

Sistema Multiagentes Utilizando a Linguagem AgentSpeak(L) para Criar Estratégias de Armadilha e Cooperação em um Jogo Tipo PacMan

Alisson Rafael Appio (Titan Informática)

alissonr_appio@yahoo.com.br

Jomi Fred Hübner (FURB/DSC)

jomi@inf.furb.br

Resumo. Este artigo descreve um Sistema Multiagentes (SMA) atuando sobre um jogo tipo PacMan, onde os personagens fantasmas são concebidos como agentes. A arquitetura e a linguagem utilizadas são *Belief-Desires-Intentions* (BDI) e *AgentSpeak(L)*, respectivamente. Os agentes têm como objetivo criar estratégias de armadilha e cooperar na execução das armadilhas criadas, dificultando a vitória do personagem come-comer, que é controlado por um usuário. A ferramenta *Jason* é utilizada para interpretar e executar o código *AgentSpeak(L)*.

Palavras-chave: Cooperação em sistemas multiagentes; jogos; estratégias de armadilha em jogos; arquitetura BDI; *AgentSpeak(L)*.

1 Introdução

Os jogos de computadores cada vez mais possuem um mercado atrativo, recebendo a atenção dos cientistas no desenvolvimento de técnicas computacionais sofisticadas com o uso de várias mídias, animações com gráficos 2D e 3D, vídeos, som, etc. A construção de jogos é uma das tarefas mais difíceis dentro da computação.

Um dos jogos mais populares é o PacMan. Neste jogo existem dois tipos de personagens, sendo eles: os fantasmas e o come-comer. O personagem come-comer é controlado por um usuário e tem como objetivo comer todas as bolinhas que estão situadas no cenário. Quando terminar de comer as bolinhas, o usuário vence o jogo. Os personagens fantasmas são controlados pelo computador e têm como objetivo evitar que o come-comer vença o jogo. Para isso, os fantasmas tentam matar o personagem come-comer.

Neste artigo, é descrito um jogo tipo PacMan, onde os personagens fantasmas são concebidos como agentes BDI, utilizando técnicas de cooperação em Sistema Multiagentes (SMA). O objetivo dos agentes é evitar que o usuário vença o jogo, uma das formas disto acontecer é criar estratégias de armadilha, prendendo o come-comer em algum lugar do mundo para conseguir matá-lo e conseqüentemente não deixando que o jogador vença. Este artigo visa mostrar um sistema que utiliza agentes BDI programados na linguagem *AgentSpeak(L)*. O código *AgentSpeak(L)* é interpretado e executado pela ferramenta *Jason*.

2 Sistema multiagentes

A área de SMA estuda o comportamento de um grupo de agentes (aplicando as técnicas de Inteligência Artificial (IA) clássica) que cooperam para resolver um problema que normalmente um único agente não seria capaz de resolver. Ocupa-se da construção de sistemas computacionais a partir da criação de entidades de software autônomas que interagem através de um ambiente compartilhado por outros agentes de uma sociedade e atuam sobre esses ambientes, alterando seu estado.

Definições mais detalhadas de SMA, seus problemas e aplicações podem ser encontradas nas seguintes referências: Alvares e Sichman (1997), Bordini, Vieira e Moreira (2001), Demazeau e Müller (1990), Jennings e Wooldridge (1998), Weiß (2000), Wooldridge (2002), Bordini e Vieira (2003).

Em SMA, existe a necessidade de coordenar as interações que ocorrem entre os agentes. Vários autores apresentam o tema coordenação, entre os quais cita-se (WEISS, 2000; WOOLDRIDGE, 2002; OLIVEIRA, 2001; RUSSEL; NORVIG, 2003). Os tipos de coordenação são descritos nas seções seguintes.

2.1 Coordenação

Pode-se dizer que coordenação em SMA é um processo no qual um agente raciocina sobre suas ações locais e ações de outros agentes com o objetivo de garantir que a comunidade funcione de maneira coerente (JENNINGS, 1996). É um ato de trabalhar em conjunto no sentido de atingir um acordo com objetivo(s) comum(ns) de forma harmoniosa. A necessidade de coordenação surge do fato da existência de dependências entre as ações dos agentes e da impossibilidade de resolução de um problema por um único agente, seja pela insuficiência de recursos, informações ou capacidade dos agentes (JENNINGS; WOOLDRIDGE, 1998).

Um exemplo de coordenação entre agentes acontece quando dois ou mais robôs precisam carregar uma mesa. Eles devem decidir quem vai pegar uma determinada parte da mesa através de uma negociação entre eles e também coordenar suas ações para conseguirem carregar a mesa, pois se eles não coordenarem suas ações um robô poderá ficar parado enquanto o outro tentará carregá-la, conseqüentemente derrubando a mesa. A coordenação acontece quando os robôs trabalham juntos para resolver um problema, ou seja, mover a mesa para outro lugar.

2.2 Cooperação

Cooperação é um tipo de coordenação entre os agentes que exercem ações com objetivo de atingir um bem social¹, *i.e.*, estão preocupados em atingir objetivos partilhados com outros agentes. Para que os agentes consigam uma cooperação satisfatória cada agente deve manter um modelo dos outros agentes e também desenvolver um modelo das futuras interações (WEISS, 2000).

Quando os agentes estão trabalhando cooperativamente, pode-se dizer que estão trabalhando como equipes e comportam-se de forma a incrementar a utilidade global do sistema e não sua utilidade individual, por exemplo:

- a) cooperação entre agentes acontece em um ambiente de direção de um táxi, onde o agente deve evitar colisões para maximizar a medida de desempenho de todos os agentes (RUSSEL; NORVIG, 2003);
- b) o planejamento de equipes em tênis de duplas. Dois agentes que jogam em uma equipe de tênis de duplas têm como objetivo comum vencer a partida, dando origem a vários sub-objetivos. Um dos sub-objetivos gerado é que eles tenham que devolver a bola que foi lançada para eles e assegurar que pelo menos um deles estará cobrindo a rede, pois essa é uma boa estratégia de jogo (RUSSEL; NORVIG, 2003).

Muitas vezes são adotadas “convenções” ou “leis sociais” para os planejamentos em SMA. Por exemplo, em jogos de tênis de dupla, a bola pode estar aproximadamente equidistante dos dois parceiros. Para resolver esse impasse, um dos agentes poderia gritar “é minha!” ou “é sua!”, através da comunicação estabelecida entre eles (RUSSEL; NORVIG, 2003).

¹Bem social, no sentido onde os agentes não entrem em discussão sobre seus objetivos, quando estes objetivos são de interesse da sociedade como um todo

2.3 Agentes

Vários autores apresentam definições de agentes, e entre as definições mais aceitas destacam-se as de Alvares e Sichman (1997), Barone et al. (2004), Bordini e Vieira (2003). Segundo estes autores, um agente é uma entidade real ou virtual, que está inserida em um ambiente, podendo perceber, agir, deliberar e comunicar-se com outros agentes e possui comportamentos autônomos. Os agentes podem ser divididos em duas categorias: reativos e cognitivos. Agentes reativos possuem comportamentos simples, não possuindo nenhum modelo do mundo onde estão atuando e possuem comportamento estímulo-resposta. Agentes cognitivos possuem comportamentos complexos onde eles deliberam e negociam suas ações com os outros agentes. Na construção de agentes cognitivos em SMA é importante mencionar alguns aspectos, como: percepção; ação; comunicação; representação; motivação; deliberação; raciocínio e aprendizagem.

Existem diversas arquiteturas para os agentes. As arquiteturas mais conhecidas na literatura são: Reativa, *Subsumption*, BDI, Deliberativa e em Camadas. Uma arquitetura para o modelo cognitivo é a arquitetura BDI (as demais arquiteturas não são abordadas neste trabalho).

BDI é uma arquitetura caracterizada por três atitudes mentais que são as crenças, os desejos e as intenções. Os principais criadores da arquitetura BDI foram Georgeff e Rao (BORDINI; VIEIRA, 2003). A fundamentação filosófica para esta concepção de agentes vem do trabalho de Dennett (1987) sobre sistemas intencionais e de Bratman (1987) sobre raciocínio prático. Uma arquitetura BDI genérica é mostrada na figura 1.

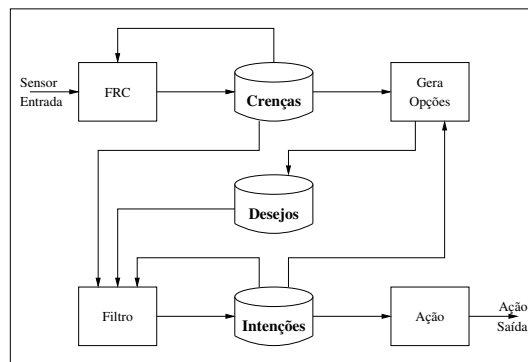


Figura 1: Arquitetura BDI genérica. Fonte: (WOOLDRIDGE, 1999)

As crenças representam tudo aquilo que o agente sabe sobre o ambiente e sobre os agentes daquele ambiente (inclusive sobre si mesmo). Os desejos representam os estados do mundo que o agente quer atingir. As intenções representam a seqüência de ações que um agente compromete-se a executar para atingir sua meta.

A função FRC (Função de Revisão de Crenças) recebe as informações do ambiente, podendo ler e atualizar a base de crenças do agente. Com as alterações do estado do ambiente, podem ser gerados novos objetivos. A função Gera Opções verifica quais estados devem ser atingidos de acordo com o estado atual e as intenções com que o agente está comprometido. A função filtro serve para atualizar o conjunto de intenções do agente com base nas crenças e desejos que ele possui. A função de Ação representa a escolha de uma determina ação para ser executada.

Uma das linguagens que considera os conceitos da arquitetura BDI é a linguagem *AgentSpeak(L)*, descrita na próxima seção.

3 Linguagem AgentSpeak(L)

Agentes cognitivos podem ser especificados através da linguagem *AgentSpeak(L)*. Um programa *AgentSpeak(L)* é especificado por um conjunto de crenças, planos, eventos ativadores e um conjunto de ações básicas que o agente executa no ambiente. Os programas feitos em *AgentSpeak(L)* são interpretados de maneira similar à programas escritos em lógica (como por exemplo, programas escritos em Prolog).

Uma crença é um predicado de primeira ordem na notação lógica usual (ou fatos, no sentido de programação lógica) e literais de crenças são átomos de crenças ou suas negações que formarão a base de crenças do agente.

Planos fazem referência a ações básicas que um agente é capaz de executar em seu ambiente, sendo composto por um evento ativador, contexto e corpo. Os planos são sensíveis ao contexto, i.e., necessitam que certas condições sejam satisfeitas para serem executados, sendo que o contexto deve ser uma consequência lógica da base de crenças do agente no momento em que o evento é selecionado pelo agente para o plano ser considerável aplicável. O corpo do plano é uma seqüência de ações básicas ou subobjetivos que o agente deve atingir ou testar quando uma instância do plano é selecionada para execução.

A linguagem *AgentSpeak(L)* distingue dois tipos de objetivos: objetivos de realização e objetivos de teste. Objetivos de realização e teste são predicados, tais como crenças, porém com operadores prefixados “!” e “?” respectivamente. Objetivos de realização expressam os desejos do agente e objetivos de teste retornam a unificação do predicado de teste com uma crença do agente ou pode falhar quando não existir nenhuma crença que seja satisfeita (BORDINI; VIEIRA, 2003).

Quando o agente percebe informações sobre o ambiente, é gerado um evento com esta percepção, sendo adicionado a sua base de crenças. É gerada uma lista com os planos que podem ser executados com essa percepção (i.e. que tenham o predicado do evento gerado pelo ambiente) e testado o contexto do plano para verificar quais planos podem ser aplicados. Por fim é selecionado um plano da lista de planos e o agente executa o plano selecionado.

3.1 Ferramenta Jason

A ferramenta **Jason**² (do inglês: *A Java-based AgentSpeak Interpreter Used with Saci For Multi-Agent Distribution Over the Net*) proporciona a interpretação e execução de programas escritos na linguagem *AgentSpeak(L)*. SMA são facilmente configurados nesta ferramenta, podendo ser executado em vários computadores através do SACL³ (HÜBNER; SICHMAN, 2000). **Jason** é implementado na linguagem Java (executada em múltiplas plataformas), sendo *Open Source* e distribuído sob a licença GNU LGPL.

Um SMA desenvolvido na ferramenta **Jason**, possui um ambiente onde os agentes estão situados e um conjunto de instâncias de agentes *AgentSpeak(L)*. O ambiente dos agentes, deve ser desenvolvido na linguagem Java. **Jason** possui os seguintes recursos (BORDINI; HÜBNER et al., 2004):

- a) negação forte (*strong negation*), portanto é possível construir sistemas que consideram mundo-fechado (*closed-world*) e mundo-aberto (*open-world*);
- b) tratamento de falhas em planos;

²Disponível em <http://jason.sourceforge.net>

³Disponível em <http://www.lti.pcs.usp.br/saci/>

- c) comunicação baseada em atos de fala (incluindo informações de fontes como anotações de crenças);
- d) anotações em identificadores de planos, que podem ser utilizados na elaboração de funções personalizadas para seleção de planos;
- e) suporte para desenvolvimento de ambientes (que normalmente não é programado em *AgentSpeak(L)*);
- f) possibilidade de executar o SMA distribuídamente em uma rede (usando o SACI);
- g) possibilidade de especificar (em Java) as funções de seleção de planos, as funções de confiança e toda a arquitetura do agente (percepção, revisão de crenças, comunicação e atuação);
- h) possui uma biblioteca básica de “ações internas”;
- i) possibilitar a extensão da biblioteca de ações internas.

A configuração do SMA é feita em arquivos com extensão *.mas2j*, basicamente é informado qual a arquitetura do SMA, “Centralized” ou “Saci” (centralizado ou distribuída), qual o ambiente onde os agentes estão situados e os agentes. A programação dos agentes *AgentSpeak(L)* é feita em arquivos com extensão *.asl*. Para ver a BNF da linguagem *AgentSpeak(L)* e como configurar um SMA na ferramenta *Jason* pode-se consultar (BORDINI; HÜBNER et al., 2004).

4 Especificação do jogo

O *software* desenvolvido (denominado PacMan_MAS) está separado em camadas (figura 2): camada de persistência; camada de lógica do jogo e a camada de SMA (agentes).

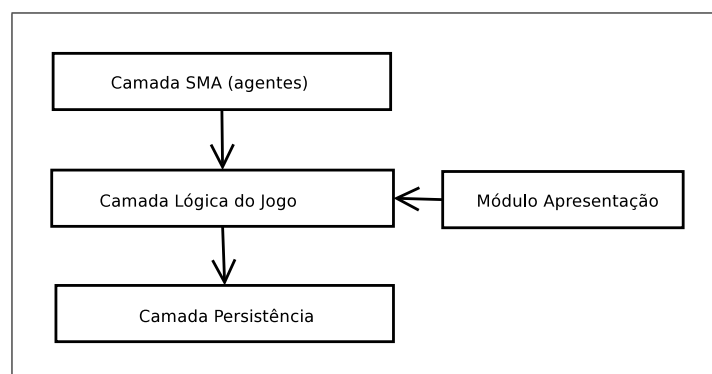


Figura 2: Camadas do jogo PacMan_MAS

A camada *Lógica do Jogo* contém a lógica de um jogo tipo PacMan, contendo as classes *GameObject* (objetos do jogo) que é uma classe abstrata, contendo os atributos que representam as coordenadas de um objeto no jogo (x e y) e possui dois métodos abstratos *getHeight()* e *getWidth()* que são chamados quando o jogo necessita saber a altura e largura de um determinado objeto. Ainda, é nesta classe que é implementado o método *collision(GameObject)* para verificar se um objeto do jogo está colidindo com outro objeto (o diagrama de classes desta camada pode ser visto na figura 3).

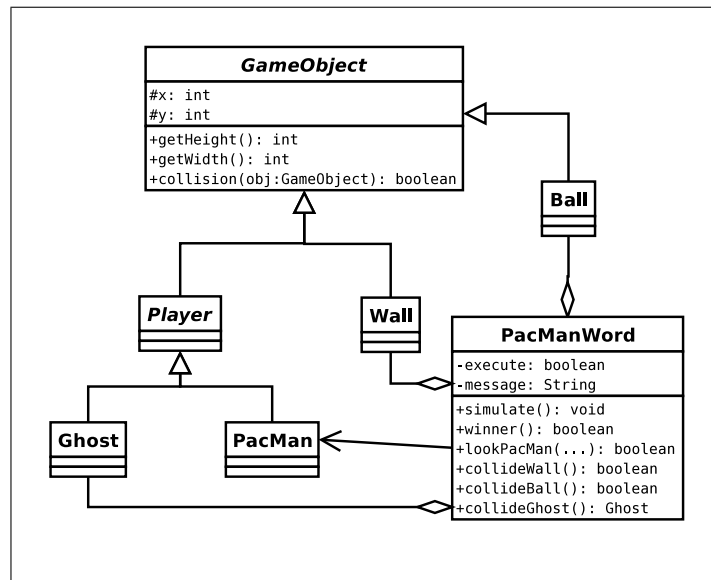


Figura 3: Diagrama de classes da camada lógica do jogo

A classe *Player* (Jogador) é uma classe abstrata, possui como super classe *GameObject*. É classe base para *PacMan* (Come-come) e *Ghost* (Fantasma) e tem como principal método, `paint(Graphics g, Component c)` que desenha a figura do jogador (come-come ou fantasma) no componente passado por parâmetro.

A classe *PacManWorld* (mundo do *PacMan*) representa a principal classe do jogo, possuindo listas de *Ghost* (Fantasma), *Ball* (Bola), *Wall* (Parede) e um objeto *PacMan* (Come-come). Os principais métodos dessa classe são: `simulate()`; `lookPacMan(String ghostName, char direction)`; `winner()`; `collideWall(Player player)`; `collideBall()` e `collideGhost()`.

A classe *Ball* representa uma bolinha do jogo. Possui como atributos uma imagem de uma bolinha (atributo de classe), seu estado (visível ou invisível) e os atributos herdados de *GameObject*. A classe *Wall* possui como atributos a sua largura, altura e os atributos herdados de *GameObject*.

O módulo de apresentação, desenha o jogo *PacMan.MAS*. Foi utilizada a estratégia de *double buffer*⁴ para desenhar as bolinhas, paredes, come-come e os fantasmas.

O usuário controla a direção do come-come, pressionando as teclas *UP*, *DOWN*, *LEFT*, *RIGHT* que move o fantasma para cima, baixo, esquerda e direita, respectivamente.

A camada de persistência faz todo o acesso de leitura de arquivos texto contendo as coordenadas dos objetos do jogo (paredes, bolinhas, etc). O *layout* do arquivo de bolinhas, contém uma coordenada *x*, *y*. O *layout* do arquivo de paredes, possui quatro valores numéricos (coordenada *x*, coordenada *y*, a largura e a altura da parede).

Os fantasmas são movimentados através de um grafo não dirigido, cada esquina do labirinto do jogo é um vertice e as arestas são as estradas que ligam uma esquina a outra. O *layout* do arquivo de

⁴É criado uma imagem em segundo plano, feito todo o desenho do jogo nesta imagem, para depois desenhar a imagem final na tela.

vértices contém três valores numéricos (coordenada x , coordenada y e um identificador (único) de cada vértice). O *layout* do arquivo de arestas possui dois valores numéricos (identificador de origem e identificador de destino da aresta). Mais detalhes da especificação e implementação destas duas camadas podem ser obtidas em (APPIO, 2004).

5 Camada SMA

Nesta camada são implementadas as classes: `EnvironmentPacMan` para efetuar uma ponte entre o jogo (ambiente) e o interpretador *Jason*; a classe `AgentGhost` que customiza a escolha de planos do agente; e a classe `EstadoJogo` utilizada para encontrar caminhos ótimos no labirinto do jogo.

5.1 Ambiente do SMA

Nesta camada é implementada a classe `EnvironmentPacMan` que estende a classe `Environment` (classe que a ferramenta *Jason* especifica) para se tornar um ambiente para o SMA, assim, efetuando uma ligação entre o ambiente e o *Jason*.

Os principais métodos da classe `EnvironmentPacMan` são:

- `executeAction(String ag, Term action)`, este método executa uma ação no ambiente. É passado por parâmetro o nome do agente e a ação que deve ser executada, e retorna *true* se conseguiu executar a ação ou *false* se falhar;
- `getPercepts(String agName)`, este método é chamado por cada agente do SMA, sendo passado como parâmetro o nome do agente que está solicitando as percepções do ambiente. É retornada uma lista com as percepções (naquele instante) do agente.

Constantemente o agente solicita as suas percepções ao ambiente e verifica se recebeu alguma mensagem de outro agente (figura 4). O agente recebe do ambiente a sua posição (coordenada x, y), através da adição da crença `pos(X, Y)`. Ele também consegue perceber o come-como (coordenada x, y), através da adição da crença `pm(X, Y)`, desde que o come-como esteja em seu campo de visão, *i.e.*, não deve existir nenhuma parede entre o come-como e o agente nas quatro direções: norte; sul; leste e oeste.

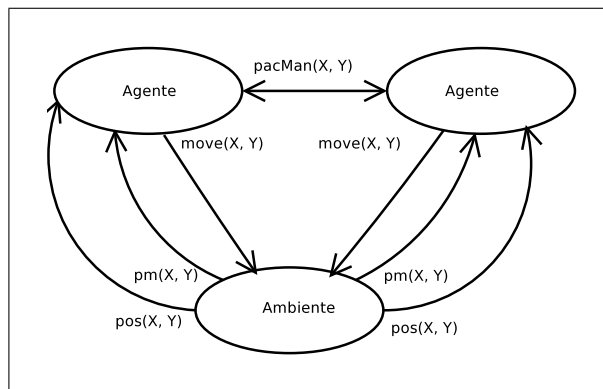


Figura 4: Comunicação entre agentes e ambiente

5.2 Especificação dos planos de movimentação do agente

Quando o agente recebe a percepção de sua posição no ambiente, por meio do evento `+pos(X, Y)`, o plano do `pos` (logo abaixo) é aplicável. Caso o agente não esteja se movendo e não esteja em modo de cooperação, o plano é selecionado e vira uma intenção do agente.

```
+ pos(X, Y) : not moving & not cooperate ←
    .nodoMaisProximo(X, Y, null, null,
        Xresult, Yresult, CornerGoal);
+ moving;
!go(Xresult, Yresult, CornerGoal).
```

O código do plano apresentado faz o agente andar aleatoriamente. No corpo do plano, o agente executa a ação interna `.nodoMaisProximo(...)` para buscar a esquina mais próxima a ele naquele momento. É adicionada uma crença `moving` e por fim é executado o plano `go(...)` (objetivo de realização).

Para o agente descobrir uma esquina, ele executa a ação interna `.nodoMaisProximo(Xg, Yg, Xpm, Ypm, Xresult, Yresult, CornerGoal)`. Esta ação retorna as coordenadas x, y e o identificador da esquina nos parâmetros `Xresult, Yresult, CornerGoal`. Esta ação pode ser executada de duas maneiras: quando o agente está andando aleatoriamente ou quando entra em modo de cooperação (modo de cooperação é explicado a seguir). Para mover o agente de uma esquina para outra aleatoriamente (os parâmetros `Xpm` e `Ypm` não estão instanciados), é necessário efetuar uma consulta ao grafo, descobrindo quais são as esquinas vizinhas. Após descobrir quais são as esquinas vizinhas, é sorteada uma esquina aleatoriamente para simular que o agente está se movendo de uma esquina para outra ao acaso.

Para o agente mover-se de uma esquina para outra, ele executa o plano `go(X, Y, CornerGoal)` recursivamente. O primeiro plano `go(X, Y, CornerGoal)` é executado quando o agente conseguiu chegar em uma esquina passada como parâmetro, quando o contexto deste plano falha é executado o segundo plano, que possui contexto igual a `true`. O código *AgentS-peak(L)* abaixo, implementa estes conceitos.

```
+! go(X, Y, CornerGoal): pos(X, Y) ←
    - moving;
    + pos(X, Y).
+! go(X, Y, CornerGoal): true ←
    ? pos(Xpos, Ypos);
    .nextPosition(Xpos, Ypos, CornerGoal, Xresult, Yresult);
    move(Xresult, Yresult);
    !go(X, Y, CornerGoal).
```

A ação interna `.nextPosition(...)` retorna a próxima coordenada x, y (nos parâmetros `Xresult, Yresult`) em direção a uma esquina que ele pretende ir. A esquina é representada pelo parâmetro `CornerGoal`, os parâmetros `Xpos, Ypos` representam a última posição do agente no ambiente.

O agente modifica o ambiente (alterar o estado do jogo) através da ação representada pelo literal `move(X, Y)`, que atualiza as coordenadas x, y do agente que executou esta ação.

5.3 Estratégia de armadilha no jogo

Quando um fantasma percebe o come-come, ele gera dois objetivos: mover-se para as coordenadas x e y do come-come (para matar o come-come) e o outro objetivo gerado é de enviar uma mensagem para os outros agentes do SMA, informando a posição do come-come. Quando um fantasma recebe uma mensagem de um outro fantasma sobre a posição do come-come, ele entra em modo de cooperação.

Ao entrar em modo de cooperação, a ação interna `.nodoMaisProximo(...)` efetua uma busca heurística, utilizando o algoritmo A* (RUSSEL; NORVIG, 2003) para saber qual é a esquina mais próxima a ele e ao come-come. É feita uma busca com heurística da menor distância em linha reta entre dois pontos. Os estados sucessores (algoritmo de busca) possuem como *custo* a distância de uma esquina até outra. O g é o custo acumulado de todo o caminho (soma do *custo*). O h é a distância do vértice corrente até o vértice meta. O f representa a soma do $g + h$ (mais detalhes sobre o algoritmo de A* consulte (RUSSEL; NORVIG, 2003)).

O plano abaixo é selecionado quando um agente está vendo o come-come, este plano torna-se uma intenção quando é percebido o evento `+pm(X, Y)`. O agente deve comunicar-se com os outros agentes para avisar onde o come-come esta (a posição x, y do come-come) e mover-se em direção ao come-come. Para o agente se mover em direção ao come-come, ele executa a ação interna `.nodoMaisLonge(...)`, que busca a esquina mais longe em linha reta em relação ao come-come e o agente, *i.e.*, fazendo o agente matar o come-come. Para o agente comunicar-se com os outros agentes ele executa o plano `sendCoordinatePacMan(X, Y, ghost, 1)`, os parâmetros x, y representam a posição do come-come, o terceiro parâmetro (`ghost`) será concatenado com um número que estiver no quarto parâmetro (o número começa com 1 e termina com 5), representando os nomes dos agentes (`ghost1, ghost2, ..., ghost5`).

```
+ pm(X, Y) : //...
  ← //...
  ? pos(Xg, Yg);
  .nodoMaisLonge(Xg, Yg, X, Y, Xresult, Yresult, GoalEsquina);
  ! sendCoordinatePacMan(X, Y, ghost, 1);
+ moving;
  ! go(Xresult, Yresult, GoalEsquina).
```

A comunicação entre os agentes é feita através do envio de uma mensagem. Para isso, é usada a ação interna `.send(ghost, tell, literal)`, o primeiro parâmetro representa quem vai receber a mensagem, o segundo parâmetro representa que a mensagem é afirmativa (LABROU; FININ, 1997), o último parâmetro representa um literal que será enviado. Ao perceber o come-come, o agente tem como objetivo avisar os outros agentes da posição do come-come. Para isso, ele executa o plano `sendCoordinatePacMan(...)` (apresentado abaixo). No corpo deste plano é executada a ação interna `.send(...)` que envia a mensagem com o literal `pacMan(X, Y)`, para um determinado agente. Para evitar que o agente mande uma mensagem para ele mesmo, ele executa a ação interna `.myName(MyName)`, no contexto do plano `sendCoordinatePacMan`, esta ação retorna o nome do agente (que executou esta ação) no parâmetro `MyName`.

```
// notice that the pacMan is in position X an Y
+! sendCoordinatePacMan(X, Y, Name, Number): .myName(N) &
  numberAg(Num) & .concat(Name, Number, NewName) &
  N != NewName & Number < Num ←
```

```

        .send(NewName, tell, pacMan(X,Y));
        ! sendCoordinatePacMan(X, Y, Name, Number+1).
+! sendCoordinatePacMan(X, Y, Name, Number):
    numberAg(N) & Number < N ←
        ! sendCoordinatePacMan(X, Y, Name, Number+1).
+! sendCoordinatePacMan(X, Y, Name, Number): true ← true.

```

Ao receber a mensagem contendo o literal `pacMan(X, Y)`, é ativado o plano abaixo, deixando o agente em modo de cooperação. O agente sabe que um outro agente está vindo o come-come na coordenada x, y , então ele executa uma busca para mover-se na direção do come-come, criando uma estratégia de armadilha.

```

+ pacMan(X,Y) : true
    ← + cooperate.

```

Os planos podem ser executados em paralelo. Para a estratégia de armadilha dar certo o agente deve preferir pelos planos de movimentação (`go(...)`) e pelos planos de percepção do come-come (`pm(...)`), para fazer isto, é necessário implementar a classe `AgentGhost` que customiza a seleção de planos do agente. Caso não seja implementado uma seleção de planos, a busca heurística pode ser executada com uma visão parcial do mundo.

6 Dificuldades encontradas

No desenvolvimento deste trabalho, uma das dificuldades encontradas foi que, por se tratar de um SMA atuando sobre um jogo (tempo real) o “estado mental” de cada agente e o ambiente estão mudando constantemente, dificultando a depuração dos agentes. A ferramenta *Jason* permite acompanhar as mudanças nas crenças, desejos e intenções de cada agente gerando um *log* do “estado mental de cada agente”.

Outra dificuldade, está relacionada com a mudança de paradigma de programação Orientação a Objetos (OO) para orientada a agentes.

Foram encontradas poucas bibliografias sobre a linguagem *AgentSpeak(L)* (linguagem especificada a poucos anos por Rao (1996)), existindo poucas aplicações com esta linguagem. Contudo, os poucos códigos que estão disponíveis na ferramenta *Jason* foram de grande utilidade para compreensão da linguagem e construção deste trabalho. Alguns planos podem ser executados em paralelo, causando inconsistências nas técnicas de cooperação utilizadas caso não seja implementada uma seleção de intenção.

7 Resultados e conclusões

O principal resultado é que a estratégia de armadilha proposta neste trabalho é criada pelos agentes. Porém, quando os fantasmas estão muito longe do come-come a estratégia não tem muito sucesso, visto que, a estratégia adotada é de se mover para uma esquina em relação ao come-come para prendê-lo em algum lugar do mundo, fica difícil para eles conseguirem prender (matar) o come-come. Outro fator que pode causar a falha na estratégia de armadilha é quando o come-come sai fora do campo de visão de um dos fantasmas e assim os outros fantasmas saíram do modo de cooperação (ver seção 5.3).

Uma limitação do *software* desenvolvido é que, não foi implementada uma negociação entre os agentes, evitando que eles tenham objetivos de ir para mesma esquina, isso pode acontecer quando a

ação interna de `nodoMaisProximo` retornar a mesma esquina (naquele momento) para os agentes. Uma forma de negociação para resolver este problema poderia ser o agente que estiver mais próximo da esquina se mover em relação a esta esquina e o outro agente deveria escolher uma outra esquina. Quando a distância para chegar na esquina for a mesma para os agentes, poderia ser feita uma escolha aleatória de duas esquinas, uma para cada agente.

Este trabalho representa uma das maiores implementações (utilizando grafo, busca heurística, cooperação, representação gráfica dos agentes *AgentSpeak(L)*, etc) que se conhece, usando a ferramenta **Jason**. A ferramenta demonstrou-se adequada na execução de códigos *AgentSpeak(L)*. A comunicação entre os agentes é feita de maneira transparente e de alto nível para o programador, pois o **Jason** garante o envio e o recebimento das mensagens através da ferramenta SACI (HÜBNER; SICHMAN, 2000). O SACI permite trocar mensagens entre agentes, que podem estar distribuídos em uma rede, através do padrão *Knowledge Query Markup Language (KQML)*.

O objetivo deste trabalho não era testar a ferramenta **Jason**, mas como ela está em desenvolvimento, foram feitos vários testes (com o desenvolvimento deste trabalho), conseqüentemente encontrando alguns *bugs* (um dos *bugs* encontrados, o código do ambiente não podia estar dentro de pacotes) que foram comunicados aos desenvolvedores da ferramenta e corrigidos.

O jogo ficou um pouco lento, em conseqüência dos fantasmas se moverem de dois em dois *pixels*. Contudo, como o *software* está separado em camadas, fazer alterações para os fantasmas se moverem mais rápido, envolve alterar a ação interna `nextPosition` e o código *AgentSpeak(L)*. Para desenvolver novas técnicas de cooperação para o jogo, basta apenas alterar o código *AgentSpeak(L)* e ações internas dos agentes, alterando apenas a camada SMA (agentes).

Fazer os agentes cooperarem não é uma tarefa trivial, pois os agentes devem fazer um planejamento, este planejamento pode ser centralizado ou distribuído, no caso deste trabalho é feito um planejamento distribuído.

A proposta deste trabalho, usar a arquitetura BDI e cooperação para criar armadilha no jogo tipo PacMan, mostrou-se, de forma geral bastante adequada. A arquitetura BDI proporciona construir jogos onde os personagens devem ter um certo nível de inteligência, as atitudes dos personagens estão mudando constantemente de acordo com o estado corrente do jogo e de sua “mente”.

Uma das vantagens da abordagem adotada, é que o comportamento de cada agente é muito simples, não existindo nenhum plano explícito para criar armadilhas, os planos são apenas de movimentação de uma esquina para outra. A partir da interação dos agentes surge (emerge) o comportamento de criar uma armadilha (seção 5.3).

A linguagem *AgentSpeak(L)* é muito poderosa para desenvolver aplicações onde o ambiente dos agentes muda constantemente. Os planos são facilmente construídos e ampliados. Por ser uma linguagem declarativa, a programação em *AgentSpeak(L)* se torna mais elegante, proporcionando um alto nível de abstração na especificação (crenças, desejos e intenções) do agente.

Mesmo os jogos mais sofisticados (que usam busca, possuindo a localização global da posição do come-come) de PacMan, não garantem que o computador vença o jogo, em muitos casos, os fantasmas ficam correndo atrás do personagem come-come até o fim do jogo. Neste trabalho, quando os agentes estão perto das esquinas próximas ao come-come e algum fantasma está vendo o come-come, é quase impossível o come-come fugir da estratégia de armadilha, pois a armadilha vai fechar todas as passagens pelas esquinas próxima do come-come.

Referências

ALVARES, Luiz Otavio; SICHMAN, Jaime Simão. Introdução aos sistemas multiagentes. In: MEDEIROS, Cláudia Maria Bauzer (Ed.). *Jornada de Atualização em Informática (JAI'97)*. Brasília: UnB, 1997. cap. 1, p.

APPIO, Alisson Rafael. *Sistema Multiagentes Utilizando a Linguagem AgentSpeak(L) para Criar Estratégias de Armadilha e Cooperação em um Jogo Tipo PacMan*. Blumenau: FURB, 2004. pag. 11–49 p. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação).

BARONE, Dante Augusto Couto et al. *Sociedades artificiais: a nova fronteira da inteligência nas máquinas*. Porto Alegre: Bookman, 2004.

BORDINI, Rafael H.; HÜBNER, Jomi F. et al. *Jason: a java-based agentspeak interpreter used with SACI for multi-agent distribution over the net*. Manual, first release. [S.l.], Jan 2004. Disponível em: <<http://jason.sourceforge.net/>>. Acesso em: 20 set. 2004.

BORDINI, Rafael H.; VIEIRA, Renata. Linguagens de programação orientadas a agentes: uma introdução baseada em AgentSpeak(L). *Revista de Informática Teórica e Aplicada*, v. 10, p. 7–38, Agosto 2003. Instituto de Informática da UFRGS, Brazil.

BORDINI, Rafael Heitor; VIEIRA, Renata; MOREIRA, Álvaro Freitas. Fundamentos de sistemas multiagentes. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (SBC2001), XX JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA (JAI), 21. *Anais...* Fortaleza-CE, Brasil: Sociedade Brasileira de Computação, 2001. p. 3–41.

BRATMAN, Michael E. *Intentions, plans and practical reason*. Cambridge, MA: Harvard University Press, 1987.

DEMAZEAU, Yves; MÜLLER, Jean Pierre. *Decentralized artificial intelligence 1*. North-Holland: Elsevier Science Publishers, 1990.

DENNETT, Daniel C. *The intentional stance*. Cambridge, MA: The MIT Press, 1987.

HÜBNER, Jomi Fred; SICHMAN, Jaime Simão. SACI: Uma ferramenta para implementação e monitoração da comunicação entre agentes. International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI (IBERAMIA/SBIA 2000), 7/15, Atibaia, São Paulo, Brazil, 2000. *Proceedings (Open Discussion Track)*. São Carlos: ICMC/USP, 2000. p. 47–56.

JENNINGS, N. R. Coordination techniques for distributed artificial intelligence. In: O'Hare G.M.P Jennings N.R. (Ed.). *Foundations of distributed Artificial Intelligence*. [S.l.]: John Wiley & Sons, Inc, 1996.

JENNINGS, Nicholas R.; WOOLDRIDGE, Michael J. (Ed.). *Agent technology: foundations, applications, and markets*. Berlin: Springer-Verlag, 1998.

LABROU, Yannis; FININ, Tim. *A proposal for a new KQML specification*. Baltimore, 1997.

OLIVEIRA, Eugênio. Agents advanced features for negotiation and coordination. In: LUCK, Michael et al. (Ed.). *Multi-agent systems and applicatons*. Prague: Springer, 2001. p. 173–186.

RAO, Anand S. AgentSpeak(L): BDI agents speak out in a logical computable language. In: WORKSHOP ON MODELLING AUTONOMOUS AGENTS IN A MULTI-AGENT WORLD (MAAMAW'96), 7., 1996, Eindhoven, The Netherlands. *Proceedings...* London: Springer-Verlag, 1996. (Lecture Notes in Artificial Intelligence), p. 42–55.

RUSSEL, Stuard; NORVIG, Peter. *Artificial intelligence: a modern approach*. 2^o. ed. New Jersey: Prentice Hall, 2003. ISBN 0-13-790395-2.

WEISS, Gerhard. *Multiagent systems: a modern approach to distributed artificial intelligence*. Cambridge, MA: MIT Press, 2000.

WOOLDRIDGE, Michael. Intelligent agents. In: WEISS, Gerhard (Ed.). *Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence*. Cambridge, MA: MIT Press, 1999. cap. 1, p. 27–77.

WOOLDRIDGE, Michael. *An introduction to multiagent systems*. New York: John Wiley & Sons, 2002.