 primeiros destes requisitos, a interface gráfica com o usuário foi projetada de maneira a ser a mais clara e objetiva possível, com painéis de preenchimento intuitivos como mostrado na figura 4.10. A figura 4.12 traz uma visão abrangente de todo o simulador, construído de maneira a seguir o esboço planejado mostrado na figura 4.2

Com o objetivo de atender o requisito 4, as classes que compõe o sistema foram projetadas de maneira que comportamentos comuns e re-aproveitáveis fossem definidos em uma classe base de maneira a servir de ponto de extensão para as demais. Dessa maneira a classe básica para construção de agentes é `Actor`, do pacote `model`, sendo tão abrangente que até mesmo as classes para representação do ninho, `Nest`, e da fonte de alimento, `FoodSource`, se estendem dela. O diagrama da figura A.6 ilustra esse relacionamento. Seguindo o mesmo padrão, a construção do ambiente de simulação representado pela classe `Environment` se deu a partir outra classe implementada com

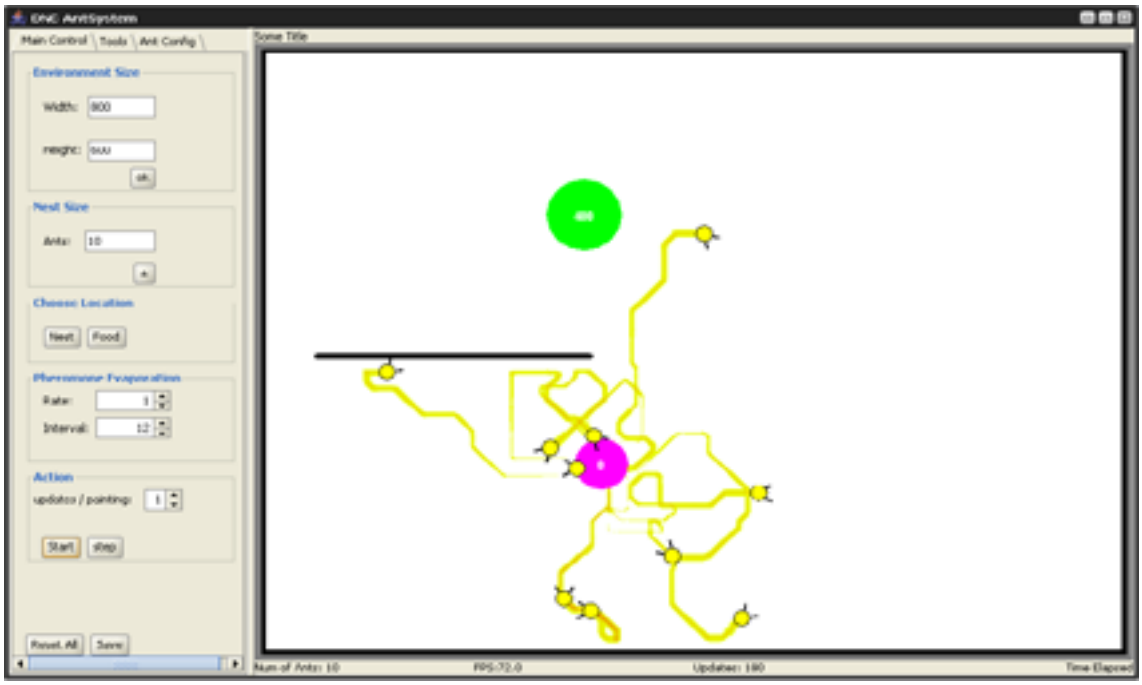

Figura 4.12: Visão geral do ambiente de simulação.

funcionalidades comuns à criação de ambientes iterativos de agentes, a classe `Stage`, conforme ilustrado no diagrama da figura A.7.

Documentação

De acordo com o último dos requisitos não funcionais(5) (boa documentação), uma adequada documentação do sistema desenvolvido é importante para permitir sua extensibilidade e manutenabilidade. Para tanto, foram criados os diagramas de classes e pacotes mais importantes, apresentados no Apêndice 1, e foi adotada uma metodologia de documentação, padronizada pela *Sun Microsystem*, e largamente utilizada para códigos fontes em java (Hemrajani, 2006). Ela baseia-se em padrões de comentários inseridos em locais predeterminados como mostra código 4.2.1, extraído da classe `Actor`, observe que todo o texto entre os caracteres "/**" e "*/" fazem parte da documentação. Após a inserção dos comentários, utiliza-se a ferramenta `javadoc`, parte integrante do conjunto de ferramentas para desenvolvimento em Java, para a compilação de um conjunto de páginas *HTML* com toda a documentação do sistema. O Apêndice 2 apresenta a documentação gerada para as classes de maior relevância nesse projeto.

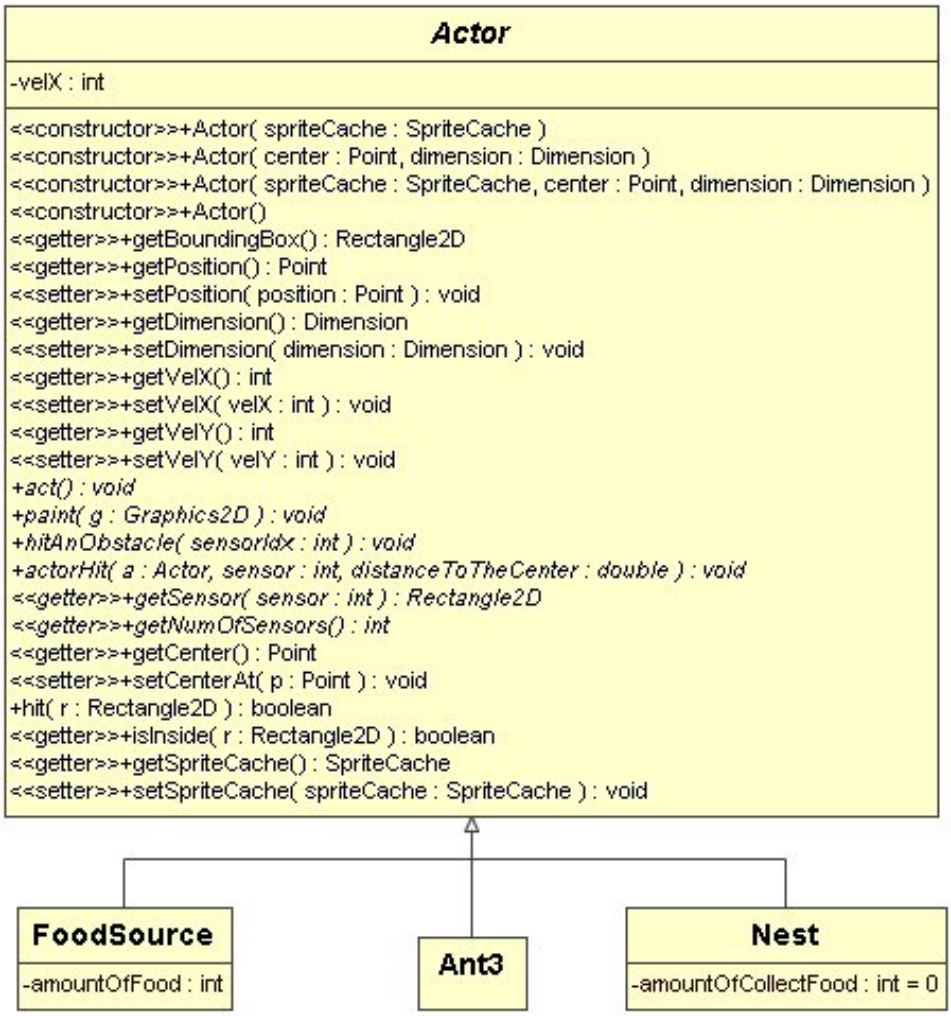

Figura 4.13: Classe Actor e seus descendentes.

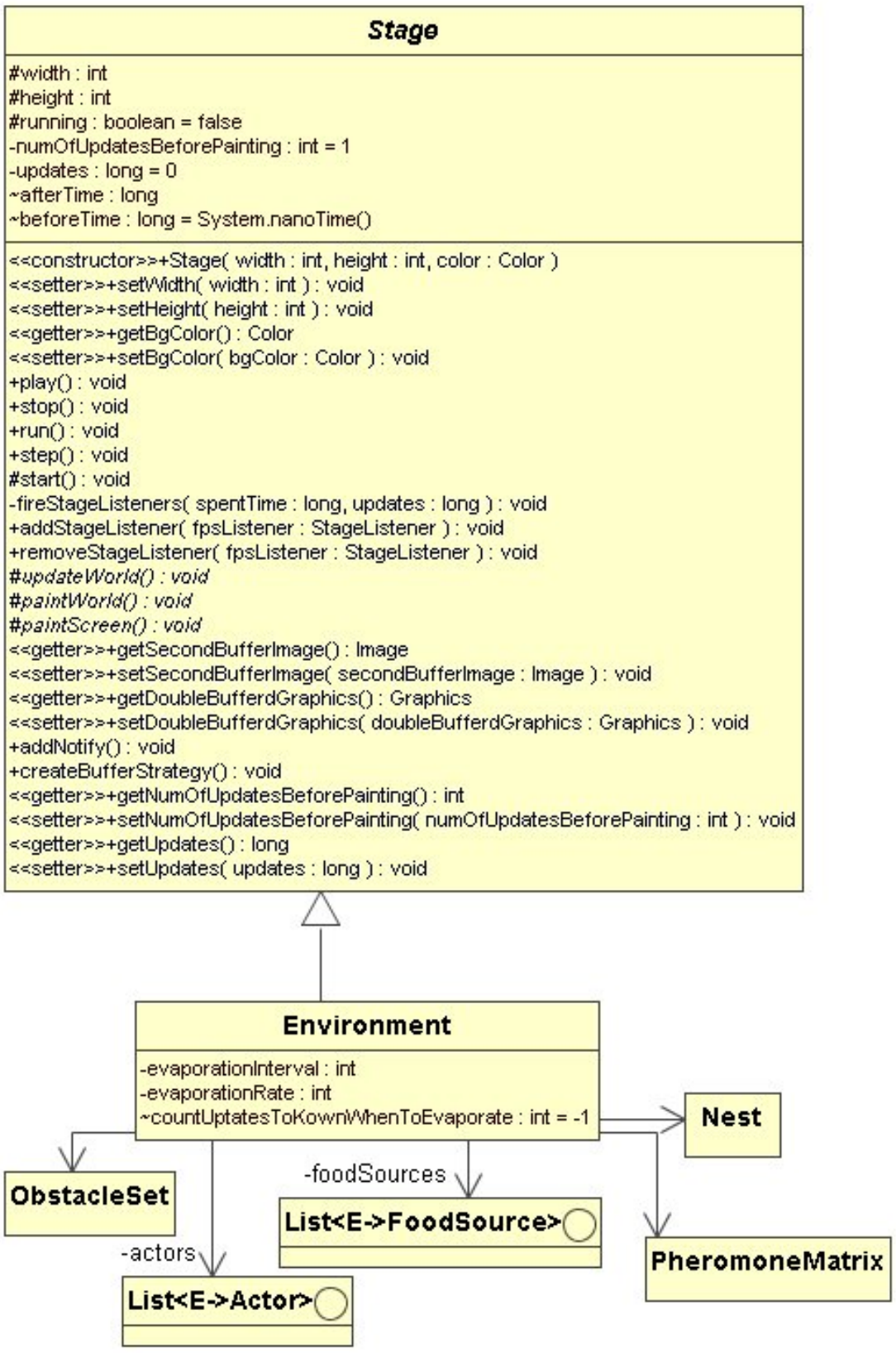

Figura 4.14: As classes envolvidas na criação do ambiente de simulação.

4.3 Discussão

A implementação de um SBA para o estudo de uma propriedade específica de um sistema multi-agente, no caso a comunicação indireta, é um processo trabalhoso. O simulador deve ser embasado por um modelo sólido e robusto, com suas propriedades e regras que guiam os agentes, definidas de maneira clara e formal. As propriedades expostas como parâmetros configuráveis devem ser cuidadosamente escolhidas, pois elas darão a flexibilidade necessária para o refinamento do modelo, permitindo que se chegue a comportamentos relevantes para aplicação em questão, como por exemplo a navegação de robôs móveis.

Dessa forma, a adoção de uma metodologia de desenvolvimento formalmente definida foi determinante para a obtenção dos resultados almejados. A escolha da plataforma, assim como a questão de partir de um trabalho já existente na área ou não, envolveu questões delicadas. Contudo, a escolha da plataforma Java de desenvolvimento trouxe flexibilidade, robustez e desempenho necessários para este tipo de aplicação, bem como um padrão e ferramentas necessários para uma documentação precisa do sistema, o que por sua vez, permitirá uma fácil compreensão dos modelos e funções codificados por pesquisadores da área de ciência da computação e engenharia, público alvo desta ferramenta.

Isso vem permitir extensões futuras do simulador para atender às necessidades de novas pesquisas no Grupo de Sistema Distribuídos do ICMC-USP. Desta maneira, esta ferramenta vem ser o ponto de partida para pesquisas na área de IE neste grupo.

```
1  /**
2   * Actor is an abstract base class for all actors to be be put in the
3   *{@link antsystem.model.Environment}.
4   * These actors includes mainly all the agents in the system like {@link Ant3},
5   * and other types of actor
6   * such as {@link antsystem.model.FoodSource} and {@link antsystem.model.Nest}
7   *
8   * @author Danilo N. Costa
9   */
10 public abstract class Actor {
11     private Point position = new Point(0,0);
12     private Dimension dimension;
13     private int velX, velY;
14     private SpriteCache spriteCache;
15
16
17
18     /** Creates a new instance of Actor */
19     public Actor(SpriteCache spriteCache) {
20         this(spriteCache, null, null);
21     }
22
23
24     /**
25      * The bounding box of the actor
26      * @return The bounding box of the actor
27      */
28     public Rectangle2D getBoundingBox(){
29         return new Rectangle2D.Float(position.x,
30                 position.y,
31                 dimension.width,
32                 dimension.height);
33     }
34
35     /**
36      * Returns the current position of the Actor.
37      * @return The position is based on the most
38      *  top left point of the Actor.
39      */
40     public Point getPosition() {
41         return position;
42     }
43
44
```

Código 4.2.1: *Trecho do código da classe Actor*

Capítulo 5
Testes e Resultados

Os experimentos realizados têm como objetivo verificar se o *software* construído está de acordo com sua especificação e se satisfaz as expectativas do ponto de vista do usuário do sistema. Desta maneira os esforços são concentrados nos requisitos funcionais do *software* durante a fase de validação.

Tendo isso em vista, foram definidos dois conjuntos de testes funcionais para o simulador implementado. O primeiro deles incide nos testes de pontes, fortemente inspirado nos experimentos de pontes binárias realizados por Deneubourg et al. (1990) e Goss et al. (1989) com formigas reais, elucidados na seção 2.1.2, que consistem na determinação de dois caminhos diferentes com início em um mesmo local, próximo ao ninho, que levam até uma fonte de alimento comum. Durante a execução destes testes as escolhas feitas pelas formigas foram analisadas enquanto realizavam a tarefa de forrageamento, observando as possíveis alterações quantitativas na escolha pelas pontes, ao decorrer de cada experimento. Já o segundo tipo de teste versa sobre ambientes mais abertos, sem pré-definições de possíveis rotas, contendo, virtualmente, infinitos caminhos factíveis entre o ninho e a fonte de alimento, podendo inclusive, ser estabelecida mais de uma fonte de alimento. Em ambos os casos o objetivo foi verificar a capacidade do grupo de forrageadores de estabelecerem uma rota específica entre o ninho e a fonte de alimento, e aferir a influência que as diferentes configurações dos parâmetros exercem sobre os testes.

5.1 Testes Com Pontes

Os testes com as pontes foram divididos em dois grupos: um com duas pontes de mesmo comprimento, e outro com duas pontes de comprimentos diferentes. Para ambos, as ferramentas de criação de obstáculos foram utilizadas de maneira que barreiras fossem inseridas compondo dois corredores distintos, tendo uma das extremidades desembocando no ninho e a outra na fonte de alimento. Em todos os testes foi verificado o número de agentes em cada uma das pontes ao longo do experimento. Cada experimento foi realizado 50 vezes e em cada um deles contou-se, visualmente, o número de agentes presente em cada uma das pontes, denominadas de ponte A e ponte B, nos seguintes intervalos de iteração: 100, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 5500, 6000, 6500 e 7000. Foram utilizados um total de 20 agentes em cada teste, contudo, nem sempre a soma total dos agentes nas pontes A e B resulta em 20, pois alguns poderiam estar tanto no ninho como na fonte de alimento.

5.1.1 Caminhos iguais

Para a realização dos testes com caminhos iguais, foram criados dois corredores de comprimento muito semelhante, aferido visualmente, com o ninho localizado na parte inferior e a fonte de alimento na parte superior, como mostra a figura 5.1. No primeiro teste não foi considerada a evaporação de feromônios, configurando-se a taxa de evaporação em zero. A tabela 5.1 mostra o valor utilizado na configuração de todos os parâmetros disponíveis. Essa configuração foi resultado de uma série de pequenos testes realizados com o intuito de se avaliar o melhor conjunto de parâmetros a ser empregado nos experimentos.

Os valores médios obtidos pelos 50 experimentos, com a contagem do número de agentes em cada uma das pontes, nos mesmos intervalos de iteração, se encontram na tabela 5.2. Como pode ser visto na figura 5.2 com o gráfico plotado com esses valores, a média de resultados mostra que houve um equilíbrio na alternância pelo uso das pontes, ou seja, as duas pontes foram utilizadas igualmente. No entanto, através da análise individual de cada um dos testes o comportamento global mostrou que inicialmente

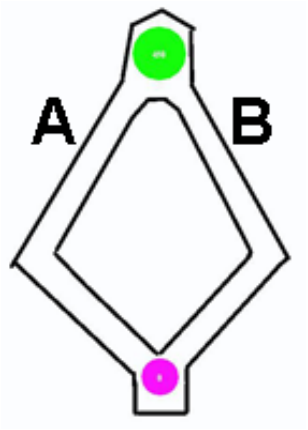

Figura 5.1: Pontes utilizada para os experimentos de caminhos de mesmo comprimento.

existe uma fase de grande desequilíbrio que com o passar do tempo diminui até chegar bem próxima do equilíbrio, como mostra por exemplo os dados do experimento número 1, representado pela figura 5.3 com a captura de tela em três momentos distintos: (a) por volta da iteração 550, com predomínio dos agentes na ponte A; (b) e (c) por volta de 3000 e 5500 iterações, respectivamente, que mostram ambas as pontes reforçadas de feromônio. A tabela 5.3 apresenta os valores obtidos para este experimento em questão, demonstrado graficamente pela figura 5.3.

Em cada experimento também observou-se o momento em que todo o alimento havia se esgotado, sendo que em média isso ocorreu por volta de 6300 iterações.

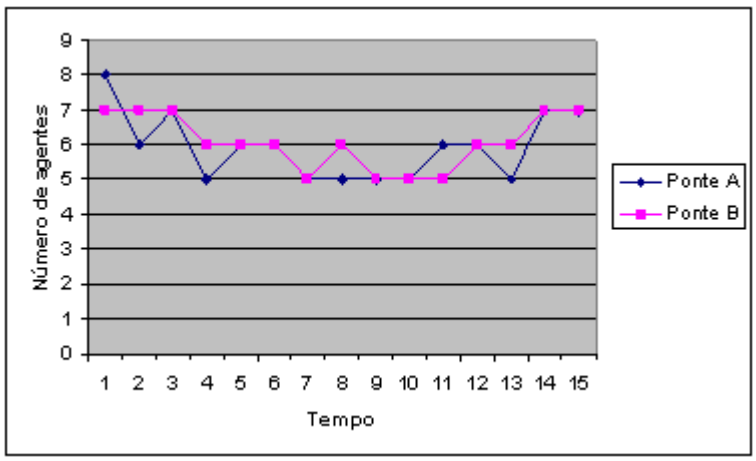

Figura 5.2: Média do número de agentes nas pontes de mesmo comprimento, sem evaporação do feromônio.

No segundo teste, foi utilizado exatamente o mesmo ambiente, respeitando as configurações dos parâmetros conforme descritos na tabela 5.1 com exceção dos valores para

Tabela 5.1: Configuração dos parâmetros para experimentos de duas pontes sem evaporação do feromônio.

Parâmetro	**Valor**
Pheromone Evaporation Interval	N/A
Pheromone Evaporation Rate	0
Nest Size	20
No Turn	96%
Clock wise 45°	2%
Clock wise 90°	0%
Clock wise 135°	0%
Clock wise 180°	0%
C.Clock wise 45°	2%
C.Clock wise 90°	0%
C.Clock wise 135°	0%
Without Food Pheromone Weight	5
Carrying Food Pheromone Weight	8
Without Food Pheromone Threshold	10
Carrying Food Pheromone Threshold	10
Without Food Pheromone Deposit	15
Carrying Food Pheromone Deposit	20
Velocity	2

intervalo e taxa de evaporação que foram configurados conforme exposto na tabela 5.4. A figura 5.5 exibe o experimento em três momentos distintos: (a) no momento da iteração 550, quando a maior parte dos agentes está carregando comida no trajeto de volta ao ninho; (b) na iteração 3000, com um reforço maior de feromônio na ponte A, e (c) na iteração por volta do número 5500, na qual a fonte de alimento já havia se extinguido. As médias de valores para esse caso são apresentadas na tabela 5.5. A média de iterações no qual a fonte de alimento havia se extinguido foi de 3500. A figura 5.6 mostra o gráfico plotado para esses valores.

Tabela 5.2: Média do número de agentes em cada um dos ramos para testes realizados com pontes de mesmo comprimento e sem a evaporação do feromônio.

Iterações	**Ponte A**	**Ponte B**
100	8	7
500	6	7
1000	7	7
1500	5	6
2000	6	6
2500	6	6
3000	5	5
3500	5	6
4000	5	5
4500	5	5
5000	6	5
5500	6	6
6000	5	6
6500	7	7
7000	7	7

Discussão dos resultados

Na execução dos dois tipos de teste (sem e com evaporação do feromônio), pode-se observar que em ambos os casos os agentes foram capazes de encontrar a fonte de alimento e retornar ao ninho. Pela análise dos resultados do primeiro caso, percebe-se que após uma variação inicial, ambas as pontes foram escolhidas de maneira aleatória recebendo reforço de feromônio, e o número de agentes praticamente equilibrado em cada uma delas ao longo de todo o experimento. Isso indica que a capacidade de análise implícita da colônia sobre a qualidade dos caminhos classificou ambos como equivalentes, o que de fato são. Já com uma taxa de evaporação considerável no sistema, observou-se que a mesma exerceu forte influência no comportamento e análise implícita dos resultados, realizada pelos agentes. Após uma variação inicial, a colônia se concentrou principalmente em apenas um dos possíveis caminhos. Isto permite inferir que o reforço positivo implicado pelo agente que por ventura completou um dos caminhos em menor tempo, aliado ao reforço negativo da evaporação do feromônio em ambas as pontes, acabou por causar um desequilíbrio considerável nas concentrações de feromônio nas pontes e fez com que, no decorrer do experimento, a colônia passasse

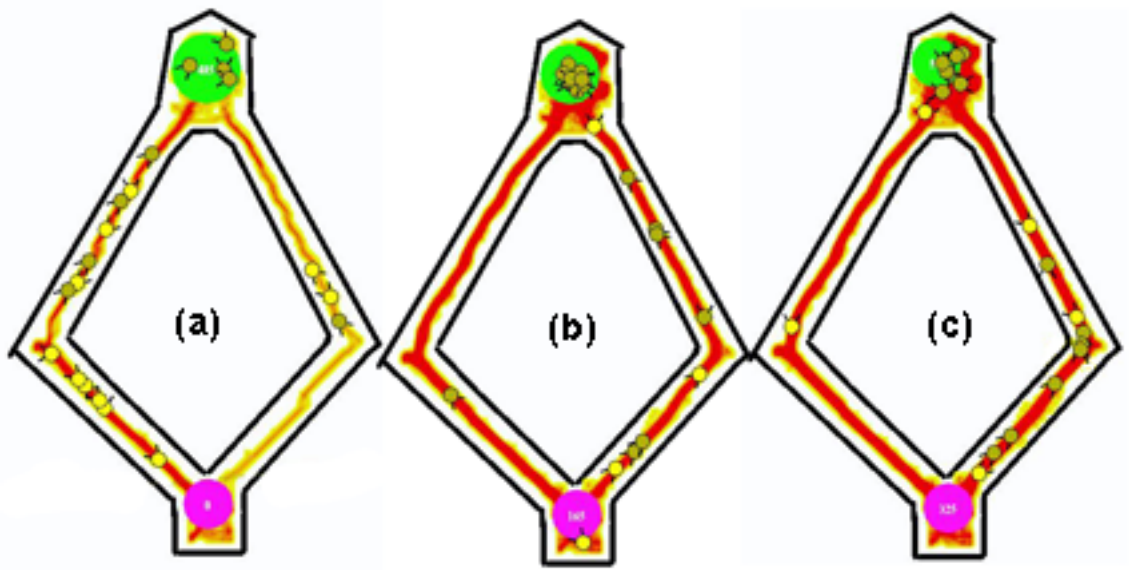

Figura 5.3: Primeiro experimento com pontes de mesmo comprimento, sem evaporação do feromônio.

a utilizar a ponte com maior reforço, evidenciando-a ainda mais, através do princípio da auto-catálise (Pasteels et al., 1987). Ainda sobre este experimento percebeu-se que com a evaporação do feromônio a eficiência global foi maior, fato observado através do momento em que a fonte de alimento se extingue, que para esse caso foi menor, por volta da iteração 3500, contra 6300 para o caso sem evaporação. Observou-se ainda, que após a extinção da fonte de alimento para o segundo caso, o desequilíbrio entre as pontes ficou menor, isso pode ser explicado pelo fato de que quando não estão carregando alimento o reforço no nível de feromônio exercido por um agente em um dado caminho não é tão intenso quando está com alimento.

5.1.2 Caminhos Diferentes

Os testes com caminhos diferentes foram realizados nos mesmos moldes dos anteriores, conforme descrito na seção 5.1.1. O que caracteriza estes experimentos são os comprimentos de cada ponte, dessa vez uma delas é maior que a outra, sendo a menor denominada ponte A e a menor B.

A figura 5.7 ilustra um dos experimentos realizados para o caso onde não havia a evaporação do feromônio em três momentos distintos, (a) por volta de 100 iterações (b) 1300 e (c) 2700. A média para a contagem do número de agentes em cada uma das

Tabela 5.3: Distribuição do número de agentes para o primeiro experimento, com pontes de mesmo comprimento, sem a evaporação do feromônio.

Iterações	**Ponte A**	**Ponte B**
100	10	3
500	14	6
1000	14	2
1500	5	5
2000	5	6
2500	4	7
3000	3	8
3500	5	7
4000	3	5
4500	6	8
5000	5	8
5500	4	4
6000	5	6
6500	8	7
7000	7	6

Tabela 5.4: Configuração dos parâmetros para experimentos de duas pontes iguais, com evaporação do feromônio.

Parâmetro	**Valor**
Pheromone Evaporation Interval	20
Pheromone Evaporation Rate	4

pontes para este caso encontra-se na tabela 5.6 ilustrada pelo gráfico da figura 5.8. O número médio de iterações para a extinção da fonte de alimento foi 6800.

A figura 5.9 ilustra o comportamento dos agentes nas pontes de comprimentos diferentes, com o uso da evaporação do feromônio. A tabela 5.7 corresponde a uma média dos resultados obtidos para estes experimentos, e é representada graficamente pela figura 5.10. Neste caso a extinção da fonte de alimento se deu por volta de 4000.

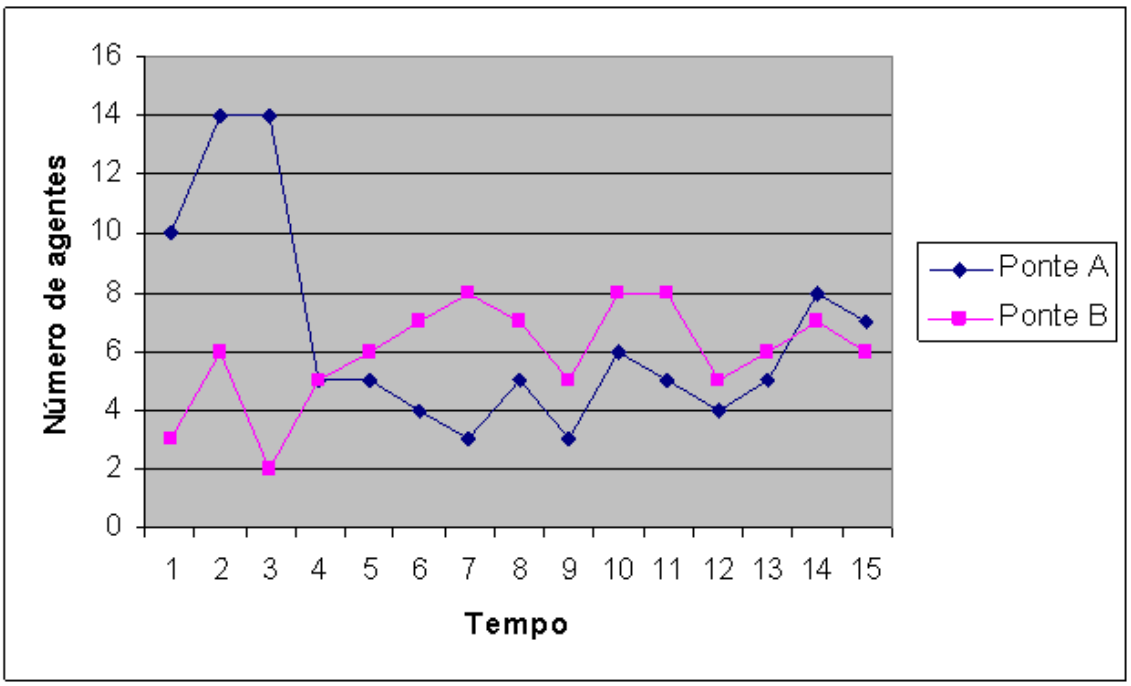

Figura 5.4: Distribuição do número de agentes entre as duas pontes sem evaporação do feromônio.

Discussão dos resultados

A análise dos resultados para os experimentos com pontes de tamanhos diferentes, novamente mostrou que os agentes foram capazes de encontrar a comida e retornar ao ninho. E ainda, como no caso anterior, a influência da evaporação do feromônio foi confirmada. Na situação em que a evaporação não estava presente, houve novamente uma tendência de equilíbrio na escolha entre os dois caminhos, mesmo sendo um deles mais comprido. O pequeno desequilíbrio se deu a favor do caminho mais curto e, como era de se esperar, essa diferença na escolha do trajeto foi amplificada no experimento com a evaporação de feromônio, evidenciando ainda mais a escolha pelo caminho mais curto. Por fim a eficiência global do sistema também foi maior para o caso da evaporação com o feromônio, na qual a extinção da fonte de alimento se deu por volta de 4000 iterações, em contraste com o caso sem a evaporação que ocorreu por volta de 6800 iterações.

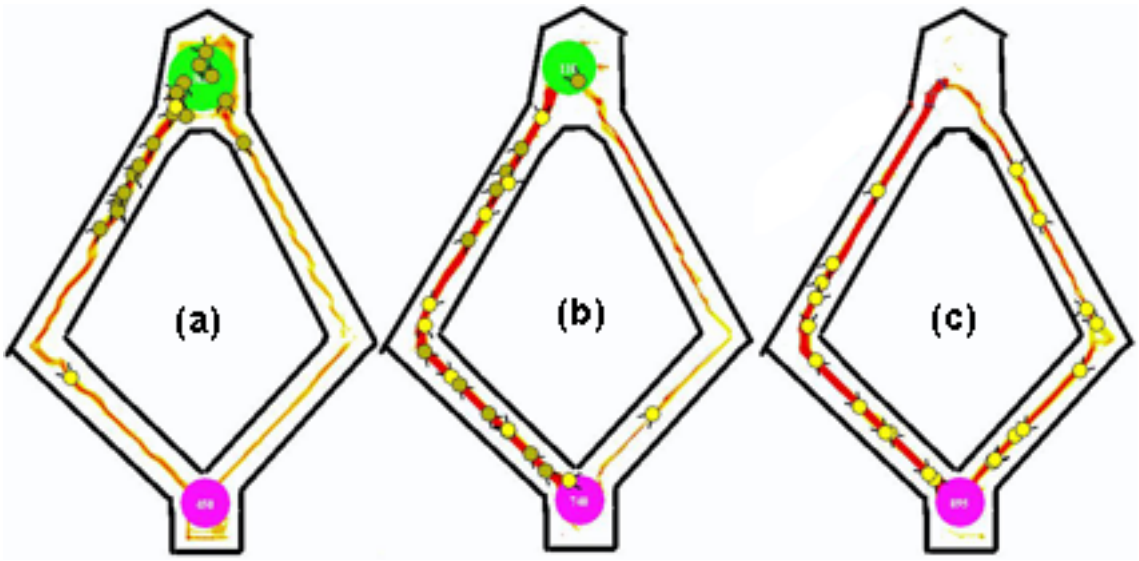

Figura 5.5: Pontes de mesmo comprimento em experimento com evaporação do feromônio.

5.2 Testes em Ambientes Abertos

Uma segunda categoria de testes para esse simulador foi realizada com o objetivo de analisar como os agentes se comportam em ambientes menos restritos do que aqueles contendo pontes, dando maior liberdade de movimentação. Dessa forma diversos cenários foram criados, com posições diversificadas para ninho, fonte de alimento e obstáculos. Diversas configurações dos parâmetros foram utilizadas na tentativa de compreender sua influência no comportamento do agente e do enxame. A figura 5.11 mostra dois exemplos nos quais os agentes foram capazes de encontrar um caminho entre a fonte de alimento e o ninho. Já a figura 5.12 mostra três momentos diferentes de um mesmo experimento. Em (a) tem-se a distribuição inicial que exibe a formação em dendrito, assim como observado nas formigas reais já citado na seção 2.1.1, em (b) observa-se uma trilha formada entre o ninho e a fonte de alimento criada a partir da formação de dendritos, em (c) é mostrado que a exploração do ambiente foi realizada de forma a criar trilhas complexas entre o alimento e o ninho.

5.3 Discussão

A utilização de feromônios como comunicação indireta durante a navegação de agentes permite a sinalização e o recrutamento indireto para uma dada rota. A retro-

Tabela 5.5: Média do número de agentes em cada um dos ramos para testes realizados com pontes de mesmo comprimento e sem a evaporação do feromônio.

Iterações	**Ponte A**	**Ponte B**
100	9	8
500	15	3
1000	16	4
1500	15	3
2000	15	5
2500	17	3
3000	19	1
3500	18	2
4000	17	3
4500	15	5
5000	10	7
5500	14	4
6000	11	9
6500	13	4
7000	13	5

alimentação positiva, através do reforço de uma trilha já marcada, juntamente com a retroalimentação negativa, através da evaporação do feromônio, propicia uma avaliação implícita da qualidade da solução como sugerido por Dorigo et al. (1999) e verificado nos experimentos com as pontes binárias. Contudo percebeu-se que a taxa de evaporação tem extrema influência sobre o sistema nos experimentos de ambiente aberto.

Na exploração de ambientes abertos a tarefa de forrageamento mostrou-se difícil de ser executada no modelo implementado, sendo bastante sensível às configurações de evaporação de feromônio (*Pheromone Evaporation Rate*), quantidade de feromônio depositada (*Pheromone Deposit*) e peso que ele exerce na tomada de decisão do movimento do agente (*Pheromone Weight*). Tal dificuldade sugere a possibilidade das formigas se utilizarem de outros meios no processo de formação das trilhas, tais como memória visual e orientação através da posição do sol, descritas nas pesquisas de Hölldobler e Wilson (1990), e , podem ser essenciais para o forrageamento eficiente em ambiente aberto.

O tamanho da área a ser explorada se mostrou bastante determinante no processo de escolha dos valores para os parâmetros. Para ambientes maiores a taxa de evaporação deve ser mais baixa do que para ambientes menores, para garantir que ao

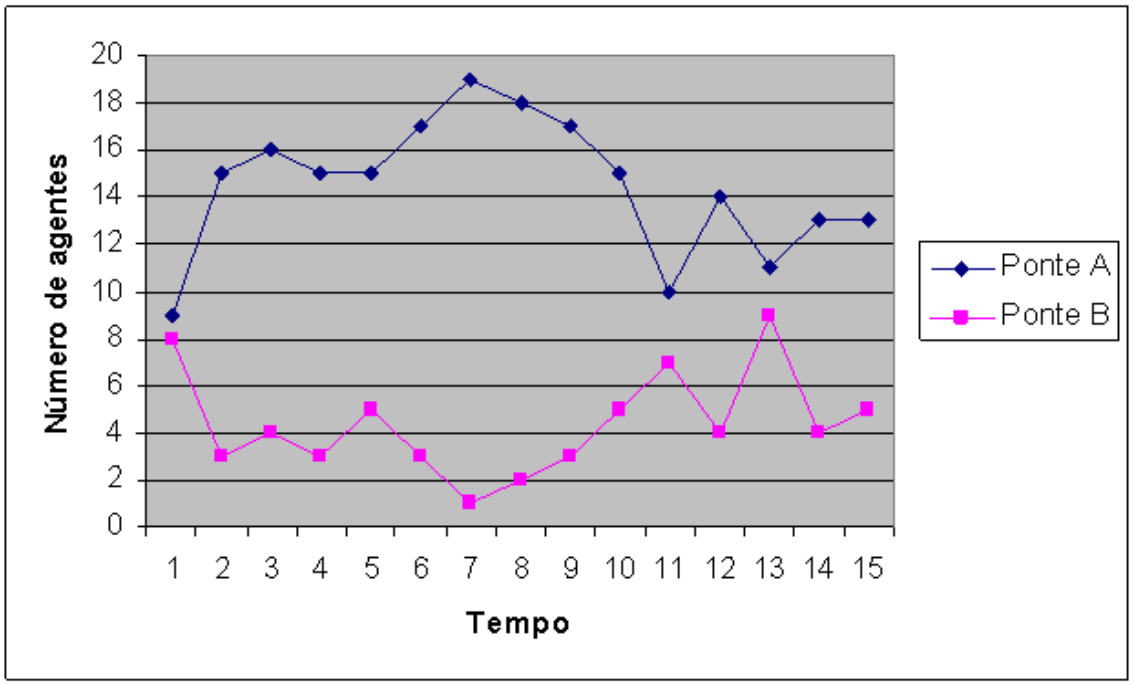

Figura 5.6: Distribuição do número de agentes entre as duas pontes considerando a evaporação do feromônio.

chegar na fonte de alimento, ainda exista uma trilha completa para guiar o agente de volta ao ninho, contudo isso propicia a criação de um ambiente mais carregado de feromônio, podendo deixar o agente confuso e perdido. Como tentativa de solução pode-se aumentar o limiar de ativação (*Pheromone Threshold*) e/ou diminuir o peso que o feromônio tem em sua tomada de decisão (*Pheromone Weight*). Essa grande quantidade de parâmetros distintos, aparentemente interdependentes, cria um modelo atraente para a aplicação de técnicas evolutivas de configuração de valores tais como os algoritmos genéticos, o que é proposto como trabalho futuro.

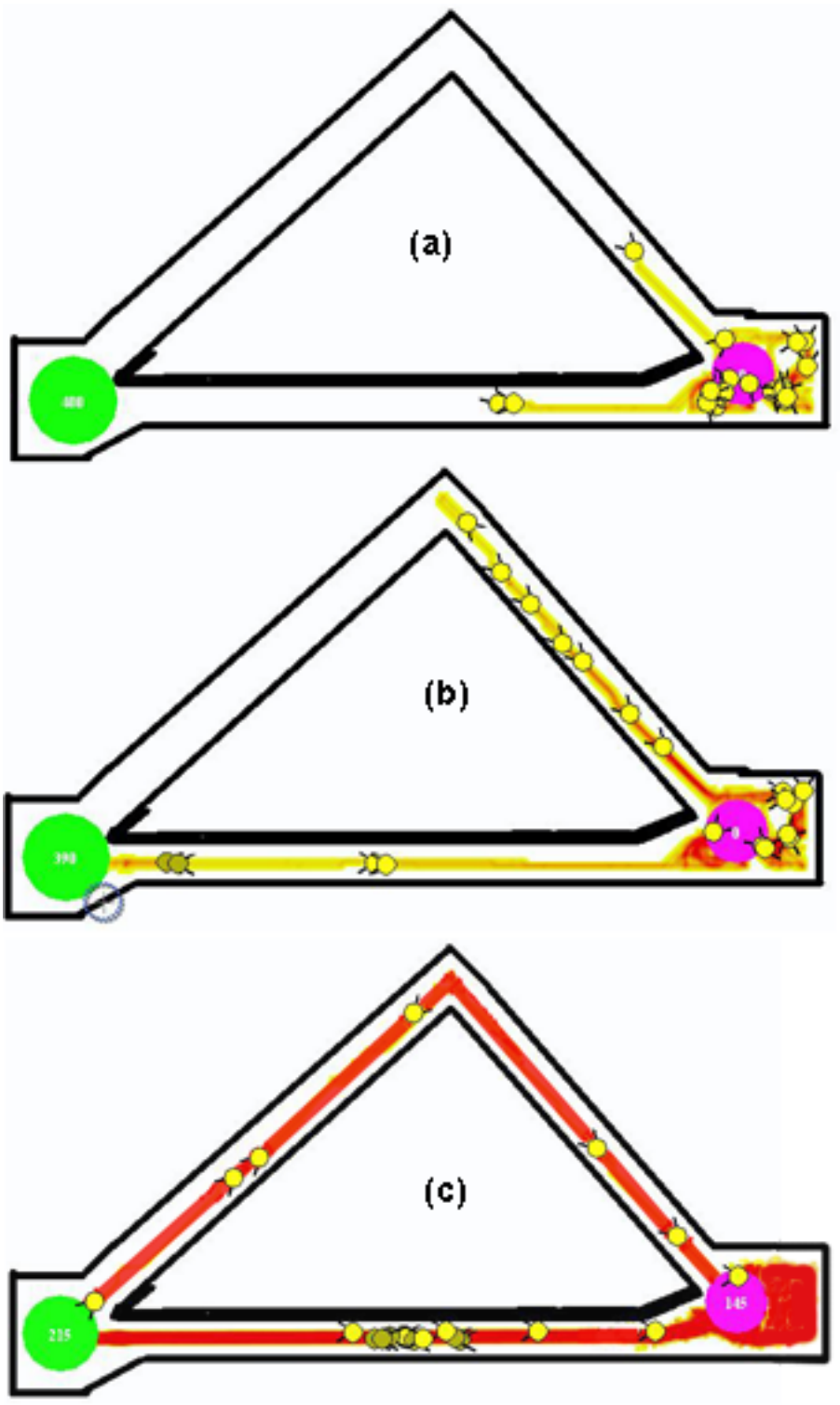

Figura 5.7: Pontes de comprimentos diferentes, sem evaporação de feromônio. Iterações: (a) 100, (b) 1300 e (c) 2700.

Tabela 5.6: Média do número de agentes em cada um dos ramos para testes realizados com pontes de comprimento diferente, sem evaporação do feromônio.

Iterações	**Ponte A**	**Ponte B**
100	9	2
500	13	6
1000	11	9
1500	12	8
2000	10	9
2500	9	10
3000	11	9
3500	9	9
4000	11	10
4500	10	9
5000	11	9
5500	10	8
6000	10	10
6500	10	8
7000	10	9

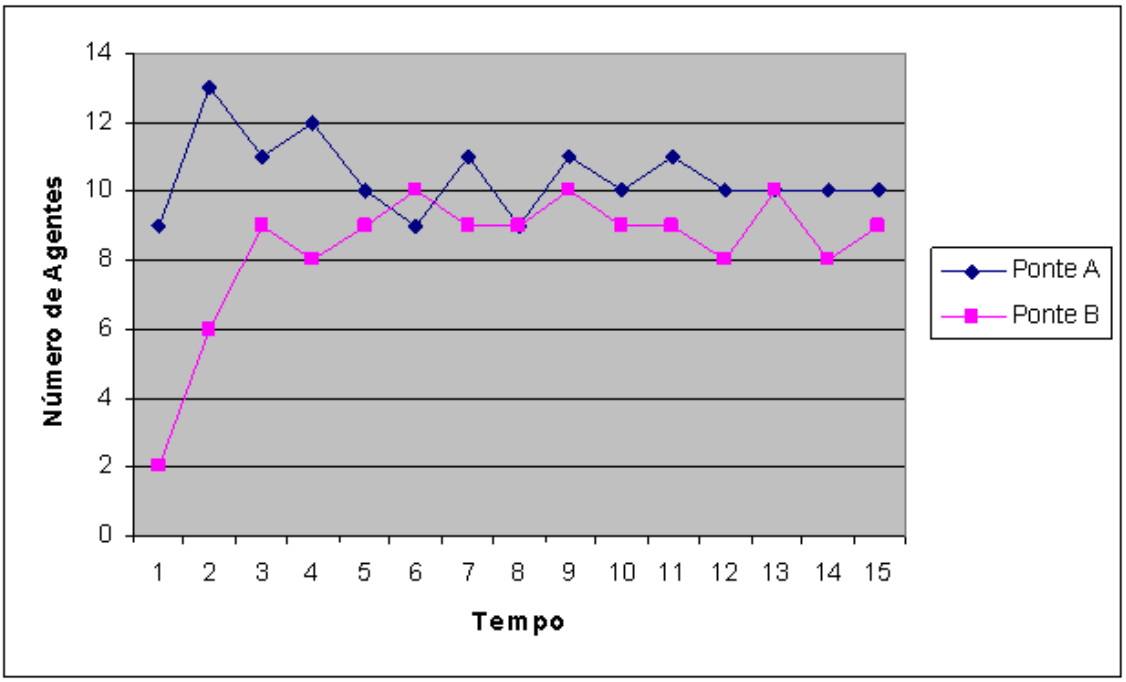

Figura 5.8: Gráfico de pontes de caminhos diferentes, sem evaporação.

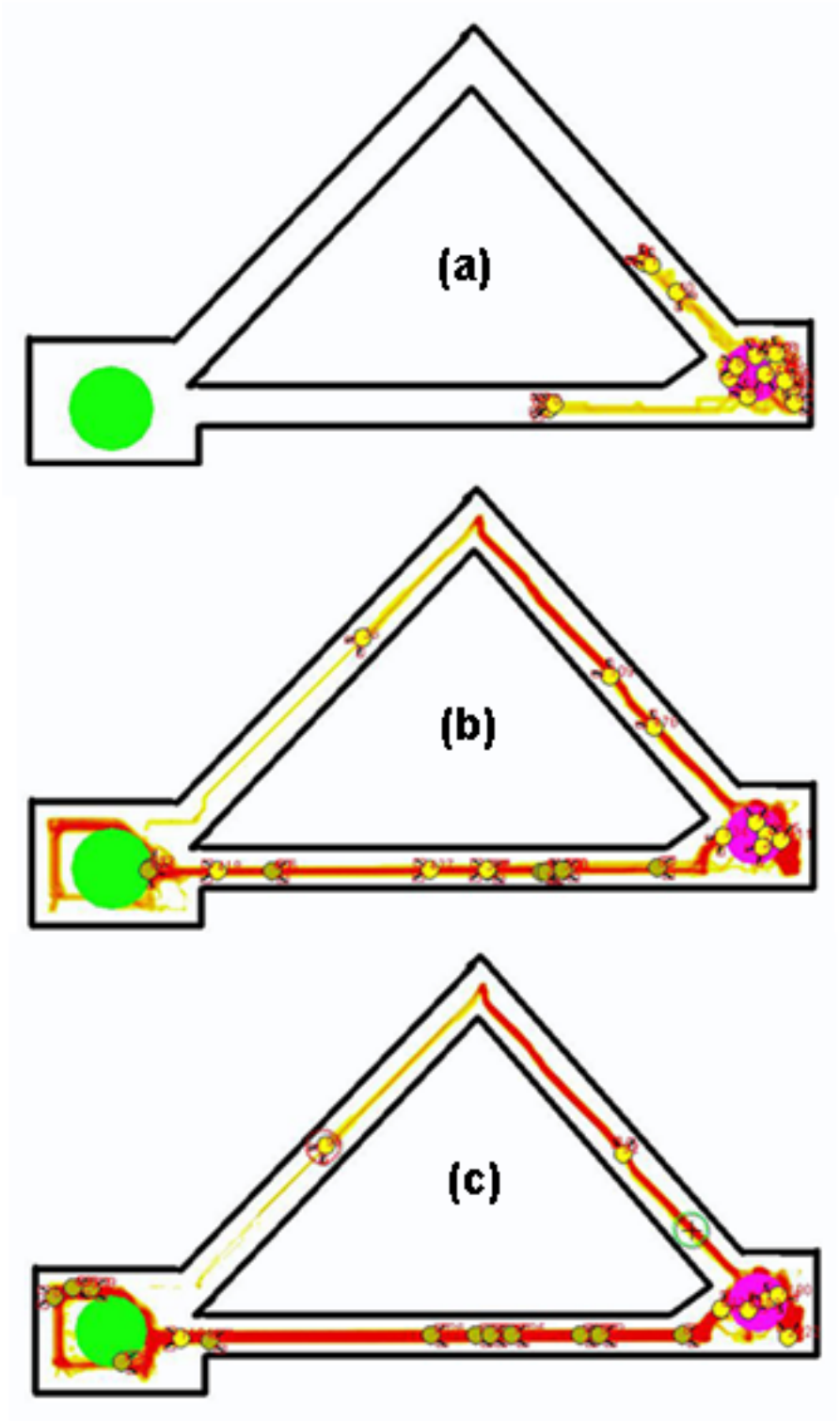

Figura 5.9: Pontes de comprimentos diferentes, com evaporação de feromônio.Iterações: (a) 100, (b) 1300 e (c) 2700.

Tabela 5.7: Média do número de agentes em cada um dos ramos para testes realizados com pontes de comprimento diferente, com evaporação do feromônio.

Iterações	**Ponte A**	**Ponte B**
100	5	4
500	11	6
1000	14	6
1500	17	3
2000	15	5
2500	14	6
3000	13	7
3500	15	5
4000	12	8
4500	12	8
5000	13	7
5500	11	8
6000	11	9
6500	11	9
7000	12	8

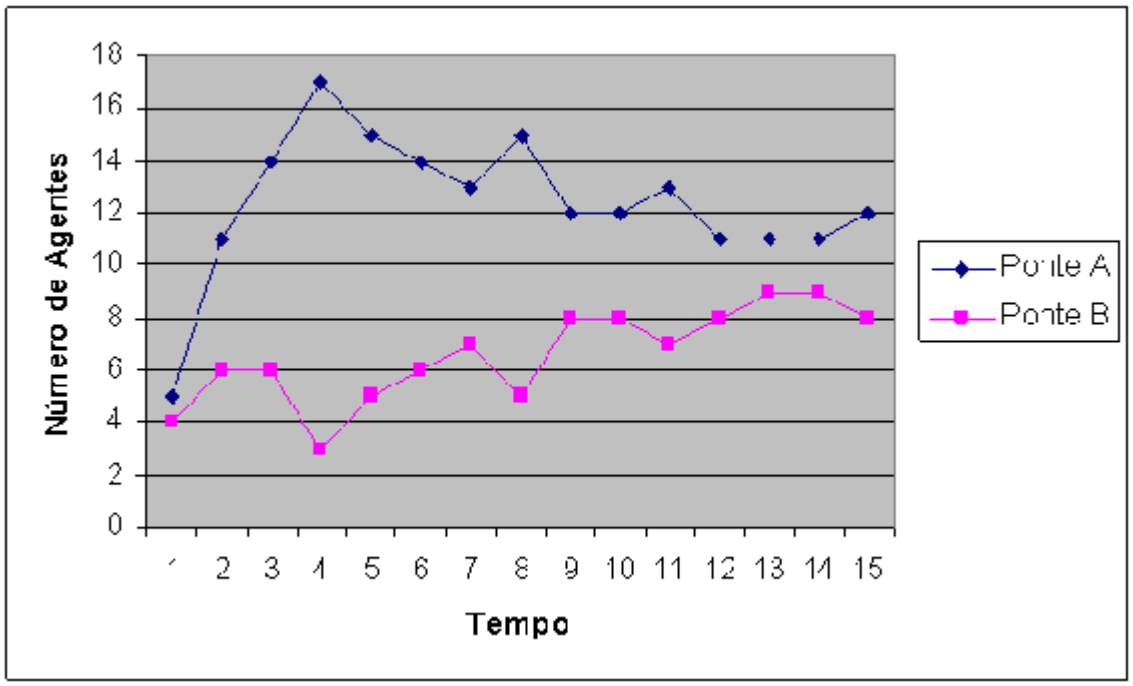

Figura 5.10: Gráfico de pontes diferentes, com evaporação de feromônio.

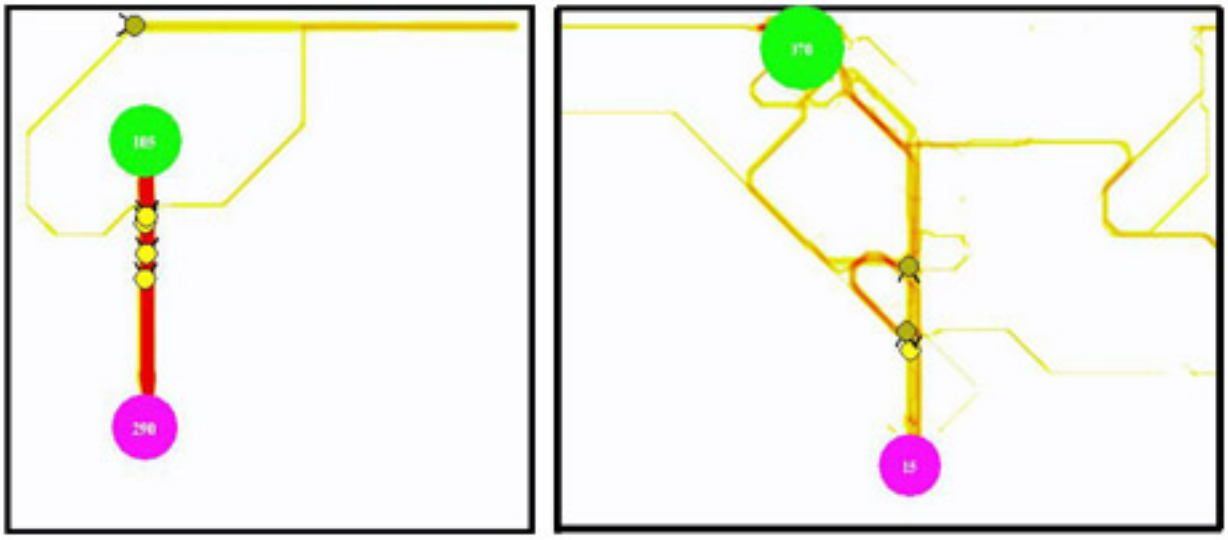

Figura 5.11: Dois experimentos distintos em ambiente aberto.

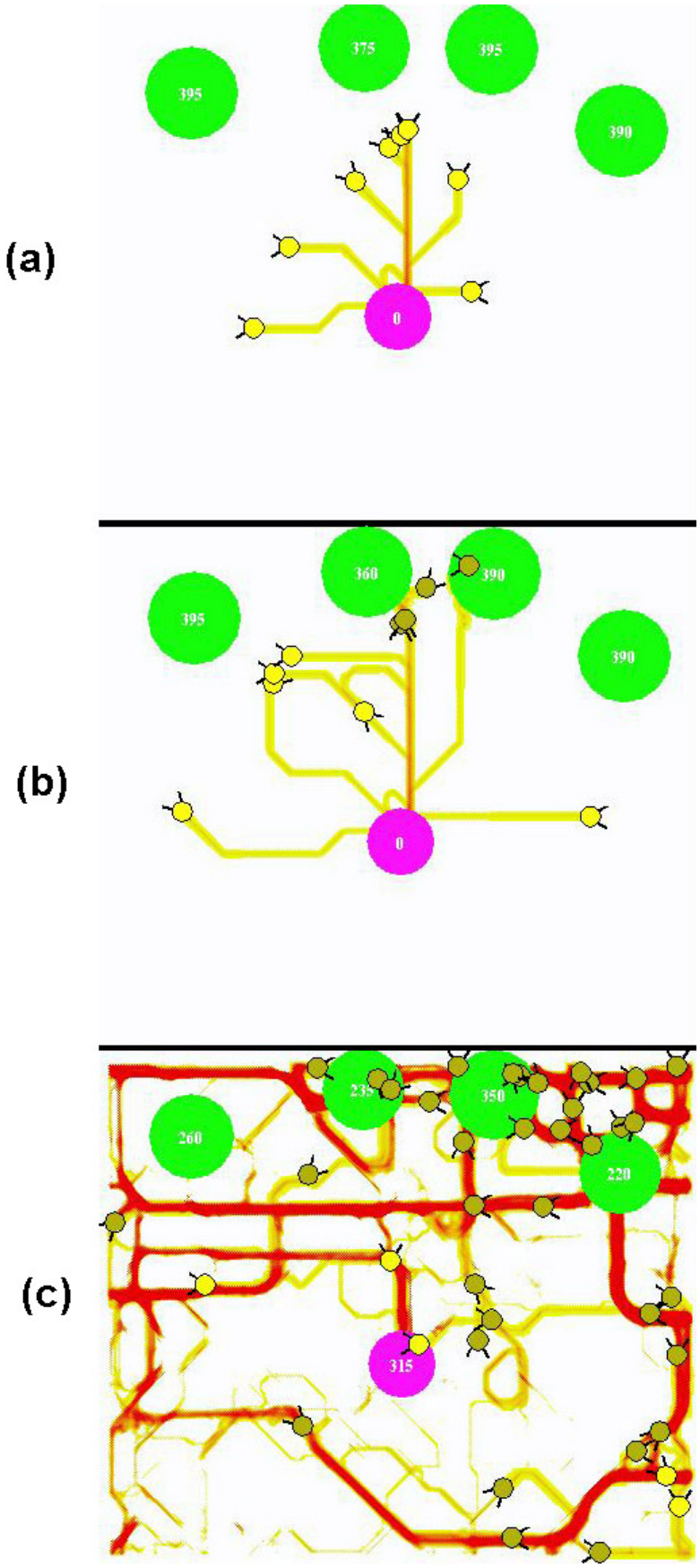

Figura 5.12: Sequência de um experimento em ambiente aberto. (a) formação inicial em dendrito, (b) uma trilha formada por um dos ramos do dendrito e (c) trilhas complexos criadas entre o ninho e as fontes de alimento.

Capítulo 6
Conclusão

O presente trabalho estudou as propriedades das colônias de formigas e seu comportamento adaptativo na tarefa de forrageamento, com enfoque na comunicação indireta através da modificação do ambiente, e aplicou estes conceitos na criação de um simulador de multi-agentes. Um modelo foi definido de maneira que parâmetros importantes fossem configurados para o estudo de sua influência no comportamento global do sistema. O simulador foi concebido tendo em mente que sua arquitetura fosse tal que permitisse o seu reuso e expansão, para tanto buscou fundamentos na engenharia de *software*. Outro ponto considerado durante seu desenvolvimento foi em relação a sua interface, que deveria ser de fácil utilização e intuitiva, de modo que sua configuração e execução fossem simples.

Essa dissertação analisou as principais plataformas existentes para o desenvolvimento de simulação baseada em agentes, ponderando suas qualidades e fraquezas, contudo, nenhuma delas mostrou atender as especificidades requeridas, descritas na seção 4.1.4. Tendo isso em vista, implementou-se um novo simulador que conta com algumas características importantes: (i) como o uso de projeto orientado a objeto, que trouxe um alto grau de modularidade o qual, por sua vez, admite futuras modificações e fácil manutenção, (ii) uma ampla documentação que inclui diagramas *UML*, análise de requisitos, descrição do projeto e da fase de testes, bem como páginas *HTML* com a documentação do código implementado, o que simplifica estudos e referências futuras, (iii) e o desenvolvimento em linguagem Java, de maneira que viabilizou o sistema ser completamente independente de sistema operacional, somado ao fato de ser uma linguagem amplamente difundida e utilizada que permite, através da disponibilização de

código aberto, uma fácil adoção por outros grupos de pesquisa e desenvolvedores em todo o mundo, sendo esta a razão para a adoção da língua inglesa na escrita do código e documentação do mesmo.

No entanto, a maior contribuição deste projeto foi a criação de uma ferramenta de simulação pronta para a execução de diversos experimentos acerca da comunicação indireta em sistemas de múltiplos agentes autônomos, através da criação de uma interface rica em painéis de configuração claros e objetivos, dando suporte ao estudo e controle de diversas características. Essas características, tais como taxa de evaporação e limiar de ativação do feromônio, por sua vez se encontram amparadas e fundamentadas na revisão bibliográfica apresentada nessa dissertação.

6.1 Trabalhos Futuros

A possibilidade de extensão da ferramenta advinda desse trabalho torna-a propícia e natural para expansões, dentre elas pode-se destacar:

- A utilização conjunta de mais de um tipo de feromônio analisando sua eficiência na exploração de ambientes.
- A modelagem e implementação de diferentes tipos de agentes, permitindo que interajam ao mesmo tempo no ambiente.
- Criação de um módulo de extensão para execução automática de uma série de experimentos em lote.
- Criação de um módulo de extensão para seleção automática de parâmetros através de algoritmos evolutivos.

Além destes, pretende-se elaborar um conjunto de testes a fim de abstrair um algoritmo para navegação de ambientes a ser embarcado em um time de robôs móveis autônomos.

Referências Bibliográficas

Ahmed, K. Z. e Umrysh, C. E. (2003). *Developing Enterprise Java Aplications with J2EE and UML*. Addison-Wesley.

An, G. (2001). Agent-based computer simulation and sirs: Building a bridge between basic science and clinical trials. *Shock*, v. 16:p. 266–273.

Angelis, D. L. D. e Mooij, W. M. (2005). Individual-based modeling of ecological and evolutionary processes. *Annual Review of Ecology Evolution and Systematics*, v. 36:p. 147–68.

Appleby, S. e Steward, S. (2000). Mobile software agents for control in telecommunication networks. *British Telecom Technical Journal*, v. 18:p. 68–70.

Arkin, R. (1998). Behavior-based robotics. *Cambridge, MA: MIT Press*.

Aron, S., Beckers, R., Deneubourg, J. L., e Pasteels, J. M. (1993). Memory and chemical communication in the orientation of two mass-recruiting ant species. *Insect Society*, v. 40:p. 369–380.

Aron, S., Deneubourg, J.-L., Goss, S., e Pasteels, J. (1990). *Biological Motion, Lecture Notes in Biomathematics.*, chapter Functional self-organization illustrated by internest traffic in the argentine ant Iridomyrmex humilis., pag. p. 533–547. Berlim: Springer-Verlag.

Beckers, R., Deneubourg, J.-L., e Goss, S. (1992). Trail laying behaviour during food recruitment in the ant lasius niger (l.). *Insects Sociaux*, v. 39:p. 59–72.

Beckers, R., Deneubourg, J.-L., e Goss, S. (1993). Modulation of trail laying in the ants lasius niger (hymenoptera: Formicidae) and its role in the collective selection of a food source. *Journal of Insect Behavior*, v. 6:p. 751–759.

Belew, R. K. e Mitchell, M. (1996). *Adaptive Individuals in Evolving Populations: Models and Algorithms*, volume v. 26. SFI Studies in the Sciences of Complexity.

Beni, G. e Wang, J. (1989). Swarm intelligence in cellular robotics systems. *Proceeding of NATO Advanced Workshop on Robots and Biological System*.

Bonabeau, E., Dorigo, M., e Theraulaz, G. (1999). Swarm intelligence: From natural to artificial systems. *NY: Oxford University Press*.

Bonabeau, E. e Theraulaz, G. (2002). Swarm smarts. *Scientific American*, pag. p. 82–90.

Bonabeau, E., Theraulaz, G., e Deneubourg, J. L. (1998). Group and mass recruitment in ant colonies: The influence of contact rates. *Journal of Theoretical Biology*, v. 195:p. 157–166.

Bonabeau, E., Theraulaz, G., L., D. J., S., A., e Camazine, S. (1997). Self-organisation in social insects. *TREE, n^{o}5*, v. 12:p. 188–193.

Bossert, W. H. e Wilson, E. (1963). The analysis of olfactory communication among animals. *Journal of Theoretical Biology*, v. 5:p. 443–469.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., e Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons.

Camazine, S., Deneubourg, J., Franks, N., Sneyd, S., Bonabeau, E., e Theraulaz, G. (2001). Self-organizing biological systems. *Princeton University Press*.

Cammaerts, M. C. e Cammaerts, R. (1980). Food recruitment strategies of the ants myrmica sabuleti and myrmica ruginodis. *Behaviour Process*, v. 5:p. 251–270.

Campione, M., Walrath, K., e Huml, A. (2000). *The Java Tutorial, Third Edition: A Short Course on the Basics*. Addison Wesley.

Campos, L. d., Gamez, J. A., e Puerta, J. M. (2002). Learning bayesian networks by ant colony optimization: Searching in two different spaces. *Mathware & Soft Computing*.

Caro, G. e Dorigo, M. (1998). Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, v. 9:p. 317–365.

Cassillas, J., Cordon, O., e Herrera, F. (2000). Learning fuzzy rules using ant colony optimization algorithms. *Proceedings of ANTS2000 - From Ant Colonies to Artificial Ants*, pag. p. 13–21.

Cassillas, J., Gordon, O., Viana, I. F. d., e Herrera, F. (2005). Learning cooperative linguistic fuzzy rules using the best-worst ant system agorithm. *Int. Journal of Intelligent Systems*, v. 20:p. 433–452.

Colorni, A., Dorigo, M., Maniezzo, V., e Trubian, M. (1994). Ant system for job shop scheduling. *Belgian Journal Operations Research*, v. 34:p. 39–53.

Davison, A. (2005). *Killer Game Programming in Java*. O'Reilly.

Deneubourg, J. L., Aron, S., Goss, S., e Pasteels, J. M. (1990). The self-organizing exploratory pattern of the argentine ant. *Journal of Insect Behavior*, v. 3:p. 159–168.

Deneubourg, J. L., GOSS, S., Franks, N., Sendova-Franks, A., DETRAIN, C., e CHRETIEN, L. (1991). The dynamics of collective sorting: Robot-like and ant-like robot. In *Proceedings of the First Conference on Simulation of Adaptative Behavior: From Animals to Animats*. Cambridge, MA: The MIT Press.

Deneubourg, J. L., Goss, S., Pasteels, J. M., Fresneau, D., e Lachaud, J. P. (1987). *Behavior in Social Insects*, volume v. 54, chapter Self-Organisation Mechanisms in Ant Societies (II): Learning in Foraging and Division of Labor, pag. p. 177–196. Experientia Supplementum, Birkhäuser Verlag.

Dorigo, M. (1992). *Optimization, learning and natural algorithms.* Tese de doutorado, Politecnico de Milano, IT.

Dorigo, M., Caro, G., e Gambardella, L. M. (1999). Ant algorithms for discrete optimization. *Artificial Life 5. Massachusetts: Institute of Technology*, pag. p. 137–172.

Dorigo, M. e Gambardella, L. (1997). Ant colony system: A cooperative learning aproach to the travelling salesman problem. *IEEE Transactions on Evolutionary Computation*, v. 1:p. 53–66.

Dorigo, M., Maniezzo, V., e Colorni, A. (1991). Positive feedback as a search strategy. Relatório técnico, Politecnico di Milano, IT, Dipartimento di Elettronica.

Dorigo, M., Maniezzo, V., e Colorni, A. (1996). The ant system: Optimization by a colony of cooperating agents. *Submitted to IEEE Transactions on Systems, Man, and Cybernetics. Part-B*, pag. p. 29–41.

Dudek, G., Jenkin, M., Milios, E., e Wilkes, D. (1993). A taxonomy for swarm robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems '93*.

Epstein, J. M. e Axtell, R. (1996). Growing artificial societies: Social science from the bottom up. *The MIT Press, Cambridge, MA*.

Flake, G. (1999). The computational beauty of nature. *Cambridge, MA: MIT Press.*

Fogel, D. B. (1995). Evolutionary computation. *Piscataway, NJ: IEEE Press*.

Fowler, M. (2004). *UML Distilled Third Edition*. Addison-Wesley.

Franklin, S. e Graesser, A. (1996). Is it an agent, or just a program?: A taxonomy for autonomous agents. *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*.

Franks, N. R., Gomez, N., Goss, S., e Deneubourg, J. L. (1991). The blind leading the blind in army ant raid patterns: Testing a model of self-organisation (hymenoptera: Formicidae). *Journal of Insect Behavior*, v. 4:p. 583–607.

Freeman, E. e Freeman, E. (2004). *Head First Design Patterns*. O'Reilly.

Fukuda, T. (1998). Plenary address to the world congress on computational intelligence. *Anchorage, Alaska*.

Gordon, D. M. (1992). *Phenotypic Plasticity*. Cambridge, MA: The Harvard University Press.

Gordon, D. M. (1996). The organisation of work in social insect colonies. *Nature*, v. 380:p. 121–124.

Gordon, D. M. (1999a). Ants at work. how an insect society is organised. *New York, Free Press.*

Gordon, D. M. (1999b). *Information Processing in Social Insects*, chapter Interaction Patterns and Task Allocation in Ant Colonies, pag. p. 51–67. Basel, Switzerland: Birkhauser Verlag.

Goss, S., Aron, S., Deneubourg, J.-L., e Pasteels, J. M. (1989). Self-organized shortcuts in the argentine ant. *Naturwissenschaften.*, v. 76:p. 579–581.

Grassé, P. P. (1946). Les insects dans leur univers. *Paris: ´Editions du Palais de la d´ecouverte.*

Grassé, P. P. (1959). La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. la th´eorie de la stigmergie: Essai d'interpr´etation du comportement des termites constructeurs. *Insectes Sociaux*, v. 6:p. 41–81.

Grimm, V. e Railsback, S. F. (2005). Individual-based modeling and ecology. *Princeton, NJ: Princeton University Press.*

Grimm, V., Revilla, E., Berger, U., Jeltsch, F., Mooij, W. M., e Railsback, S. F. e. a. (2005). Pattern-oriented modeling of agent-based complex systems: Lessons from ecology. *Science*, v. 310:p. 987–991.

Gutiahr, W. J. (2003). A converging aco algorithm for stochastic combinatorial optimization. *Lecture Notes in Computer Science*, pag. p. 10–25.

Hackwood, S. e Beni, S. (1992). Self-organization of sensors for swarm intelligence. *IEEE Int. Conference on Robotics and Automation*, pag. p. 819–829.

Haken, H. (1997). Synergetics. *Springer Verlag, Berlin, Germany.*

Hangartner, W. (1969). Carbon dioxide, a releaser for digging behavior in solenopsis geminata (hymenoptera: Formicidae). *Psyche*, pag. p. 58–67.

Hemrajani, A. (2006). *Agile Java Development with Spring, Hibernate and Eclipse.* Sams.

Hölldobler, B., Möglich, M., e Maschwitz, U. (1974). Communication by tandem running in the ant camponotus sericeus. *Journal of Comparative Physiology*, v. 90:p. 105–127.

Hölldobler, B. e Wilson, E. O. (1990). *The Ants.* The Belknap Press of Harvard University Press-Cambridge, MA.

Hodgins, J. e Brogan, D. (1994). Robot herds: Group behaviors for systems with significant dynamics. *Proceedings of Artificial Life IV*, pag. p. 319–324.

Holland, J. H. (1975). Adaptation in natural and artificial systems. *The University of Michigan Press, Ann Arbor.*

Hoskins, D. A. (1995). An interated function systems approach to emergence. In *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA, MIT Press.

Karlson, S. e Lüshcer, M. (1959). "pheromones", a new term for a class of biologically active substances. *Nature*, v. 183:p. 55–56.

Kennedy, J., Eberhart, R. C., e Shi, Y. (2001). *Swarm Intelligence*. Morgan Kaufmann Publishers.

Langton, C. G. (1989). *Artificial Life.*, volume v. 6. Addison-Wesley.

Law, J. e Regnier, F. (1971). Pheromones. *Annual Review of Biochemistry*, v. 40:p. 533–548.

Lorenz, K. (1973). *Behind the mirror: A Search for a Natural History of Human Knowledge*. New York: Harcourt Brace Jovanovich.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., e Balan, G. (2005). Mason: A multiagent simulation environment. *Simulation*, v. 81:p. 517–527.

Marietto, M., David, N., Sichman, J., e Coelho, H. (2002). Requirements analysis of agent-based simulation platforms. *Proceedings of Multi-Agent Based Simulation Workshop*.

Martinoli, A. (1999). *Swarm Intelligence in Autonomous Collective Robotics: from tools to the analysis and synthesis of distributed control strategies.* Tese (phd em ciências), École Polytechnique Fédérale de Lausanne, Lausanne.

Mataric, M. (1992). Minimizing complexity in controlling a mobile robot population. *IEEE Int. Conference on Robotics and Automation*.

Minar, N., Burkhart, R., Langton, C., e Askenazi, M. (1996). The swarm simulation system:a toolkit for building multi-agent simulations. Report no. 96-06-042., Santa Fe, NM: Santa Fe Institute.

Nicolis, G. e Prigogine, I. (1977). Self-organisation in non-equilibrium systems. *Wiley*.

Pacala, S. W., Gordon, D. M., e J., G. H. C. (1996). Effects of social group size on information transfer and task allocation. *Evolutionary Ecology*, v. 10:p. 127–165.

Pachter, M. e Chandler, P. (1998). Challenges of autonomous control. *IEEE Control Systems Magazine*, pag. p. 92–97.

Pasteels, J. M., Deneubourg, J. L., e Goss, S. (1987). Self-organization in ants societies (i):trail recruitment to newly discovered food sources. *In Behavior in Social Insects, Experientia Supplementum, Birkhäuser Verlag*, v. 4:p. 155–175.

Portha, S., Deneubourg, J. L., e Detrain, C. (2002). Self-organized asymmetries in ant foraging: a functional response to food type and colony needs. *Behavioral Ecology*, v. 13:p. 776–791.

Railsback, S. F., Lytinen, S. L., e Jackson, S. K. (2006). Agent-based simulation platforms: Review and development recommendations. *SIMULATION*, v. 82:p. 609–623.

Ramos, V. e Merelo, J. J. (2002). *Self-Organized Stigmergic Document Maps: Environment as a Mechanism for Context Learning*. Centro Univ. de Merida, Merida, Spain.

Raspinelli, J. M., Lopes, H. S., e Freitas, A. A. (2002). Data mining with an ant colony optimization algorithm. *IEEE Trans. Evolutionary Computation (Special Issue on Ant Colony Algorithm)*, v. 6:p. 321–332.

Reynolds, C. (1987). Flocks, herds, and schools: A distributed behavioral model. In *SIGGRAPH*.

Robinson, G. E. (1992). Regulation of vision of labor in insect societies. *Ann. Rev. Entomol.*, v. 37:p. 637–665.

Shtovba, S. (2005). Ant algorithms: Theory and applications. *Programming and Computer Software*, v. 31(nº 4):p. 167–178.

Sommerville, I. (2000). *Software Engineering*. Addison-Wesley.

Tobias, R. e Hofmann, C. (2004). Evaluation of free java-libraries for social-scientific agent based simulation. *Journal of Artificial Societies and Social Simulation*, v. 10.

Van Vorhis Key, S. E. e Baker, T. (1986). Observations on the trail deposition and recruitment behavior of the argentine ant, iridomyrmex humilis (hymenoptera: Formicidae). *Annals of the Entomological Society of America*, v. 79:p. 283–288.

Verhaeghe, J. C. (1982). Group recruitment in tetramorium caespitum. *Insectes Sociaux*, v. 29:p. 67–85.

Verhaeghe, J. C., Champagne, P., e Pasteels, J. (1980). Le recrutement alimentaire chez tapinoma erraticum (hym.,form). *Insectes Sociaux*, v. 30:p. 347–360.

Vittori, K. (2005). *Estudo experimental, modelagem e implementação do comportamento de colônias de formigas em um ambiente dinâmico*. Tese de doutorado, Escola de Engenharia de São Carlos, USP.

Vrionides, P. (2006). Swarm intelligence: An artificial life simulator exploring the role of local communication in task alocation and switching in ants. Master's thesis, University of Sheffield.

Wilson, E. O. (1971). The insect societies. *The Belknap Press of Harvard University Press*.

Wilson, E. O. (1978). *On Human Nature*. Harvard University Press-Cambridge, MA.

Yamins, D. (2005). Towards a theory of "local to global" in distributed multi-agent systems. *Harvard University*, pag. p. 183–190.

Apêndice A

Principais Diagramas UML

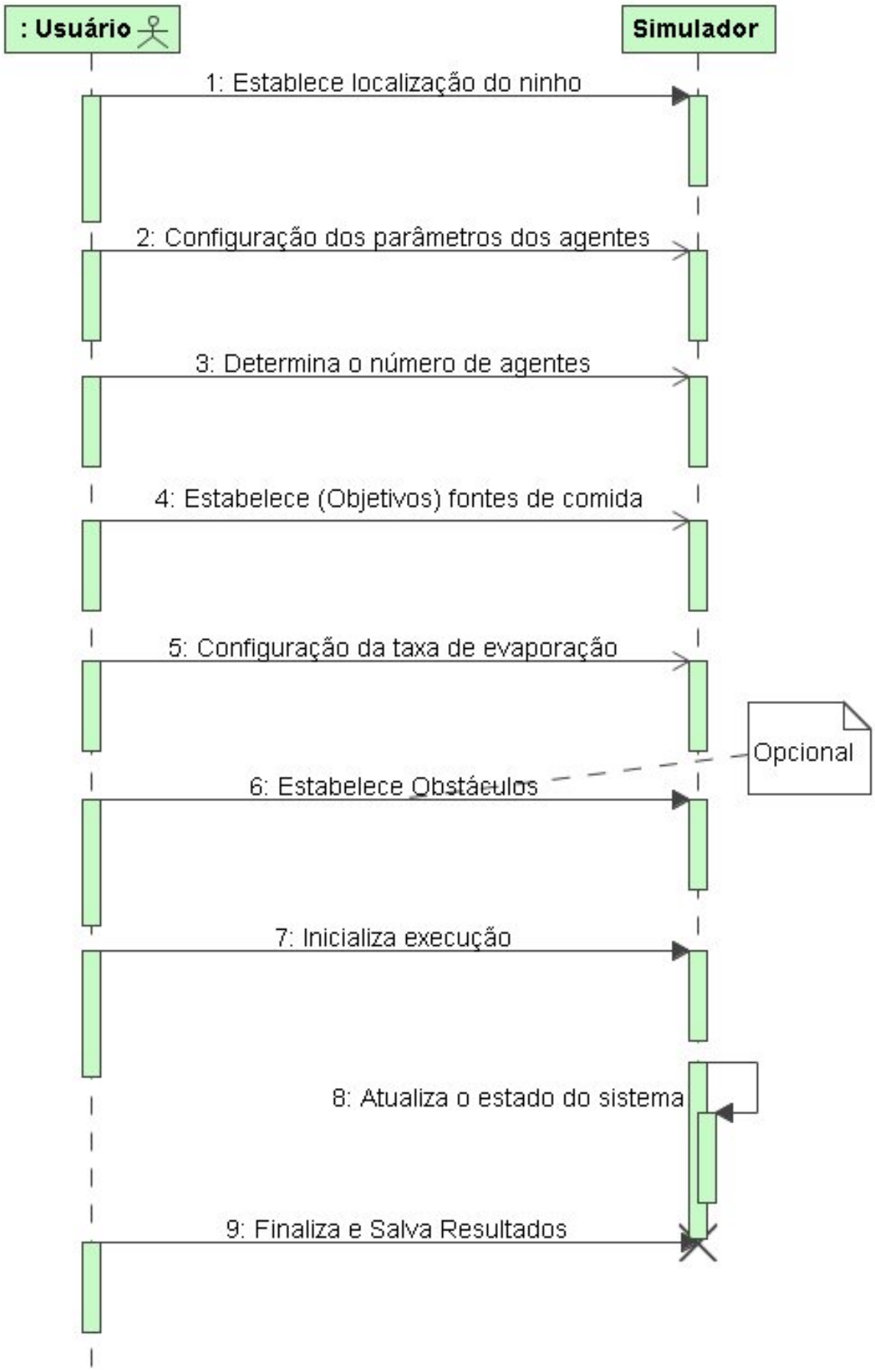

Figura A.1: Diagrama de sequencia de caso de uso.

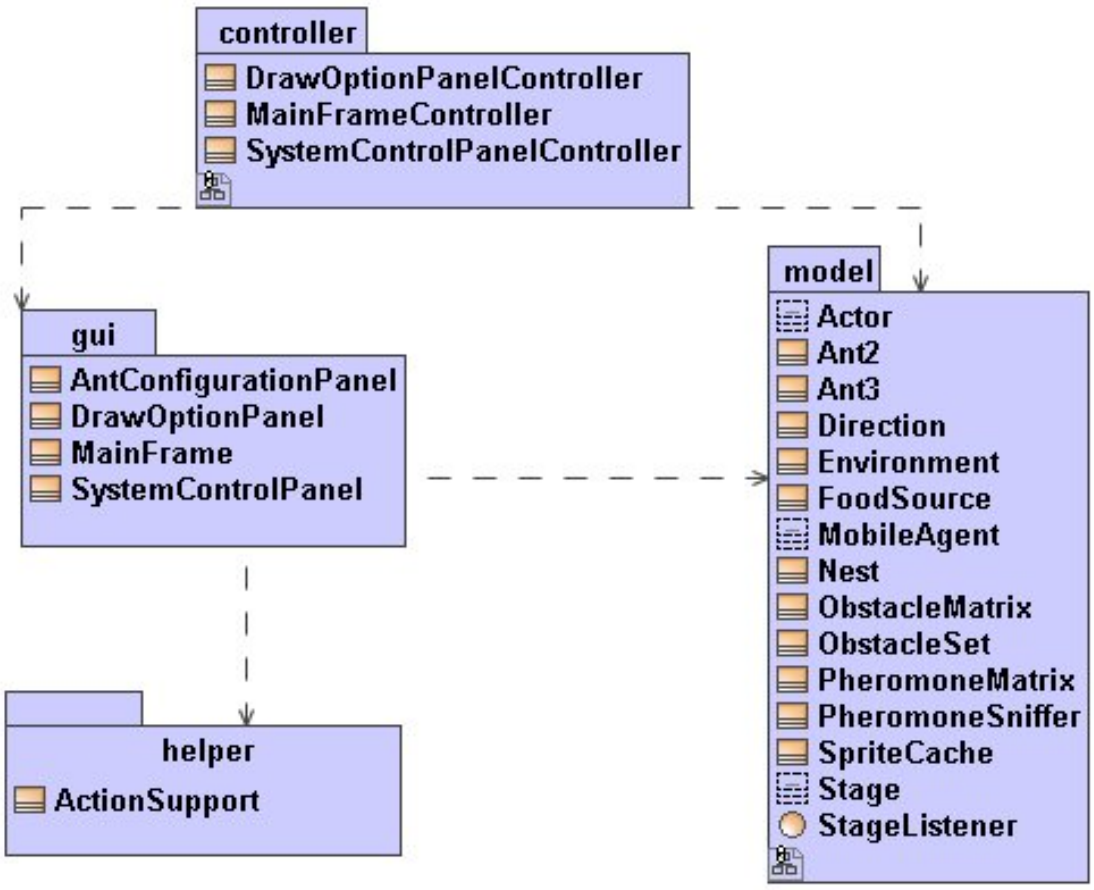

Figura A.2: Visao geral - Pacotes e suas Classes

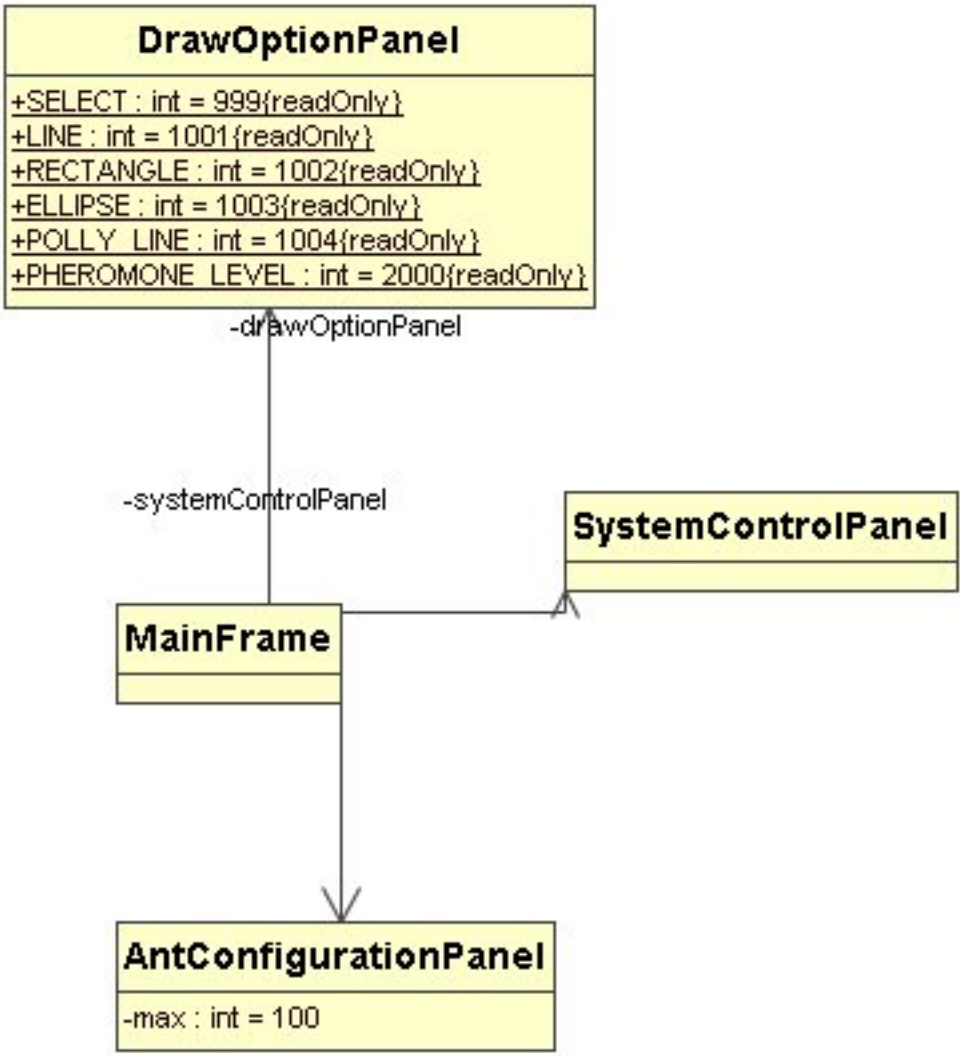

Figura A.3: Pacote gui.

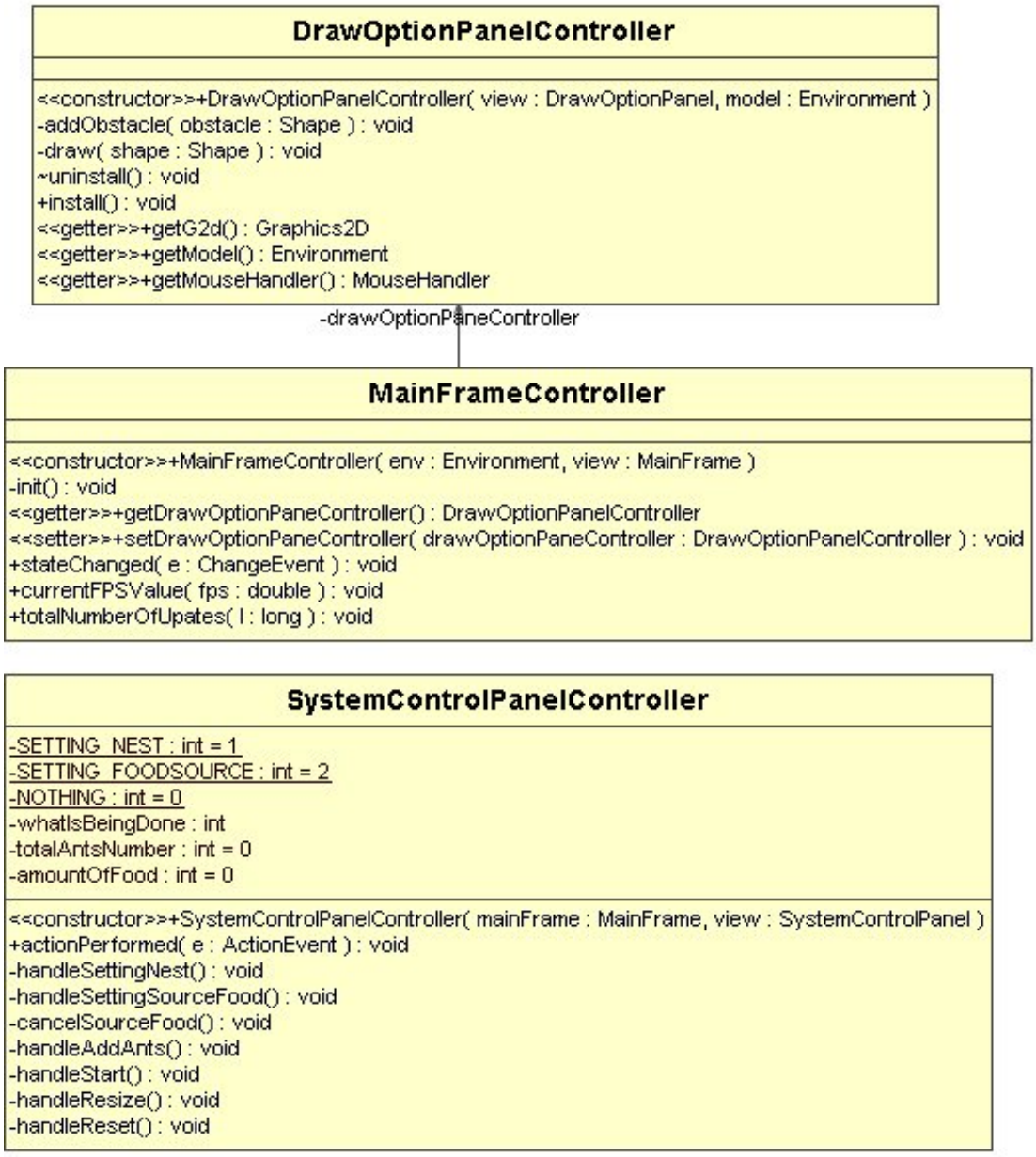

Figura A.4: Pacote controller

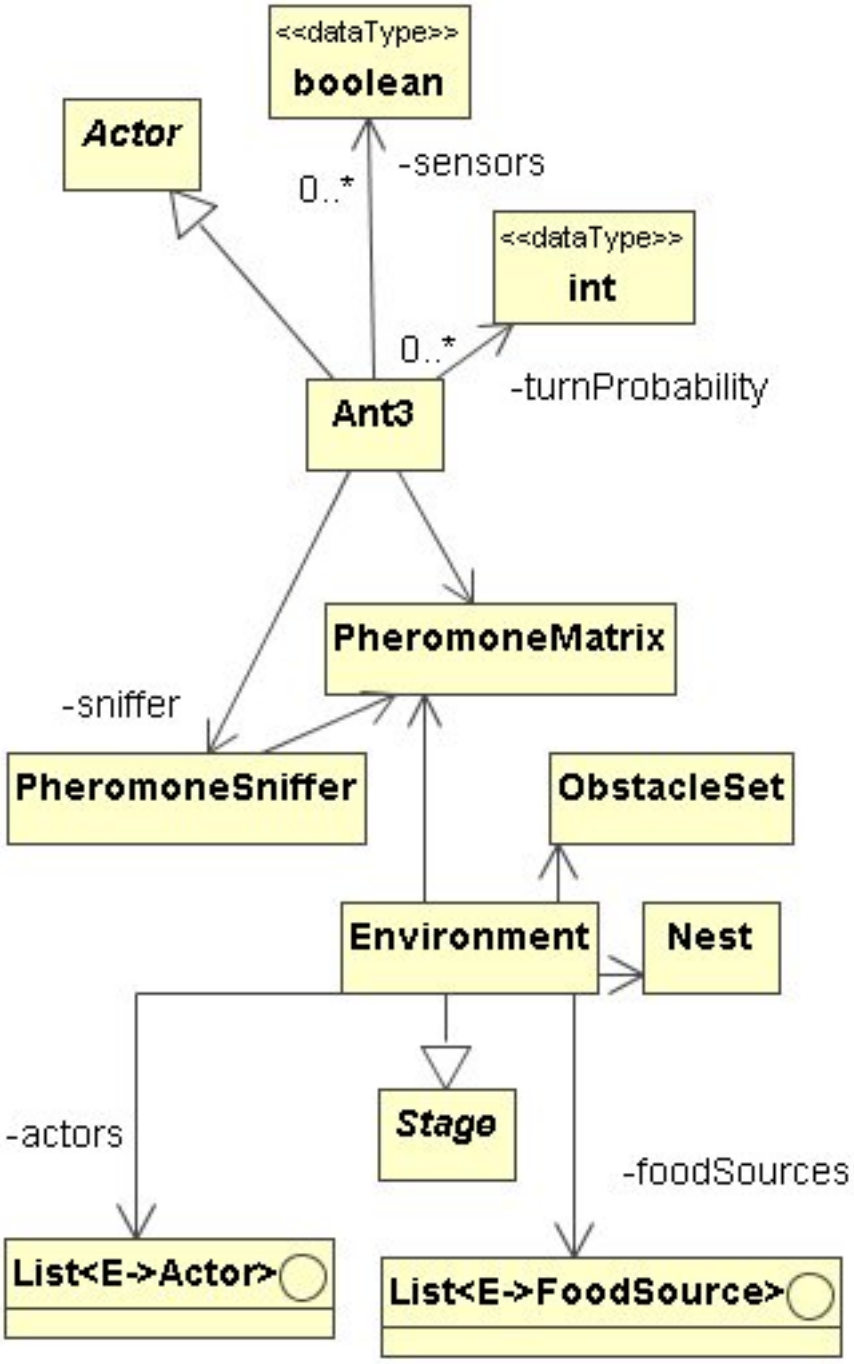

Figura A.5: Pacote model

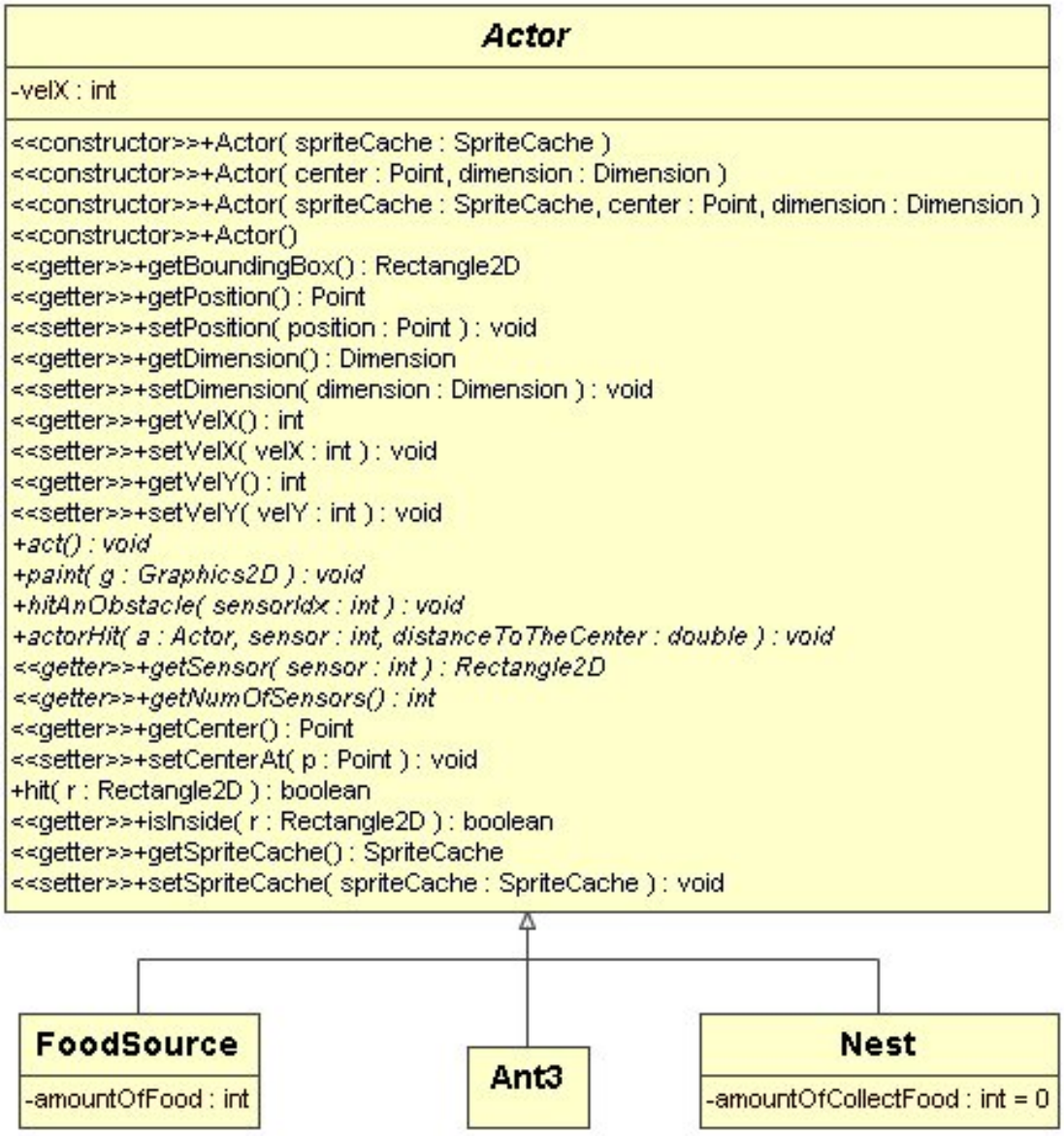

Figura A.6: Atores

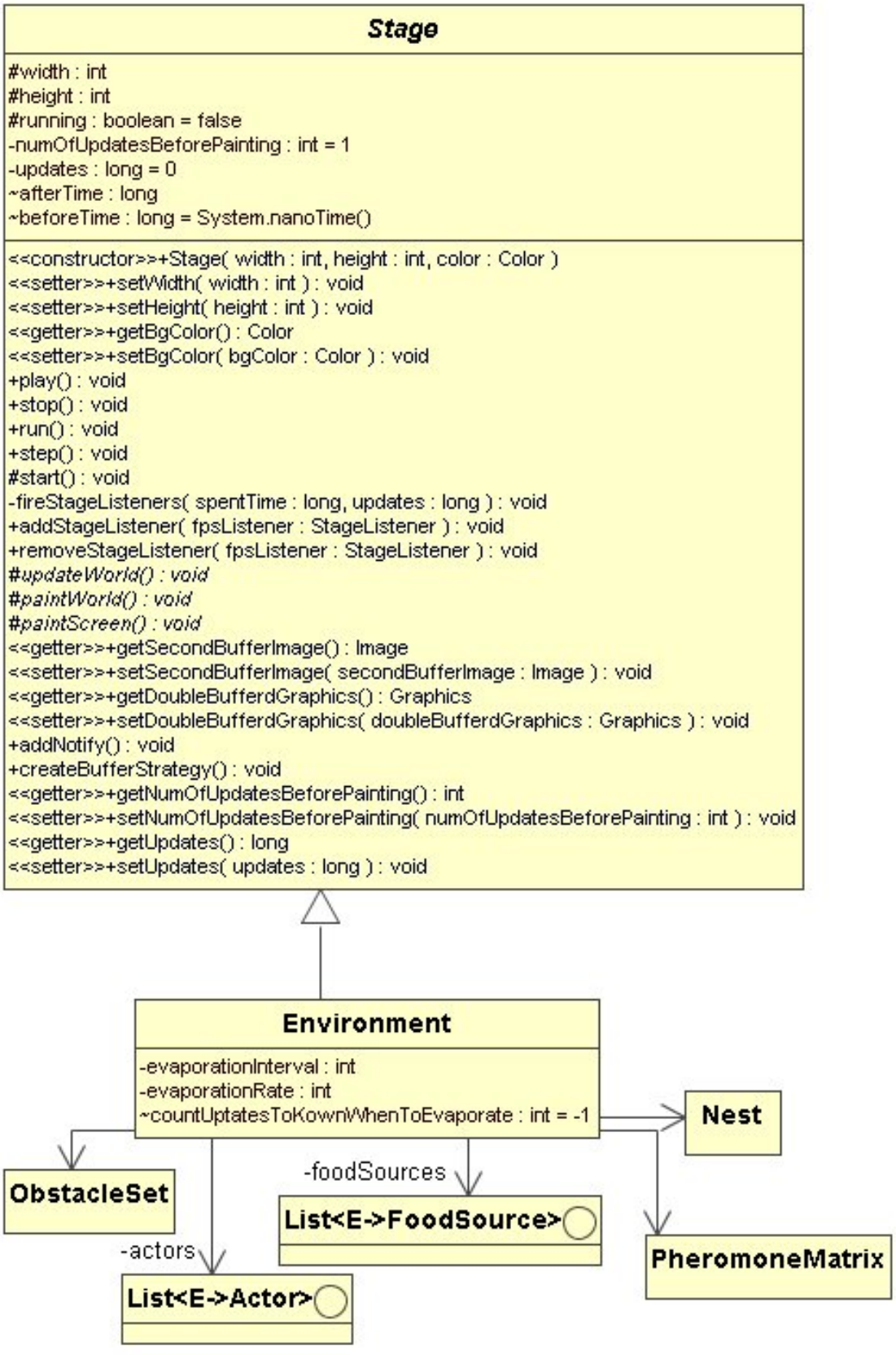

Figura A.7: Stage e Environmente

Apêndice B

Documentação Javadoc
Principais classes

antsystem.model

Class Stage

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └javax.swing.JComponent
              └javax.swing.JPanel
                  └antsystem.model.Stage
```

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.lang.Runnable, javax.accessibility.Accessible

Direct Known Subclasses:

Environment

```
public abstract class Stage
extends javax.swing.JPanel
implements java.lang.Runnable
```

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JPanel
javax.swing.JPanel.AccessibleJPanel

Nested classes/interfaces inherited from class javax.swing.JComponent
javax.swing.JComponent.AccessibleJComponent

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

protected java.lang.Thread	**animator**
static java.awt.Graphics	**doubleBufferdGraphics**
protected int	**height**
protected boolean	**running**
protected java.awt.Image	**secondBufferImage**
protected SpriteCache	**spriteCache**
protected int	**width**

Fields inherited from class javax.swing.JComponent
accessibleContext, listenerList, TOOL_TIP_TEXT_KEY, ui, UNDEFINED_CONDITION, WHEN_ANCESTOR_OF_FOCUSED_COMPONENT, WHEN_FOCUSED, WHEN_IN_FOCUSED_WINDOW

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

Stage(int width, int height, java.awt.Color color) Creates a new instance of Stage

Method Summary

void	**addNotify**()
void	**addStageListener**(StageListener fpsListener)
void	**createBufferStrategy**()
java.awt.Color	**getBgColor**()
java.awt.Graphics	**getDoubleBufferdGraphics**()
int	**getNumOfUpdatesBeforePainting**()
java.awt.Image	**getSecondBufferImage**()
long	**getUpdates**()
protected abstract void	**paintScreen**()
protected abstract void	**paintWorld**()
void	**play**()
void	**removeStageListener**(StageListener fpsListener)
void	**run**()
void	**setBgColor**(java.awt.Color bgColor)
void	**setDoubleBufferdGraphics**(java.awt.Graphics doubleBufferdGraphics)
void	**setHeight**(int height)
void	**setNumOfUpdatesBeforePainting**(int numOfUpdatesBeforePainting)
void	**setSecondBufferImage**(java.awt.Image secondBufferImage)
void	**setUpdates**(long updates)
void	**setWidth**(int width)

protected void	**start**()
void	**step**()
void	**stop**()
protected abstract void	**updateWorld**()

Methods inherited from class javax.swing.JPanel

getAccessibleContext, getUI, getUIClassID, paramString, setUI, updateUI

Methods inherited from class javax.swing.JComponent

addAncestorListener, addVetoableChangeListener, computeVisibleRect, contains, createToolTip, disable, enable, firePropertyChange, firePropertyChange, firePropertyChange, fireVetoableChange, getActionForKeyStroke, getActionMap, getAlignmentX, getAlignmentY, getAncestorListeners, getAutoscrolls, getBorder, getBounds, getClientProperty, getComponentGraphics, getComponentPopupMenu, getConditionForKeyStroke, getDebugGraphicsOptions, getDefaultLocale, getFontMetrics, getGraphics, getHeight, getInheritsPopupMenu, getInputMap, getInputMap, getInputVerifier, getInsets, getInsets, getListeners, getLocation, getMaximumSize, getMinimumSize, getNextFocusableComponent, getPopupLocation, getPreferredSize, getRegisteredKeyStrokes, getRootPane, getSize, getToolTipLocation, getToolTipText, getToolTipText, getTopLevelAncestor, getTransferHandler, getVerifyInputWhenFocusTarget, getVetoableChangeListeners, getVisibleRect, getWidth, getX, getY, grabFocus, isDoubleBuffered, isLightweightComponent, isManagingFocus, isOpaque, isOptimizedDrawingEnabled, isPaintingTile, isRequestFocusEnabled, isValidateRoot, paint, paintBorder, paintChildren, paintComponent, paintImmediately, paintImmediately, print, printAll, printBorder, printChildren, printComponent, processComponentKeyEvent, processKeyBinding, processKeyEvent, processMouseEvent, processMouseMotionEvent, putClientProperty, registerKeyboardAction, registerKeyboardAction, removeAncestorListener, removeNotify, removeVetoableChangeListener, repaint, repaint, requestDefaultFocus, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resetKeyboardActions, reshape, revalidate, scrollRectToVisible, setActionMap, setAlignmentX, setAlignmentY, setAutoscrolls, setBackground, setBorder, setComponentPopupMenu, setDebugGraphicsOptions, setDefaultLocale, setDoubleBuffered, setEnabled, setFocusTraversalKeys, setFont, setForeground, setInheritsPopupMenu, setInputMap, setInputVerifier, setMaximumSize, setMinimumSize, setNextFocusableComponent, setOpaque, setPreferredSize, setRequestFocusEnabled, setToolTipText, setTransferHandler, setUI, setVerifyInputWhenFocusTarget, setVisible, unregisterKeyboardAction, update

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, addImpl, addPropertyChangeListener, addPropertyChangeListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalKeys, getFocusTraversalPolicy, getLayout, getMousePosition, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, printComponents, processContainerEvent, processEvent, remove, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusCycleRoot, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setLayout, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, createImage, createImage, createVolatileImage, createVolatileImage, disableEvents, dispatchEvent, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getForeground, getGraphicsConfiguration, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputContext, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocale, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getToolkit, getTreeLock, gotFocus, handleEvent, hasFocus, hide, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isPreferredSizeSet, isShowing, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, postEvent, prepareImage, prepareImage, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processMouseWheelEvent, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, resize, resize, setBounds, setBounds, setComponentOrientation, setCursor, setDropTarget, setFocusable, setFocusTraversalKeysEnabled, setIgnoreRepaint, setLocale, setLocation, setLocation, setName, setSize, setSize, show, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

width

```
protected int width
```

---

height

```
protected int height
```

---

spriteCache

```
protected SpriteCache spriteCache
```

---

running

```
protected volatile boolean running
```

---

secondBufferImage

```
protected java.awt.Image secondBufferImage
```

---

doubleBufferdGraphics

```
public static java.awt.Graphics doubleBufferdGraphics
```

---

animator

```
protected java.lang.Thread animator
```

Constructor Detail

Stage

```
public Stage(int width,
             int height,
             java.awt.Color color)
```

Creates a new instance of Stage

Method Detail

setWidth

```
public void setWidth(int width)
```

---

setHeight

```
public void setHeight(int height)
```

---

getBgColor

```
public java.awt.Color getBgColor()
```

---

setBgColor

```
public void setBgColor(java.awt.Color bgColor)
```

---

play

```
public void play()
```

---

stop

```
public void stop()
```

---

run

```
public void run()
```

Specified by:
run in interface `java.lang.Runnable`

---

step

```
public void step()
```

---

start

```
protected void start()
```

---

addStageListener

```
public void addStageListener(StageListener fpsListener)
```

---

removeStageListener

```
public void removeStageListener(StageListener fpsListener)
```

---

updateWorld

```
protected abstract void updateWorld()
```

---

paintWorld

```
protected abstract void paintWorld()
```

---

paintScreen

```
protected abstract void paintScreen()
```

---

getSecondBufferImage

```
public java.awt.Image getSecondBufferImage()
```

---

setSecondBufferImage

```
public void setSecondBufferImage(java.awt.Image secondBufferImage)
```

---

getDoubleBufferdGraphics

```
public java.awt.Graphics getDoubleBufferdGraphics()
```

---

setDoubleBufferdGraphics

```
public void setDoubleBufferdGraphics(java.awt.Graphics doubleBufferdGraphics)
```

---

addNotify

```
public void addNotify()
```

Overrides:

addNotify in class javax.swing.JComponent

---

createBufferStrategy

```
public void createBufferStrategy()
```

---

getNumOfUpdatesBeforePainting

```
public int getNumOfUpdatesBeforePainting()
```

---

setNumOfUpdatesBeforePainting

```
public void setNumOfUpdatesBeforePainting(int numOfUpdatesBeforePainting)
```

---

getUpdates

```
public long getUpdates()
```

---

setUpdates

```
public void setUpdates(long updates)
```

---

antsystem.model

Class Ant3

```
java.lang.Object
  └antsystem.model.Actor
      └antsystem.model.Ant3
```

```
public class Ant3
extends Actor
```

Constructor Summary

Ant3() Creates a new instance of Ant2
Ant3(PheromoneMatrix pheromoneMatrix, int velocity)

Method Summary

void	**act**() The main method of the Actor.
void	**actorHit**(Actor a, int sensor, double distanceToTheCenter) Its called by the Environmnet whenever it is collided to another Ac Environment .
double	**getActorDistance**()
java.awt.Color	**getCarryingFoodColor**()
int	**getCarryingFoodPheromoneDepositAmount**()
int	**getCarryingFoodPheromoneThreshold**()
int	**getCarryingFoodPheromoneWeight**()
int	**getClockwise_135_Probability**()
int	**getClockwise_180_Probability**()
int	**getClockwise_45_Probability**()

int	**getClockwise_90_Probability**()
int	**getCounterclockwise_135_Probability**()
int	**getCounterclockwise_45_Probability**()
int	**getCounterclockwise_90_Probability**()
int	**getDirection**()
int	**getIn135LeftPheromoneConcentration**()
int	**getIn135RightPheromoneConcentration**()
int	**getIn45LeftPheromoneConcentration**()
int	**getIn45RightPheromoneConcentration**()
int	**getIn90LeftPheromoneConcentration**()
int	**getIn90RightPheromoneConcentration**()
int	**getInBackPheromoneConcentration**()
int	**getInFrontPheromoneConcentration**()
java.awt.Color	**getNormalColor**()
int	**getNoTurnProbability**()
int	**getNumOfSensors**() Returns the number of Sensors in the Actor
int	**getPheromoneDepositAmount**()
int	**getPheromoneLevelAround**()
PheromoneMatrix	**getPheromoneMatrix**()
int	**getPheromoneThreshold**()
int	**getPheromoneWeight**()
java.awt.geom.Rectangle2D	**getSensor**(int i) Gets the sensor Rectangle

int	**getVelocity**()
void	**hitAnObstacle**(int sensorIdx) To be called when the ant hits an obstacle
void	**initRadomDirection**() Chooses a initial direction randomly.
boolean	**isCarryingFood**() Verifies whether this Ant is carrying food or not.
void	**paint**(java.awt.Graphics2D g2d) Its called by Environment every time it needs to update the visual r World This method must be overridden to provides the correctly visual re
protected java.awt.geom.AffineTransform	**randomlyTurn**() Makes a turn randomly based on the respective probabilities set in array.
void	**setActorDistance**(double actorDistance)
void	**setCarryingFood**(boolean carryingFood) Sets the state of carrying food.
void	**setCarryingFoodColor**(java.awt.Color carryingFoodColor)
void	**setCarryingFoodPheromoneDepositAmount**(int carryingFoodPheromo Sets the amount of pheromone that will be deposited when the ant i each turn.
void	**setCarryingFoodPheromoneThreshold**(int carryingFoodPheromoneTh
void	**setCarryingFoodPheromoneWeight**(int carryingFoodPheromoneWeigh
void	**setClockwise_135_Probability**(int clockwise_135_Probability)
void	**setClockwise_180_Probability**(int clockwise_180_Probability)
void	**setClockwise_45_Probability**(int clockwise_45_Probability)
void	**setClockwise_90_Probability**(int clockwise_90_Probability)
void	**setCounterclockwise_135_Probability**(int counterclockwise_135_
void	**setCounterclockwise_45_Probability**(int counterclockwise_45_Pr
void	**setCounterclockwise_90_Probability**(int counterclockwise_90_Pr
void	**setDirection**(Direction direction)
void	**setDirection**(int direction)

void	**setIn135LeftPheromoneConcentration**(int in135LeftPheromoneConc
void	**setIn135RightPheromoneConcentration**(int in135RightPheromoneCo
void	**setIn45LeftPheromoneConcentration**(int in45LeftPheromoneConcer
void	**setIn45RightPheromoneConcentration**(int in45RightPheromoneConc
void	**setIn90LeftPheromoneConcentration**(int in90LeftPheromoneConcer
void	**setIn90RightPheromoneConcentration**(int in90RightPheromoneConc
void	**setInBackPheromoneConcentration**(int inBackPheromoneConcentrat
void	**setInFrontPheromoneConcentration**(int inFrontPheromoneConcentr
void	**setNormalColor**(java.awt.Color normalColor)
void	**setNoTurnProbability**(int noTurnProbability)
void	**setPheromoneDepositAmount**(int pheromoneDepositAmount)
void	**setPheromoneLevelAround**(int pheromoneLevelAround)
void	**setPheromoneMatrix**(PheromoneMatrix pheromoneMatrix)
void	**setPheromoneThreshold**(int pheromoneThreshold)
void	**setPheromoneWeight**(int pheromoneWeight)
void	**setPosition**(java.awt.Point position) Sets a new position for the Actor
void	**setVelocity**(int velocity) Set constant for velocity

Methods inherited from class antsystem.model.Actor
getBoundingBox, getCenter, getDimension, getPosition, getSpriteCache, getVelX, getVelY, hit, isInside, setCenterAt, setDimension, setSpriteCache, setVelX, setVelY

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Ant3

```
public Ant3()
```

Creates a new instance of Ant2

Ant3

```
public Ant3(PheromoneMatrix pheromoneMatrix,
            int velocity)
```

Method Detail

isCarryingFood

```
public boolean isCarryingFood()
```

Verifies whether this Ant is carrying food or not.

Returns:

`True` if it is carrying food.

---

setCarryingFood

```
public void setCarryingFood(boolean carryingFood)
```

Sets the state of carrying food.

Parameters:

`carryingFood` -

---

initRadomDirection

```
public void initRadomDirection()
```

Chooses a initial direction randomly.

---

randomlyTurn

```
protected java.awt.geom.AffineTransform randomlyTurn()
```

Makes a turn randomly based on the respective probabilities set in `turnProbability` array.

Returns:

The correct `AffineTransform`
to be used in the graphical environment to perform the correspondent turn.

---

act

```
public void act()
```

Description copied from class: Actor
The main method of the Actor. It will be called by the Environment in every update of the World. This method must be overridden to provides the correctly action for the Actor, for example, if the Actor is moving, it should update its velocity values here.

Specified by:
act in class Actor

---

paint

```
public void paint(java.awt.Graphics2D g2d)
```

Description copied from class: Actor
Its called by Environment every time it needs to update the visual representation of the World This method must be overridden to provides the correctly visual representation.

Specified by:
paint in class Actor
Parameters:
g2d - The Graphics context in which the actor will be painted.

---

setDirection

```
public void setDirection(Direction direction)
```

---

setDirection

```
public void setDirection(int direction)
```

---

getDirection

```
public int getDirection()
```

---

getVelocity

```
public int getVelocity()
```

---

setPosition

```
public void setPosition(java.awt.Point position)
```

Description copied from class: Actor
Sets a new position for the Actor

Overrides:
setPosition in class Actor
Parameters:
position - The most top left point of the new position

---

setVelocity

```
public void setVelocity(int velocity)
```

Set constant for velocity

Parameters:
`velocity` - in pixels

---

getPheromoneMatrix

```
public PheromoneMatrix getPheromoneMatrix()
```

---

setPheromoneMatrix

```
public void setPheromoneMatrix(PheromoneMatrix pheromoneMatrix)
```

---

actorHit

```
public void actorHit(Actor a,
                     int sensor,
                     double distanceToTheCenter)
```

Description copied from class: `Actor`
Its called by the `Environmnet` whenever it is collided to another `Actor` in the `Environment` .

Specified by:
`actorHit` in class `Actor`
Parameters:
`a` - The `Actor` tho whom it is collided.
`sensor` - The index of the sensor collided.
`distanceToTheCenter` - The distance to the center point of the collided `Actor`.

---

getSensor

```
public java.awt.geom.Rectangle2D getSensor(int i)
```

Gets the sensor Rectangle

Specified by:
`getSensor` in class `Actor`
Parameters:
`i` - which of the sensors
Returns:
Returns the correspondent Rectangle2D of the request sensor

---

hitAnObstacle

```
public void hitAnObstacle(int sensorIdx)
```

To be called when the ant hits an obstacle

Specified by:
`hitAnObstacle` in class `Actor`
Parameters:
`sensorIdx` - The index of the sensor which has been hit

---

getNumOfSensors

```
public int getNumOfSensors()
```

Description copied from class: `Actor`
Returns the number of Sensors in the Actor

Specified by:
`getNumOfSensors` in class `Actor`
Returns:
the number of Sensors

---

getNoTurnProbability

```
public int getNoTurnProbability()
```

---

setNoTurnProbability

```
public void setNoTurnProbability(int noTurnProbability)
```

---

getClockwise_45_Probability

```
public int getClockwise_45_Probability()
```

---

setClockwise_45_Probability

```
public void setClockwise_45_Probability(int clockwise_45_Probability)
```

---

getClockwise_90_Probability

```
public int getClockwise_90_Probability()
```

---

setClockwise_90_Probability

```
public void setClockwise_90_Probability(int clockwise_90_Probability)
```

---

getClockwise_135_Probability

```
public int getClockwise_135_Probability()
```

---

setClockwise_135_Probability

```
public void setClockwise_135_Probability(int clockwise_135_Probability)
```

---

getClockwise_180_Probability

```
public int getClockwise_180_Probability()
```

---

setClockwise_180_Probability

```
public void setClockwise_180_Probability(int clockwise_180_Probability)
```

---

getCounterclockwise_45_Probability

```
public int getCounterclockwise_45_Probability()
```

---

setCounterclockwise_45_Probability

```
public void setCounterclockwise_45_Probability(int counterclockwise_45_Probability)
```

---

getCounterclockwise_90_Probability

```
public int getCounterclockwise_90_Probability()
```

---

setCounterclockwise_90_Probability

```
public void setCounterclockwise_90_Probability(int counterclockwise_90_Probability)
```

---

getCounterclockwise_135_Probability

```
public int getCounterclockwise_135_Probability()
```

---

setCounterclockwise_135_Probability

```
public void setCounterclockwise_135_Probability(int counterclockwise_135_Probability)
```

---

getPheromoneWeight

```
public int getPheromoneWeight()
```

---

setPheromoneWeight

```
public void setPheromoneWeight(int pheromoneWeight)
```

---

getInFrontPheromoneConcentration

```
public int getInFrontPheromoneConcentration()
```

---

setInFrontPheromoneConcentration

```
public void setInFrontPheromoneConcentration(int inFrontPheromoneConcentration)
```

---

getIn45RightPheromoneConcentration

```
public int getIn45RightPheromoneConcentration()
```

---

setIn45RightPheromoneConcentration

```
public void setIn45RightPheromoneConcentration(int in45RightPheromoneConcentration)
```

---

getIn45LeftPheromoneConcentration

```
public int getIn45LeftPheromoneConcentration()
```

---

setIn45LeftPheromoneConcentration

```
public void setIn45LeftPheromoneConcentration(int in45LeftPheromoneConcentration)
```

---

getIn90RightPheromoneConcentration

```
public int getIn90RightPheromoneConcentration()
```

---

setIn90RightPheromoneConcentration

```
public void setIn90RightPheromoneConcentration(int in90RightPheromoneConcentration)
```

---

getIn90LeftPheromoneConcentration

```
public int getIn90LeftPheromoneConcentration()
```

---

setIn90LeftPheromoneConcentration

```
public void setIn90LeftPheromoneConcentration(int in90LeftPheromoneConcentration)
```

---

getInBackPheromoneConcentration

```
public int getInBackPheromoneConcentration()
```

---

setInBackPheromoneConcentration

```
public void setInBackPheromoneConcentration(int inBackPheromoneConcentration)
```

---

getIn135RightPheromoneConcentration

```
public int getIn135RightPheromoneConcentration()
```

---

setIn135RightPheromoneConcentration

```
public void setIn135RightPheromoneConcentration(int in135RightPheromoneConcentration)
```

---

getIn135LeftPheromoneConcentration

```
public int getIn135LeftPheromoneConcentration()
```

---

setIn135LeftPheromoneConcentration

```
public void setIn135LeftPheromoneConcentration(int in135LeftPheromoneConcentration)
```

---

getPheromoneThreshold

```
public int getPheromoneThreshold()
```

---

setPheromoneThreshold

```
public void setPheromoneThreshold(int pheromoneThreshold)
```

---

getPheromoneLevelAround

```
public int getPheromoneLevelAround()
```

---

setPheromoneLevelAround

```
public void setPheromoneLevelAround(int pheromoneLevelAround)
```

---

getCarryingFoodPheromoneWeight

```
public int getCarryingFoodPheromoneWeight()
```

---

setCarryingFoodPheromoneWeight

```
public void setCarryingFoodPheromoneWeight(int carryingFoodPheromoneWeight)
```

Parameters:

carryingFoodPheromoneWeight -

---

getCarryingFoodPheromoneThreshold

```
public int getCarryingFoodPheromoneThreshold()
```

---

setCarryingFoodPheromoneThreshold

```
public void setCarryingFoodPheromoneThreshold(int carryingFoodPheromoneThreshold)
```

---

getPheromoneDepositAmount

```
public int getPheromoneDepositAmount()
```

---

setPheromoneDepositAmount

```
public void setPheromoneDepositAmount(int pheromoneDepositAmount)
```

---

getCarryingFoodPheromoneDepositAmount

```
public int getCarryingFoodPheromoneDepositAmount()
```

---

setCarryingFoodPheromoneDepositAmount

```
public void setCarryingFoodPheromoneDepositAmount(int carryingFoodPheromoneDepositAmount)
```

Sets the amount of pheromone that will be deposited when the ant is carrying food in each turn.

Parameters:

`carryingFoodPheromoneDepositAmount` - the amount to be deposited.

---

getNormalColor

```
public java.awt.Color getNormalColor()
```

---

setNormalColor

```
public void setNormalColor(java.awt.Color normalColor)
```

---

getCarryingFoodColor

```
public java.awt.Color getCarryingFoodColor()
```

---

setCarryingFoodColor

```
public void setCarryingFoodColor(java.awt.Color carryingFoodColor)
```

---

getActorDistance

```
public double getActorDistance()
```

---

setActorDistance

```
public void setActorDistance(double actorDistance)
```

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

antsystem.model

Class Environment

```
java.lang.Object
  └java.awt.Component
      └java.awt.Container
          └javax.swing.JComponent
              └javax.swing.JPanel
                  └antsystem.model.Stage
                      └antsystem.model.Environment
```

All Implemented Interfaces:

java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable, java.lang.Runnable, javax.accessibility.Accessible

---

```
public class Environment
extends Stage
```

See Also:

Serialized Form

Nested Class Summary

Nested classes/interfaces inherited from class javax.swing.JPanel
javax.swing.JPanel.AccessibleJPanel

Nested classes/interfaces inherited from class javax.swing.JComponent
javax.swing.JComponent.AccessibleJComponent

Nested classes/interfaces inherited from class java.awt.Container
java.awt.Container.AccessibleAWTContainer

Nested classes/interfaces inherited from class java.awt.Component
java.awt.Component.AccessibleAWTComponent, java.awt.Component.BltBufferStrategy, java.awt.Component.FlipBufferStrategy

Field Summary

Fields inherited from class antsystem.model.Stage
animator, doubleBufferdGraphics, height, running, secondBufferImage, spriteCache, width

Fields inherited from class javax.swing.JComponent
accessibleContext, listenerList, TOOL_TIP_TEXT_KEY, ui, UNDEFINED_CONDITION, WHEN_ANCESTOR_OF_FOCUSED_COMPONENT, WHEN_FOCUSED, WHEN_IN_FOCUSED_WINDOW

Fields inherited from class java.awt.Component
BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface java.awt.image.ImageObserver
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Constructor Summary

Environment(int width, int height, java.awt.Color color) Creates a new instance of Environment

Method Summary

void	**activeRepaint**() This method should be used to instantly show an model's Environment change.
void	**addActor**(Actor a) Adds a new Actor in the Environment
void	**addFoodSource**(FoodSource fs) Adds a new Source of food in the Environment
void	**clearPheromoneMatrix**() Clear all the pheromone existent.
void	**constructBorders**(int width, int height)
void	**creatAnts**(int numOfAnts) **Deprecated.**
protected void	**foundFoodTest**(Actor a) Verify if the Actor has hit an FoodSource.
protected void	**foundNestTest**(Actor a) Verify if the Actor has hit an Nest.
java.awt.Shape	**getDrawingShape**() Gets the currently set drawing shape.
int	**getEvaporationInterval**() The evaporation interval currently used.
int	**getEvaporationRate**() The evaporation rate currently used
Nest	**getNest**() Gets the Nest in the Environment
java.awt.Point	**getNestLocation**() Gets the current location of the nest
ObstacleSet	**getObstacles**() Gets the set of obstacles (shapes) currently existent in the Environment.

PheromoneMatrix	**getPherormoneMatrix**() Gets the currently set PheromoneMatrix
protected void	**hitObstacleTest**(Actor a) Verify if the Actor a has hit an obstacle.
void	**paint**(java.awt.Graphics g) Paints this Environment
protected void	**paintScreen**() Paints the Buffered Graphics to the Screen
protected void	**paintWorld**() Creates a World's representation in the Buffered Graphics
void	**removeAllActors**() Removes all actors in the Environment
void	**removeAllObstacles**() Removes all obstacles, except the borders, in the Environment
void	**setDrawingShape**(java.awt.Shape drawingShape) Sets the shape which is being drawing in the moment.
void	**setEvaporationInterval**(int evaporationInterval) Sets the interval between two evaporation events.
void	**setEvaporationRate**(int evaporationRate) Sets the evaporation rate
void	**setNest**(Nest nest) Sets the nest
void	**setNest**(java.awt.Point location) Creates a new Nest and sets it as the nest
void	**setNestLocation**(java.awt.Point nestLocation) Places the nest in the new location.
void	**setPherormoneMatrix**(PheromoneMatrix pherormoneMatrix) Sets the pheromone matrix to be used in the Environment.
protected void	**updateWorld**()

Methods inherited from class antsystem.model.Stage

addNotify, addStageListener, createBufferStrategy, getBgColor, getDoubleBufferdGraphics, getNumOfUpdatesBeforePainting, getSecondBufferImage, getUpdates, play, removeStageListener, run, setBgColor, setDoubleBufferdGraphics, setHeight, setNumOfUpdatesBeforePainting, setSecondBufferImage, setUpdates, setWidth, start, step, stop

Methods inherited from class javax.swing.JPanel

getAccessibleContext, getUI, getUIClassID, paramString, setUI, updateUI

Methods inherited from class javax.swing.JComponent

addAncestorListener, addVetoableChangeListener, computeVisibleRect, contains, createToolTip, disable, enable, firePropertyChange, firePropertyChange, firePropertyChange, fireVetoableChange, getActionForKeyStroke, getActionMap, getAlignmentX, getAlignmentY, getAncestorListeners, getAutoscrolls, getBorder, getBounds, getClientProperty, getComponentGraphics, getComponentPopupMenu, getConditionForKeyStroke, getDebugGraphicsOptions, getDefaultLocale, getFontMetrics, getGraphics, getHeight, getInheritsPopupMenu, getInputMap, getInputMap,

getInputVerifier, getInsets, getInsets, getListeners, getLocation, getMaximumSize, getMinimumSize, getNextFocusableComponent, getPopupLocation, getPreferredSize, getRegisteredKeyStrokes, getRootPane, getSize, getToolTipLocation, getToolTipText, getToolTipText, getTopLevelAncestor, getTransferHandler, getVerifyInputWhenFocusTarget, getVetoableChangeListeners, getVisibleRect, getWidth, getX, getY, grabFocus, isDoubleBuffered, isLightweightComponent, isManagingFocus, isOpaque, isOptimizedDrawingEnabled, isPaintingTile, isRequestFocusEnabled, isValidateRoot, paintBorder, paintChildren, paintComponent, paintImmediately, paintImmediately, print, printAll, printBorder, printChildren, printComponent, processComponentKeyEvent, processKeyBinding, processKeyEvent, processMouseEvent, processMouseMotionEvent, putClientProperty, registerKeyboardAction, registerKeyboardAction, removeAncestorListener, removeNotify, removeVetoableChangeListener, repaint, repaint, requestDefaultFocus, requestFocus, requestFocus, requestFocusInWindow, requestFocusInWindow, resetKeyboardActions, reshape, revalidate, scrollRectToVisible, setActionMap, setAlignmentX, setAlignmentY, setAutoscrolls, setBackground, setBorder, setComponentPopupMenu, setDebugGraphicsOptions, setDefaultLocale, setDoubleBuffered, setEnabled, setFocusTraversalKeys, setFont, setForeground, setInheritsPopupMenu, setInputMap, setInputVerifier, setMaximumSize, setMinimumSize, setNextFocusableComponent, setOpaque, setPreferredSize, setRequestFocusEnabled, setToolTipText, setTransferHandler, setUI, setVerifyInputWhenFocusTarget, setVisible, unregisterKeyboardAction, update

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, addImpl, addPropertyChangeListener, addPropertyChangeListener, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getComponentZOrder, getContainerListeners, getFocusTraversalKeys, getFocusTraversalPolicy, getLayout, getMousePosition, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusCycleRoot, isFocusTraversalPolicyProvider, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, printComponents, processContainerEvent, processEvent, remove, remove, removeAll, removeContainerListener, setComponentZOrder, setFocusCycleRoot, setFocusTraversalPolicy, setFocusTraversalPolicyProvider, setLayout, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, createImage, createImage, createVolatileImage, createVolatileImage, disableEvents, dispatchEvent, enable, enableEvents, enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getBackground, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor, getFocusListeners, getFocusTraversalKeysEnabled, getFont, getForeground, getGraphicsConfiguration, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputContext, getInputMethodListeners, getInputMethodRequests, getKeyListeners, getLocale, getLocation, getLocationOnScreen, getMouseListeners, getMouseMotionListeners, getMousePosition, getMouseWheelListeners, getName, getParent, getPeer, getPropertyChangeListeners, getPropertyChangeListeners, getSize, getToolkit, getTreeLock, gotFocus, handleEvent, hasFocus, hide, imageUpdate, inside, isBackgroundSet, isCursorSet, isDisplayable, isEnabled, isFocusable, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight, isMaximumSizeSet, isMinimumSizeSet, isPreferredSizeSet, isShowing, isValid, isVisible, keyDown, keyUp, list, list, list, location, lostFocus, mouseDown, mouseDrag, mouseEnter, mouseExit, mouseMove, mouseUp, move, nextFocus, paintAll, postEvent, prepareImage, prepareImage, processComponentEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent, processInputMethodEvent, processMouseWheelEvent, remove, removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener, removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, resize, resize, setBounds, setBounds, setComponentOrientation, setCursor, setDropTarget, setFocusable, setFocusTraversalKeysEnabled, setIgnoreRepaint, setLocale, setLocation, setLocation, setName, setSize, setSize, show, show, size, toString, transferFocus, transferFocusUpCycle

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Environment

```
public Environment(int width,
                   int height,
                   java.awt.Color color)
```

Creates a new instance of Environment

Method Detail

constructBorders

```
public void constructBorders(int width,
                             int height)
```

---

getNestLocation

```
public java.awt.Point getNestLocation()
```

Gets the current location of the nest

Returns:
the nest position

---

setNestLocation

```
public void setNestLocation(java.awt.Point nestLocation)
```

Places the nest in the new location.

Parameters:
`nestLocation` - new location of the nest

---

hitObstacleTest

```
protected void hitObstacleTest(Actor a)
```

Verify if the Actor a has hit an obstacle.

Parameters:
`a` - The Actor to be tested against all the existing obstacles.

---

foundFoodTest

```
protected void foundFoodTest(Actor a)
```

Verify if the Actor has hit an FoodSource.

Parameters:
`a` - The Actor to be tested.

---

foundNestTest

```
protected void foundNestTest(Actor a)
```

Verify if the Actor has hit an Nest.

Parameters:
`a` - The Actor to be tested.

---

updateWorld

```
protected void updateWorld()
```

Specified by:
`updateWorld` in class `Stage`

---

paintWorld

```
protected void paintWorld()
```

Creates a World's representation in the Buffered Graphics

Specified by:
`paintWorld` in class `Stage`

---

paintScreen

```
protected void paintScreen()
```

Paints the Buffered Graphics to the Screen

Specified by:
`paintScreen` in class `Stage`

---

activeRepaint

```
public void activeRepaint()
```

This method should be used to instantly show an model's Environment change. It calls the `paintWorld` and `paintScreen`

---

addFoodSource

```
public void addFoodSource(FoodSource fs)
```

Adds a new Source of food in the Environment

Parameters:
`fs` - the new `FoodSource`

---

getNest

```
public Nest getNest()
```

Gets the `Nest` in the `Environment`

Returns:
the nest or `null`

---

setNest

```
public void setNest(Nest nest)
```

Sets the nest

Parameters:
`nest` - `Nest` to be set

---

setNest

```
public void setNest(java.awt.Point location)
```

Creates a new `Nest` and sets it as the nest

Parameters:
`location` - of the nest

---

paint

```
public void paint(java.awt.Graphics g)
```

Paints this `Environment`

Overrides:
`paint` in class `javax.swing.JComponent`
Parameters:
`g` - the Graphics context to be painted

---

creatAnts

```
@Deprecated
public void creatAnts(int numOfAnts)
```

Deprecated.

Creates new actors

Parameters:
`numOfAnts` - number of actors to be created

---

addActor

`public void` **`addActor`**`(Actor a)`

Adds a new Actor in the Environment

Parameters:
`a` - the actor to be added

---

getObstacles

`public ObstacleSet` **`getObstacles`**`()`

Gets the set of obstacles (shapes) currently existent in the Environment.

Returns:
The ObstacleSet.

---

getDrawingShape

`public java.awt.Shape` **`getDrawingShape`**`()`

Gets the currently set drawing shape.

Returns:
The shape.

---

setDrawingShape

`public void` **`setDrawingShape`**`(java.awt.Shape drawingShape)`

Sets the shape which is being drawing in the moment. This way it will be rendered in a different color.

Parameters:
`drawingShape` - The shape being drawing.

---

getPherormoneMatrix

`public PheromoneMatrix` **`getPherormoneMatrix`**`()`

Gets the currently set PheromoneMatrix

Returns:
The instance currently used.

---

setPherormoneMatrix

```
public void setPherormoneMatrix(PheromoneMatrix pherormoneMatrix)
```

Sets the pheromone matrix to be used in the Environment.

Parameters:

`pherormoneMatrix` - The instance to be set.

---

getEvaporationInterval

```
public int getEvaporationInterval()
```

The evaporation interval currently used.

Returns:

The number of updates before one evaporation.

---

setEvaporationInterval

```
public void setEvaporationInterval(int evaporationInterval)
```

Sets the interval between two evaporation events.

Parameters:

`evaporationInterval` - The number of updates before one evaporation.

---

getEvaporationRate

```
public int getEvaporationRate()
```

The evaporation rate currently used

Returns:

How much of pheromone is evaporated per turn.

---

setEvaporationRate

```
public void setEvaporationRate(int evaporationRate)
```

Sets the evaporation rate

Parameters:

`evaporationRate`
- How much of pheromone will be subtracted in each location which has pheromone.

---

removeAllActors

```
public void removeAllActors()
```

Removes all actors in the Environment

---

removeAllObstacles

```
public void removeAllObstacles()
```

Removes all obstacles, except the borders, in the Environment

---

clearPheromoneMatrix

```
public void clearPheromoneMatrix()
```

Clear all the pheromone existent.

---

antsystem.model

Class FoodSource

```
java.lang.Object
  └antsystem.model.Actor
      └antsystem.model.FoodSource
```

```
public class FoodSource
extends Actor
```

Constructor Summary

FoodSource() Creates a new instance of FoodSource
FoodSource(int x, int y)
FoodSource(java.awt.Point location)
FoodSource(java.awt.Point center, int amountOfFood)
FoodSource(java.awt.Point center, int amountOfFood, SpriteCache spriteCache)
FoodSource(java.awt.Point center, SpriteCache spriteCache)

Method Summary

void	**act**() The main method of the Actor.
void	**actorHit**(Actor a, int sensor, double distanceToTheCenter) Its called by the Environmnet whenever it is collided to another Actor in the Environment .
int	**getAmountOfFood**() Returns the amount of food currently existent.
int	**getNumOfSensors**() Returns the number of Sensors in the Actor
java.awt.geom.Rectangle2D	**getSensor**(int i) Returns the correspondent bounding box of sensor index.
void	**hitAnObstacle**(int sensorIdx) Its called by the Environment every time the Actor is collided with some obstacle in the Environment.

void	**paint**(java.awt.Graphics2D g) Its called by Environment every time it needs to update the visual representation of the World This method must be overridden to provides the correctly visual representation.
void	**setAmountOfFood**(int amountOfFood) Sets the total of food.
int	**withDraw**(int amount) Withdraw the specified amount of food.

Methods inherited from class antsystem.model.Actor
getBoundingBox, getCenter, getDimension, getPosition, getSpriteCache, getVelX, getVelY, hit, isInside, setCenterAt, setDimension, setPosition, setSpriteCache, setVelX, setVelY

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

FoodSource

```
public FoodSource()
```

Creates a new instance of FoodSource

FoodSource

```
public FoodSource(int x,
                  int y)
```

FoodSource

```
public FoodSource(java.awt.Point location)
```

FoodSource

```
public FoodSource(java.awt.Point center,
                  SpriteCache spriteCache)
```

FoodSource

```
public FoodSource(java.awt.Point center,
                  int amountOfFood)
```

FoodSource

```
public FoodSource(java.awt.Point center,
                  int amountOfFood,
```

SpriteCache spriteCache)

Method Detail

getAmountOfFood

```
public int getAmountOfFood()
```

Returns the amount of food currently existent.

Returns:
The amount of food.

---

setAmountOfFood

```
public void setAmountOfFood(int amountOfFood)
```

Sets the total of food.

Parameters:
`amountOfFood` - Amount of food to be set as the total.

---

withDraw

```
public int withDraw(int amount)
```

Withdraw the specified amount of food.

Parameters:
`amount` - The amount to be subtract from the total.
Returns:

---

paint

```
public void paint(java.awt.Graphics2D g)
```

Description copied from class: Actor
Its called by Environment every time it needs to update the visual representation of the World This method must be overridden to provides the correctly visual representation.

Specified by:
paint in class Actor
Parameters:
`g` - The Graphics context in which the actor will be painted.

---

act

```
public void act()
```

Description copied from class: Actor
The main method of the Actor. It will be called by the Environment in every update of the World. This method must be overridden to provides the correctly action for the Actor, for example, if the Actor is moving, it should update its velocity values here.

Specified by:
act in class Actor

---

hitAnObstacle

```
public void hitAnObstacle(int sensorIdx)
```

Description copied from class: Actor
Its called by the Environment every time the Actor is collided with some obstacle in the Environment.

Specified by:
hitAnObstacle in class Actor
Parameters:
sensorIdx - The number corresponding to the sensor that is collided.

---

getSensor

```
public java.awt.geom.Rectangle2D getSensor(int i)
```

Description copied from class: Actor
Returns the correspondent bounding box of sensor index.

Specified by:
getSensor in class Actor
Parameters:
i - The index of the desired sensor.
Returns:
A Rectangle2D correspondent to the desired sensor.

---

getNumOfSensors

```
public int getNumOfSensors()
```

Description copied from class: Actor
Returns the number of Sensors in the Actor

Specified by:
getNumOfSensors in class Actor
Returns:
the number of Sensors

---

actorHit

```
public void actorHit(Actor a,
                     int sensor,
                     double distanceToTheCenter)
```

Description copied from class: Actor
Its called by the Environmnet whenever it is collided to another Actor in the Environment .

Specified by:
actorHit in class Actor
Parameters:

`a` - The `Actor` tho whom it is collided.
`sensor` - The index of the sensor collided.
`distanceToTheCenter` - The distance to the center point of the collided `Actor`.

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

antsystem

Class Main

```
java.lang.Object
  └antsystem.Main
```

```
public class Main
extends java.lang.Object
```

Constructor Summary

Main() Creates a new instance of Main

Method Summary

static void	**main**(java.lang.String[] args)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Main

```
public Main()
```

Creates a new instance of Main

Method Detail

main

```
public static void main(java.lang.String[] args)
```

Parameters:

`args` - the command line arguments

antsystem.model

Class PheromoneMatrix

```
java.lang.Object
  └ antsystem.model.PheromoneMatrix
```

```
public class PheromoneMatrix
extends java.lang.Object
```

Field Summary

static java.lang.Object	**lock**

Constructor Summary

PheromoneMatrix()
PheromoneMatrix(int width, int height) Creates a new instance of EnvironmentMatrix

Method Summary

void	**addPheromoneLevel**(java.awt.Point p, int val)
void	**clear**() Clears all the pheromone
void	**depositPheromone**(int x, int y, int val)
void	**evaporate**(int level)
java.util.Set<java.awt.Point>	**existentPoints**()
int	**getHeight**()
int	**getPheromoneLevel**(int x, int y)
int	**getPheromoneLevel**(java.awt.Point p)
int	**getValue**(int x, int y) **Deprecated.**

int	**getWidth**()
void	**pait**(java.awt.Graphics g)
void	**setHeight**(int height)
void	**setValue**(int x, int y, int val) **Deprecated.**
void	**setWidth**(int width)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

lock

```
public static java.lang.Object lock
```

Constructor Detail

PheromoneMatrix

```
public PheromoneMatrix(int width,
                       int height)
```

Creates a new instance of EnvironmentMatrix

PheromoneMatrix

```
public PheromoneMatrix()
```

Method Detail

getValue

```
@Deprecated
public int getValue(int x,
                    int y)
```

Deprecated.

---

setValue

```
@Deprecated
public void setValue(int x,
```

```
                                int y,
                                int val)
```

Deprecated.

---

getPheromoneLevel

```
public int getPheromoneLevel(int x,
                             int y)
```

---

getPheromoneLevel

```
public int getPheromoneLevel(java.awt.Point p)
```

---

depositPheromone

```
public void depositPheromone(int x,
                             int y,
                             int val)
```

---

addPheromoneLevel

```
public void addPheromoneLevel(java.awt.Point p,
                              int val)
```

---

existentPoints

```
public java.util.Set<java.awt.Point> existentPoints()
```

---

evaporate

```
public void evaporate(int level)
```

---

pait

```
public void pait(java.awt.Graphics g)
```

---

getWidth

```
public int getWidth()
```

---

setWidth

```
public void setWidth(int width)
```

---

getHeight

```
public int getHeight()
```

---

setHeight

```
public void setHeight(int height)
```

---

clear

```
public void clear()
```

Clears all the pheromone

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

antsystem.model

Class PheromoneSniffer

```
java.lang.Object
  └antsystem.model.PheromoneSniffer
```

```
public class PheromoneSniffer
extends java.lang.Object
```

Field Summary

static int	**BACK**
static int	**FRONT**
static int	**LEFT_135**
static int	**LEFT_45**
static int	**LEFT_90**
static int	**RIGHT_135**
static int	**RIGHT_45**
static int	**RIGHT_90**

Constructor Summary

PheromoneSniffer() Creates a new instance of PheromoneSniffer
PheromoneSniffer(int distance, PheromoneMatrix pheromoneMatrix)
PheromoneSniffer(PheromoneMatrix pheromoneMatrix)

Method Summary

int	**getDistance**()

void	**setDistance**(int distance)
int	**sniff**(java.awt.geom.Point2D from, Direction antDirection, int sniffTo)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

FRONT

public static final int **FRONT**

See Also:
Constant Field Values

RIGHT_45

public static final int **RIGHT_45**

See Also:
Constant Field Values

RIGHT_90

public static final int **RIGHT_90**

See Also:
Constant Field Values

RIGHT_135

public static final int **RIGHT_135**

See Also:
Constant Field Values

BACK

public static final int **BACK**

See Also:
Constant Field Values

LEFT_45

```
public static final int LEFT_45
```

See Also:
Constant Field Values

LEFT_90

```
public static final int LEFT_90
```

See Also:
Constant Field Values

LEFT_135

```
public static final int LEFT_135
```

See Also:
Constant Field Values

Constructor Detail

PheromoneSniffer

```
public PheromoneSniffer()
```

Creates a new instance of PheromoneSniffer

PheromoneSniffer

```
public PheromoneSniffer(PheromoneMatrix pheromoneMatrix)
```

PheromoneSniffer

```
public PheromoneSniffer(int distance,
                        PheromoneMatrix pheromoneMatrix)
```

Method Detail

getDistance

```
public int getDistance()
```

---

setDistance

```
public void setDistance(int distance)
```

---

sniff

```
public int sniff(java.awt.geom.Point2D from,
                 Direction antDirection,
                 int sniffTo)
```

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS **FRAMES NO FRAMES All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

antsystem.model

Class Actor

```
java.lang.Object
  └ antsystem.model.Actor
```

Direct Known Subclasses:
Ant2, Ant3, FoodSource, Nest

```
public abstract class Actor
extends java.lang.Object
```

Actor is an abstract base class for all actors to be be put in the `Environment`. These actors includes mainly all the agents in the system like `Ant3`, and other types of actor such as `FoodSource` and `Nest`

Constructor Summary

`Actor``()`
`Actor``(java.awt.Point center, java.awt.Dimension dimension)`
`Actor``(SpriteCache spriteCache)` Creates a new instance of Actor
`Actor``(SpriteCache spriteCache, java.awt.Point center, java.awt.Dimension dimension)`

Method Summary

`abstract void`	**`act`**`()` The main method of the Actor.
`abstract void`	**`actorHit`**`(Actor a, int sensor, double distanceToTheCenter)` Its called by the `Environmnet` whenever it is collided to another `Actor` in the `Environment` .
`java.awt.geom.Rectangle2D`	**`getBoundingBox`**`()` The bounding box of the actor
`java.awt.Point`	**`getCenter`**`()` Returns the actual center position of the `Actor`
`java.awt.Dimension`	**`getDimension`**`()` Returns the current dimension of the Actor.
`abstract int`	**`getNumOfSensors`**`()` Returns the number of Sensors in the Actor
`java.awt.Point`	**`getPosition`**`()` Returns the current position of the Actor.
`abstract java.awt.geom.Rectangle2D`	**`getSensor`**`(int sensor)` Returns the correspondent bounding box of sensor index.

SpriteCache	**getSpriteCache**()
int	**getVelX**() Returns the current velocity in X axis of the Actor.
int	**getVelY**() Returns the current velocity in Y axis of the Actor.
boolean	**hit**(java.awt.geom.Rectangle2D r) Tests whether the Rectangle2D hits the Actor
abstract void	**hitAnObstacle**(int sensorIdx) Its called by the Environment every time the Actor is collided with some obstacle in the Environment.
boolean	**isInside**(java.awt.geom.Rectangle2D r) Tests whether the Rectangle2D is inside the Actor
abstract void	**paint**(java.awt.Graphics2D g) Its called by Environment every time it needs to update the visual representation of the World This method must be overridden to provides the correctly visual representation.
void	**setCenterAt**(java.awt.Point p) Sets the new position for the Actor using the desired center.
void	**setDimension**(java.awt.Dimension dimension) Sets the new dimension of the Actor
void	**setPosition**(java.awt.Point position) Sets a new position for the Actor
void	**setSpriteCache**(SpriteCache spriteCache)
void	**setVelX**(int velX) Sets the new velocity in X axis.
void	**setVelY**(int velY) Sets the new velocity in Y axis.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Actor

```
public Actor(SpriteCache spriteCache)
```

Creates a new instance of Actor

Actor

```
public Actor(java.awt.Point center,
             java.awt.Dimension dimension)
```

Actor

```
public Actor(SpriteCache spriteCache,
             java.awt.Point center,
             java.awt.Dimension dimension)
```

Actor

```
public Actor()
```

Method Detail

getBoundingBox

```
public java.awt.geom.Rectangle2D getBoundingBox()
```

The bounding box of the actor

Returns:
The bounding box of the actor

---

getPosition

```
public java.awt.Point getPosition()
```

Returns the current position of the Actor.

Returns:
The position is based on the most top left point of the Actor.

---

setPosition

```
public void setPosition(java.awt.Point position)
```

Sets a new position for the Actor

Parameters:
`position` - The most top left point of the new position

---

getDimension

```
public java.awt.Dimension getDimension()
```

Returns the current dimension of the Actor.

Returns:
The `Dimension` of the Actor.

---

setDimension

```
public void setDimension(java.awt.Dimension dimension)
```

Sets the new dimension of the Actor

Parameters:
`dimension` - The new `Dimension` to be set in the Actor.

getVelX

```
public int getVelX()
```

Returns the current velocity in X axis of the Actor.

Returns:
The current X velocity in `int` precision.

---

setVelX

```
public void setVelX(int velX)
```

Sets the new velocity in X axis.

Parameters:
`velX` - The value in `int` precision.

---

getVelY

```
public int getVelY()
```

Returns the current velocity in Y axis of the Actor.

Returns:
The current Y velocity in `int` precision.

---

setVelY

```
public void setVelY(int velY)
```

Sets the new velocity in Y axis.

Parameters:
`velY` - The value in `int` precision.

---

act

```
public abstract void act()
```

The main method of the Actor. It will be called by the `Environment` in every update of the World. This method must be overridden to provides the correctly action for the Actor, for example, if the Actor is moving, it should update its velocity values here.

---

paint

```
public abstract void paint(java.awt.Graphics2D g)
```

Its called by Environment every time it needs to update the visual representation of the World This method must be overridden to provides the correctly visual representation.

Parameters:
`g` - The Graphics context in which the actor will be painted.

hitAnObstacle

```
public abstract void hitAnObstacle(int sensorIdx)
```

Its called by the Environment every time the Actor is collided with some obstacle in the `Environment`.

Parameters:

`sensorIdx` - The number corresponding to the sensor that is collided.

---

actorHit

```
public abstract void actorHit(Actor a,
                              int sensor,
                              double distanceToTheCenter)
```

Its called by the `Environmnet` whenever it is collided to another `Actor` in the `Environment` .

Parameters:

`a` - The `Actor` tho whom it is collided.
`sensor` - The index of the sensor collided.
`distanceToTheCenter` - The distance to the center point of the collided `Actor`.

---

getSensor

```
public abstract java.awt.geom.Rectangle2D getSensor(int sensor)
```

Returns the correspondent bounding box of sensor index.

Parameters:

`sensor` - The index of the desired sensor.

Returns:

A `Rectangle2D` correspondent to the desired sensor.

---

getNumOfSensors

```
public abstract int getNumOfSensors()
```

Returns the number of Sensors in the Actor

Returns:

the number of Sensors

---

getCenter

```
public java.awt.Point getCenter()
```

Returns the actual center position of the `Actor`

Returns:

`Point` instance corresponding to the actual center position.

---

setCenterAt

```
public void setCenterAt(java.awt.Point p)
```

Sets the new position for the Actor using the desired center.

Parameters:
`p` - The `Point` to the new center position.

---

hit

```
public boolean hit(java.awt.geom.Rectangle2D r)
```

Tests whether the `Rectangle2D` hits the `Actor`

Parameters:
`r` - The rectangle against which the test will be done.
Returns:
True if the rectangle hits the `Actor`, false other way.

---

isInside

```
public boolean isInside(java.awt.geom.Rectangle2D r)
```

Tests whether the `Rectangle2D` is inside the `Actor`

Parameters:
`r` - The rectangle against which the test will be done.
Returns:
True if the rectangle is inside the `Actor`, false other way.

---

getSpriteCache

```
public SpriteCache getSpriteCache()
```

---

setSpriteCache

```
public void setSpriteCache(SpriteCache spriteCache)
```

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES** **All Classes**
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD