Universidade Estadual de Londrina

Departamento de Computação
Trabalho de Conclusão de Curso

EDUARDO FUJITA

ALGORITMOS DE IA PARA JOGOS

Londrina
2005

EDUARDO FUJITA

ALGORITMOS DE IA PARA JOGOS

Trabalho apresentado ao Curso de Ciência da Computação, da Universidade Estadual de Londrina, como requisito à obtenção do título de Bacharel.

Orientador: Prof. Dr. Pedro Paulo da Silva Ayrosa

Londrina
2005

EDUARDO FUJITA

ALGORITMOS DE IA PARA JOGOS

COMISSÃO EXAMINADORA

_____

Prof. Dr. Pedro Paulo da Silva Ayrosa
Universidade Estadual de Londrina

_____

Profª. Ms. Liana Dessandre D. Garanhani
Universidade Estadual de Londrina

_____

Prof. Ms. Rafael Robson Negrão
Universidade Estadual de Londrina

Londrina, 1 de dezembro de 2005

"If God does not exist in our world,

then I will create God with my own hands!".

Krelian - Xenogears

AGRADECIMENTOS

Ao Prof. Dr. Pedro Paulo da Silva Ayrosa, por sua maneira descontraída de ministrar as aulas e sua orientação na elaboração do presente trabalho.

À minha família.

Aos amigos, professores e funcionários do departamento de computação da UEL.

FUJITA, Eduardo. **Algoritmos de IA para jogos**. 2005. Monografia (Graduação em Ciência da Computação) – Universidade Estadual de Londrina.

RESUMO

Este trabalho aborda as principais técnicas da Inteligência Artificial (IA) aplicadas na área de desenvolvimento de jogos, como máquinas de estado finito, *scripts*, algoritmos genéticos e redes neurais. Apresenta de maneira sucinta a história da IA bem como as principais características do desenvolvimento de jogos. Demonstra a importância da IA como fator diferencial na criação de jogos mais atraentes ao consumidor. Por fim, faz uma análise do estado da arte nas pesquisas nesta área e conclui que a tendência é a criação de jogos que utilizem técnicas não-determinísticas, aprendendo com seus erros e adaptando-se ao modo de jogar de seu adversário humano.

Palavras-chave: Inteligência Artificial; jogos.

FUJITA, Eduardo. **AI algorithms for games**. 2005. Monograph (Graduation in Computer Science) – Universidade Estadual de Londrina.

ABSTRACT

This work approaches the main Artificial Intelligence (AI) techniques applied in the area of game development, like finite state machines, scripts, genetic algorithms and neural networks. Presents in a succinct way the history of AI as well as the main characteristics of game development. Demonstrates the importance of AI as a differential factor in the creation of more attractive games for the final consumer. Ultimately, makes an analysis of the state-of-the-art in the researches in this area and concludes that the tendency is the creation of games using non-deterministic techniques, learning with its own mistakes and adapting to the playing manner of its human opponent.

Key-words: Artificial Intelligence; games.

LISTA DE TABELAS

LISTA DE EQUAÇÕES

LISTA DE ILUSTRAÇÕES

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

A área de jogos por computador é uma das áreas que mais amadureceu nas últimas décadas; tanto em popularidade quanto em abrangência de áreas. O seu desenvolvimento hoje requer investimentos milionários em uma grande equipe de programadores, artistas, músicos, escritores e também em Inteligência Artificial (IA).

Quando surgiram os primeiros consoles[1] populares, por volta das décadas de 60 e 70, o poder computacional oferecido pelos mesmos era bastante escasso, portanto existia pouco espaço para a implementação de algoritmos de IA. Essa limitação perdurou até meados da década de 90, quando os computadores e consoles avançaram muito em tecnologia, e assim os jogadores passaram a exigir dos jogos um nível de dificuldade compatível com a de um ser humano, pois muitos se queixavam de um alto grau de previsibilidade que os antigos jogos possuíam.

Assim, o estudo da IA aplicada aos jogos foi ganhando apoio no mundo acadêmico, pois além de ser uma área nova e interessante do ponto de vista mercadológico, ela tem se mostrado útil para a implementação de teorias da IA.

O próximo capítulo fará um breve resumo da história da IA, desde os primórdios até os dias atuais.

No terceiro capítulo, são apontados os principais estilos de jogos, bem como as características acerca do desenvolvimento dos mesmos.

O quarto capítulo engloba o tema principal deste trabalho, apresentando as principais técnicas da IA aplicadas aos jogos, bem como exemplos de tais aplicações.

---

[1] Este se difere de um computador pois seu *hardware* é dedicado exclusivamente para a execução de jogos. Neste trabalho não há distinção entre jogos para computador e jogos para consoles.

Já no quinto capítulo, é feito um estudo do estado da arte da IA em jogos, mostrando quais as áreas mais promissoras para pesquisas futuras.

Finalmente, no sexto capítulo, é feita a conclusão deste trabalho, enfatizando as idéias principais apresentadas no mesmo.

2 INTELIGÊNCIA ARTIFICIAL

2.1 Desenvolvimento da Teoria de IA

Muitos autores, tais como RUSSEL (2004), consideram o filósofo grego Aristóteles (384-322 A.C) como um dos principais precursores dos estudos de IA. Segundo ele, o estudo sobre o pensamento era a base para todo o conhecimento. Em uma de suas obras, Aristóteles introduziu o conceito de *silogismo*, que provia argumentos para provar a veracidade de certas sentenças a partir de outras que são sabidamente verdadeiras. Um famoso exemplo de um silogismo é: “Sócrates é homem; todos os homens são mortais; logo, Sócrates é mortal”.

Em sua obra *Lógica*, Aristóteles desenvolveu um dos conceitos fundamentais da IA e da Ciência Cognitiva. Segundo ele, a lógica é um instrumento, e o estudo do pensamento é o alicerce para o conhecimento. Foi a partir desses estudos da lógica, que bem mais tarde (por volta do séc.XX) foi concebida a Prova Automática de Teoremas.

Com a entrada do Renascimento, o homem percebeu certa distinção entre a mente e a matéria, ou seja, entre aquilo que vemos e aquilo que realmente existe:

> “Pela primeira vez, provavelmente, as nossas idéias acerca do mundo foram vistas como fundamentalmente distintas de sua aparência.” (LUGER, 2004).

Esta separação entre mente e realidade foi de fundamental importância no entendimento do nosso pensamento, tendo sido esta idéia reforçada por René Descartes (1596-1650), que discutiu as diferenças entre a mente e a matéria. Desta maneira, percebeu-se que os processos mentais vivem em um mundo à parte, tendo suas próprias leis.

Mais tarde, observou-se a necessidade de reagrupar os processos físicos com os mentais, visto que somos um sistema composto por estas duas entidades. Assim, concluiu-se que tanto os processos mentais quanto os físicos podem ser caracterizados a partir da matemática formal.

Tendo-se chegado à conclusão de que o pensamento é uma forma de computação, surgiu a necessidade de formalizá-lo. Tal formalização veio com Leibniz, que introduziu o primeiro sistema de lógica formal, e Euler, com a teoria dos grafos; esta, é de grande utilidade neste estudo: com os nós dos grafos, é possível representar estados de um agente em determinado momento do jogo. Os arcos representam eventos, que podem perpetuar a mudança de estado (nó) e, consequentemente, a mudança de comportamento do agente.

Nó século XIX, Charles Babbage e Ada Byron trabalharam no estudo de máquinas mecânicas programáveis para o cálculo de valores de funções polinomiais. Embora tal máquina não tenha sido construída com sucesso, seus estudos contribuíram muito com a Ciência da Computação, pois incluía conceitos como a programabilidade, memória e abstração de dados. Ada chegou até mesmo a construir programas para estas máquinas, o que a torna talvez a primeira programadora da história.

Também no século XIX, o matemático George Boole formalizou as leis da lógica, desenvolvendo a Álgebra Booleana, que é a base de toda a Ciência da Computação até os dias de hoje. Boole demonstrou que, com apenas os valores booleanos 0 e 1 sería possível captar todo o poderio da lógica.

Já no iníco do século XX, teve o início da história moderna da IA. Alfred North Whitehead e Bertrand Russel, no *Principia Mathematica*, publicado em 1910-1913 tentaram derivar todas as verdades matemáticas a partir de um conjunto bem definido de axiomas e regras de inferência para a lógica simbólica, lidando com a matemática de uma maneira estritamente formal.

2.2 Nascimento da IA

Embora os avanços científicos citados anteriormente tenham formado a base para o estudo da IA moderna, somente na metade do século XX, com o surgimento dos computadores é que a IA se tornou viável do ponto de vista científico. Graças à sua memória e velocidade de processamento, tornou-se muito mais prático a implementação de grafos, resolução de sistemas de raciocínio formal e heurísticas, bem como fazer testes acerca dos mesmos. Alan Turing (1950) foi um dos primeiros a tratar da inteligência de máquina em relação ao computador digital moderno. Ele propôs um teste empírico para verificar a qualidade da inteligência de máquina em relação ao desempenho humano. Apesar de algumas críticas, o teste tem sobrevivido até os dias atuais.

Baseado na análise da lógica proposicional de Russel, no conhecimento fisiológico e funcional dos neurônios no cérebro, Warren McCulloch e Walter Pitts proporam em 1943 um modelo de neurônios artificiais no qual cada neurônio poderia estar em um estado “ligado” ou “desligado”. Quando um número adequado de neurônios vizinhos estimulasse um certo neurônio, este podería mudar do estado “desligado” para “ligado”, sendo que o estado de um neurônio representa a proposição do qual ele é resultado. Donald Hebb (1949) demonstrou como a modificação dos pesos entre as conexões poderia acrescentar a aprendizagem ao sistema. Em 1951 foi construído o primeiro computador baseado em redes neurais pelos estudantes Marvin Misky e Dean Edmonds –ambos graduados no curso de matemática de Princeton. O computador simulava uma rede de 40 neurônios.

Em 1956, John McCarthy convidou muitos pesquisadores renomeados como o próprio Misky, além de outros como Claude Shannon e Nathaniel Rochester cujo interesse se baseava em tópicos avançados como teoria da computabilidade, redes neurais, bem como o estudo da inteligência. O principal objeto de estudo era tão novo que foi preciso designar a ele um novo nome: Inteligência Artificial (IA).

O evento teve duração de dois meses e é conhecido como “A conferência de IA de verão” (*Summer AI Conference)*.

A conferência teve grande repercussão e construiu a base para uma pesquisa ambiciosa envolvendo diversas áreas como a engenharia, matemática, ciência da computação, psicologia, entre outras. Muitos participantes se convenceram de que com o avanço acentuado da velocidade de processamento conseguida através do *hardware* a cada ano, a possibilidade de se conceber máquinas com capacidade intelectual humana deixaria de ser uma questão de “como” e sim “quando” isto ocorreria. Embora cada um dos tópicos envolvidos possuíssem objetivos similares, foi necessário separar a IA como um ramo de pesquisa completamente diferente, pois somente a IA abraçou a idéia de reproduzir artificialmente habilidades humanas como o aprendizado e uso da linguagem.

O período que se sucedeu a esta conferência foi um período de grande euforia para a pesquisa em IA, pois até então, os computadores eram vistos apenas como simples máquinas aritméticas e nada mais. Uma criação importante no período foi o GPS (*General Problem Solver*), que foi projetado para resolver problemas como um ser humano o faría. Isto faz com que este tenha sido provavelmente um dos primeiros exemplos da abordagem “pensando como um humano”, que será descrita um pouco mais adiante.

Uma das primeiras contribuições da IA na área de jogos se deu por volta de 1952, quando Arthur Samuel escreveu um programa que jogava damas. Tal programa aprendeu a jogar tão bem que chegou ao ponto de derrotar seu próprio criador; provando que era possível construir um programa que fosse além do que fora lhe ensinado.

Em 1958, John McCarthy contribuiu mais uma vez de maneira histórica para a IA. Após mudar-se de Dartmouth para o MIT, McCarthy concebeu uma linguagem de alto nível denominada *Lisp*, que mais tarde se tornaría a linguagem de IA predominante. Ainda no mesmo ano, McCarthy publicou um artigo que descrevia o que pode ser considerado o primeiro sistema completo de IA. Segundo McCarthy, o programa utilizaria uma base de conhecimentos na resolução

de problemas. O diferencial deste em relação ao que havia sido pesquisado até então, é que foi introduzido o conceito de “senso comum”, ou seja, um conhecimento generalizado sobre o mundo. O programa, denominado *Advice Taker* seria capaz, por exemplo, de exibir um plano para dirigir-se até o aeroporto, através de simples axiomas. O programa também seria capaz de adquirir competências em novas áreas, sem a necessidade de ser reprogramado; sendo portanto, capaz de adquirir aprendizagem.

2.3 IA Atualmente

O desenvolvimento da Inteligência Artificial foi tão grande que seu potencial de aplicação se tornou bastante abrangente. Algumas áreas de aplicação incluem:

- planejamento e agendamento autônomo: usado pela NASA para o controle de agentes remotos (robôs enviados em missões interplanetárias);

- entretenimento: na forma de jogos eletrônicos ou no teste científico do poder computacional da IA em comparação com o de um ser humano, como foi feito pela IBM ao lançar o programa jogador de xadrez *Deep Blue*;

- medicina: através de sistemas especialistas é possível fazer diagnósticos cuja qualidade fica próxima à de um especialista na área.

- robótica: sempre foi de interesse do homem desenvolver máquinas à sua semelhança. Atualmente elas não chegaram neste nível ainda, mas já executam tarefas simples como varrer um casa e desviar-se de obstáculos;

- compreensão da linguagem: ainda não foi construído um

programa capaz de compreender perfeitamente a linguagem humana, devido a sua ambiguidade e contextualidade, mas existem programas que traduzem com certa fidelidade alguns textos de uma língua para outra.

2.4 Definição

Computadores conseguem fazer cálculos a velocidades infinitamente maiores que a de um ser humano, e devido a sua grande capacidade de armazenamento, essas máquinas podem ser muito mais “inteligentes” que qualquer humano em certas tarefas como jogar xadrez ou calcular rotas através de um grafo de distâncias.

Todavia, não podemos dizer que tais sistemas possam ser inteligentes, e sim “inteligentes até certo ponto”, pelo fato de os mesmos utilizarem certos mecanismos intelectuais em detrimento de outros. Em um jogo de xadrez, por exemplo, um computador analisa milhões de possíveis caminhos de um grafo de jogadas factíveis, enquanto que um homem analisaria cerca de duas ou três delas em um mesmo intervalo de tempo. A razão pela qual um homem consegue realizar uma mesma tarefa que a máquina mas com uma quantidade tão pequena de computação reside na capacidade de entendimento do problema que os seres humanos possuem (McCarthy, 2004). Se algum dia for possível desenvolver uma máquina capaz de jogar xadrez analisando o tabuleiro da mesma forma como um ser humano o faz, seria possível desenvolver um programa tão inteligente quanto o *Deep Blue*, que derrotou o campeão *Garry Kasparov* em 1997, mas com um poder computacional disponível décadas atrás.

Alguns autores consideram que o grau de inteligência está fortemente ligado à complexidade do meio vivido pelo ser, e quanto mais complexo é o ambiente, mais inteligente serão os indivíduos que vivem nele, pois estes deverão sobreviver em um ambiente em constante evolução. Segundo estes autores, inteligência é “saber sobreviver”. Esta visão é difundida na abordagem dos agentes, que será vista mais adiante.

Muitos entram em contradição ao encontrar uma definiçao do que é realmente a inteligência. Segundo Russel (2004), existem dois ramos de pensamento em relação ao significado de IA: o primeiro mede a inteligência artificial em termos de fidelidade ao desempenho humano, e a segunda, em termos de um conceito ideal de inteligência, denominado racionalidade. Um sistema é dito racional quando ele age de forma “correta”, segundo sua base de dados. Ainda segundo o autor, para cada ramo, existem duas maneiras de se medir a inteligência do sistema, que é analisar se ele “age” ou “pensa” humanamente, somando se a um total de quatro abordagens, que serão discutidas a seguir. Devemos notar que nenhuma das abordagens escapa da comparação entre a máquina e um ser humano, pois a inteligência humana sempre foi considerada o modelo ideal de inteligência.

2.4.1 Sistemas que agem como humanos

Como a maioria das definições sobre o que é ou quais os requisitos necessários para se ter um ser inteligente recaem na subjetiva definição da própria inteligência, Alan Turing (1950), também conhecido como o pai da computação por suas contribuições à teoria da computabilidade, propôs um teste empírico para avaliar a inteligência de um agente computacional. O teste consiste em colocar uma máquina e um humano em salas separadas entre si, além de um interrogador, também humano, mas que não tem nenhum contato físico ou visual com os dois participantes (homem e máquina).

O interrogador não sabe qual dos participantes é humano nem tampouco qual deles é a máquina e seu objetivo é distinguir quem é humano e quem é a máquina utilizando-se apenas de um dispositivo que utiliza a escrita como forma de comunicação, enviando perguntas e recebendo respostas dos participantes. O uso deste terminal é muito importante para o teste, já que elimina quaisquer diferenças físicas entre humano e máquina, como habilidades mecânicas ou uso da voz, embora esses traços possam ser considerados características de seres inteligentes. A máquina é aprovada no teste, ou seja, é dotada de inteligência caso o interrogador, após fazer determinadas perguntas, não conseguir distinguir o

humano da máquina.

Um computador capaz de passar no Teste de Turing, possuiria as seguintes características:

- **processamento de linguagem natural**: para que seja capaz de comunicar com o interrogador. A compreensão da fala humana vai muito além de procurar o significado das palavras em um dicionário, recaindo na solução de ambigüidades da fala e na análise de todo o contexto do discurso;

- **representação de conhecimento**: para armazenar informações recebidas é necessário desenvolver uma forma de abstração e armazenamento do conhecimento;

- **raciocínio automático**: usa sua base de dados para fazer inferências e chegar à novas conclusões;

- **aprendizado de máquina**: um sistema computacional, diferentemente de um ser humano, irá efetuar toda a computação feita na resolução de um problema, não importa quantas vezes este mesmo problema lhe foi apresentado. Para fazê-lo adaptar-se à novas situações, além de detectar e transceder padrões é necessário embutir alguma forma de aprendizado nesses sistemas.

2.4.2 Sistemas que pensam como humanos

O homem só conseguiu criar o avião depois que parou de tentar criar máquinas que imitam o vôo dos pássaros e passou a focar nos estudos de aerodinâmica. É dentro desta filosofica que se baseia a Ciência Cognitiva, uma área interdisciplinar que une IA com a psicologia. Segundo esta, somente tendo conhecimento suficiente sobre nossas mentes, se torna possível implementá-las em um programa de computador.

2.4.3 Sistemas que agem racionalmente

> "[..]definimos um agente como um elemento de uma sociedade que pode perceber aspectos (freqüentemente limitados) de seu ambiente e afetá-lo, quer diretamente ou através da cooperação com outros agentes." (LUGER, 2004).

Agente vem do Latim *agere*, que significa "fazer", e é simplesmente algo que "age". Eles diferem de outros programas pelo fato de serem autônomos (completa ou parcialmente), sendo que cada agente contribui com a solução geral do problema, podendo inclusive interagir com outros agentes.

Este tipo de abordagem visualiza a inteligência como parte da cultura de uma sociedade, e que as interações entre seus agentes produz inteligência. Para que essas interações sejam possíveis, cada agente deve possuir uma habilidade sensorial (sendo apto a perceber o que está à sua volta), ser capaz de se adaptar a mudanças, e, principalmente, ser orientado a objetivos. Mesmo que um agente não saiba qual é a sua verdadeira função no sistema como um todo, o conjunto de soluções (ações) viabilizadas por cada um dos agentes produz um resultado global.

2.4.4 Sistemas que pensam racionalmente

Estes sistemas são fortemente baseados nos estudos de lógica, que se iniciaram desde os primeiros estudos sobre silogismos de Aristóteles, até o *Solucionador Geral de Problemas* de Newel e Simon (1963).

A lógica pode ser facilmente automatizada, pelo fato de ser um sistema formal. Muitos problemas podem ser formulados e resolvidos através desta abordagem, fato que serviu de motivação no desenvolvimento de provadores automáticos de teoremas.

No entanto, estes sistemas enfrentam duas grandes dificuldades:

- às vezes, não é possível formular um problema em notação lógica, pois a base de conhecimentos não é completamente verdadeira;

- embora haja programas que resolvam problemas descritos em notação lógica, pode não haver recursos computacionais suficientes para a resolução destes. Alguns sistemas lógicos podem ter um nível de complexidade suficiente para gerar infinitos teoremas prováveis.

A solução para tais problemas está no uso de técnicas heurísticas. Elas reduzem o tamanho do espaço de busca através de passos que podem ajudar a alcançar a solução desejada, mas que não garantem que esta solução será ótima. Na maioria dos casos, quando não se alcança uma solução ótima, obtém-se uma boa aproximação para a mesma.

2.5 IA nos Jogos

IA no contexto dos jogos implica que os oponentes controlados pelo computador apresentem um certo grau de cognitividade (percepção, esperteza) ao enfrentar o jogador humano. Existem várias maneiras de se implementá-la, mas o mais importante é que eles pareçam apresentar certo grau de inteligência.

As técnicas de IA no contexto de jogos foram usadas muito tempo antes destes fazerem parte do ramo da computação. Inicialmente a IA fora utilizada nas pesquisas em busca em espaço de estados; em jogos como xadrez, damas e o jogo-dos-15[2]. A vantagem do uso de jogos no estudo da IA reside no fato de os mesmos possuirem regras bem definidas e serem fáceis de se praticar.

---

[2] Jogo estilo quebra-cabeça onde quinze peças são embaralhadas e dispostas em uma matriz 4x4. As peças só podem ser movidas em direção ao único espaço em branco, e o objetivo do jogo é restaurar a matriz ao seu estado original.

Deve-se atentar para as diferenças entre a utilização de técnicas acadêmicas de IA e do uso de *cheating*, termo comum entre os jogadores que denomina o uso de trapaça pelo sistema, fazendo com que os jogadores tenham a ilusão de que o sistema é inteligente. De acordo com Buckland (2005), uma pesquisa foi feita entre vários testadores do jogo de tiro *Halo*, e constatou-se que os mesmos atribuíam maior nota à qualidade da IA do programa quando aumentaram-se os pontos de vida dos NPCs[3]; e esta mesma nota foi menor quando também propositalmente estes pontos de vida foram diminuídos, fazendo com que os NPCs ficassem mais fáceis de se destruir.

Outra técnica comum de trapaça é o uso da vantagem de que o sistema conhece todos os caminhos e perigos de um cenário, tendo deste modo uma visão muito mais ampla que a do jogador. Em um labirinto, por exemplo, pode-se fazer com que o inimigo surpreenda o jogador, utilizando algoritmos de busca em caminhos, que serão vistos mais adiante.

O aumento da capacidade computacional trouxe também mais realismo aos jogos. Com o advento dos gráficos em 3D, a IA ganhou ainda mais espaço nesta área. Com ambientes cada vez mais complexos, é de suma importância o uso de algoritmos de IA para evitar que os NPCs atravessem paredes, fiquem presos em cantos do cenário ou ajam de maneira indiferente a estímulos visuais e sonoros produzidos pelo jogador.

Na prática, o propósito da IA em um jogo é controlar cada aspecto do NPC, provendo as seguintes facilidades (CHAMPANDARD, 2003) :

- **Comportamentos primitivos:** capturar itens, apertar botões, usar objetos, gesticular, entre outros;

- **Movimento:** entre determinadas áreas do cenário, desviando-se de obstáculos, plataformas, entre outros;

---

[3] *Non Player Character* (Personagem não controlado pelo jogador, mas pela máquina).

- **Tomada de decisão:** situa-se um nível acima das outras facilidades, computando as ações necessárias para cumprir seus objetivos.

Tais facilidades podem até ser implementadas com a programação tradicional, no entanto, a IA traz soluções mais elegantes, gerando comportamentos mais qualitativos. Isto é possível através da utilização de funcionalidades tais como o reconhecimento de padrões, predição, aproximação e controle motor.

3 CARACTERÍSTICAS DO DESENVOLVIMENTO DE JOGOS

Para garantir o sucesso de um jogo no mercado, é necessário, além de uma boa equipe de desenvolvimento e capital financeiro, um bom planejamento visando a fatia de mercado desejada e o público-alvo que se deseja alcançar. Os aspectos fundamentais que devem ser levados em consideração antes mesmo de se iniciar seu desenvolvimento são mostrados a seguir.

3.1 Categorias

Há uma vasta gama de estilos diferentes de jogos no mercado, e, dependendo do mesmo, seu grau de complexidade no tocante ao desenvolvimento é diferente. Jogos infantis e de plataforma possuem lógica relativamente simples se comparados aos simuladores e RPGs, por exemplo.

Não há uma fórmula específica para se enquadrar cada jogo dentro de uma determinada categoria, visto que a cada ano surgem muitas novidades, e alguns softwares são uma mesclagem de vários estilos diferentes. Os principais estilos vendidos no mercado são:

- **RPG:** do inglês *Role Playing Game*, é um jogo no qual o jogador assume o papel de um personagem tomando todas as suas decisões no desenrolar da estória, que geralmente assume proporções épicas. Como o jogador possui imensas possibilidades de escolha que acarretarão em caminhos diferentes na estória, os RPGs possuem uma complexa base de dados. Exemplos: *Final Fantasy* da Squaresoft e *Diablo* da Blizzard.

- **Aventura:** não deve ser confundido com o RPG. Nesta classe de software, a ênfase é dada à resolução de quebra-cabeças e enigmas ao invés da construção de personagens, batalhas, etc.

Exemplos: *The Dig* e *Full Throttle*, ambas da Lucas Arts.

- **Simulação:** seu foco principal é a física do ambiente em questão, proporcionando o maior grau de realismo possível. Estes jogos buscam elementos gráficos de alta qualidade e jogabilidade fiel à realidade. A programação do software exige muito conhecimento de física e matemática, além de possuir grau de complexidade elevado no que tange seu desenvolvimento. Uma boa IA é fundamental para aumentar o realismo da simulação. Exemplos: *Flight Simulator* da Microsoft (simulador de vôo) e *Gran Turismo* da Polyphony (simulador de carros).

- **Esportes:** simulam esportes como futebol, basquete, vôlei, golfe, entre outros. Assim como os jogos de simulação, têm interfaces 3D bastante complexas e seu desenvolvimento possui as mesmas dificuldades inerentes ao desenvolvimento de um jogo estilo simulação (física, matemática e IA). Exemplo: *FIFA Soccer* da Electronic Arts (simula partidas de Futebol).

- **Plataforma:** é um dos estilos de jogos mais clássicos. Sua lógica é simples se comparado aos simuladores e RPGs: basicamente, o jogador tem seus movimentos limitados ao eixo X (no caso 2D); movendo-se para a direita ou à esquerda e pulando obstáculos. Inicialmente, a maioria dos jogos de plataforma eram 2D, mas devido aos avanços de hardware muitos estão sendo feitos utilizando tecnologias 3D. Exemplos: *Sonic* (SEGA) e *Super Mario Bros*. (Nintendo).

- **Educação/treinamento:** Possuem um foco mais didático e pedagógico do que de entretenimento. São jogos educacionais que podem ser utilizados no aprendizado e/ou treinamento, de acordo com os critérios que se desejam transmitir. Exemplos: Simuladores de vôo para treino de pilotos, jogos de aventura

cujos enigmas envolvem o aprendizado de história, matemática, entre outros.

- **Estratégia:** são jogos que forçam o jogador a utilizar o seu lado intelectual ao invés dos reflexos com o controle. Através de uma análise cuidadosa da situação, o jogador toma decisões críticas que envolvem o planejamento de cidades (*Sim City 3000,* Maxis) ou o comando de exércitos (*WarCraft,* Blizzard) por exemplo. Um algoritmo comumente utilizado em jogos de estratégia é o A* (A-estrela) na busca de caminhos (*pathfinding*), que será vista adiante.

- **Infantil:** seu público alvo são as crianças e enfocam histórias simples e quebra-cabeças com o intuito de educar e divertir a criança. Possuem visual colorido e interface simples. O jogo segue trajetória linear, por isso sua programação é relativamente simples. Exemplo: *Putt Putt Joins the Circus* (Humongous).

- **Tabuleiro:** seguem a linha dos jogos clássicos como xadrez, damas, 8-peças, entre outros.

3.2 Público Alvo

Quando se desenvolve um jogo, é necessário definir um público-alvo para o mesmo. Esta questão é de grande relevância, visto que ela pode afetar inclusive o seu faturamento. Dependendo do estilo do enredo, falas ou da inclusão de cenas sugestivas, um jogo pode ser classificado em uma faixa-etária mais elevada, fazendo com que uma menor variedade de consumidores tenham acesso ao produto. Este tipo de classificação etária também varia de um país para outro devido às diferenças culturais entre estes. O maior consumidor de jogos eletrônicos não é o público infantil, mas são as pessoas jovens entre 16 a 25 anos em média.

Logo, é necessário definir antecipadamente à produção, qual será o público-alvo do produto e também se este poderá ser futuramente lançado em outros países, sendo traduzido para outras linguas.

3.3 Plataforma

Os jogos eletrônicos estão disponíveis nas mais diversas plataformas, sendo os computadores pessoais, consoles e os *arcades*[4] as plataformas mais comuns. Atualmente, tem havido uma grande expansão na área de jogos para celulares, devido à grande base de usuários em todo o mundo –que tende a ultrapassar a quantidade de dispositivos das outras plataformas. Geralmente, um jogo é lançado para uma plataforma-alvo e então adaptado posteriormente para as demais.

3.4 Área de Atuação

A área de atuação no mercado de jogos é ampla e pode variar em cada companhia ou tipo de jogo. Além disso, novos empregos surgem à medida que emergem novas tecnologias (tanto de *software* como *hardware*).

Apesar dessas disparidades, a maioria dos estúdios de desenvolvimento são formados por quatro equipes principais: *design*, artística, programação e testes. A equipe de *design* cuida dos conceitos básicos do funcionamento do jogo. A equipe artística cria imagens e compõe músicas e outros efeitos sonoros. A de programação planeja e programa o código para o funcionamento do *software*. Já a equipe de testes verifica quais são os erros de código (*bugs*) do jogo antes deste ser publicado.

---

[4] Também conhecidos como “fliperamas”.

4 ALGORITMOS DE IA PARA JOGOS

As técnicas de IA para jogos podem ser divididas em duas principais categorias, que são denominadas “determinísticas” e “não-determinísticas”:

- **determinística:** tem comportamento previsível, consome poucos recursos da máquina e é mais fácil de ser implementada. Um exemplo de comportamento determinístico é o algoritmo básico de perseguição (cap. 4.1.1). O problema desta abordagem é que se faz necessário prever todas as ações possíveis em um determinado momento, fazendo-se uso de muitas regras do tipo “se-então”; além disso, após pouco tempo de jogo, o jogador consegue prever facilmente o comportamento da máquina, pois, para um mesmo conjunto de entradas, as mesmas respostas são atribuídas;

- **não-determinística:** possui um grau de incerteza, que pode variar a cada implementação; um exemplo de comportamento não determinístico inclui o aprendizado que um personagem da máquina adquire ao jogar contra um humano após um determinado período. Tal aprendizado pode ser implementado por uma rede neural ou algoritmo genético, por exemplo. É possível fazer com que a máquina possa inclusive exibir comportamentos que vão além do que lhe foi programado –mas há um porém, pois, em alguns casos, a máquina pode exibir um comportamento não desejado. Outra dificuldade se encontra nos testes, pois fica mais difícil testar todas as combinações de comportamento que a máquina poderá exibir durante a execução do jogo.

4.1 Perseguição e Fuga

A locomoção é uma das premissas básicas de um agente. Tal característica é fundamental para que este consiga atingir seus objetivos. Um algoritmo de perseguição e fuga pode ser divido em duas partes (BOURG e SEEMAN, 2004): uma, é a tomada de decisão para que se inicie o processo, e outra é a ação propriamente dita. Pode-se dizer também que existe um terceiro processo a ser considerado, que é a evasão de possíveis obstáculos que possam estar no caminho encontrado pelo algoritmo.

Um algoritmo de busca simples envolve o decremento das distâncias entre as coordenadas do caçador e da presa.

4.1.1 Algoritmo básico de perseguição:

Código 1: algoritmo básico de perseguição

```
se (caçadorX > caçaX)
     caçadorX--;
senão se (caçadorX < caçaX)
     caçadorX++;

se (caçadorY > presaY)
     caçadorY--;
senão se (caçadorY < presaY)
     caçadorY++;
```

Pra transformar o algoritmo de perseguição em um algoritmo de evasão, basta fazer exatamente o oposto, invertendo as coordenadas da presa de modo que sua distância aumente em relação à caça.

4.1.2 Exemplo de um algoritmo básico de evasão:

Código 2: algoritmo básico de evação

```
se (presaX > caçadorX)
     presaX++;
senão se (presaX < caçadorX)
     presaX--;
se (presaY > caçadorY)
     presaY++;
senão se (presaY < caçadorY)
     presaY--;
```

Ambos os algoritmos anteriores não consideram a situação atual do cenário, ou seja, não levam em conta os obstáculos que possam estar no caminho do movimento entre o caçador e a presa.

Outro revés, é que o algoritmo não apresenta um movimento suave, pois o perseguidor sempre começa com um movimento na diagonal em uma coordenada, e depois termina fazendo um movimento reto na outra coordenada, o que não aparenta ser um movimento natural. Vide figura a seguir:

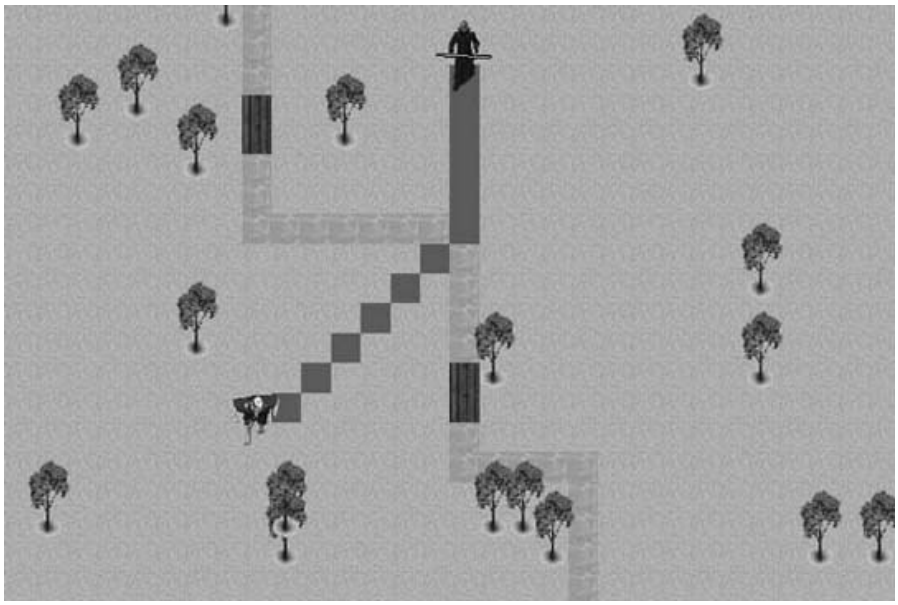

Figura 1 - uma melhora é alcançada utilizando a "perseguição com linha de visão" (*line-of-sight chasing*) (BOURG e SEEMAN, 2004)

4.1.3 Perseguição com linha de visão

Na perseguição com linha de visão, o predador move-se em direção à presa através de uma linha reta traçada entre as coordenadas deste e aquele. Se a presa encontra-se inerte, o caminho será uma reta; e, se a presa encontrar-se em movimento, o caminho a ser seguido pelo predador será formado por curvas –pois, a cada iteração do laço do jogo, o predador traçará uma reta entre a sua posição e a posição da presa, e, como esta encontra-se em alteração, esta reta ganhará nova direção a cada iteração.

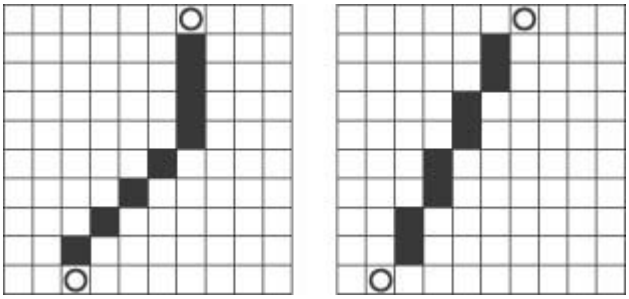

Figura 2 – Perseguição simples (esq) e com linha de visão (dir)

Um algoritmo interessante na implementação da perseguição com linha de visão é o algoritmo de Bresenham, que é muito utilizado na computação gráfica para o desenho de retas. A razão pela qual o algoritmo de Bresenham é útil em perseguição e fuga é o fato de que ele nunca desenha dois *pixels*[5] adjacentes no menor eixo da reta, o que implica que ele sempre encontrará o caminho mais curto entre ambos os pontos.

Observe que, devido à sua natureza, o algoritmo de Bresenham só funciona em jogos que usam *tiles*, ou seja, onde o cenário é formado por ladrilhos dispostos em linhas e colunas. Uma versão do algoritmo de linha de visão será mostrada mais tarde.

---

[5] Pixel: menor unidade que compõe uma imagem em um monitor ou tela de televisão.

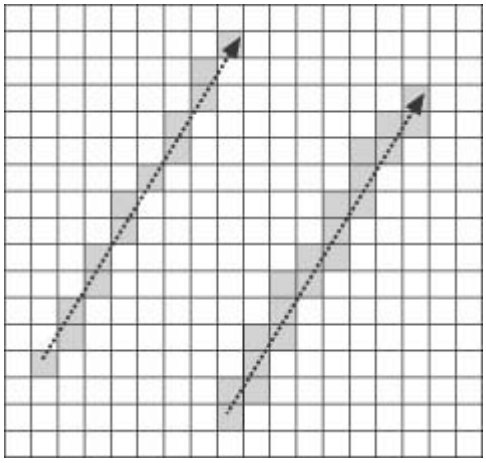

Figura 3 - o algoritmo de Bresenham (esq.) nunca desenha dois pixels adjacentes no menor eixo da reta

Consulte o ANEXO A para ver o algoritmo de Bresenham na íntegra.

4.1.4 Linha de visão para ambientes contínuos

O algoritmo de Bresenham é bastante satisfatório para cenários compostos por *tiles*[6]. Para ambientes contínuos, nos quais os agentes (tanto os controlados pelo computador quanto pelo jogador) são submetidos à forças físicas como aceleração, velocidade e direção, existe um algoritmo de perseguição e fuga bastante eficiente, que utiliza as técnicas de linha de visão. Tendo conhecimento das coordenadas da presa, o predador, a cada iteração do laço de jogo, ajusta a direção de seu movimento para que este se dirija de encontro à presa.

Após o cálculo da direção para qual o agente deverá seguir, a mecânica do jogo fará o tratamento de tal locomoção, que deve ser limitada por uma velocidade máxima e também por uma taxa de mudança de direção máxima. Esta última não deve ser muito alta para que o agente não tenha uma trajetória muito brusca.

Código 3: algoritmo de linha de visão para ambientes contínuos

```
void    DoLineOfSightChase (void)
{
     Vector   u;
     bool     left = false;
     bool     right = false;

     u = VRotate2D(-Predator.fOrientation,
        (Prey.vPosition - Predator.vPosition));
     u.Normalize();

     if (u.x <  -_TOL)
          left = true;
     else if (u.x > _TOL)
          right = true;

     Predator.SetThrusters(left, right); //ajusta a direção
}
```

[6] Tile: do inglês “ladrilho”. É uma unidade que compõe um cenário em ambientes 2D. Cada cenário é composto de uma matriz de *tiles*.

As variáveis *left* e *right* indicarão quais forças de direção o agente deve tomar. O código “Prey.vPosition - Predator.vPosition” calcula o vetor distância relativa entre o predador e a presa em coordenadas globais, que então é passada à função “VRotate2D”, que converte estas coordenadas globais para as coordenadas locais do predador. Coordenadas globais são as coordenadas fixas (imutáveis) do sistema. Já as coordenadas locais de um agente são mutáveis e rotacionam juntamente com o objeto ao qual estão aderidas.

Figura 4 - sistema de coordenadas em linha de visão para ambientes contínuos

É possível transformar coordenadas globais (X, Y) em coordenadas locais (x, y) em termos da orientação do agente em relação ao sistema de coordenadas globais (teta) com a seguinte fórmula:

$$x = X\cos(\theta) + Y\sin(\theta)$$
$$y = -X\sin(\theta) + Y\cos(\theta)$$

Equação 1: transformando coordenadas globais em locais

Agora o vetor “u” armazena a direção do predador à presa. Assim sendo, “u” é então normalizado para que ele contenha um valor unitário. Com este valor, podemos verificar se a presa encontra-se à esquerda, à direita, ou logo à frente do predador, de acordo com suas coordenadas locais. No primeiro caso, *left* recebe *true*, no segundo, *right* recebe true, e, no terceiro, nada precisa ser feito, pois não é necessário mudar de direção.

4.1.5 Interceptação

Os algoritmos vistos até aqui fazem com que, em dado momento, o predador sempre vá em direção à presa. Porém, esta solução nem sempre é a mais otimizada em termos de espaço percorrido e tempo gasto. O algoritmo de linha de visão na maioria das vezes faz com que o predador fique atrás da presa, ou, caso ele seja mais rápido, ele acabará ultrapassando-a.

Pode-se imaginar que o problema da interceptação se resume a encontrar o caminho perpendicular mais curto ao longo da trajetória da presa. Mas isto poderia fazer com que o predador alcançasse este ponto muito rápidamente -e ele teria que esperar pela presa (dando-lhe a oportunidade de fugir). Isto ocorre quando não se leva em consideração a velocidade relativa entre os dois corpos.

Para que o predador possa prever corretamente o ponto de encontro com a presa, além de suas posições relativas, é necessário que o predador tenha também o conhecimento de suas velocidades relativas:

$$Vr = Vpresa - Vpredador$$
$$Sr = Spresa - Spredador$$

Equação 2: cálculo da velocidade e posição relativa

Com estas duas informações, é possível calcular o tempo de abordagem (T_A), que é o tempo médio gasto para percorrer uma distância igual à S_r, à uma velocidade V_r:

$$T_A = |S_r| / |V_r|$$

Equação 3: cálculo do tempo de abordatem (T_A)

Observe que S_r e V_r são vetores, portanto T_A é a magnitude de S_r dividida pela magnitude de V_r.

Agora é possível prever a posição da presa (S_f) no tempo T_A futuro:

$$S_f = S_{presa} + (V_{presa})(T_A)$$

Equação 4: posição da presa no tempo (T_A) futuro

Agora, o ponto S_f será o ponto para o qual o predador deverá seguir no algoritmo de linha de visão em ambientes contínuos apresentado anteriormente. Modificando aquele algoritmo, temos o algoritmo de interceptação:

Código 4: algoritmo de interceptação

```
void     DoIntercept(void)
{
     Vector   u;
     Bool     left = false;
     Bool     right = false;
     Vector   Vr, Sr, St;
     Double   tc

     Vr = Prey.vVelocity - Predator.vVelocity;
     Sr = Prey.vPosition - Predator.vPosition;
     Ta = Sr.Magnitude() / Vr.Magnitude();
     Sf = Prey.vPosition + (Prey.vVelocity * Ta);
     u = VRotate2D(-Predator.fOrientation,(Sf - Predator.vPosition));

     //daqui em diante, permanece o mesmo código do algoritmo anterior
     u.Normalize();
     if (u.x <  -_TOL)
          left = true;
     else if (u.x > _TOL)
          right = true;

     Predator.SetThrusters(left, right); //ajusta a direção
}
```

Uma grande vantagem do algoritmo de interceptação é que o predador não precisa necessariamente ser mais rápido que a presa. No algoritmo de linha de visão, o predador na maioria dos casos acaba ficando atrás da presa; tendo que alcançá-la. Com a interceptação, é suficiente que o predador seja rápido apenas o suficiente para alcançar o ponto de interceptação dentro do tempo previsto; caso contrário, a interceptação não será possível.

Para conseguir um maior nível de inteligência, pode-se combinar os algoritmos de perseguição e fuga com outros, que lidam com outras situações do problema –como o desvio de obstáculos, e a tomada de decisão quanto a efetuar ou não a perseguição (fuga).

4.2 Sistemas Baseados em Regras

Sistemas baseados em regras são muito utilizados, tanto para aplicações da vida real quanto para jogos. São sistemas que consistem de inúmeras regras do tipo “se-então”, que serão aplicadas a um conjunto de entradas. De acordo com o conjunto de entrada, a parte “então” da regra define qual ação será tomada.

Em IA, estes sistemas também são chamados de “Sistemas Especialistas”, e são utilizados para substituir um especialista humano em uma determinada área do conhecimento, como o diagnóstico de doenças, classificação de doenças em determinada cultura, diagnóstico de erros de engenharia, entre outras. Uma grande vantagem dos sitemas especialistas é que eles imitam a maneira como o ser humano tende a pensar e raciocinar, dado um conjunto de fatos e seu próprio conhecimento sobre um domínio de conhecimento (BOURG e SEEMAN, 2004).

Outra vantagem em usar sistemas especialistas é que devido à sua natureza modular, sua implementação é fácil, e o conjunto de regras pode ser disposto em qualquer ordem.

4.2.1 Elementos de um sistema baseado em regras

Todo sistema especialista ou baseado em regras deve possuir os seguintes elementos:

- memória de trabalho (*working memory*): guarda os fatos já sabidos e também as asserções feitas pela aplicações das regras;

- regras: conjunto de regras no estilo “se-então” que operam sobre os fatos gravados na memória de trabalho. Quando uma

regra é aplicada, ela dispara a ação vinculada à sua parte "então", ou faz com que o sistema mude de estado, como numa máquina de estado finito.

- interpretador: também conhecido como "motor de inferências", é a parte do sistema responsável pela escolha e aplicação de determinada regra a partir de seu estado atual. O interpretador utiliza duas maneiras para fazer inferências: o *forward e* o *backward chaining*.

4.2.2 *Forward chaining*

O sistema *forward chaining* (encadeamento progressivo) é o mais comum dos sistemas de inferência. Nele, o sistema inicia com um conjunto de fatos, e a partir destes, aplica as regras repetidamente até que o resultado desejado seja alcançado. O método é dividido em três etapas:

1. *matching*: nesta fase, o sistema procura identificar todas regras que se aplicam ao conjunto de entrada, de acordo com seu estado atual;

2. resolução de conflitos: na primeira etapa, mais de uma regra pode satisfazer o conjunto de entrada atual. Para resolver este conflito, existem várias abordagens, como por exemplo:

    a. escolher sempre a primeira regra aplicável;

    b. escolher uma das regras aleatóriamente;

    c. atribuir pesos para as regras e escolher a de maior peso. Tais pesos podem ser atualizados de acordo com a frequência com que cada regra é escolhida.

3. execução: depois de eliminadas as ambiguidades, a regra

escolhida é executada, ou seja, a sua parte “então”.

4.2.3 *Backward chaining*

O *backward chaining*, ou encadeamento regressivo, é basicamente o contrário do *forward chaining.* A partir de um objetivo, o sistema verifica quais regras poderíam ser aplicadas para que o mesmo seja atingido. É por este motivo que o *backward chaining* é dito ser um algoritmo dirijido ao objetivo (*goal driven*). Devido à sua natureza recursiva, o *backward chaining* não é amplamente utilizado, por questões computacionais.

4.2.4 Aplicação em jogos

Uma implementação pode conter os dois tipos de algoritmo. Em um jogo de estratégia, por exemplo, considere o pequeno trecho de um sistema de regras:

If (templo=true) then paladino=true

If (templo=true and trincheira=true) then soldado=true

O trecho de regras acima indica quais unidades o jogador pode construir se possuir determinada construção. Por exemplo, se o jogador possuir um templo, então ele está apto a construir paladinos.

Sabendo que o jogador possui um templo, utilizando *forward chaining*, o computador pode presumir que o jogador tem grandes chances de possuir paladinos, desta forma, ele poderá preparar sua defesa de modo que consiga superar um possível ataque de paladinos.

O contrário também é possivel -descobrindo que o jogador possui soldados, o computador, utilizando *backward chaining*, pode presumir que o jogador possui trincheiras. Desta forma, o computador pode armar seu exército de modo

que tenha força suficiente para superar a trincheira.

4.3 Lógica Nebulosa (*fuzzy*)

Na lógica *fuzzy*, ou nebulosa, é possível mapear problemas discretos ou booleanos da mesma maneira como seres humanos o fazem. Por exemplo, ao invés de dizermos que um objeto está à uma distância de 0,5, 1 ou 10 metros, podemos dizer que o mesmo está “muito perto”, “perto” ou “longe”.

Na lógica booleana tradicional, existiriam limites discretos para “muito perto”, “perto” e “longe”. E tais limites são completamente disjuntos. Já na lógica *fuzzy*, todo o problema pode ser mapeado em vários níveis, ou seja, áreas nebulosas, permitindo uma certa sobreposição entre os limites.

É pelo fato de não ser determinística, que a lógica nebulosa tem sido bastante difundida em aplicações da vida cotidiana. Um ar condicionado, por exemplo, ao invés de ligar e desligar seu termostado constantemente quando a temperatura ambiente se desvinculasse da temperatura estipulada, causando danos ao aparelho, poderia simplesmente ajustar sua potência de modo que a temperatura ficasse próxima à ideal. Utilizando lógica nebulosa, também sería possível controlar o sistema de freios de um metrô, fazendo com que este parasse de maneira tão suave que seus passageiros teríam a mínima sensação de desconforto.

4.3.1 Lógica nebulosa nos jogos

Pode-se trabalhar com lógica nebulosa nos jogos de maneira parecida como é construído um sistema baseado em regras, ou uma máquina de estados finitos. Contudo, a lógica nebulosa permite trabalharmos com variáveis mais próximas à linguagem humana -ao invés de trabalharmos com valores discretos, trabalhamos com variáveis do tipo “perto”, “longe”, ou “quente”, “muito quente”, por exemplo.

Veja a seguir um exemplo de uso de lógica nebulosa em um jogo.

Neste, o computador toma decisões de acordo com o estado atual do jogador, que é medido segundo a lógica nebulosa:

Se (muito_forte) then fugir;

Se (forte) then procurar_arma;

Se (fraco) then atacar;

Note que “muito_forte”, “forte” e “fraco” são funções *fuzzy*, cujo cálculo será visto adiante. É relevante salientar que, se fosse utilizada a lógica tradicional, teríamos de estabelecer inúmeros limites (regras) para definir qual é o significado de cada uma dessas funções, enquanto que na lógica nebulosa, é possível fazê-lo com um número bem menor de regras.

4.3.2 Características

O processo *fuzzy* é dividido em três etapas. Entrada (fuzzificação), saída (saída fuzzy), e quantização (defuzzificação).

4.3.3 Fuzzificação

Na fuzzificação, os dados (números) são mapeados em dados nebulosos. Este processo de mapeamento involve encontrar o grau de associatividade entre os dados de entrada e os conjuntos *fuzzy* pré-definidos. No exemplo acima, isto significaria descobrir, segundo os dados numéricos de entrada, em qual dos grupos se encontra a força do jogador (muito forte, forte ou fraco). Isto é feito através de funções de associatividade.

4.3.4 Funções de associatividade

Praticamente qualquer função pode ser usada como função de associtatividade, isto vai depender da facilidade de implementação, e também da precisão que se deseja alcançar como resultado. O seu valor resultante irá dizer, no nosso exemplo, o quão forte está o jogador que o computador irá enfrentar. Este valor vai de 0 (zero, fraco) a 1 (um, forte). Veja a função de associatividade abaixo:

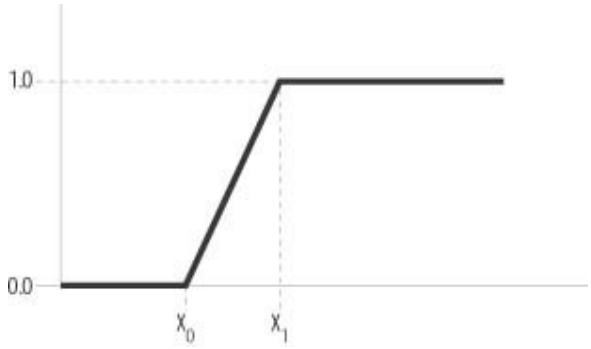

Figura 5 – exemplo de função de associatividade (gráfico)

Nesta, valores abaixo de x_0 indicam valor fraco, e valores acima de x_1 indicam valor muito forte. Um valor entre esses dois limites indica um certo grau de força. A formula para esta função é mostrada a seguir:

$$f(x) = \begin{cases} 0; x \leq x_0 \\ \dfrac{x}{x_1 - x_0} - \dfrac{x_0}{x_1 - x_0}; x_0 < x < x_1 \\ 1; x \geq x_1 \end{cases}$$

Equação 5: função de associatividade (fórmula)

Geralmente é utilizada uma função de associatividade para cada grupo de valores que se deseja avaliar. Veja o exemplo abaixo:

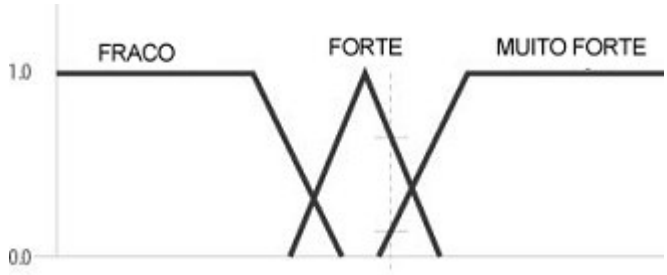

Figura 6 – múltiplas funções de associatividade

Neste caso, há três funções de associatividade. Note que existem inúmeras dessas funções, como a trapezoidal, e a triangular, cuja implementação foge ao escopo deste trabalho. Maiores detalhes sobre tais funções podem ser encontrados em (MCNEIL e THRO, 1994). O importante é salientar que, tendo as três funções de associatividade, basta usar os valores de entrada como parâmetro em cada uma delas para obter cada um dos graus de associatividade.

Portanto, para uma dada entrada, podemos encontrar um valor de 0 para fraco, 0,75 para forte e 0,15 para muito forte. Daí podemos inferir que o jogador é substancialmente forte.

4.3.5 Saída fuzzy

Depois de fuzzificar os valores de entrada, se faz necessário produzir uma saída, que será a ação tomada pelo computador naquele momento. Isto é feito através da avaliação de regras, de maneira semelhante àquela feita nos sistemas baseados em regras. A diferença aqui é que todas as regras são avaliadas e produzem uma resposta cuja intensidade é expressa segundo a lógica *fuzzy*. Os operadores lógicos *fuzzy* são os seguintes:

Disjunção: VERDADE(A OU B) = MAX(VERDADE(A), VERDADE(B))

Conjunção: VERDADE(A E B) = MIN (VERDADE (A), VERDADE(B))

Negação: VERDADE(NÃO A) = 1 – VERDADE(A)

Lembremos que na lógica *fuzzy*, o valor “VERDADE” é um número real, que varia de 0 a 1. Retomando nosso exemplo, poderíamos incrementá-lo adicionando novas variáveis *fuzzy*, produzindo o seguinte conjunto de regras:

- Se (inimigo.muito_forte E computador.desarmado) then fugir;
- Se (inimigo.forte E desarmado) then procurar_arma;
- Se (inimigo.fraco OU inimigo.desarmado) then atacar.

Este é apenas um exemplo. Inúmeras outras regras poderíam ser adicionadas para lidar com todas as possibilidades. O resultado da avaliação destas regras podería ser o seguinte:

- fugir com intensidade 0.5;
- procurar arma com intensidade 0.2;
- atacar com intensidade 0.76.

A intensidade associada a cada ação representa a força de cada regra. Uma maneira de escolher qual será decisão tomada é escolher a de maior peso (neste caso, atacar). Note que os pesos acima não indicam a intensidade com que será feita a fuga, procura da arma ou ataque, e sim o peso de cada decisão. O cálculo intensidade com que a ação deverá ser executada, quando necessária, será calculada na fase de desfuzzificação.

4.3.6 Desfuzzificação

A desfuzzificação é necessária quando se deseja um saída quantitativa, ou seja, um número como saída do sistema nebuloso. Em um caso de perseguição e fuga, por exemplo, um sistema *fuzzy* podería indicar ao caçador um

valor quantitativo de quanto ele deveria mudar de direção à esquerda ou à direita para interceptar a presa. Exemplos de funções de defuzzificação podem ser encontrados em (MCNEIL e THRO, 1994).

4.4 *Scripts*

Scripts são linguagens de programação bastante simplificadas, escritas especificamente para lidar com tarefas específicas dentro de um jogo, como controle de fluxo de diálogos, controle de ações, entre outras.

Os *scripts* ficam em arquivos separados do código principal do programa, na forma de arquivos de texto simples. Para dificultar a alteração desses arquivos de *script*, alguns desenvolvedores criam compiladores para a linguagem, armazenando os mesmos na sua forma compilada -de modo que sua leitura seja irreconhecível ao usuário. Isto evita que os mesmos sejam alterados com intenções maliciosas.

Uma grande vantagem da utilização de *scripts* é que, conforme o tamanho do projeto aumenta, a alteração de constantes dentro do código fica cada vez mais difícil, já que o tempo de compilação é bastante demorado. Armazenando essas constantes na forma de *scripts*, o código em linguagem de programação fica separado do código referente às regras do jogo, e isto permite ainda que pessoas leigas em programação possam ser incumbidas de tal tarefa. Mas isso não é tudo:

> “Uma linguagem script mais avançada aumenta a interação entre o script e o executável, permitindo não apenas inicializar variáveis mas criar a lógica do jogo e até mesmo objetos do jogo, tudo a partir de um ou mais arquivos de script.” (BUCKLAND, 2005) (trad.nossa).

4.4.1 Construção

Para que os *scripts* funcionem de maneira independente do programa executável, é necessário que este último tenha consigo um interpretador compatível com a sua linguagem. Este interpretador irá carregar o arquivo de *script* para inicializar as variáveis dentro do jogo, como no exemplo abaixo:

Código 5: exemplo de arquivo *script*

```
FORCA = 30;
AGILIDADE=10;
VIDAS=2;
```

O exemplo acima é um pequeno arquivo de *script* que especifica os atributos de um personagem controlado pelo computador. Veja o exemplo abaixo:

Código 6: exemplo de *script* controlando a lógica do computador

```
Se (Jogador.armado=TRUE) então  Fuja();
Senão  Ataque();
```

Repare como a linguagem *script* pode ser utilizada para controlar aspectos lógicos do computador. Este exemplo utiliza uma linguagem de alto nível, mais próxima à linguagem natural. *Scripts* como este requerem maior tempo gasto no desenvolvimento de interpretadores e compiladores para a linguagem, mas permitem a criação de "mods".

Mods são extensões de um jogo criadas através de *scripts* pelos próprios jogadores. Através de sua linguagem, o desenvolvedor pode permitir quais aspectos do jogo o jogador poderá alterar, ou criar, dando a ele a oportunidade de criar novos mundos, armas, inimigos, entre outros. Isto tem sido uma tendência em jogos recentes para aumentar o tempo de duração do jogo e alavancar as vendas. Um grande exemplo de utilização de mods é o jogo *Unreal Tournament* 2004, da Epic Games.

4.5 Máquinas de Estado Finito

As máquinas de estado finito ou *finite state machines* (FSMs) têm sido há muito tempo a principal escolha na hora de implementar a IA de agentes para jogos. Sua principal idéia consiste em dividir o comportamento do agente em vários estados, com transições entre eles. A transição de um estado para outro dependerá de fatores de entrada, que podem ser estímulos provocados pelo jogador ou pelo ambiente. Historicamente, um dos mais famosos exemplos de FSM é a Máquina de Turing, que foi proposta por Alan Turing, em 1936.

Existem vários motivos pelos quais o uso de FSMs tenha sobrevivido até hoje, entre eles podemos citar:

- facilidade de compreensão – pois é natural para nós humanos pensarmos nos objetos como estando em um estado ou outro;
- são fáceis de serem implementados;
- facilidade de teste – como possuem um número finito de estados, é possível traçar a rota de eventos que causaram o mal-funcionamento do agente defeituoso;
- são rápidos – há pouca sobregarga de processamento, pois todas as regras são implementadas via programação e são da forma “se-então”;
- são flexiveis – estados e regras adicionais podem ser implementados rapidamente. Além disso, podem ser combinadas com técnicas não-determinísticas da IA, como redes neurais e lógica nebulosa.

4.5.1 Características

Os elementos básicos que constituem uma FSM são:

- estados: representam uma posição no tempo que, consequentemente irá influir no comportamento do agente;
- transições: são ligações entre os estados. Podem ser uni ou bi-direcionais;
- eventos: são ações que ocorrem externamente ao agente;
- condições: são regras que devem ser preenchidas para que ocorra a mudança de estado.

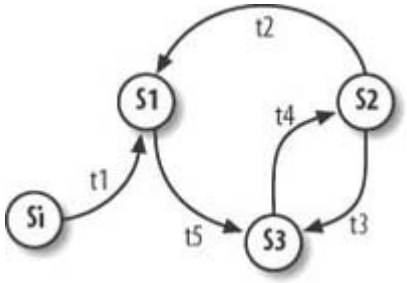

Figura 7 – máquina de estados finitos (s: estados; t:transições)

4.5.2 Implementação

Para implementar uma FSM, é necessário ter uma variável de controle, que indicará o estado atual do agente. O valor inicial desta variável irá determinar o estado inicial. A partir disso, a cada interação do agente com seu ambiente é verificado seu estado atual, e, a partir deste, é verificado se alguma condição de transição é satisfeita. Caso isto ocorra, a mudança do valor da variável de controle indica a mudança de estado. Isto pode ser feito através do uso de condições “se-então” ou através da declaração *switch* das linguagens C e Java.

A adição de estados adicionais pode resultar na dificuldade do entendimento do código. Uma solução para o problema está na utilização de **tabelas de transição de estados**. Tais tabelas são divididas em colunas, que contém a condição e o novo estado (transição), caso a condição seja satisfeita.

Tabela 1 – tabela de transição de dados

Estado Atual	Condição	Novo estado
Fugindo	Seguro	Patrulhando
Atacando	MaisFracoQueInimigo	Fugindo
Patrulhando	Ameacado E MaisForteQueInimigo	Atacando
Patrulhando	Ameacado E MaisFracoQueInimigo	Fugindo

4.5.3 Máquinas de estado finito não-determinísticas

Uma grande desvantagem no uso de FSMs é o fator previsibilidade, que acaba tornando o jogo muito fácil para o jogador humano após algum tempo de jogo. Uma maneira de tornar as ações praticadas pelo agente mais difíceis de se prever é misturar esta com um outra técnica já mencionada neste trabalho: a lógica nebulosa.

Deste modo, podemos tornar a FSM menos determinística, utilizando as técnicas abaixo (BROWNLEE, 2005):

- fuzzificar as condições de transição de estados. Quando for encontrado um conflito, escolher a transição de maior valor *fuzzy*;

- fuzzificar os valores de entrada, representando a força (peso) com que determinado evento tenha ocorrido. O sistema então utilizaria esses dados numéricos, disparando apenas as transições de estado cujos pesos estiverem acima de um limiar.

Outra maneira de tornar o sistema menos determinístico é selecionando os estados de transição aleatóriamente dentro de uma grande gama de estados, criando uma sensação de imprevisibilidade.

4.6 Busca Heurística

A busca heurística é uma técnica antiga da IA e é muito utilizada em jogos de tabuleiro e, recentemente na busca de caminhos (*pathfinding*). Tais jogos são bastante propícios para a implementação dessas técnicas de IA, pois possuem um espaço de estados finito, mas suficientemente grande de modo que a busca exaustiva (busca em largura, busca em profundidade, etc) seja proibitiva, pois não conseguiriam encontrar uma solução em um tempo computacional aceitável.

As heurísticas agilizam a busca de uma solução pois utilizam de informações para escolher os ramos, dentre todos os ramos de uma árvore de espaço de estados, que possuem uma maior probabilidade de levarem a uma solução aceitável para o problema (LUGER, 2004). No entanto, elas possuem um revés, que é a possibilidade de não ser encontrada uma solução ótima, ou então nem mesmo encontrar-se uma solução. Isto pode ser contornado utilizando heurísticas ou algoritmos de busca mais eficientes (Garey e Johnson, 1979 apud LUGER, 2004 pág.134).

Considere o jogo-da-velha. Uma busca exaustiva teria uma quantidade de 9! ou 362880 jogadas. Esta ainda não é uma quantidade tão grande se comparada ao tamanho do espaço de busca de um jogo mais complexo como o xadrez. No entanto, podemos reduzir drasticamente o espaço de busca utilizando uma heurística simples: supondo que estejamos realizando a jogada do jogador “X”, escolhemos sempre a posição que garante a X o “maior número de vitórias” possível.

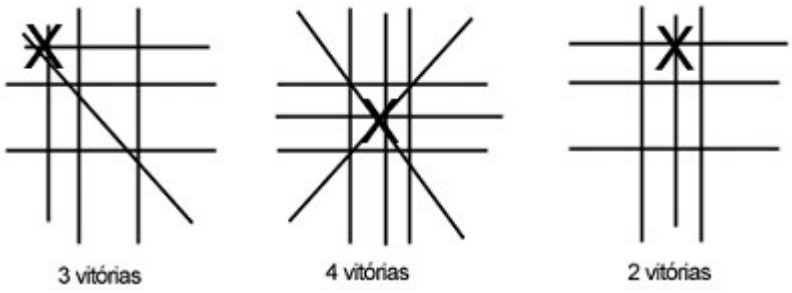

Figura 8 – heurística para o "jogo da velha'

Segundo a heurística do maior número de vitórias, a jogada escolhida dentre as três acima deveria ser a segunda (4 vitórias). O limite superior aproximado para a quantidade de estado utilizando essa técnica é de 72 estados – um número consideravelmente inferior à quantidade inicial de 9! (nove fatorial) estados.

Um algoritmo que utiliza a heurística do maior número de vitórias é a "subida de encosta":

> "As estratégias de subida de encosta expandem o estado corrente da busca e avaliam os seus filhos. O melhor filho é selecionado para expansão futura; nem os seus irmãos nem os seus pais são considerados. A busca pára quando ela alcança um estado que é melhor que qualquer um de seus filhos" (LUGER, 2004).

Um algoritmo heurístico muito utilizado em jogos, principalmente os de estratégia e tiro em primeira pessoa é o A* (lê-se "A estrela"), que será visto adiante quando for abordado o tópico *pathfinding*.

4.7 *Pathfinding*

Pathfinding, ou busca de caminhos é um dos problemas mais comuns acerca do desenvolvimento de jogos, e está presente na maioria dos gêneros. Boa parte das entidades de IA precisam do *pathfinding*: sejam elas tanques, pessoas, veículo ou unidade de combate. E não é apenas para movimentação que se aplica o *pathfinding*; podemos utilizá-lo também para resolver problemas como:

- patrulhamento: consiste na movimentação de uma unidade

através de pontos pré-estabelecidos do cenário e em ordem. Isto faz com que as unidades pareçam mais “vivas” e aumenta as suas chances de encontrar inimigos;

- desvio de obstáculos: requer que a unidade tenha conhecimento do que está ao seu redor, de modo que evite colisões com este;

- perseguição: fazer com que a unidade vá em direção ao seu alvo, como abordado no tópico “Perseguição e Fuga”;

- mirar e atirar: um problema que ocorre quando uma unidade tenta atingir a outra é que obstáculos podem interromper a tragetória. Uma unidade inteligente deverá prever a existência de um obstáculo na rota de colisão do tiro antes de efetuá-lo (ou mesmo mover-se para uma posição aonde o mesmo possa ser efetuado).

Um algoritmo típico de *pathfinding* com desvio de obstáculos em pseudo-código funciona da seguinte maneira:

Código 7: algoritmo de *pathfinding* em pseudo-código

```
Início
  ENQUANTO não chegou ao destino
    ESCOLHA um local próximo rumo ao destino
    SE o local está vazio, mova-se até ele
    SENÃO escolha outra direção segundo estratégia de desvio de obstáculos
  FIM ENQUANTO
Fim
```

4.7.1 Estratégias para desvio de obstáculos

O problema de *pathfinding* sería demasiadamente simples se não houvesse o problema de desvio de obstáculos. Os algoritmos estudados anteriormente no capítulo sobre perseguição e fuga poderíam ser facilmente adaptados para o *pathfinding*. No desvio de obstáculos, as seguintes estratégias

podem ser adotadas:

- escolher uma direção aleatória: se os obstáculos forem pequenos, convexos[7] e distribuídos de maneira dispersa, esta técnica funciona satisfatóriamente e requer pouco processamento se comparado a outras abordagens. Entretanto, para cenários com muitos quartos e passagens estreitas entre eles, o uso desta técnica não é recomendado, pois pode fazer com que o agente fique preso ou demore muito tempo para encontrar o caminho correto;

- contornar o obstáculo: o agente controlado pelo computador seguiria utilizando um algoritmo simples de *pathfinding* até encontrar um obstáculo; ocorrendo isto, o mesmo entraria em um estado "de contorno", seguindo as bordas do obstáculo na tentativa de contorná-lo. O problema aqui consiste em saber quando o algoritmo deve parar de contornar o objeto, caso contrário há o risco de se retornar ao ponto inicial (quando o agente entrou no estado de contorno). Isto pode ser solucionado traçando uma linha reta entre a posição atual e a posição de destino, no momento em que o obstáculo for encontrado; logo, quando esta reta for cruzada, o agente saberá que o obstáculo foi contornado e assim ele retomará o trajeto original utilizando o algoritmo de *pathfinding.*

- calcular o caminho antecipadamente: os métodos anteriores geralmente são satisfatórios, mas não lidam com problemas que incluem regiões com peso, como terrenos cujos custos das rotas são variados (STOUT, 1997); estes problemas são resolvidos com algoritmos de teoria de grafos, como o *Breadth-first search* (BFS), *Depth-first search* e *Dijkstra*.

---

[7] Um poliedro convexo é aquele que, dados quaisquer dois pontos pertencentes a ele, o segmento contendo estes pontos nas extremidades deverá estar inteiramente contido no poliedro.

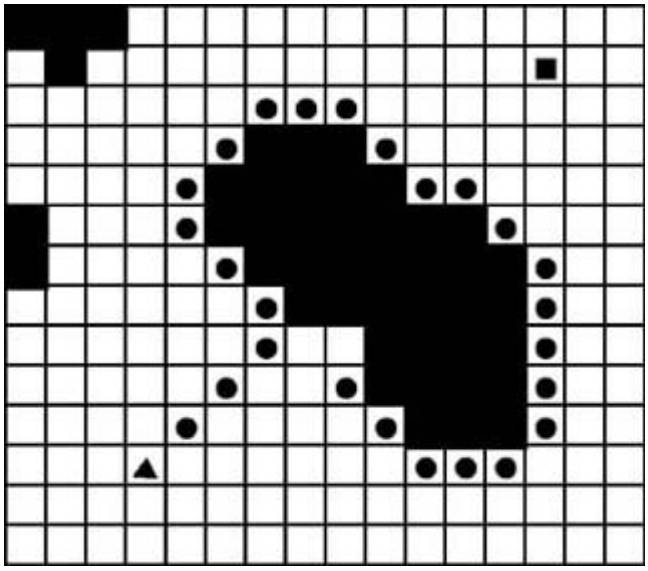

Figura 9 - problema do ciclo no desvio de obstáculos

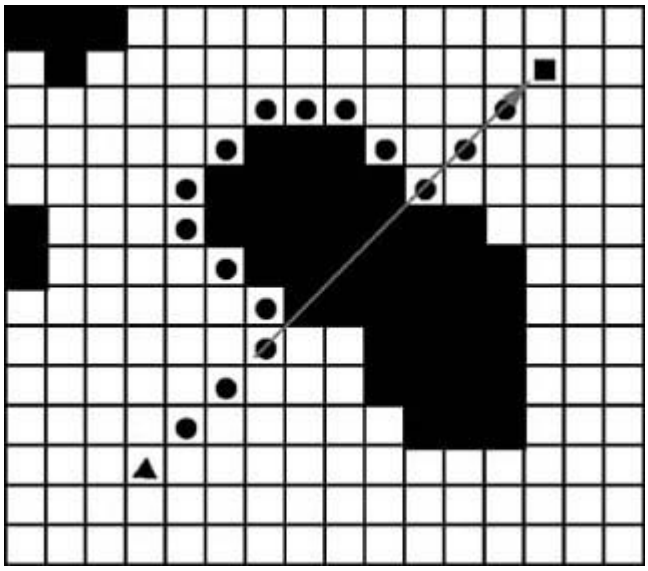

Figura 10 - problema do ciclo resolvido

4.7.2 Algoritmo A*

Este é um algoritmo muito importante, tanto pra IA tradicional quanto para a IA para jogos. Motivo pelo qual um tópico específico foi dedicado a ele. O A* (pronuncia-se “A estrela”) é um algoritmo admissível, ou seja, para qualquer grafo ele encontrará um caminho ótimo entre os estados inicial e final, caso exista um caminho entre esses estados.

Para utilizar o A* em jogos, dividimos o terreno de um cenário ou mapa em pequenos quadrados (ladrilhos) de igual tamanho, e consideramos cada ladrilho como sendo um nó no grafo. A distância total a ser percorrida no *pathfinding* corresponde ao número de ladrilhos percorridos pela unidade do ponto (estado) inicial ao final. Os nós-filho de cada nó são representados pelos ladrilhos adjacentes ao ladrilho em questão.

O A* mantém duas listas: a de nós abertos e a de nós fechados. A primeira armazena todos os ladrilhos que precisam ser verificados, sendo que um deles será escolhido como o próximo nó-destino da entidade segundo uma estratégia de escolha; já a segunda, armazena os nós que não precisam mais serem verificados no momento.

O algoritmo é realizado da seguinte maneira:

1) coloca-se o nó (ladrilho inicial) na lista de abertos;

2) feito isto, colocamos todos os seus nós adjacentes (filhos) também na lista de abertos, ignorando aqueles ladrilhos que representam paredes ou outras áreas do cenário que não são alcançáveis, bem como aqueles que já se incluem na lista de nós fechados;

3) coloca-se o ladrilho inicial na lista de nós fechados;

4) escolhe-se um dos nós da lista de abertos para continuar o algoritmo. O nó atual é feito “pai” do nó escolhido, para que seja possível traçar o caminho a ser percorrido pela entidade; a estratégia para escolher este nó será vista adiante;

5) o processo se repete até que o nó-destino seja adicionado à lista de abertos (neste caso, basta fazer o caminho inverso na árvore de estados para encontrar o caminho, e o algoritmo termina em “sucesso”), ou a lista de abertos fique vazia (indicando falha na busca de um caminho).

O ponto crucial aqui é escolher qual dos nós fará parte do caminho a ser percorrido. Isto é feito utilizando uma função de avaliação F(n). Esta função é resultado da soma da distância percorrida do nó inicial ao nó avaliado, G(n), com uma função heurística, H(n), onde n representa o nó avaliado:

$$F(n) = G(n) + H(n)$$

Equação 6: função de avaliação do A*

H(n) pode ser estimado de várias maneiras, no entanto, deve ser sempre menor ou igual a G(n), “subestimando” o custo real para chegar ao nó-destino. Um exemplo de função heurística é calcular a mesma utilizando uma linha reta entre o nó atual e o nó-destino, ignorando obstáculos intermediários.

Feito isto, escolhemos, dentre os nós abertos, aquele nó que possuir o menor valor F(n). Se o nó escolhido já estiver na lista de abertos, verificamos se o valor de G, utilizando o nó atual para se chegar até ele é menor. Se isto for verdadeiro, fazemos o nó atual ser o pai do nó escolhido e recalculamos os valores F e G para este nó; caso contrário, nada precisa ser feito.

Lester (LESTER, 2003), propõe o uso de algumas melhorias para o algoritmo A* se tornar ainda mais eficiente:

- custo de terreno variável: nem sempre o menor custo significa encontrar a menor distância entre os dois pontos. Certos mapas

possuem certas características como rampas, relevo acidentado, armadilhas, entre outras. Pode-se adaptar o algoritmo acima para este problema adicionando o custo de atravessar tais empecilhos à função custo G de um dado ladrilho;

- caminhos suaves: o caminho resultante do A* é ótimo, mas nem sempre é o mais visualmente aceitável. No cálculo do caminho, pode-se penalizar a escolha de ladrilhos que resultariam numa mudança brusca de direção, adicionando um custo à sua função G;

- lidar com áreas inexploradas: dependendo do jogo, sería muito irrealístico o computador ter conhecimento de toda a topologia do mapa, sabendo exatamente que caminho percorrer em todas as ocasiões; uma solução é manter um vetor de “nós conhecidos” para o computador, ou seja, todos os nós contidos neste vetor já foram explorados por ele, e o resto do mapa é presumido “andável”, até provado o contrário. Com esta abordagem, as unidades irão cometer erros, entrar em becos, até explorarem o espaço ao seu redor. Quando o mapa for explorado em toda sua extensão, o *pathfinding* será executado normalmente;

- outras unidades: o A* considera apenas a topologia do terreno, ignorando outras unidades. Por isso, é necessário adaptá-lo para que detecte a colisão com outros elementos do jogo.

4.8 Redes Neurais

Foi só recentemente que as redes neurais, ou RNs, começaram a ser utilizadas em jogos comerciais. Devido à falta de recursos disponíveis tanto nos PCs quanto nos consoles das gerações passadas, era preferível utilizar técnicas

mais simples e determinísticas na implementação da IA. Mas conforme o tempo foi passando, tanto a quantidade de memória quanto de processamento disponível evoluiu muito, e agora é cada vez mais comum o uso das redes neurais em jogos.

As redes neurais são baseadas no modelo de ligações entre neurônios dos seres vivos, onde que cada neurônio possui três partes básicas: dendrito, corpo celular e axônio. O cérebro possui aproximadamente 100 bilhões desses neurônios, que se comunicam através de sinais eletro-químicos. A troca de informações entre neurônios é denominada “sinapse”, que é a ligação entre o axônio de um neurônio com o dendrito de outro. Quando um neurônio recebe um sinal, ele executa uma espécie de “cálculo”, que, ao ultrapassar um limiar, dispara o neurônio, e este envia um sinal ao próximo neurônio da rede, através de seu axônio.

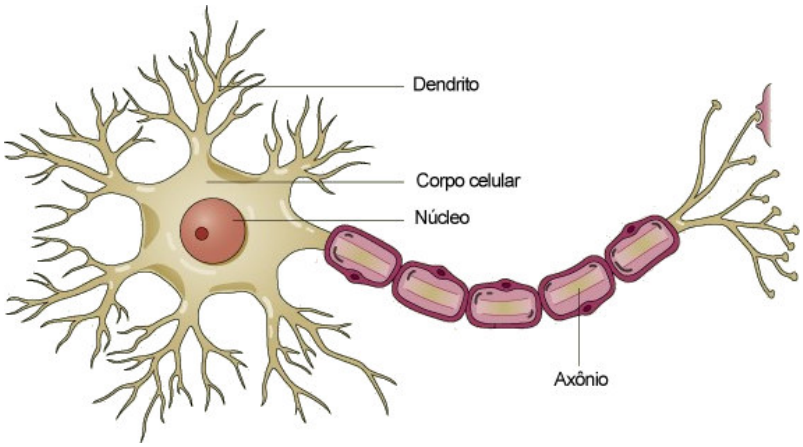

Figura 11 - Estrutura básica de um neurônio

4.8.1 O neurônio artificial

Redes neurais são formadas por vários neurônios artificiais. Um neurônio artificial é modelado à semelhança de um neurônio biológico. Cada neurônio possui um conjunto de entradas, uma função de ativação e uma saída. Cada entrada do neurônio possui um peso associado.

O número de neurônios do qual a rede é formada depende da tarefa que ela irá desempenhar, podendo variar de poucos neurônios (3 ou 4) a milhares deles. Em aplicações da vida real, as RNs são utilizadas no reconhecimento de

padrões, categorização, aprendizado, competição, auto-organização, entre outras áreas.

Uma vez criada a rede, ela deverá ser treinada para que funcione da maneira esperada. Treinar uma rede neural nada mais é que ajustar os pesos, ou seja, a força entre suas conexões. Cada peso está associado a uma entrada do neurônio, e é representado por um número de ponto flutuante negativo ou positivo (exercendo influência excitatória ou inibitória) na entrada. Quando uma entrada é apresentada ao neurônio, ela é multiplicada pelo seu peso. O neurônio então soma todas essas entradas, gerando um valor de ativação. Se este valor for maior que um certo limiar, o neurônio gera um sinal de saída, que dependendo da topologia da rede pode ser a saída desejada, ou uma saída que servirá como entrada para outro neurônio.

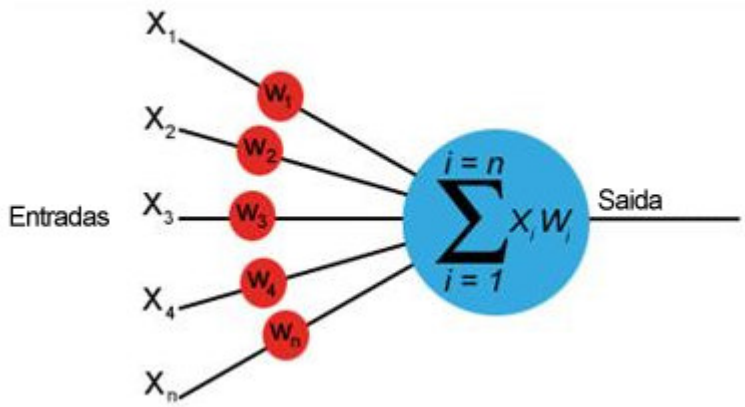

Figura 12 - neurônio artificial

As principais vantagens das redes neurais são:

- tolerância a falhas: as redes neurais podem ser treinadas para reconhecerem padrões distorcidos e até mesmo incompletos, além serem capazes de operar, até certo ponto, mesmo quando parte da rede for danificada;

- compatibilidade com a tecnologia atual: é possível treinar uma rede para executar uma tarefa especializada e depois implementá-la em um *hardware* de baixo-custo. Assim, elas

poem ser facilmente inseridas em sistemas existentes;

- auto-organização: redes neurais possuem uma capacidade de aprendizagem adaptativa, auto-organizando as informações recebidas. Isto permite que a rede responda satisfatoriamente mesmo quando forem apresentadas informações novas.

4.8.2 Redes neurais em jogos

Até o momento, não existem técnicas para modelagem de uma rede com inteligência complexa (ao nível de um animal, por exemplo), mesmo sendo possível, com a tecnologia de hoje, montar uma rede com milhares de neurônios. A maioria das aplicações atuais de RNs utilizam cerca de uma dúzia ou mais neurônios, e elas são feitas para desempenhar tarefas específicas. E em jogos isto não é diferente. Redes neurais são usadas em conjunto com outras técnicas, executando tarefas que necessitam um certo grau de aprendizado ou para realizar tarefas em que as unidades controladas pelo computador sejam menos previsíveis.

Uma das redes mais utilizadas em jogos é a rede *feedforward* de três camadas. Ela é assim chamada devido ao fato de seus neurônios enviarem suas saídas à próxima camada até que seja obtida a saída final da rede.

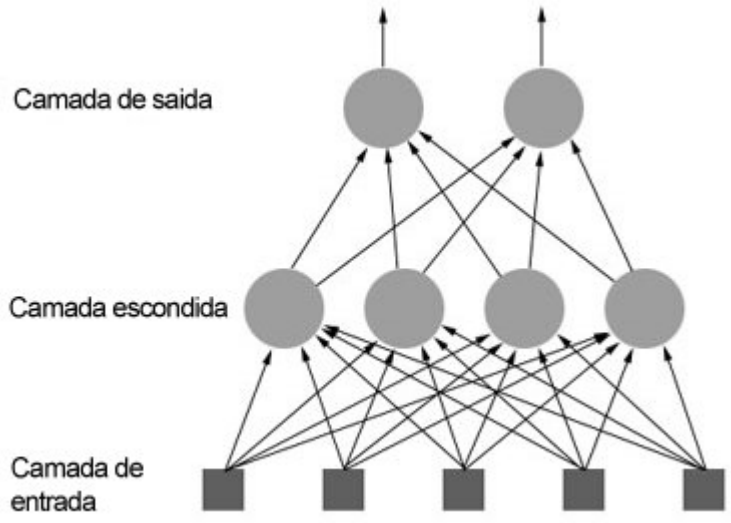

Figura 13 - Rede *feedforward* de três camadas

A rede possui uma camada de entrada, uma camada escondida e uma camada de saída. Não há restrições quanto ao número de neurônios em cada camada, e a quantidade ideal deles depende do problema a ser resolvido. Cada neurônio da camada de entrada é ligado a todos os neurônios da camada escondida, e cada neurônio da camada escondida é ligado a todos os neurônios da camada de saída. Maiores detalhes sobre o cálculo de funções de ativação, bem como de outros tipos de RN pode ser encontrados em FREEMAN 1991.

A rede podería ser utilizada, por exemplo, para controlar a direção de uma unidade, tendo como entrada as informações enviadas por seus sensores visuais e como saída, para qual direção a unidade deve seguir. Observe que apenas a parte da tomada de decisão é controlada pela RN. Para a movimentação pode-se utilizar um algoritmo simples de perseguição e fuga. Veja abaixo uma ilustração para o problema.

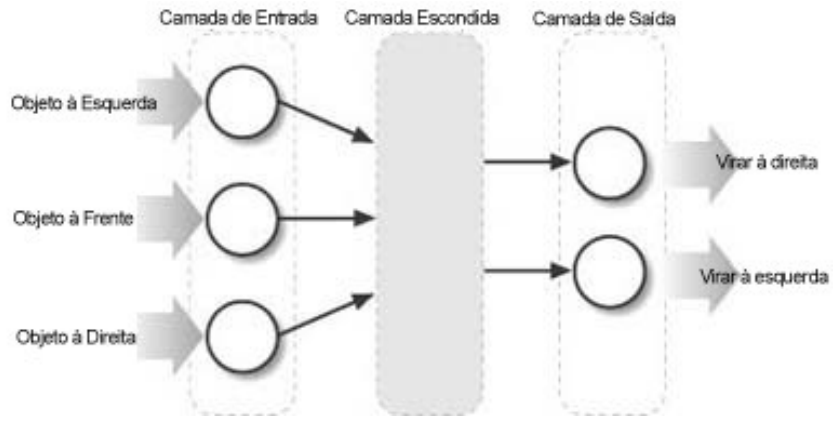

Figura 14 - rede *feedforward* tripla camada controlando uma unidade

4.8.3 Mapeando o problema

Toda rede neural recebe como entrada um conjunto de números reais. O problema ocorre quando precisamos utilizar outros tipos de valores de entrada, como enumerações e valores lógicos verdadeiro/falso –o que é comum na implementação de jogos.

O mapeamento pode ser feito utilizando os valores 0 (zero) para o valor lógico “falso” e 1 (um) para o valor lógico “verdadeiro”. Quando a variável for uma enumeração, utiliza-se um vetor de entrada, sendo que cada posição do vetor possui valor 0 ou 1. Por exemplo: se temos uma variável enumerada que pode assumir os valores “espada”, “revólver” e “canhão”, por exemplo, teríamos um vetor de três posições com os valores “{0, 0, 1}”, caso o valor desta variável seja “canhão”, por exemplo (aqui assumimos que as variáveis se excluem mutuamente, não podendo assumir mais de um valor ao mesmo tempo). Se a enumeração tivesse um valor “revólver”, o vetor de entrada sería “{0, 1, 0}”.

Quando nosso problema já se encontra no domínio contínuo, apenas precisamos enviar suas variáveis como entrada na rede neural. Deve-se atentar para entradas com diferenças acentuadas entre seus elementos. Por exemplo, se uma rede possuir uma entrada com valor 20 e outra com valor 200, pode-se obter um valor influenciado pela entrada de maior valor. Para tal problema, pode-se dividir ambas as entradas pelo maior valor (200), obtendo valores em termos percentuais.

4.9 Algoritmos Genéticos

A pesquisa em algoritmos genéticos, assim como as redes neurais, é outra área da IA influenciada pela biologia. Ela é baseada nos estudos sobre a evolução das espécies, de Charles Darwin, moldando a população de indivíduos através da sobrevivência dos membros mais aptos a viver naquele ambiente.

No mundo real, as espécies evoluem constantemente na tentativa de melhor se adaptarem aos seus ambientes. Os seres mais “fortes” e capazes de sobreviver passam adiante suas características para a próxima geração. Tais características são codificadas nos cromossomos. A geração resultante é fruto da combinação de dois cromossomos da geração atual, num processo denominado *crossover.* Nessa combinação podem ocorrer mutações aleatórias, e, se estas mutações contribuírem para a sobrevivência da espécie, elas serão também passadas para as gerações futuras.

4.9.1 Algoritmos genéticos em jogos

No desenvolvimento de jogos, os algoritmos genéticos visam a busca de uma solução ótima para um determinado problema. Cada um dos indivídios mencionados anteriormente são possíveis soluções para o problema em questão. Geralmente, os algoritmos genéticos são utilizados quando se deseja uma IA que atue mesmo em situações não previstas pelo desenvolvedor, como em tomadas de decisão de acordo com o comportamento do jogador. Para problemas como o do *pathfinding*, (nos quais já existem algoritmos sólidos para sua resolução) não é recomendado o uso dos algoritmos genéticos, pois estes são mais custosos computacionalmente.

A implementação dos algoritmos genéticos em jogos pode ser dividida em quatro etapas, que são a inicialização da população, validação da aptidão, seleção e evolução.

1) Inicialização: primeiramente, é preciso definir um modo de codificar os cromossomos da população de criaturas de IA que iremos criar. Esta codificação pode ser feita de maneira simples, utilizando um vetor de caracteres; cada índice do vetor pode representar uma característica do ambiente externo e o valor contido nesta posição representa a ação que a criatura deverá executar.

Exemplo: Suponha que a população de criaturas controladas pelo computador tenham que responder de acordo com a arma em uso pelo jogador (espada, arco, lança e machado), e as respostas possíveis são: usarEscudo (1), ataqueComEspada (2), fugir (3), esconder (4) e ataqueComArco (5). Então os cromomossomos destas criaturas serão representados por um vetor de quatro posições, onde a posição 1 contém o identificador da ação a ser executada caso o jogador tenha uma espada, a posição 2 será executada caso o jogador tenha um arco, e assim por diante. Cada posição do vetor assume um valor de 1 a 5.

Na etapa de inicialização, espera-se obter uma população com alto grau de variabilidade genética, portanto atribui-se valores aleatórios para os cromossomos. Veja o exemplo da tabela abaixo.

Tabela 2 – exemplo de codificação de um cromossomo

Arma	Espada	Arco	Lança	Machado
Ação	[1..5]	[1..5]	[1..5]	[1..5]

Definidos os métodos de codificação dos genes da população, inicializamos ela conforme descrito acima, com valores aleatórios para os cromossomos das criaturas. Note que o exemplo acima possui apenas quatro situações possíveis. Numa aplicação real poderíamos ter um número muito maior de situações a considerar.

2) Validação da aptidão: nesta etapa do processo evolucionário, deve-se avaliar quais indivíduos da população são os mais aptos a resolver o problema em questão. Para isto, utilizamos uma função de avaliação de aptidão (também denominada função *fitness*). Para o exemplo acima, uma boa função avaliação é contabilizar a diferença entre o dano que a criatura causou ao jogador e o dano recebido. As criaturas com a maior aptidão segundo esta função terão a maior probabilidade de passar seus genes à geração seguinte.

3) Seleção: nesta etapa é utilizada a função calculada na etapa anterior para que sejam escolhidos os indivíduos que participarão do processo evolucionário. Na vida real, geralmente dois pais contribuem com seus cromossomos para a geração seguinte. Já no mundo dos jogos, podemos escolher qualquer número dentre os melhores indivíduos (segundo a função de aptidão).

4) Evolução: nesta última etapa, serão criados os novos indivíduos que serão introduzidos no ambiente do jogo. São selecionados os melhores indivíduos e combinados seus genes no processo de *crossover*. Aqui também são introduzidas mutações aleatórias. Observe a tabela abaixo.

Tabela 3 – exemplo de *crossover*

Pai 1	**Pai 2**	**Filho**
5	3	5
4	1	1
3	5	4
4	2	2

Observe que no terceiro cromossomo do indivíduo resultante (terceira linha da tabela) a característica não foi herdada de nenhum dos pais. Isto é resultado de uma mutação aleatória. Por exemplo, na hora do *crossover*, podemos fazer com que a característica resultante tenha, digamos, 5% de chances de ser gerada aleatóriamente, ou seja, não estando presente em nenhum dos indivíduos da geração anterior.

Agora o algoritmo retornará à segunda etapa, e a cada iteração, serão criados indivíduos mais aptos a enfrentar o jogador em diferentes situações.

A implementação do algoritmo genético é resumida abaixo, em pseudo-código.

Código 8: algoritmo genético em pseudo-código

```
Procedimento algoritmo genético
  Início
    Inicialize a população;
    Enquanto a condição de parada não for satisfeita faça
    Início
      Avalie a aptidão de cada indivíduo da população;
      Selecione os membros da população com base na aptidão;
      Produza os descendentes destes pares;
      Introduza esses descendentes na população
   Fim
Fim
```

4.10 *Flocking*

É bastante comum na implementação de jogos a necessidade de prover uma movimentação em grupos para personagens controlados pelo computador, principalmente em jogos de estratégia e RPG. Estes grupos podem ser de animais, pessoas, unidades de combate, entre outros. É claro que também é possível implementar esse tipo de movimentação com *scripts* bem elaborados, mas a utilização de *flocking* elimina a necessidade de criar um *script* para controlar cada criatura do jogo –ao invés disso, precisamos de apenas um *script* para controlar o grupo como um todo.

Um algoritmo básico de *flocking* foi apresentado por Craig Reynolds, em seu artigo de 1987 intitulado “Flocks, Herds, and Schools: A Distributed Behavioral Model”. A idéia principal do algoritmo é fazer os NPCs moverem-se de maneira coerciva, como se houvesse um objetivo em comum –o oposto de se ter várias unidades movendo-se de maneira independente, sem coordenaçao alguma.

Para determinar a direção resultante de cada unidade, cada unidade controlada pelo *flocking* mantém informações sobre outras unidades em um raio circular de tamanho pré-definido, centrado na própria unidade. Este raio é denominado “ráio de vizinhança” (*neighborhood radius*) (BUCKLAND, 2005). Ainda segundo BUCKLAND, *flocking* também é bastante utilizado em filmes, como em “*Batman Returns”* e “O Senhor dos Anéis”.

As três principais regras que regem o algoritmo de *flocking* são: separação, alinhamento e coesão:

- separação: é uma força que faz com que as unidades mantenham certa distância uns dos outros, evitando colisões com seus vizinhos;

- alinhamento: mantém a direção de cada unidade alinhada à direção de seus vizinhos. Isto é feito calculando-se a média dos

vetores de direção de cada vizinho e subtraindo-se desta média a direção atual da unidade. O resultado é a direção que a unidade deve seguir para manter-se em rumo com seus vizinhos;

- coesão: é um tanto similar à regra anterior, mas neste caso a unidade dirige-se ao centro de massa (posição média) de seus vizinhos. A média é calculada levando em consideração o vetor de posições dos vizinhos. O vetor resultante é o centro de massa da vizinhança e indica a posição à qual a unidade deve se dirigir.

Flocking é considerado uma área de estudo da *Artificial Life*, ou simplesmente *A-Life* (vida artificial).

"[...] *Artificial Life* (*Alife*) basicamente procura estudar a vida e suas propriedades no computador –criando a vida digitalmente dentro do computador, simulando-a ou ambos. *Alife* pode ser especialmente interessante para pessoas de outras áreas como a teoria do caos, pois muitos aspectos são comuns entre ambas as áreas. " (MATTHEWS, 2002, trad.nossa).

Para emular a vida, a *A-Life* faz uso de técnicas como algoritmos genéticos, *flocking*, sistemas baseados em regras, entre outras. O comportamento das criaturas é dividido em problemas menores, que são mais tarde unidos por uma hierarquia de decisões que cada uma delas possui, determinando que ações devem ser feitas para satisfazer suas necessidades. Segundo Woodcock (2000):

"[...] as interações que ocorrem entre comportamentos de baixo-nível, explicitamente codificados e as motivaçoes/necessidades dos personagens causa o surgimento de um comportamento de alto-nível, mais 'inteligente', sem qualquer programação explicíta ou complexa." (trad.nossa).

5 ALGORITMOS DE IA PARA JOGOS – ESTADO DA ARTE

Com o surgimento de placas aceleradoras gráficas 3D para lidar especificamente com gráficos pesados, hoje há uma fatia muito maior da CPU para processamento da IA. Além disso, bons gráficos deixaram de ser um diferencial e passaram a ser algo essencial para o sucesso de vendas do produto. Deste modo, a IA acaba tornando-se esse grande diferencial entre um produto e outro.

Segundo Woodcock, (1999) na maioria das empresas já existem equipes especializadas em desenvolver a IA para o jogo –algo incomum há pouco tempo atrás. Em estúdios menores, onde não há recursos financeiros para tal, é contratado pessoal terceirizado para trabalhar com a IA –mostrando uma viabilidade mercadológica para pesquisadores de IA neste segmento.

Ultimamente, tem-se procurado muito a implementação de IA não-determinística, devido à sua menor previsibilidade e de sua capacidade de aprender com o jogador. Estes, por sua vez, esperam unidades de IA cada vez mais inteligentes e desafiadoras, sem o uso de *cheating*.

A seguir, apresentaremos os principais tópicos de estudo da IA para jogos atualmente.

IA extensível: presente principalmente nos jogos de tiro em primeira pessoa, um grande número de jogos recentes tem apresentado esta técnica. Com ela os jogadores podem criar sua própria IA, seja através de uma linguagem *script* criada especialmente para o jogo ou através de uma interface mais simples, pela qual o jogador define o comportamento do computador. Exemplos de sucesso são os jogos *Quake* e *Unreal*. Existem muitos lugares na internet aonde os jogadores compartilham suas criaturas criadas através de IA extensível –aumentando o tempo de vida útil do produto.

Pathfinding: o algoritmo A* continua sendo o mais utilizado para o *pathfinding*, seja na sua forma padrão ou com a inclusão de algumas modificações

–a comunidade de jogos considera este problema bem resolvido, e agora busca implementações específicas para cada jogo.

Atualmente, o que mais se tem pesquisado nesta área é a inclusão da análise de terreno em *pathfinding*. A análise de terreno procura por características naturais do cenário, como lugares para emboscadas e combates, recursos naturais, entre outras; tais características são usadas na criação de um *pathfinding* mais robusto.

Formações: com o crescente sucesso de jogos de simulação militar, este tópico tem sido alvo de muito estudo por desenvolvedores de IA. O objetivo é fazer grupos de unidades militares comportarem-se de maneira realística. Sistemas baseados em regras, *flocking*, máquinas de estado finito e lógica nebulosa são as técnicas preferidas devido a uma maior facilidade na implementação e testes, em detrimento de técnicas como algoritmos genéticos e redes neurais.

A-Life e seu modo de quebrar o problema em situações menores também tem sido muito utilizada, dando uma visão hierárquica ao desenvolvimento da IA. Jogos como *Starfleet Command* (Interplay) e *Force 21* (Red Storm) utilizam esta técnica. Nesses jogos, as unidades são divididas em grupos, onde cada um deles possui um comandante; as ordens à cada grupo são interpretadas pelo comandante, que então atribui comandos à cada unidade individualmente.

Aprendizagem: futuramente, espera-se criar jogos que evoluam constantemente, aprendendo com o jogador e melhorando suas aptidões mesmo após saírem das prateleiras. Existem dois principais problemas com relação a isso: um deles é a dificuldade de testar o produto, pois o comportamento de uma rede neural, por exemplo, pode ser imprevisível; outro problema é a possibilidade de que a IA, após certo tempo de treinamento com o jogador possa tornar-se estúpida ou até mesmo difícil demais para o jogador –uma solução podería ser a inclusão de uma opção pela qual o jogador pudesse reiniciar o aprendizado da máquina para suas configurações de fábrica.

Outra abordagem para o problema é a criação de um banco de dados

de situações passadas, como foi feito no jogo *Magic & Mayhem*, da Mythos Games. Diante da necessidade de atacar o jogador, o jogo recorre ao seu banco de dados procurando aquelas estratégias que surtiram efeito em situações parecidas que ocorreram anteriormente.

Alguns jogos utilizam redes neurais que são treinadas pelos próprios desenvolvedores e que são mais tarde “desligadas” antes de chegarem ao consumidor. Tal abordagem pode criar uma boa IA, mas infelizmente anula qualquer possibilidade de aprendizagem com o jogador.

Também é importante ressaltar que os algoritmos genéticos têm sido pouco utilizados recentemente, ficando sua implementação mais limitada aos jogos “simuladores de vida”. Mas em seu lugar, as técnicas de *flocking* e *A-Life* continuam sendo bastante utilizadas na IA para jogos.

6 CONCLUSÃO

É bastante evidente a importância que a IA tem mostrado na implementação de jogos. Devido ao fato dos gráficos deixarem de ser um grande diferencial e devido aos avanços de *hardware*, há cada vez mais espaço para a implementação de técnicas outrora consideradas muito “acadêmicas” para uma implementação comercial. Também pode-se observar cada vez mais a utilização de técnicas não-determinísticas em conjunto com já conhecidas técnicas como as máquinas de estado finito, *scripts* e sistemas baseados em regras.

Muito se argumentava sobre a possibilidade da criação de jogos multi-jogador via rede fazer com que as pesquisas em IA ficassem estagnadas, pois, jogar contra humanos é muito mais desafiador que fazê-lo contra o computador. No entanto, já foi comprovado que, seja por questões financeiras ou mesmo pessoais, muitos ainda preferem jogar contra a máquina.

Futuramente, espera-se que a IA aprenda cada vez mais com o jogador, utilizando redes neurais artificiais e até mesmo os algoritmos genéticos, que até o momento têm sido pouco utilizados, salvo em implementações específicas. Mesmo em tópicos em que já existem algoritmos de sucesso como o *pathfinding* e seu A*, ainda há muito a ser pesquisado, o que torna esta uma área de pesquisa bastante promissora. E o resultado dessas pesquisas é a criação de jogos cada vez mais divertidos.

REFERÊNCIAS BIBLIOGRÁFICAS

BOURG, David; SEEMAN, Glenn. **AI for Game Developers**. O'Reilly, 2004.

BROWNLEE, Jason. **Finite State Machines (FSM)**. AI Depot. Disponível em: <http://ai-depot.com/FiniteStateMachines/FSM.html>. Acesso em 15 out. 2005.

BUCKLAND, Mat. **Programming Game AI by Example**. Texas, EUA: Wordware Publishing, Inc, 2005.

BUCKLAND, Mat. **AI Techniques for Game Programming**. Cincinnati, Ohio. EUA: Premier Press, 2002.

CHAMPANDARD, Alex. **AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors**. New Riders Publishing, 2003.

FREEMAN, James A.; SKAPURA, David M. **Neural Networks: Algorithms, Applications and Programming Techniques**. Loral Space Information Systems and Adjunct Faculty, School of Natural and Applied Sciences University of Houston at Clear Lake: Addison-Wesley Publishing Company, 1991.

LESTER, Patrick. **A* Pathfinding for Beginners**. GameDev.net. Disponível em:<http://www.gamedev.net/reference/articles/article2003.asp>. Acesso em 8 mar. 2005.

LUGER, George F. **Inteligência Artificial: estruturas e estratégias para a solução de problemas complexos**. - 4.ed. - Porto Alegre: Bookmann, 2004.

MCCARTHY, John. **What is Artificial Intelligence?**. Stanford University, Computer Science Department, 2004. Disponível em:<http://www-formal.stanford.edu/jmc/whatisai/>.

MCNEIL, Martin F.; THRO, Ellen. **Fuzzy Logic: A Practical Approach**. Chestnut Hill, MA, EUA: AP Professional, 1994.

RUSSEL, Stuart J.; NORVIG, Peter. **Inteligência Artificial: Uma Abordagem Moderna**. São Paulo: Editora Campus, 2004.

STOUT, Brian. **Smart Moves: Intelligent Pathfinding**. Gamasutra, 1997. Disponível em:< http://www.gamasutra.com/features/19970801/pathfinding.htm>. Acesso em 24 mai. 2005.

WOODCOCK, Steve. **Game AI: The State of the Industry**. Gamasutra, 1999. Disponível em:<http://www.gamasutra.com/features/19990820/game_ai_01.htm>. Acesso em 18 mai. 2005.

WOODCOCK, Steve. **Game AI: The State of the Industry**. Gamasutra, 2000. Disponível em:<http://www.gamasutra.com/features/20001101/woodcock_01.htm>. Acesso em 18 mai. 2005.

BIBLIOGRAFIA CONSULTADA

CHAMPANDARD, Alex. **The Future of Game AI: Intelligent Agents**. Disponível em:< http://ai-depot.com/GameAI/Agent-Intelligence.html>. Acesso em 16 out. 2005.

FUNGE, John D.; TU, Xiaoyuan. **Hardcore AI for Computer Games and Animation**. In: SIGGRAPH, 1998. Wesmont, Illinois. EUA.

LAMOTHE, Andre. **Building Brains into Your Games**. Gamasutra. Disponível em: <http://www.gamasutra.com/features/19970601/build_brains_into_games.htm>. Acesso em 18 mai. 2005.

MATTHEWS, James.**How to Get Started with Artificial Life**. Generation5, 2002. Disponível em:<http://www.generation5.org/content/2002/howto04.asp>. Acesso em 1 nov. 2005.

PINTO, Paulo. **Minimax Explained**. AI Depot. Disponível em: <http://ai-depot.com/LogicGames/MiniMax.html>. Acesso em 2 nov. 2005.

SALTZMAN, Marc. **Game Creation and Careers: Insider Secrets from Industry Experts**. New Riders Publishing, 2004.

SMITH, Leslie. **An Introduction to Neural Networks**. Centre for Cognitive and Computational Neuroscience. University of Stirling. Disponível em:<http://www.cs.stir.ac.uk/~lss/NNIntro/InvSlides.html>. Acesso em 8 mar. 2005.

ANEXOS

ANEXO A – Algoritmo de Bresenhan

```
void ai_Entity::BuildPathToTarget (void)
{
     int nextCol=col; //col e row são valores iniciais do caminho
     int nextRow=row;

     //endRow e endCol são coordenadas de destino
     //deltaRow e deltaCol são usados para determinar a direção
     int deltaRow=endRow-row;
     int deltaCol=endCol-col;
     int stepCol, stepRow;
     int currentStep, fraction;

     for (currentStep=0;currentStep<kMaxPathLength; currentStep++)
     {
       //inicialização dos vetores do caminho
       pathRow[currentStep]=-1;
       pathCol[currentStep]=-1;
     }

     currentStep=0;
     pathRowTarget=endRow;
     pathColTarget=endCol;

     //determina a direção usando os deltas
     if (deltaRow <  0) stepRow=-1;  else stepRow=1;
     if (deltaCol <  0) stepCol=-1;  else stepCol=1;

     deltaRow=abs(deltaRow*2);
     deltaCol=abs(deltaCol*2);

     //inicia o caminho com as coordenadas atuais do predador
     pathRow[currentStep]=nextRow;
     pathCol[currentStep]=nextCol;
     currentStep++;

     if (deltaCol >deltaRow) //determina qual eixo é o mais longo
     {
       fraction = deltaRow *2-deltaCol;
       while (nextCol != endCol)
       {
         if (fraction >=0)
         {
           nextRow =nextRow +stepRow;
           fraction =fraction -deltaCol;
         }

         nextCol=nextCol+stepCol;
         fraction=fraction +deltaRow;
         pathRow[currentStep]=nextRow;
         pathCol[currentStep]=nextCol;
         currentStep++;
       }
     }
     else
     {
```

```
        fraction =deltaCol *2-deltaRow;
        while (nextRow !=endRow)
        {
          if (fraction >=0)
          {
            nextCol=nextCol+stepCol;
            fraction=fraction -deltaRow;
          }

          nextRow =nextRow +stepRow;
          fraction=fraction +deltaCol;
          pathRow[currentStep]=nextRow;
          pathCol[currentStep]=nextCol;
          currentStep++;

        }
      }

}
```