Estratégias Adaptativas e Evolutivas em Tempo Real para Jogos Eletrônicos

Pedro Demasi
Orientador: Adriano Joaquim de Oliveira Cruz

UFRJ

2003

Estratégias Adaptativas e Evolutivas em Tempo Real para Jogos Eletrônicos

por

Pedro Demasi

Dissertação submetida ao Corpo Docente do Instituto de Matemática e Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências.

Área de Concentração: Ciência da Computação

Aprovada por:

Adriano Joaquim de Oliveira Cruz, Ph.D
(Presidente)

Geber Lisboa Ramalho, DSc.

Josefino Cabral Melo Lima, DSc.

Antônio Carlos Gay Thomé, Ph.D

À Universidade Federal do Rio de Janeiro e seu Colégio de Aplicação, aos quais devo muito do que sou hoje.

Agradecimentos

Agradeço, primeiramente, à minha mãe, não só pelos motivos óbvios como, também, pela paciência em me ajudar a corrigir este texto.

Quero agradecer ao meu orientador pela oportunidade que poucos teriam coragem de dar, de desenvolver um trabalho na tão subestimada área de jogos.

Agradeço à CAPES e ao NCE pelo financiamento dado à pesquisa deste trabalho.

É necessário, mesmo após a dedicatória, agradecer explicitamente à Universidade Federal do Rio de Janeiro e seu Colégio de Aplicação, não só por tudo aquilo que representaram e ainda representam para mim como, também, por mostrarem que ainda é possível haver ensino público de excelente qualidade mesmo frente a todas as enormes dificuldades que tais instituições são obrigadas a enfrentar. Servem, para todos, como exemplo e modelo a serem seguidos.

Devo, também, agradecer a todos aqueles que, direta ou indiretamente, colaboraram para que este trabalho fosse possível, principalmente os voluntários que se propuseram a testar e avaliar os jogos implementados como resultado da pesquisa.

E, finalmente, gostaria de agradecer aos colegas do Laboratório de Inteligência Computacional (LabIC-NCE), do qual faço parte, pelo apoio e companheirismo durante este período.

Resumo

Este trabalho apresenta um estudo a respeito de técnicas adaptativas e evolutivas aplicadas a problemas de tempo real, em particular para o caso de jogos eletrônicos. Alguns desses métodos propostos são contribuições originais (coevolução em tempo real e padrões adaptativos), enquanto outros são modificações de estratégias já existentes (máquinas de estado nebulosas, aprendizado de regras nebulosas, previsão de cadeias e navegação) de maneira a possibilitar seu aproveitamento em tempo real.

Todos os métodos e estratégias propostos e comentados foram implementados em jogos eletrônicos desenvolvidos exclusivamente para este fim, e os resultados obtidos devidamente analisados.

Abstract

This work presents a study about adaptative and evolutionary techniques applied to real-time problems, particularly to video games. Some of these proposed methods are original contributions (online coevolution and adaptative patterns), whereas others are modifications of existing strategies (fuzzy state machines, fuzzy rule learning, sequence prediction and navigation) in order to make possible apply them to real time problems.

All proposed and discussed methods and strategies were implemented on video games developed specially to this end, and the results were analysed.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

"Prazos largos são fáceis de subscrever. A imaginação os faz infinitos."
Machado de Assis, *Dom Casmurro*

1.1 Jogos Eletrônicos

1.1.1 Por que Jogos?

A necessidade por diversão faz parte da natureza do ser humano. Talvez por ser um anseio tão comum a tantas pessoas pode parecer algo banal e até insignificante. Na área de Ciência da Computação, o entretenimento é praticamente ignorado no meio acadêmico, sendo considerado e explorado quase que exclusivamente na área comercial. Os jogos eletrônicos, contudo, podem ser encarados como uma fonte quase inesgotável de possibilidades de pesquisa e descobertas científicas, se levados a sério. O problema é que raramente o são.

Normalmente, quando vistos sob uma perspectiva livre de preconceitos, os jogos podem servir como um apoio de importância inestimável para qualquer pesquisa em desenvolvimento ou atividade de ensino. É comum, porém, que se qualifique essa área como não sendo "séria" e, assim, confunde-se, equivocadamente, seriedade com diversão. Afinal, a qualidade de uma determinada descoberta (ou, ao menos, a busca pela mesma) não pode ser medida através do nível de diversão associado a seu meio. Se algum método é criado, implementado, testado e avaliado, usando-se um instrumento divertido para tal, o que o torna, em teoria, menos importante do que outro a que se rotula "sério"?

Seriedade não implica importância, assim como diversão não a exclui. Concluir

o contrário, de forma puramente retórica, não passa de preconceito.

Muitos avanços experimentados pela computação, nos últimos anos, devem-se a jogos, ainda que indiretamente. A febre de placas de vídeo com aceleração em 3D surgiu devido ao crescimento exponencial de complexidade gráfica dos jogos, a partir de meados da década de 90 (HADWIGER, 2000). Se, por um lado, as pesquisas em hardware nessa área permitiram que os jogadores, ávidos por qualidade e realismo cada vez maiores em seus jogos, tivessem seus desejos atendidos, por outro lado permitiu que esses dispositivos gráficos avançadíssimos pudessem ser usados por outros tipos de usuários, como agências de publicidade, designers, artistas e, até mesmo, o usuário doméstico comum. Temos hoje, em casa, equipamentos gráficos com a potência que, há meia década, nem grandes estações gráficas tinham.

A importância dos jogos, entretanto, não fica restrita apenas ao campo da informática. A grande explosão da internet no final do século passado possibilitou, também, o surgimento dos chamados jogos *"Massive Multi-player"*, em que milhares (literalmente) de pessoas jogam simultaneamente através da grande rede, imersas num mundo virtual, interagindo entre si e com o ambiente. Além de estudos e algoritmos a respeito de distribuição de cargas entre servidores, também podem ser feitas pesquisas sociais a respeito do comportamento das pessoas nesses ambientes virtuais e suas interações.

Um potencial enorme como esse não poderia ficar imerso na completa obscuridade. Iniciativas surgem pelo mundo com alguma força, e mesmo o mais sisudo dos ambientes acadêmicos terá, mais cedo ou mais tarde, de admitir e aceitar essa área de pesquisa como promissora. Já há congressos e conferências acadêmicos, na área de Ciência da Computação, ao redor do mundo que discutem pesquisa em jogos, como, por exemplo, ADCOG (Application and Development of Computer Games), CG (International Conference on Computers and Games) e ICEC (International Conference on Entertainment Computing) e até sobre aspectos específicos como Game On (International Conference on Intelligent Games and Simulation) No Brasil, em outubro de 2002 realizou-se, em Fortaleza, o I Workshop Brasileiro de Jogos e Entretenimento digital (WJogos), patrocinado pela Sociedade Brasileira de Computação (SBC). Apesar de o apelo comercial da área ainda ser muito forte, o meio acadêmico parece estar começando a abrir os olhos para as demais possibilidades que essa área nos oferece, se bem explorada.

1.1.2 Vantagens e Desvantagens

Como não poderia deixar de ser, há vantagens e desvantagens em se orientar uma pesquisa através de jogos. Quais são essas e a natureza de cada uma pode depender do que e como se está pesquisando, quais recursos (humanos e tecnológicos) se tem à disposição etc. Muitos fatores, contudo, podem ser vistos como genéricos o suficiente para serem encarados em qualquer pesquisa. Podemos, então, analisar o caso deste trabalho, e como todos os fatores podem influenciar de forma conjunta o andamento duma pesquisa. Não vamos nem entrar no mérito da questão financeira, uma vez que já é, de alguma maneira, lugar comum lembrar a importância econômica da produção de jogos em todo o mundo (BATTAIOLA, 2000). Supomos, assim, que não há muita necessidade de nos alongarmos neste ramo específico.

Uma das grandes vantagens dos jogos é o apelo. Este desempenha um papel importante como motivador para testes. De forma simples, após um modelo ser criado e implementado num jogo, a probabilidade de ele ser testado exaustivamente é alta. Pode-se espalhá-lo pela Internet, ou mesmo entre amigos, e, dependendo da capacidade do jogo em prender a atenção do jogador, tem-se uma boa possibilidade de testar o modelo de várias maneiras, seja pelas pessoas que irão jogar (diversidade e quantidade de adversários), seja pelo número de vezes que cada um irá jogar (vários testes realizados), entre outros fatores (diferentes configurações de máquina, situações inusitadas ou inesperadas etc). A grande desvantagem é que, para isso se concretizar, possivelmente a qualidade do jogo deverá ser razoavelmente alta em comparação com a qualidade média de jogos comerciais aos quais o jogador médio está acostumado, o que foge um pouco da proposta inicial do projeto. Isso pode fazer com que muito tempo seja perdido com detalhes que não importam diretamente na pesquisa, como a qualidade gráfica do jogo, uma interface bem elaborada etc. Outra dificuldade é recuperar os dados de jogo para análise, pois normalmente é necessário contar com a boa vontade dos usuários em mandar as informações, a não ser que o jogo seja feito usando-se uma arquitetura cliente/servidor, pois neste caso os dados são coletados diretamente pelo servidor. Esta última solução, porém, requer mais recursos computacionais que nem sempre estão disponíveis.

Uma outra vantagem importante é a motivação. Tendo uma equipe trabalhando em torno de um projeto, costuma ser uma excelente vantagem fazer com que todos tenham prazer naquilo que estão fazendo, e o fato de estarem trabalhando em con-

junto de forma a implementar todo aquele trabalho teórico num jogo pode ser um fator importantíssimo para motivá-los. No curso de Computação I do Bacharelado em Informática da UFRJ, por exemplo, os alunos são incentivados a implementar um jogo simples como trabalho final. Para alunos de primeiro período, muitos dos quais aprenderam a programar naquele semestre, os resultados são animadores. Por não ser uma atividade extremamente trivial, contudo, a criação do jogo exige conhecimentos sólidos de programação. Ao invés de ser uma desvantagem, entretanto, isso acaba sendo uma vantagem, pois é uma forma de se aprofundar os conhecimentos não só em programação como em outras áreas correlatas, como algoritmos, computação gráfica, inteligência artificial etc.

Um fator importante, também, é a natureza interdisciplinar dos jogos. Várias pesquisas de diferentes áreas podem ter suas idéias implementadas num mesmo projeto de jogo. Áreas como inteligência artificial, computação gráfica, compiladores, programação, redes etc. são todas importantes na elaboração do jogo. Algumas parecerão mais óbvias (como computação gráfica) e outras, menos (como compiladores), mas todas têm, sim, sua importância dentro de um projeto mais amplo. Essa diversidade não se restringe, todavia, apenas à Ciência da Computação. Outras áreas do conhecimento humano no meio acadêmico são igualmente candidatas a fazer parte de um mesmo projeto para suas pesquisas, como, por exemplo, Psicologia, Pedagogia, Sociologia, Desenho Industrial, Cinema, Teatro etc. com estudos e pesquisas pertinentes a suas respectivas áreas. Este fator é muito importante para se lidar com a estrutura complexa de um jogo, bem como lidar com cada aspecto de forma específica, além de permitir uma excelente troca entre diversas áreas de pesquisa. Uma das possíveis desvantagens é a dificuldade em se juntar toda essa diversidade dentro de um projeto (seja por falta de espaço físico ou pela excessiva distância entre os centros). Outro problema seria a dificuldade de se avaliar o êxito de cada um dos projetos de pesquisa em separado, uma vez que um pode estar interferindo no outro, dificultando a análise dos resultados obtidos de forma independente.

Temos, até como uma espécie de conseqüência do que foi exposto no parágrafo anterior, o fato de que há diversas possibilidades de se encaixar qualquer tipo de método desenvolvido dentro de um jogo para avaliação. Em questões específicas de inteligência, como algoritmos adaptativos, os jogos representam possibilidades quase infinitas. O problema é saber encaixar o que foi criado dentro do jogo. Verificam-se,

dessa forma, duas visões para a origem da concepção do método, uma seria ele ter sido feito de forma a se encaixar no jogo e a outra o jogo ter sido feito para que o método se encaixe nele. Ou seja, ou a natureza do jogo influi na criação do método ou o método criado influi na natureza do jogo. Usar um jogo já desenvolvido, assim como suas dificuldades e fraquezas, como motivação para criar uma nova técnica seria, também, uma experiência interessante. Afinal, após implementada ela viria a ser aplicada em outros contextos que não o de jogos eletrônicos, sendo bastante útil fora deles, embora motivada pelos mesmos.

1.2 Relevância de Jogos em Métodos Aproximativos

Existem diversos problemas para os quais se procura um método analítico direto que os resolva. Na área de Inteligência Artificial, por exemplo, há problemas em que tal abordagem é possível, como procurar o menor caminho de navegação dentro de um ambiente discretizado. Em alguns casos é possível um método, mas sua complexidade de computação é tão grande que ele passa a ser inviável. O problema do caixeiro viajante é um exemplo clássico, pois, mesmo usando-se diversas heurísticas, a estimativa do tempo de execução dos algoritmos é exponencial (CORMEN et al, 2002). O que nos resta, nesses casos, é procurar uma forma aproximativa que tenha a menor estimativa de erro possível, ou recorrer a um algoritmo que tenha complexidade polinomial mas que, para isso, tenha associada a ele uma probabilidade de o resultado não ser ótimo.

Há problemas, porém, para os quais não existe uma solução direta, muitas vezes devido à sua incerteza natural ou caráter aproximativo. Reconhecimento de caracteres seria um desses problemas (SILVA, 2002).

Várias situações da vida real exigem soluções que não podem ser determinadas através de um método algorítmico direto e correto. Para esses casos temos de criar estratégias aproximativas, que nos dêem o menor erro que for possível. A determinação do que seja esse "erro" será simples ou não, dependendo da natureza do problema. Para reconhecimento de caracteres, por exemplo, sabemos que o "erro" consiste em reconhecer uma letra como diferente da que se está apresentando ao modelo criado. Além, nesse caso, de ser relativamente fácil determinar a natureza do erro, também podemos binarizá-lo (ou seja, contar como "certo" ou "errado").

Uma estratégia para um jogo, todavia, não tem essas características. Afinal, se o modelo criado perde um jogo, podemos considerar que ele está "errado"? Se criarmos um programa para jogar xadrez e ele perder uma partida para um grande mestre, isso significaria um fracasso? Claro que é uma questão relativa, pois depende muito dos objetivos iniciais do projeto. Mesmo assim, porém, é difícil precisar o quão "bom" o método desenvolvido é, uma vez que depende, além dos objetivos a que se almeja, da qualidade dos adversários (dentro daquele contexto de jogo) e também do próprio andamento da partida em si. Afinal, ainda no exemplo do xadrez, perder uma partida em 3 lances é bem diferente de perder uma em 60 lances! E, mais ainda, perder duas partidas numa mesma quantidade x qualquer de lances pode ter significados completamente diferentes, dependendo do andamento das mesmas, do desempenho do programa, do adversário em cada uma etc.

A pesquisa de métodos de inteligência aproximativos em jogos é particularmente convidativa, afinal, a diversidade e a quantidade de estratégias permitem um campo de possibilidades enorme. Até porque se, por um lado, existem jogos para os quais se pode provar que há uma tática vencedora (VENKATARAMAN, 2000), para muitos outros isso não é possível. Um exemplo banal seria o "par ou ímpar". É extremamente óbvio, nesse caso, a impossibilidade de determinar uma estratégia que vença sempre. Jogos que se passem dentro de um ambiente complexo, com muitas possibilidades e em tempo real têm uma quantidade tão grande de possibilidades que, mesmo se houvesse um método direto, este seria computacionalmente inviável, como no caso do problema do caixeiro viajante. E a chance de simular esses ambientes complexos em que diversas estratégias podem coexistir é algo extremamente encorajador para pesquisas, principalmente no tocante a métodos adaptativos e evolutivos.

Um outro fator interessante é que a ausência de uma "melhor" estratégia pode levar a um cenário em que cada participante do jogo possui um leque de alternativas para as ações que vai desempenhar e a escolha depende fundamentalmente daquilo que seu(s) adversário(s) está(ão) fazendo. Voltando ao exemplo banal do "par ou ímpar", se o meu adversário sempre escolhe um número par (e ele ganha com número pares), então, se eu escolher sempre ímpar, eu ganho. Claro que esse é um exemplo extremamente simples, mas dá uma idéia das possibilidades, afinal, num jogo muito mais complexo, diversas estratégias conflitantes podem surgir e ser usadas. E o computador pode ser testado de forma impiedosa, já que a criatividade humana é enorme o suficiente para sempre criar novas formas de derrotar o adversário dentro

daquele contexto.

1.3 Objetivos

O principal objetivo deste trabalho é criar modelos de inteligência que evoluam em tempo real dentro das condições de jogo, isto é, de acordo com o adversário. Ou, de uma forma mais simples, criar uma série de métodos adaptativos e evolutivos em tempo real para jogos.

Há duas características fundamentais nesse objetivo. Pretendemos, em primeiro lugar, apresentar métodos que possam se adaptar em tempo real. Ou seja, que evoluam durante o jogo, aproveitando dados obtidos nesse contexto para tal. E, em segundo lugar, que essa adaptação seja dependente do adversário. Logo, não desejamos chegar ao melhor jogador possível universal para um determinado jogo. Procuramos, ao invés disso, desenvolver o melhor adversário possível **num determinado momento contra um determinado jogador**.

O conceito de "melhor" adversário não significa, necessariamente, que ele é imbatível. Entendemos como "melhor" adversário aquele que, em geral, está um nível um pouco acima do jogador, mas não tão acima que seja impossível derrotá-lo, apenas o suficiente que exija uma melhora do jogador para vencer.

O desenvolvimento do trabalho visa a aproveitar os jogos de ação como forma de propor e desenvolver os métodos. A definição mais precisa do termo "jogo de ação" é dada na seção 3.1. De forma simples, porém, podemos considerar como jogos de ação aqueles que possuem ação em tempo real e, dessa forma, são mais propícios para os objetivos deste trabalho, apresentados nesta seção.

Todos os métodos foram desenvolvidos com o intuito de criar estratégias que permitissem que, de alguma forma, o jogo pudesse se adaptar (e evoluir) de acordo com o jogador. Sendo assim, todos os modelos apresentados no capítulo 4, que são o núcleo deste trabalho, têm o objetivo principal de conseguir ou viabilizar, de alguma forma, a adaptação do jogo de acordo com o nível de habilidade do jogador, visando, assim, a alcançar o objetivo principal descrito no início desta seção.

1.4 Motivações

Motivações para este estudo não faltam. Uma delas, certamente, é mostrar a importância que a pesquisa em jogos eletrônicos pode adquirir no âmbito acadêmico e que é possível, sim, pesquisar com seriedade nesse campo.

O ambiente de um jogo, além disso, propicia uma enorme gama de possibilidades para aplicações de estratégias e algoritmos, sendo, igualmente, uma boa fonte de inspiração para a criação de métodos inteligentes.

O desafio de se enfrentar um ser humano é, também, especialmente atraente, uma vez que os métodos desenvolvidos estarão sendo testados contra a mais capaz e avançada das formas de inteligência que conhecemos. Sem contar que a própria natureza progressiva do aprendizado humano se verifica dentro do contexto dos jogos, podendo ser mais um parâmetro de comparação para o trabalho.

Acrescente-se a tudo isso o fato de se trabalhar com a junção de dois assuntos tão fascinantes e desafiadores como a programação de jogos e a criação de métodos adaptativos e evolutivos de inteligência. Temos, portanto, motivos de sobra para acreditar que pesquisas nesses campos podem (e devem) ser muito mais exploradas nos próximos anos, de forma a alcançar técnicas com níveis de sofisticação cada vez maiores, que, certamente, serão úteis não só para o entretenimento mas também para as mais diversas áreas do conhecimento humano.

1.5 Organização da Dissertação

Esta dissertação está divida em 6 capítulos. O capítulo 2 trata dos modelos e técnicas de lógica nebulosa e algoritmos evolutivos, que foram utilizados como base de conhecimento para a criação dos métodos adaptativos deste trabalho. Os conceitos fundamentais de cada um são descritos, bem como uma análise de suas vantagens e desvantagens, com destaque para o contexto deste trabalho.

O terceiro capítulo trata, especificamente, de jogos e da questão da inteligência a eles aplicada, quais as técnicas normalmente utilizadas e suas descrições, bem como o emprego que geralmente se faz das mesmas. Algumas ponderações são colocadas a respeito do uso de técnicas inteligentes pelos desenvolvedores e um apanhado geral do que se tem realizado no campo. Uma breve discussão sobre assuntos éticos também é apresentada, evitando-se, contudo, um aprofundamento maior da questão que fuja

ao escopo desta dissertação.

No capítulo 4 são apresentados os modelos teóricos propostos. Cada método desenvolvido é abordado, descrito e analisado. Os conceitos teóricos expostos nos capítulos anteriores são utilizados aqui de forma a moldar as técnicas e conferir-lhes uma base a partir da qual são desenvolvidos.

O capítulo seguinte trata da implementação dos métodos propostos e dos jogos que foram desenvolvidos como forma de utilizar as técnicas criadas. Alguns resultados experimentais são apresentados, bem como uma análise aprofundada dos comportamentos das estratégias e uma avaliação das mesmas baseada na interpretação dos dados obtidos.

O capítulo 6 traz as conclusões do trabalho, seguindo-se a elas a análise do trabalho realizado como um todo, dificuldades e problemas enfrentados, sugestões de trabalhos futuros como prosseguimento a este e algumas considerações finais.

Capítulo 2

Modelos Inteligentes

"A realidade é uma mera ilusão, ainda que bastante persistente."
- Albert Einstein

Neste capítulo, revisaremos dois modelos inteligentes, bastante famosos na literatura, os quais serão a base de que irão partir todos os métodos desenvolvidos e estudados no presente trabalho.

Tratar problemas cujo espaço de busca é exponencialmente grande ou para os quais soluções diretas são impraticáveis ou impossíveis requer o uso de modelos que permitam trabalhar tanto com incerteza quanto com imprecisão. Esse relaxamento faz com que haja introdução de conhecimento heurístico para a solução dos problemas, o que pode ser uma desvantagem.

2.1 Lógica Nebulosa

A lógica nebulosa (ou *lógica fuzzy*), de maneira geral, é um modelo capaz de generalizar a lógica clássica binária de forma a ser aproveitada em situações em que há um alto grau de incerteza.

Depois de anos de muita controvérsia, a lógica nebulosa, enfim, a partir do final da década de 80, passou a ser aceita como uma poderosa ferramenta para solução de diversos tipos de problema, graças ao sucesso de sua aplicação em várias áreas (YEN, LANGARI, 1999).

O modelo lógico tradicional, amplamente utilizado em toda a Matemática Clássica, tem se mostrado incapaz de modelar sistemas reais complexos, para os quais não

é possível obter equações que os descrevam satisfatoriamente, ou casos em que a Teoria dos Conjuntos é insuficiente para tal.

Em meados da década de 20, uma descoberta de Werner Heisenberg, que surpreendeu toda a comunidade científica, teve uma grande importância para o futuro desenvolvimento da teoria nebulosa: o princípio da incerteza. Colocando de forma simples, trata-se da constatação de que, mesmo que se tenha total controle acerca das informações pertinentes a um determinado evento, ainda assim não se pode chegar a 100% de certeza (KOSKO, 1993). Fica claro que esta nova visão coloca em cheque a verdade absoluta da lógica tradicional, nos fazendo refletir a respeito do que pode ser considerado verdade (ou não), ou quão certos estamos ao afirmar isso e se não seria natural admitirmos algo como parcialmente verdadeiro. No caso da álgebra booleana, é importante salientar que toda a sua teoria se baseia no falso e no verdadeiro, ambos absolutos e inquestionáveis.

A Albert Einstein é atribuído o pensamento "Quando as leis da matemática referem-se à realidade, elas não estão certas. Quando estas leis estão certas, elas não se referem à realidade", o que seria uma constatação da incapacidade intrínseca da lógica tradicional de lidar com problemas realistas.

Durante a década de 60, o professor Lotfi Zadeh, da Universidade da Califórnia, introduziu a teoria de uma lógica mais flexível, a que ele denominou "Fuzzy Logic", ou seja, "Lógica Nebulosa" ou "Lógica Difusa" (ZADEH, 1965). Os novos conceitos por ele introduzidos, na verdade, constituem um novo conjunto o qual abrange a Lógica Booleana convencional, fazendo uso do conceito de verdade parcial, valores entre totalmente verdade e totalmente falso.

É também de sua autoria o princípio da incompatibilidade, assim formulado: *Na medida em que a complexidade de um sistema aumenta, nossa habilidade para fazer afirmações precisas e que sejam significativas acerca deste sistema diminui até que um limiar é atingido, além do qual precisão e significância (ou relevância) tornam-se quase que características mutuamente exclusivas.*

Esta nova teoria torna mais realista a modelagem de sistemas que possuam características vagas (ou difusas) e que necessitem da inclusão de termos pouco precisos, como, por exemplo, "alto", "rápido" e "frio".

2.1.1 Variáveis e Conjuntos

A Teoria dos Conjuntos clássica fornece todas as regras para pertinências de conjuntos e a relação entre os mesmos, utilizadas em toda a Matemática. Essas relações são definidas através de uma função de avaliação que nos diz se um dado elemento pertence ou não a um determinado conjunto. Ou seja, ela recebe um elemento x e um conjunto X e retorna 0 (falso) se $x \notin X$ ou 1 (verdadeiro) caso $x \in X$.

Este tipo de definição nos expõe a deficiências claras, ainda mais quando tratamos de conjuntos em que a incerteza é um fator importante. Mais do que isso, o limiar entre pertencer ou não a uma dada definição é muito rígido, o que pode nos trazer problemas de perda de informação. Ao tratarmos os conjuntos com regras de relação e pertinência nebulosas, passamos a definir o retorno da função citada anteriormente como um número real no intervalo $[0, 1]$. Ou seja, ao invés de termos 0 **ou** 1, temos qualquer valor **entre** 0 e 1. Esta função denominamos de "função de inclusão" (ou pertinência) e o valor como "grau de inclusão" (ou pertinência). A figura 2.1 mostra uma comparação simples de dois possíveis gráficos para funções de pertinência entre a lógica tradicional e a nebulosa.

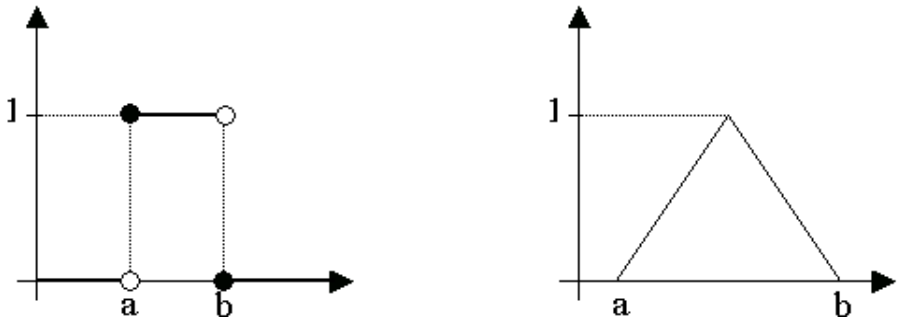

Figura 2.1: Lógica Booleana *versus* Lógica Nebulosa

Dessa forma temos uma avaliação mais sutil e flexível da pertinência de um elemento a um certo conjunto. Podemos, agora, fazer uma definição mais formal acerca dos conjuntos nebulosos e suas relações com os respectivos elementos. Assim sendo, seja uma coleção de objetos indicados genericamente por X. Então um **conjunto nebuloso** A em X é um conjunto de pares ordenados: $A = \{(x, \mu_A(x))|x \in X\}$ onde μ_A é a chamada função de pertinência e $\mu_A(x)$ é o grau de pertinência de x em A.

Existem diversos tipos de funções que podem ser utilizadas e sua escolha está

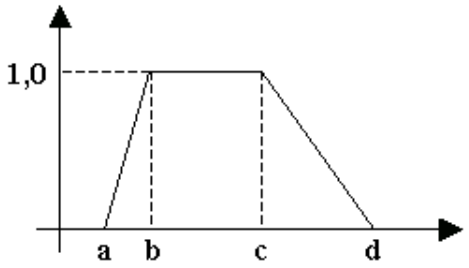

Figura 2.2: Gráfico da função trapézio

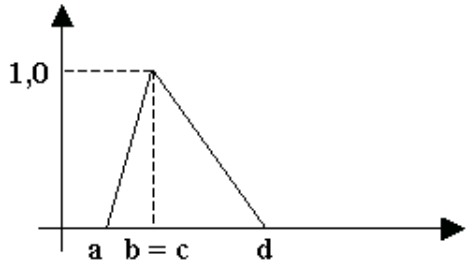

Figura 2.3: Gráfico da função triângulo

diretamente relacionada com a complexidade do modelo em questão e com o seu respectivo custo computacional. Em outras palavras, quanto mais complexa a função (por exemplo, não linear), maior será o tempo gasto para processá-la, de forma que temos uma relação inversamente proporcional entre esses dois fatores.

As mais comuns, entre as funções lineares, são as triangulares e as trapezoidais. Ambas podem ser representadas usando-se apenas quatro pontos a, b, c e $d \in \Re$. Observe-se que a função triangular é, na verdade, uma trapezoidal degenerada, com $b = c$. Os gráficos dessas funções podem ser vistos nas figuras 2.2 e 2.3.

A função trapezoidal, dados os pontos $a < b < c < d$, portanto, é definida como:

$$\begin{aligned} \mu_A(x) &= \frac{a-x}{a-b}, a \leq x \leq b \\ \mu_A(x) &= 1, b \leq x \leq c \\ \mu_A(x) &= \frac{d-x}{d-c}, c \leq x \leq d \\ \mu_A(x) &= 0, x \notin [a,d] \end{aligned} \tag{2.1}$$

Uma **variável nebulosa**, que tem um nome simbólico que a caracteriza, é definida sobre um universo de discurso e subdividida em diversos conjuntos nebulosos, (ou **rótulos**).

O nome simbólico nada mais é do que uma identificação que usamos para distinguir a variável, como, por exemplo "temperatura". O universo de discurso é o mesmo que domínio, e, no caso da temperatura, poderia estar, por exemplo, entre -20^oC e 45^oC. Este domínio pode ser subdividido em diversos conjuntos nebulosos,

aos quais damos o nome de rótulos, tais como, por exemplo, muito frio (entre -20^oC e 5^oC), frio (entre -5^oC e 17^oC), ameno (entre 15^oC e 27^oC), quente (entre 25^oC e 35^oC) e muito quente (31^oC e 45^oC). A figura 2.4 mostra uma representação gráfica da variável temperatura, com os respectivos conjuntos nebulosos.

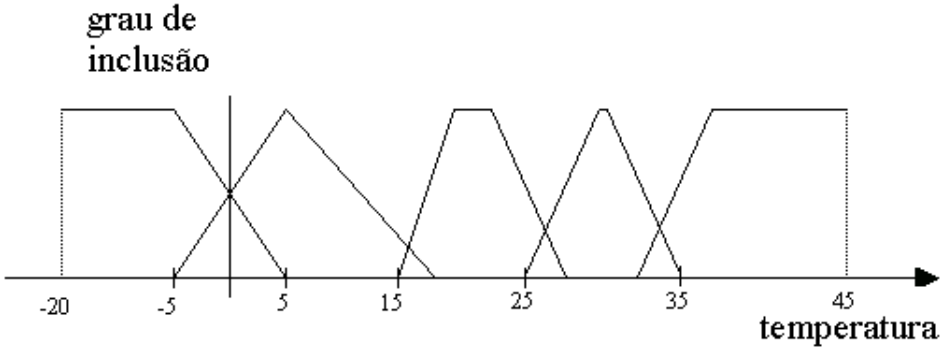

Figura 2.4: Representação gráfica da variável nebulosa temperatura

Vale notar que os rótulos não são, necessariamente, mutuamente exclusivos. Esta, aliás, é uma das principais características da lógica nebulosa e que representa uma grande vantagem em relação à tradicional, já que uma determinada variável pode assumir valores simultâneos em diferentes conjuntos. Ou seja, um mesmo elemento x pode ter valores de $\mu_A(x)$ maiores que zero para vários conjuntos.

Além disso, encontramos novas interpretações para as operações elementares da Teoria dos Conjuntos, tais como união, interseção e negação (ou complemento). Suponhamos três conjuntos A, B e C. Usando a lógica tradicional, se fizermos $C = A \cup B$, C será o conjunto cujos elementos são todos aqueles que pertencem a A **ou** a B. Por outro lado, $C = A \cap B$, faria com que C fosse formado por todos os elementos que pertencem a A **e** a B **ao mesmo tempo**. Finalmente, $C = \neg A$ atribui a C todos os elementos do universo considerado que não pertencem a A.

Definindo mais formalmente, temos:

$C = \{\forall x | x \in A \vee x \in B\}$, para $C = A \cup B$
$C = \{\forall x | x \in A \wedge x \in B\}$, para $C = A \cap B$
$C = \{\forall x | x \notin A\}$, para $C = \neg A$

Estes conceitos, mesmo expostos rapidamente, nos fazem perceber o quanto limitam o potencial de aplicações que os utilizem, mesmo porque fazem uso de relações de pertinência puramente binárias (pertence ou não).

Estas operações com conjuntos nebulosos são definidas com base nas funções de inclusão. A função de inclusão $\mu_C(x)$ da união dos conjuntos A e B ($C = A \cup B$) é definida, segundo Zadeh, como $\mu_C(x) = max(\mu_A(x), \mu_B(x))$, $x \in X$ (onde X tem o mesmo significado anteriormente utilizado). A interseção entre os conjuntos A e B ($C = A \cap B$) é definida, ainda segundo Zadeh, através da função de inclusão $\mu_C(x) = min(\mu_A(x), \mu_B(x))$, $x \in X$. Por fim, temos a função de inclusão do complemento, cuja definição é $\mu_C(x) = 1 - \mu_A(x)$, $x \in X$.

Fica claro que, através da definição da negação nebulosa, é possível para um elemento estar ao mesmo tempo num conjunto e no seu complemento, o que seria uma contradição pela lógica clássica, na qual $A \cap \neg A = \emptyset$, ou seja, um elemento está num conjunto A **ou** em sua negação. No caso da lógica nebulosa, isso só vai acontecer se $\mu_A(x) = 0$ ou $\mu_A(x) = 1$. Sendo k um número real no intervalo aberto $(0, 1)$, teremos $\mu_A(x) = k \Rightarrow \mu_{\neg A}(x) = 1 - k$ e como $k \neq 1$ e $k \neq 0$, então ambos os graus de pertinência serão maiores que zero, o que significa dizer que o elemento x pertence parcialmente ao conjunto A e ao seu complemento, simultaneamente.

Outras definições para as operações de união e interseção também podem ser encontradas. Listamos, na tabela 2.1, algumas delas, como descritas em (COX, 1994).

$A \cap B$	$A \cup B$
$min(\mu_A(x), \mu_B(x))$	$max(\mu_A(x), \mu_B(x))$
$(\mu_A(x) + \mu_B(x))/2$	$(2 \times min(\mu_A(x), \mu_B(x)) + 4 \times max(\mu_A(x), \mu_B(x)))/$
$\mu_A(x) \times \mu_B(x)$	$(\mu_A(x) + \mu_B(x)) - (\mu_A(x) \times \mu_B(x))$
$max(0, \mu_A(x) + \mu_B(x) - 1)$	$min(1, \mu_A(x) + \mu_B(x))$
$1 - min(1, ((1 - \mu_A(x))^k + (1 - \mu_B(x))^k)^{1/k})$	$min(1, (\mu_A(x)^k + \mu_B(x)^k)^{1/k})$

Tabela 2.1: Exemplos de funções de pertinência para operações de conjuntos nebulosos

2.1.2 Regras

Uma regra pode ser interpretada como uma seqüência de operações nebulosas sobre graus de pertinência de variáveis em conjuntos com uma conseqüência igualmente nebulosa.

Se dizemos que "x é X", estamos nos referindo ao grau de pertinência de x no conjunto nebuloso X, ou seja, $\mu_X(x)$. Ao usarmos proposições da forma "se-então"

para representar as regras, então os antecedentes podem ser formados por diversas operações de interseção e união de variáveis sobre conjuntos, com uma conseqüência de uma variável sobre um outro conjunto. Uma regra, então, poderia ser algo do tipo: **se** a é A **e** (b é B **ou** c é C) **então** d é D.

De certa forma, a estrutura de uma proposição condicional nebulosa é muito parecida com a da lógica clássica, apenas trocando os sinais $<$, $>$ e $=$ por um verbo. A forma como a regra é interpretada, porém, difere bastante.

No caso da lógica tradicional, o conseqüente apenas se verifica, ou é inferido, se a operação sobre os antecedentes for verdadeira. Por exemplo, no caso de uma regra formada apenas com **e** (interseção), todos os antecedentes têm de ser verdadeiros, enquanto que numa formada apenas com **ou** (união), pelo menos um tem de ser verdadeiro.

Para as regras nebulosas, esse cenário muda bastante. Interpretando, como já vimos, cada antecedente como o grau de pertinência dentro do respectivo conjunto, podemos aplicar as operações sobre conjuntos nebulosos já vistas para calcular o grau de pertinência de todo o antecedente. No caso da regra dada como exemplo acima, teríamos $\mu_A(a) \cap (\mu_B(b) \cup \mu_C(c))$. Então, por exemplo, se estamos usando as definições de mínimo e máximo para união e interseção e temos $\mu_A(a) = 0,7$, $\mu_B(b) = 0,66$ e $\mu_C(c) = 0,25$, teríamos, como avaliação dos antecedentes $\mu_A(a) \cap (\mu_B(b) \cup \mu_C(c)) = min(0,7, max(0,66, 0,25)) = min(0,7, 0,66) = 0,66$.

O valor calculado para os antecedentes é o grau de pertinência que o conseqüente passa a ter em relação ao conjunto de saída. Assim, ainda segundo o exemplo de regra dado, teríamos $\mu_D(d) = 0,66$ como resultado da avaliação.

2.1.3 Sistemas Nebulosos

Um sistema nebuloso, posto de forma simples, é formado por uma série de definições de variáveis, conjuntos e regras nebulosas e segue um processo de recepção e tratamento dos dados de entrada dentro desse contexto para poder processar a saída.

Há várias formas de encarar sistemas nebulosos, algumas mais simples, outras mais complexas (YEN, LANGARI, 1999). Diferentes modelos existem e iremos descrever, nesta seção, a forma mais simples que, inclusive, é a que utilizaremos ao longo deste trabalho, sempre que estiver presente alguma forma de sistema nebuloso.

Neste caso, temos dois tipos de variáveis nebulosas que representam os dados de entrada e saída do sistema. As entradas aparecem nos antecedentes de regras,

enquanto que as saídas nos conseqüentes.

Dadas as definições de regras, variáveis e conjuntos, e os valores de entrada, o primeiro passo é a **nebulização** (*fuzzyfication*) desses dados. Esse processo nada mais é do que a avaliação dos valores numéricos da entrada dentro dos conjuntos nebulosos das respectivas variáveis. Assim, por exemplo, um sistema que tem como entrada a velocidade de um automóvel, em km/h, teria de transformar esse valor em graus de pertinência nos respectivos conjuntos que poderiam ser, digamos, "lento", "médio" e "rápido".

Com os dados devidamente representados nebulosamente, podemos aplicar as regras para inferir as saídas. Assim, uma série de regras, tais como descritas na seção 2.1.2, são avaliadas. Quando houver mais de uma avaliação do mesmo conseqüente para o mesmo conjunto, o que é comum, sempre usaremos o grau de pertinência mais alto. Observe-se que uma mesma variável de saída pode ter graus de pertinência maiores que zero em mais de um conjunto, possivelmente até em todos.

Depois de avaliadas as regras com os devidos graus de pertinência atribuídos às variáveis de saída, temos de traduzir esses dados para valores de saída que possam ser aproveitados pelo sistema, operação que chamamos de **desnebulização** (*defuzzyfication*). Se, por exemplo, a saída do sistema é quantos graus um determinado elemento deve girar em torno do próprio eixo, de nada adianta sabermos que seu grau de pertinência no conjunto "girar muito" é $0,8$, "girar médio" é $0,5$ e "girar pouco" é $0,2$. Precisamos ter um valor, em graus, que represente essa avaliação das regras, e é isso que o processo de desnebulização faz.

Uma das formas de fazer a desnebulização é usando o centro de massa. Para a conclusão de que "y é Y", teríamos, para alguma divisão discreta do espaço:

$$y = \frac{\sum \mu_Y(y_i) \times y_i}{\sum \mu_Y(y_i)} \tag{2.2}$$

Ou seja, a fórmula (2.2) utiliza uma partição do espaço para aproximar o cálculo do centro de massa. No caso contínuo nós temos:

$$y = \frac{\int \mu_Y(y_i) \times y_i dy}{\int \mu_Y(y_i) dy} \tag{2.3}$$

Passamos a ter, após a desnebulização, um valor numérico que representa a nossa saída, e este pode ser usado pelo sistema.

2.2 Algoritmos Genéticos

Algoritmos Genéticos (AGs) constituem uma ferramenta poderosa com aplicação em problemas complexos cujos espaços de busca das soluções ótimas podem ser absurdamente grandes para que seja possível determiná-las com precisão através de um método direto. Essas soluções ótimas, em alguns casos, nem sequer existem e, muitas vezes, o que procuramos é meramente uma aproximação que nos dê um resultado satisfatório dentro do contexto do problema.

De forma geral, os algoritmos genéticos são um método estocástico de otimização baseados nos conceitos de seleção natural e processos evolucionários, sendo inicialmente propostos por John Holland, da Universidade de Michigan, em 1975 (JANG, 1997). Sua natureza é essencialmente empírica, e, de certa forma, podemos encarar os AGs como uma busca aleatória heurística.

A idéia inicial é tratar cada possível solução para um determinado problema como um **indivíduo**. Cada indivíduo é codificado como uma série de bits, denominada **cromossomo**, ou código genético. Cada parte desse código genético pode ser chamada de **gene**.

Um determinado indivíduo é avaliado, dentro do contexto do problema, por uma **função de aptidão** (*fitness*), ou função de avaliação, a qual determina quão apto o indivíduo é.

O conjunto de indivíduos é chamado de **população**. Cada população diferente constitui uma **geração**, e, à medida que novas gerações vão substituindo as antigas, a tendência é que as soluções se aproximem do ótimo procurado, ou seja, que os indivíduos das novas gerações sejam mais aptos. A população que forma a primeira geração é, costumeiramente, totalmente aleatória, ou seja, os cromossomos são criados aleatoriamente.

Os indivíduos mais aptos têm maior chance de sobreviver e passar adiante seu código genético para as gerações futuras. Aplicam-se, para criar uma nova geração, diversos operadores genéticos, como **cruzamento** (*crossover*), **elitismo** e **mutação**.

Temos, normalmente, como condição de parada para o algoritmo genético, um limite de gerações e/ou um determinado valor mínimo de aptidão a ser alcançado. O algoritmo 2.1 descreve, de maneira muito genérica, a forma básica dos AGs:

Algoritmo 2.1 Algoritmos Genéticos

```
gerar população inicial
para geracao de 1 até max_geracao faça
    para i de 1 até total_individuos faça
        avaliar indivíduo i
    fim para
    aplicar operadores genéticos para formar nova geração
fim para
```

2.2.1 Operadores Genéticos

Depois de avaliados os indivíduos, é necessário aplicar operadores genéticos de maneira a formar a geração seguinte.

Um operador simples é o **elitismo**, o qual simplesmente replica os melhores indivíduos da geração atual para a seguinte. Se temos, então, n indivíduos por geração e uma proporção de elitismo de 25%, então um quarto da nova geração será formada pelos n / 4 melhores indivíduos da atual.

O operador mais comum, porém, é o **cruzamento** (*crossover*), que simula a reprodução sexuada de espécies. A idéia básica é selecionar dois indivíduos da geração e cruzar o código genético de ambos, criando dois novos indivíduos para a geração seguinte.

A forma mais comum de seleção é o **método da roleta**, no qual cada indivíduo tem uma chance de ser escolhido proporcional à sua contribuição para a aptidão total da população. Sendo f a função de avaliação, a chance p_i de cada indivíduo i em ser selecionado, portanto, é:

$$p_i = \frac{f(i)}{\sum_j f(j)} \tag{2.4}$$

Se temos, por exemplo, quatro indivíduos com aptidões de 10, 17, 9 e 14, a chance de cada um ser selecionado será, respectivamente, $0,20$, $0,34$, $0,18$ e $0,28$. Ainda que os mais aptos tenham maior probabilidade de ser escolhidos, este método permite que mesmo os menos aptos tenham uma chance, embora pequena, de ser selecionados.

Escolhidos dois pais, aplica-se o cruzamento entre eles para formar os novos indivíduos. Há várias formas de fazer isso, e listaremos algumas principais aqui. As mais comuns são o cruzamento em um ou dois pontos. No primeiro caso, sorteia-se

um ponto no qual haverá a troca de cromossomos entre os pais. Então, por exemplo, se os indivíduos escolhidos têm código genético 111111 e 000000, se for escolhido o quarto bit (da esquerda para a direita) como ponto de cruzamento, os filhos gerados serão 111100 e 000011. No segundo caso, são escolhidos dois pontos aleatórios para realizar a troca de genes. Então, usando os mesmos pais do exemplo anterior, se fossem sorteados o segundo e o quinto bits, os filhos seriam 110001 e 0011100. Claro que podemos estender esse raciocínio de forma a permitir o cruzamento para um número arbitrário de pontos, embora cruzamentos em mais de dois pontos não sejam normalmente usados.

Uma outra forma interessante de cruzamento é por máscara. Tendo uma seqüência de bits pré-definidos como uma máscara, a mesma é aplicada aos pais selecionados para o cruzamento, gerando os filhos. Cada bit da máscara indica de qual pai virá o gene da respectiva posição no filho. Assim, por exemplo, se os indivíduos selecionados forem 110011 e 100101, definindo a máscara como sendo 101010, os filhos gerados serão 100111 e 110001. Ou seja, para o primeiro filho, cada bit 1 da máscara significa aproveitar o gene do primeiro pai e 0 do segundo, enquanto que, para o segundo filho, o contrário se verifica. Pode-se variar este método usando máscaras diferentes para cada filho, um conjunto de máscaras com escolha aleatória da que será usada no cruzamento atual etc.

É possível, ainda, associar uma probabilidade de ocorrer o cruzamento, depois de selecionados os pais. Se definirmos, assim, essa chance como sendo p, então numa proporção p dos casos o cruzamento será feito normalmente, enquanto que, proporcionalmente a $1-p$ dos casos, os filhos gerados serão simplesmente os próprios pai replicados.

Outro operador muito comum é a **mutação**, aplicada, em geral, após a nova geração ser formada e antes de a mesma ser avaliada. A este operador é associada uma taxa, que é a freqüência em que ela irá acontecer. A forma mais comum de mutação é a **troca de bit** (*bit flip*), que é calculada sobre cada bit, e sua conseqüência é negá-lo. Se temos, então, uma taxa de mutação de 1%, então 1 em cada 100 bits será trocado. Uma outra maneira, mais drástica, de aplicar a mutação é associar a taxa por indivíduo e, como conseqüência, redeterminar aleatoriamente todos os seus genes. Então, por exemplo, se a taxa de mutação é 1%, então 1 em cada 100 indivíduos de cada nova geração será refeito de maneira completamente aleatória.

Uma forma diferente de aplicar essa última mutação é definir uma proporção constante de cada nova geração formada apenas por indivíduos aleatórios. No caso da mutação, há uma chance de ocorrer, neste caso sempre ocorre.

Se tivermos uma população de 1000 indivíduos e definirmos as taxas de elitismo, cruzamento e indivíduos aleatórios como 0.15, 0.80 e 0.05 respectivamente, teremos, a cada geração, 150 indivíduos novos replicados dos 150 melhores da geração anterior, 800 por cruzamento entre 400 casais selecionados e 50 totalmente aleatórios. Sem contar que temos, ainda, as probabilidades de cruzamento e mutação.

Claro que há diversas maneiras possíveis de se codificar os indivíduos que não apenas por seqüências de bits, podendo-se fazer com outras abstrações de dados (números inteiros, reais ou mesmo estruturas).

2.3 Algoritmos Coevolucionários

A idéia básica da coevolução é que duas ou mais subpopulações tendem a encontrar uma melhor solução quando competem ou cooperam entre si. Ou seja, quando a evolução de uma está intimamente ligada à das outras, a tendência é que todas caminhem rumo a uma melhor configuração para todas.

Há vários trabalhos publicados sobre algoritmos coevolutivos (ACEs) e já foi demonstrado que eles podem ser utilizados com sucesso em diversas áreas (ANGELINE, POLLACK, 1993; WIEGAND, LILES, JONG, 2001) e até mesmo em jogos (REYNOLDS, 1994).

Os algoritmos coevolucionários são geralmente definidos por suas funções de aptidão dependentes da interação entre um indivíduo e os demais. Tal interação pode ser **cooperativa**, o que significa que os indivíduos evoluem em direção a um objetivo comum, ou **competitiva**, o que significa que eles estão competindo entre si de forma a conquistar um determinado recurso.

Em resumo, como afirma Wiegand, algoritmos coevolucionários competitivos são mais adequados a problemas para os quais uma função de avaliação externa é muito difícil de determinar, mas que têm uma maneira natural de definir a aptidão com base no sucesso competitivo, enquanto que os cooperativos são mais adequados aos casos em que se pode decompor o problema naturalmente em subproblemas que coevoluem, baseando sua aptidão em quão bem os indivíduos de diferentes subpopulações trabalham em conjunto dentro do contexto geral (WIEGAND, LILES,

JONG, 2002).

2.3.1 Competição

A idéia básica da coevolução competitiva é que a competição direta entre espécies gera uma pressão evolutiva que é responsável pela melhora dos indivíduos. Dessa forma, a aptidão dentro de uma espécie é calculada justamente levando-se em conta seu desempenho em relação às demais, ou seja, é uma definição direta de acordo com a própria natureza do problema (FUNES et al, 1998).

Supondo que haja duas espécies coevoluindo, quando uma determinada população encontra uma estratégia que representa um ótimo local dentro do espaço de busca, a outra é pressionada a encontrar uma solução que se saia bem contra essa estratégia. Ao acontecer isso, a primeira população passa a ter uma estratégia que não é mais um ótimo local, pois já não está se saindo bem contra os adversários, e a situação se inverte. Esse tipo de situação é comparada a uma "corrida armamentista".

Um exemplo simples de competição seria a situação predador / presa. A aptidão das presas é definida como a sobrevivência à caça, e, assim, as que sobrevivem são as que se reproduzem e, portanto, passam seu código genético adiante. Por outro lado, os predadores mais aptos são aqueles que conseguem capturar mais presas e sobrevivem (os que não conseguem, morrem de fome) e também passam adiante seu código genético. A aptidão de ambas as espécies, portanto, é definida pela própria natureza do problema. Se as presas desenvolverem uma estratégia melhor, os predadores tenderão a morrer mais, pois menos vezes conseguirão caçar com sucesso. Quando uma nova estratégia surgir entre os predadores que seja melhor que a das presas, os predadores com esse código genético tenderão a dominar a espécie, pois serão os mais aptos dentro do ambiente, passando com mais freqüência seu código genético. A situação continua dessa forma, com ambas as espécies alternando momentos de supremacia em relação à outra.

Um dos grandes problemas da coevolução competitiva é o chamado *Efeito da Rainha Vermelha* (CLIFF, MILLER, 1995). A Rainha Vermelha é um personagem do livro *Through The Looking Glass*, de Lewis Carroll, que fica correndo sem sair do lugar, apenas a paisagem se move. No caso da coevolução, esse efeito é responsável pela dificuldade em se determinar com precisão o progresso das espécies, uma vez que a aptidão dos indivíduos de uma população é definida, naturalmente, em função

de seu desempenho contra os das demais. Então, por exemplo, dadas duas espécies A e B, se a aptidão média de seus indivíduos é praticamente a mesma na 1^a e na 100^a gerações, podemos afirmar que não houve evolução? O Efeito da Rainha Vermelha já foi investigado em diversas situações e formas de tentar contornar esse problema foram propostas (CLIFF, MILLER, 1995; CLIFF, MILLER, 1996). Floreano mostra que esse efeito pode resultar em "ciclos" entre classes de estratégias que se alternam e tornar bastante difícil a tarefa de monitar o progresso evolucionário (FLOREANO, NOLFI, MONDADA, 1998; ØSTERGåRD, 2000).

Para casos, porém, em que a melhor estratégia de uma espécie depende justamente da adotada por outra, a coevolução competitiva, como já foi dito, tende a ser uma excelente opção, não só pela definição intuitiva da aptidão, como pela natureza dinâmica do problema.

2.3.2 Cooperação

Ao contrário da competição, os algoritmos coevolucionários cooperativos buscam uma simbiose entre as subespécies, de forma que, coexistindo, elas tendam a encontrar uma solução geral que, mesmo que para cada uma individualmente não seja muito boa, quando consideradas em conjunto constituem uma solução melhor.

A principal característica, neste caso, é a subdivisão do problema principal em subproblemas menores, representados pelas subespécies, com um certo grau de dependência entre si. Um exemplo muito simples seria otimizar uma função de três variáveis $f(x, y, z)$, em cujo caso haveria três subproblemas distintos (encontrar os valores de x, y e z), porém dependentes entre si (otimizar a função).

Quase tudo acontece, na coevolução cooperativa, separadamente para cada subespécie, como se fosse um algoritmo genético tradicional. A diferença está na avaliação dos indivíduos, uma vez que para o cálculo da aptidão são necessários representantes das demais subespécies, geralmente tomados da geração anterior (apenas para esse fim). No exemplo da otimização de $f(x, y, z)$, para avaliar um indivíduo de x, é necessário usar um indivíduo de y e um de z para isso, os quais são escolhidos entre aqueles que formaram a geração anterior nas respectivas subespécies. Claro que na primeira geração não há anterior e, portanto, esses indivíduos auxiliares para avaliação podem ser meramente aleatórios.

Como há, portanto, uma interdependência entre as espécies, a avaliação dos indivíduos de uma não pode ocorrer sem levar em conta as demais. O problema, então,

passa a ser como selecionar esses indivíduos e quantos escolher. Afinal, um representante de cada outra população é o suficiente para poder avaliar um indivíduo dentro do contexto geral? Se sim, como escolher esse representante, apenas aproveitando o mais apto da geração anterior ou fazendo-o de maneira aleatória? Se não, quantos escolher e como fazer a avaliação usando mais de um exemplar por cada outra espécie (média simples, ponderada, maior valor etc.)? Há vários estudos a respeito de como fazer essa seleção, quantos escolher, como calcular a função de aptidão para mais de uma avaliação (usando mais de um exemplar das demais populações), bem como diversos fatores, dependendo da natureza do problema o qual se deseja resolver (WIEGAND, LILES, JONG, 2001).

Capítulo 3

Jogos e Inteligência

Para alcançar o objetivo principal deste trabalho, que é evoluir de acordo com o adversário, como definido na seção 1.3, precisamos, inicialmente, investigar os métodos usualmente utilizados em jogos eletrônicos para modelar a inteligência dos mesmos.

O uso de métodos de inteligência artificial em jogos é algo relativamente antigo e algumas técnicas tradicionais vêm sendo utilizadas há bastante tempo para a criação da inteligência dos jogos. Na maioria dos casos, todavia, grande parte do tempo e esforço da produção são gastos com a parte gráfica e visual, como já observara Rouse em 1998 a respeito da ânsia dos desenvolvedores de criar ambientes em 3D e aproveitar toda a tecnologia para a criação gráfica (ROUSE, 1998). Essa tendência, felizmente, já se vem revertendo, como observava Woodcock em 2000 num artigo para a *Game Developer Magazine* (WOODCOCK, 2000b). Mesmo assim, um bom número de desenvolvedores prefere usar implementações diretas e pouco trabalhadas de métodos convencionais (os quais veremos neste capítulo), ao invés de se preocupar em desenvolver um pouco mais esse campo e criar algo que realmente desafie o jogador de alguma forma. Como observa Tozour, freqüentemente a inteligência de um jogo é deixada para a última hora, feita quase que no improviso, embora também afirme que, ultimamente, muitos têm começado, finalmente, a levar esse aspecto a sério (TOZOUR, 2002).

Pior ainda, como veremos mais adiante, muitas vezes os jogadores acabam comprando "gato por lebre", pois muito do que é dito com estardalhaço como fazendo parte do jogo, na verdade não é exatamente o que foi cantado.

Antes de iniciar, porém, com as descrições dos métodos usuais de inteligência em jogos, vamos aproveitar para definir alguns termos que serão usados daqui em

diante, neste trabalho, de forma a não haver ambigüidades de interpretação nem falta de definições.

O objetivo principal deste capítulo, portanto, é apresentar os termos que serão usados com freqüência ao longo do resto deste trabalho, bem como fazer uma revisão dos métodos mais freqüentemente utilizados para a criação da inteligência de jogos eletrônicos *independente do tipo de jogo em que são aproveitados*. Não necessariamente tudo o que for apresentado na seção 3.2 será aproveitado ou comentado futuramente no trabalho. Alguns modelos aparecem, ainda assim, para que seja possível uma visão bem geral do que é feito na área e quais sãos os modelos normalmente utilizados.

3.1 Definições

3.1.1 Jogo

Definir um jogo é algo subjetivo. Nossa intuição nos tenta a considerar um jogo tudo aquilo que, de alguma forma, nos diverte.

Para Rollings e Morris, um jogo não é apenas várias coisas legais misturadas, um monte de gráficos bonitos, uma série de enigmas e uma boa história, pois de nada adianta ter isso tudo se não for divertido (ROLLINGS, MORRIS, 2000).

Já Battaiola define um jogo como formado por três partes básicas - enredo, motor (engine) e interface interativa - e que o sucesso do mesmo estaria na perfeita combinação de tais componentes (BATTAIOLA, 2000).

Como foge ao escopo deste trabalho uma discussão mais profunda a respeito da natureza dos jogos eletrônicos, vamos considerar uma decisão muito mais intuitiva e subjetiva do que uma mais técnica e formal. Entendemos, portanto, como jogo qualquer atividade que seja considerada divertida por aquele que a desempenha e, conseqüentemente, que um jogo eletrônico seja aquele que é jogado com o uso de algum equipamento eletrônico (como um computador).

Este trabalho se concentra, como foi colocado na seção 1.3, no desenvolvimento de métodos adaptativos e evolutivos para jogos de ação. Ao contrário de considerá-los como um gênero separado (que serão resumidos na seção 3.1.2), interpretamos como sendo um conjunto dos mesmos. Dessa forma, portanto, precisamos de uma definição separada do que estaremos considerando como jogos de ação para os objetivos deste trabalho.

Um jogo, dentro do contexto deste trabalho, é considerado como "de ação" quando a interação entre jogador, agentes e ambiente tem de acontecer em tempo real. Ou seja, não há turnos definidos entre os jogadores como acontece, por exemplo, num jogo normal de xadrez. Não há restrição *a priori* de movimentação, salvo alguma condição extraordinária, para quem quer que esteja jogando, seja ele humano ou não. De forma simples, um jogador não é obrigado a esperar a sua vez, ou o seu turno, para realizar alguma ação.

A não restrição ao movimento, sem o uso de turnos, é um grande complicador. Visto que não há obrigatoriedade de esperar sua vez, o jogador pode tomar a ação constantemente. Se o agente, portanto, gastar muito tempo "pensando", vai ter uma desvantagem considerável. Afinal, mesmo alguns segundos podem ser determinantes para uma derrota e, às vezes, mesmo uma fração de segundo pode ser fatal.

Nessas condições extremas desenvolvemos este trabalho.

3.1.2 Gêneros de Jogos

Dividir jogos eletrônicos em gêneros é, muitas vezes, uma tarefa discutível. Não são poucos os jogos que normalmente podem cair em mais de uma categoria definida ou mesmo não se adequar perfeitamente a nenhuma. Ainda assim pode ser interessante fazer a divisão até para facilitar as definições ou mesmo por questões meramente didáticas.

A divisão aqui apresentada é uma versão resumida das que estão presentes em (LAMOTHE, 1999; BATTAIOLA, 2000), com pequenas alterações. Claro que há várias outras formas diferentes possíveis de se fazer a divisão em estilos, mas decidimos por uma que seja simples e que permita uma visão geral do que é feito atualmente no campo.

- **FPS**: abreviação do inglês *First Person Shooters*. São jogos em que geralmente os objetivos não são exatamente claros ou importantes. Sua característica principal é a imersão dos jogadores num ambiente completamente 3D com visão em primeira pessoa. Caracterizam-se por serem normalmente violentos, e os jogadores estão em constantes conflitos, tendo de matar diversas criaturas ou mesmo outros jogadores.

- **Esportes**: jogos que simulam partidas e competições esportivas, como futebol, basquete, corridas etc.

- **Luta**: simulam uma espécie de arena na qual, em geral, dois lutadores se enfrentam com o objetivo de derrotar o adversário.

- **Plataforma**: normalmente são jogos em 2D que exigem muita perícia do jogador.

- **Estratégia**: requerem decisões de larga escala e lidam com simulações de mundos gigantescos e vários agentes. Possibilitam a criação de estratégias elaboradas e complexas dos jogadores, com o objetivo de derrotar os adversários ou apenas prosperar dentro do mundo virtual.

- **Simulação**: têm simulações complexas de ambientes reais. Simuladores de vôo e de carro são as formas mais famosas deste tipo de jogo.

- **Adventure**: têm como principal apelo histórias bem elaboradas e requerem bastante raciocínio para a solução de uma série de enigmas por parte do jogador. A perícia do jogador praticamente não importa e dificilmente os jogos requerem interações intensas em tempo real.

- **RPG**: abreviação do inglês *Role Playing Game*. A idéia é transportar os jogos tradicionais de RPG para o meio eletrônico, embora guardem alguma semelhança com o estilo adventure.

- **Quebra-cabeças e tabuleiro**: primam pelo raciocínio. Incluem-se jogos tradicionais, como xadrez e damas, e invenções mais novas, como Tetris.

- **Educativos**: levam em conta aspectos didáticos e pedagógicos no desenvolvimento do enredo, podendo ter outras características de vários outros gêneros.

3.1.3 Termos

Nesta seção apresentamos, de forma sucinta, boa parte dos termos usuais de um jogo (inclusive entre os jogadores).

- **Jogador**: é o jogador humano do jogo, ou seja, qualquer pessoa que esteja jogando. De forma indiscriminada iremos chamar de "jogador" tanto a pessoa que joga quanto o personagem que ela comanda.

- **Agente (ou NPC)**: é todo e qualquer elemento controlado pelo computador. Um agente, dessa forma, pode tanto ser um aliado quanto um adversário do jogador. Pode ser também chamado de *bot* (corruptela de *robot*), embora esse termo esteja mais associado aos agentes de jogos do gênero FPS.

- **Inimigo**: é qualquer adversário do jogador.

- **Fase**: muitos jogos são subdivididos em fases, que nada mais são do que trechos que costumam guardar alguma coerência dentro de si. Em alguns jogos (principalmente nos mais antigos), é necessário seguir uma ordem determinada das fases de forma a terminá-los. Em outros, porém, não há uma ordem fixa e o jogador pode seguir a que achar mais adequada (possivelmente dentro de alguma restrição). Também pode ser chamada de **estágio**, **nível** ou **mundo**.

- **Chefe de fase**: é o inimigo final de uma fase que deve ser enfrentado pelo jogador. Em geral é um agente bem mais difícil de ser batido que os demais da fase, e, muitas vezes, é necessário descobrir algum truque ou macete para poder derrotá-lo.

- **Vidas**: muitos jogos têm o conceito de "vidas", que, em última instância, representam o número de chances que o jogador tem de jogar antes que o jogo acabe. Sempre que ocorrem determinados eventos (como, por exemplo, ser atingido por um disparo), o jogador perde uma vida. Muitas vezes, quando o jogador "morre", ele recomeça do início da fase atual, embora em outros casos ele possa recomeçar do mesmo ponto onde "morreu".

- **Energia**: é um conceito muito parecido com o de vidas. Em geral, a energia é usada em conjunto com as vidas, com o jogador perdendo energia devido a determinados eventos, a qual, ao esgotar-se, acaba acarretando na perda de uma vida.

- **Partida**: pode ser considerada de várias formas. Uma delas seria um jogo completo, ou seja, a partida seria o que foi jogado do início do jogo até o seu fim. Pode, também, ser considerada como uma determinada fração do jogo como, por exemplo, jogar uma fase de seu início até o seu fim. Esta última definição não é usual entre os jogadores (que geralmente designam uma partida

como um jogo completo), porém nos é conveniente estender a definição geral hierarquicamente.

- **Vencer o jogo**: diz-se que um jogador "venceu o jogo" quando ele chega ao final do mesmo. Em geral isso significa passar por todas as fases (ou pelo menos um determinado número delas) e derrotar um chefe de fase final. Em outros casos, pode ser simplesmente cumprir um determinado objetivo, como, por exemplo, recuperar um objeto perdido ou resolver uma série de enigmas. Na maioria dos casos, é apresentada uma animação final de forma a concluir o enredo do jogo.

- **Turno**: o menor instante de tempo do jogo. Se o jogo tiver, por exemplo, n quadros por segundo, haverá n turnos de jogo por segundo. Sendo que, em cada turno, todos os jogadores podem realizar uma determinada ação, salvo alguma condição especial de jogo.

- **Jogada**: pode ser a mesma definição de partida. Também pode se referir a uma determinada ação (de qualquer jogador ou agente), ou turno.

- **Continue**: ao se esgotarem todas as vidas do jogador, em muitos jogos é dada a opção de "continue", que é permitir que mais uma partida seja iniciada do mesmo ponto em que a atual terminou (por exemplo, na mesma fase). A maioria dos jogos que têm essa opção permitem um número limitado de vezes que o jogador pode utilizá-la, enquanto outros não estipulam um limite (chamado de **continue infinito**).

- **Engine (motor)**: é o coração do jogo, como o *kernel* de um sistema operacional. A engine controla a ação do jogo e implementa as principais e mais importantes funções. Muitas vezes aquilo que estará presente no código depende muito do gênero do jogo, mas de certa forma o papel da engine é mais ou menos o mesmo, sendo, como dito, o núcleo da parte de programação.

3.2 Métodos Usuais

Vamos revisar, nesta seção, os métodos de inteligência mais comumente utilizados em jogos eletrônicos. Apesar de alguns desses métodos não nos interessarem para

este trabalho em particular, ainda assim os descrevemos, uma vez que o que nos propomos nesta seção é, justamente, uma revisão genérica sobre tais modelos.

Falar a respeito de técnicas de inteligência aplicadas em jogos pode ser uma tarefa árdua. Grande parte disso se deve ao fato de a maioria da pesquisa específica deste assunto ser segredo industrial, já que o grosso do que é feito se concentra na área comercial. Podemos, ainda assim, afirmar que métodos clássicos são amplamente utilizados nos mais diversos gêneros de jogos, quer observando bem o comportamento dos agentes e deduzindo sua forma de se comportar, quer através dos (ainda poucos) artigos e livros publicados na área (DELOURA, 2000; RABIN, 2002a; LAMOTHE, 1999). Sem contar que há quem considere que esses métodos mais convencionais são os que, até hoje, dão os melhores resultados (TOZOUR, 2002), o que não deixa de ser válido.

Infelizmente poucos são os jogos que têm seu sucesso (comercialmente falando) oriundo de uma elaborada inteligência. As exceções, em geral, costumam estar nos gêneros de estratégia e simuladores.

Ainda que seja apenas especulação, não deixa de ser uma possibilidade a ser levada em conta o fato de a explosão do uso da Internet poder ter produzido, entre desenvolvedores, a falsa impressão de que criar uma boa inteligência deixou de ser importante. Afinal, muitos jogadores estão ávidos por jogar via rede contra seus amigos ou até mesmo pessoas desconhecidas do outro lado do mundo.

Isso seria, no final das contas, uma conclusão precipitada e equivocada. Se, por um lado, seres humanos certamente propiciam um nível de desafio muito maior do que qualquer agente (o que deve continuar acontecendo por um bom tempo), por outro, há sempre a necessidade de se controlar personagens que não são jogados por uma pessoa. Universos virtuais complexos tendem a necessitar muito desse tipo de agente, já que nem sempre a quantidade de jogadores é suficiente para controlar todos os seres presentes naquele mundo, ou mesmo para papéis os quais não sejam do gosto de ninguém representar, quer por serem nada atrativos, excessivamente estáticos, pouco desafiadores, quer por outro motivo qualquer. Além disso, pode acontecer de não haver a possibilidade (ou vontade) de se jogar com (ou contra) outras pessoas.

A seguir, de qualquer forma, passaremos pelos principais (e mais tradicionais) métodos usados, até hoje, para controlar a inteligência de agentes nos jogos. Alguns desses serão aproveitados neste trabalho, constituindo variações adaptativas das

formas clássicas (normalmente estáticas) utilizadas.

3.2.1 Aleatório

A forma mais ingênua e simples possível de "inteligência" é modelar o comportamento de um determinado agente de forma completamente aleatória. As decisões que ele toma e os caminhos que segue são decididos assim.

É possível fazer isso de várias formas. No que diz respeito à movimentação, uma maneira é sortear uma das direções para as quais os agente tem como se mover, e manter aquela direção por um determinado número de turnos. Esse total de turnos pode ser uma constante (às vezes apenas 1), um valor aleatório ou até uma constante somada a um valor aleatório. Quanto às decisões, uma solução é simplesmente escolher aleatoriamente uma dentre o conjunto total de possibilidades.

As escolhas aleatórias também podem ser feitas atribuindo-se pesos, de forma que algumas tenham mais chances de ser sorteadas que outras. É possível, dessa maneira, conferir determinadas *tendências* ao comportamento do agente, mesmo que ele seja basicamente aleatório.

Esta não é, obviamente, uma forma muito "inteligente" de agir, propriamente, mas tem utilidade em alguns casos. Personagens secundários dentro de um jogo, ou até mesmo elementos de cenário que não tenham influência direta no ambiente (apenas como "decoração") e que precisem ter algum movimento são, dessa maneira, manipulados de um jeito simples e, ainda assim, têm um efeito interessante. Agentes que, por algum motivo dentro do contexto do jogo, requeiram movimentos erráticos e, por que não dizer, aleatórios, também, de uma forma ou de outra, acabam sendo descritos por esse tipo de movimento.

A grande vantagem desse método é a sua simplicidade. Poucas linhas de código são o suficiente para descrever a forma de movimento do agente. Além disso, pelo seu caráter aleatório, o movimento e a tomada de decisões, de forma geral, tornam-se imprevisíveis. A imprevisibilidade também pode ser uma desvantagem e, mais do que isso, a falta de coerência, se esse tipo de estratégia for utilizada para um agente que, supostamente, deve agir de modo "inteligente".

3.2.2 Perseguir / Fugir

Um padrão de comportamento muito comum na natureza é dualidade "predador x presa". De forma simples, um predador constantemente persegue sua presa que, por sua vez, está sempre fugindo. São formas de agir opostas, mas que guardam, entre si, muita similaridade.

Se quisermos determinar um algoritmo de perseguição para um agente, tendo à disposição sua posição atual e a do objetivo, basta que diminuamos ao máximo, a cada turno, a distância que os separa. Supondo, por exemplo, que a posição do agente seja (a_x, a_y) e a do objetivo (o_x, o_y), um algoritmo simples de perseguição poderia ser:

Algoritmo 3.1 Perseguição Simples

se $a_x < o_x$ **então** $a_x \leftarrow a_x + 1$
se $a_x > o_x$ **então** $a_x \leftarrow a_x - 1$
se $a_y < o_y$ **então** $a_y \leftarrow a_y + 1$
se $a_y > o_y$ **então** $a_y \leftarrow a_y - 1$

Se o agente puder se movimentar mais de uma unidade por eixo a cada turno, então basta modificar o algoritmo para permitir isso. No caso, o incremento/decremento só não será o máximo se não for possível fazer tal movimento. Ou seja, cada condição do algoritmo 3.1 passaria a ter a seguinte forma ($MaxInc$ representa o maior incremento/decremento possível):

Algoritmo 3.2 Modificação para Perseguição

se $a_x < o_x$ **então**
    se $o_x - a_x > MaxInc$ **então**
        $a_x \leftarrow a_x + MaxInc$
    senão
        $a_x \leftarrow a_x + (o_x - a_x)$
    fim se
fim se

Outra pequenas adaptações podem ser feitas de acordo com as necessidades e restrições específicas do problema. Se não for, por exemplo, permitido ao agente se movimentar em todos os eixos ao mesmo tempo, ou se houver um ou três eixos, dependendo do número de dimensões do ambiente. Nenhuma dessas mudanças, porém, é muito trabalhosa, basta acrescentar mais condições para mais eixos ou restringir movimentos em mais de um eixo. Nada muito complicado.

Um algoritmo de fuga é análogo ao de perseguição. No caso, ao invés de tentarmos sempre diminuir a distância, queremos que esta sempre aumente. Basta pegar o algoritmo de perseguição e substituir os sinais de "maior que" para "menor que" e vice-versa. As mesmas questões de ponderação anteriores se aplicam também ao caso da fuga.

Esses algoritmos têm a grande vantagem de ser extremamente simples de implementar e muito eficientes, pois requerem uma quantidade de operações muito pequenas para ser executados. Mas essa simplicidade implica uma série de defeitos. Em primeiro lugar, o movimento tende a ser extremamente artificial, o agente tende sempre a se movimentar em linha reta fazendo viradas bruscas, quando um dos eixos está alinhado com o objetivo. Há algumas formas de atenuar esse efeito, tornando o comportamento do agente mais natural, como o descrito em (LAMOTHE, 1999). Outro problema interessante é o que fazer quando um dos eixos está alinhado mas os demais (ou pelo menos um) não? Isso, no caso da perseguição, não importa, pois essa é a finalidade. Mas, no caso da fuga, pode fazer diferença, principalmente se o agente se encontrar encurralado. Outro problema é quando há mais do que um objetivo. No caso da perseguição, elege-se um por prioridade e o encaramos como se fosse o único. No caso da fuga, contudo, isso provavelmente não vai dar muito certo. E, finalmente, a maior de todas as restrições desse método é que ele só funciona corretamente em ambientes sem obstáculos, pois se houver algum entre o agente e o caminho que está sendo feito não haverá como contorná-lo.

Apesar de ser extremamente simples e ingênuo, recheado de desvantagens e problemas, esse método, em conjunto com alguns outros, pode dar resultados até interessantes. Não é de se admirar que ainda seja usado até hoje, apesar de todas as suas limitações.

3.2.3 Padrões

Nem sempre se deseja que o agente persiga ou fuja de algo. Pode-se, às vezes, simplesmente querer que uma seqüência de passos seja obedecida. Ou, de forma mais simples, que o agente siga um *script*.

Isso equivale, na verdade, a ter um determinado algoritmo o qual o agente executa. Os padrões seriam equivalentes a blocos de programa que são executados em seqüência e/ou repetidamente, num laço. Padrões podem se extremamente simples, bem como bastante complexos. Uma forma simples poderia ser:

Algoritmo 3.3 Exemplo para Padrões

```
enquanto verdadeiro
    enquanto não encontrar um obstáculo
        mova para a esquerda
    fim enquanto
    enquanto não encontrar um obstáculo
        mova para a direita
    fim enquanto
fim enquanto
```

Padrões podem ficar mais complexos de acordo com a quantidade de comandos, de saltos etc. Encarando-os como programas, pode-se ter uma noção clara que realmente eles podem ser muito grandes e cobrir uma vasta gama de ações.

Não é necessário, porém, criar uma linguagem complexa e um interpretador para poder usar esse tipo de recurso. A "linguagem" pode ser tão simples como forem os comandos necessários. Há várias maneiras, portanto, de implementar os padrões, dependendo do tipo de recursos que se deseja ver presente.

É possível, de uma forma geral, encarar os padrões como uma linguagem de máquina, tornando a interpretação bem simples. Supondo, por exemplo, que um determinado agente tenha 4 comandos possíveis de movimento (cima, baixo, esquerda e direita) e que, cada um, requeira um argumento de quantos pontos se deve mover (entre 0 e 50, digamos). Os comandos seriam do tipo "esquerda 10", baixo "43", "cima 27" etc. Como são apenas 4 comandos, são necessários 2 bits para codificá-los. Mais 6 bits para o argumento ($2^5 < 50 < 2^6$), são necessários 8 bits (ou seja, 1 byte) para codificar cada comando nessa mini-linguagem.

Usando esse modelo, vários outros comandos podem ser acrescentados, como alterações no fluxo do programa, condições, laços, valores aleatórios etc. Ainda assim, porém, a interpretação dos comandos não será das mais complicadas, já que basta modelar a "linguagem" da forma mais simples que os recursos nela presentes exigirem. Em última instância, será uma linguagem de máquina relativamente simples. Nota-se, claramente, que esse método é um superconjunto dos anteriores, pois se consegue implementar aqueles através deste de forma particularmente direta.

Este método é muito bom quando a quantidade de padrões não é excessivamente grande, porque se pode modelar a "linguagem" de forma simples e sem muito gasto de memória. A execução também é direta, e não requer muito tempo computacional,

já que o processo de interpretação é automático, bastando usar o índice do comando a ser executado num vetor que os armazene. Além disso, como já foi dito, ele engloba os métodos anteriores, podendo substituí-los com muitas vantagens. O grande problema é a complexidade de modelar o algoritmo. Não são muitas as pessoas que têm afinidade com programação de baixo nível, e programar usando esse tipo de abordagem, é praticamente como programar em assembly. Claro que se houver poucas variantes mantendo as possibilidades simples, não haverá muitos problemas. Quanto mais complexa, porém, for a "linguagem" dos padrões, mais próxima de programar em linguagem de máquina ela será, e isso nem sempre é o desejado.

O método de padrões, com algumas sofisticações, ainda é bastante usado em diversos gêneros de jogos, como de esportes e de estratégia (LAMOTHE, 1999).

3.2.4 Máquinas de Estado

Este método é, provavelmente, o mais popular entre os desenvolvedores de jogos para modelagem de inteligência (RABIN, 2002b; DYBSAND, 2000). Apesar de tudo, é muito semelhante ao método anterior e, de certa forma, pode ser considerado, também, como um método por padrões. A diferença é que, neste caso, se modela a inteligência através de uma máquina de estados e a implementação segue diretamente do modelo.

O agente possui o seu estado atual, no qual realiza determinadas ações, e, dependendo do contexto, analisado com variáveis de entrada (geralmente o que o agente percebe do mundo), ele decide qual será o estado para o próximo turno.

A implementação não é muito complicada, seguindo um determinado padrão e aproveitando as características da linguagem na qual o jogo está sendo desenvolvido. Em geral, implementa-se uma a máquina de estados através de uma estrutura `case`, presente na maioria das linguagens. Não é difícil desenvolver programas e bibliotecas que, por meio de uma interface gráfica intuitiva, traduzam o modelo da inteligência para o código fonte da linguagem de desenvolvimento. Há muitas possibilidades e, por se tratar de um método razoavelmente poderoso e flexível, é muito bem aceito entre os desenvolvedores.

Uma característica interessante das máquinas de estado é a possibilidade de ser fazer estruturas hierárquicas sem muitas dificuldades. Ou seja, um determinado estado de uma máquina seria executado como uma outra máquina de estados. Dessa

forma teríamos máquinas dentro de máquinas, numa recursão que iria tão longe quanto se quisesse, ou fosse necessário. Essa hierarquização seria particularmente vantajosa e atraente para problemas que naturalmente exigissem tal divisão e/ou fossem mais simples de se modelar dessa forma.

Há, também, a possibilidade de se acrescentar transições aleatórias, de acordo com probabilidades programadas, conferindo ao agente um comportamento mais imprevisível.

As vantagens de usar máquina de estado são parecidas com as de padrões. É método suficientemente flexível e poderoso para modelar as ações de agentes, e torna-se tão complexo quanto se queira. Além de tudo, não é muito complicado de se trabalhar, podendo ser considerado até intuitivo. Sua implementação não é das mais complicadas, a não ser que se tenha máquinas de estado muito grandes (e até hierarquizadas), mas, por outro lado, o processo de implementação pode ser automatizado de forma satisfatória, atenuando em muito as dificuldades. O grande problema continua sendo a natureza algorítmica do método, além de não ser adaptativo. Uma máquina de estados "pura" não evolui, ou seja, ela não encerra possibilidade de aprendizado nem de adaptação a determinados eventos, permanecendo estática. Podem-se usar transições probabilísticas com pesos variáveis, que sejam modificados ao longo da "vida" do agente como uma forma ingênua de "aprendizado", quase uma espécie de rede neural muito rudimentar, mas essa forma não seria intuitiva o suficiente para ser modelada e, além disso, acabaria, possivelmente, resultando em comportamentos instáveis e estranhos no processo. De uma forma geral, porém, as máquinas de estado são bem aceitas entre os desenvolvedores visto que, com elas, se modela desde as mais simples formas de comportamento até algumas bastante complexas. Apesar de tudo, suas limitações adaptativas são evidentes, além da incapacidade de trabalhar com contextos nos quais a incerteza é muito alta.

3.2.5 Máquinas de Estado Nebulosas

Este é um método que pode vir a ser bastante popular num futuro próximo, até porque é uma extensão do método anterior, que já é bastante utilizado, como já foi comentado. As máquinas de estado nebulosas são, na verdade, máquinas de estado normais porém "acrescidas" de lógica nebulosa. Assim como esta é um superconjunto da lógica tradicional, as máquinas de estado nebulosas são um superconjunto das tradicionais.

Basicamente, as transições de estados passam a ser representadas por regras nebulosas e ter pesos associados de acordo com o grau de pertinência. O estado atual também tem um grau de pertinência, assim como os demais, e o grau do próximo estado na transição é o resultado de uma série de operações nebulosas sobre os graus do estado de origem e das regras de transição. Sendo assim, pode-se estar em mais de um estado ao mesmo tempo, ou seja, diversos estados com grau de pertinência maior que zero.

Este assunto será discutido de forma mais ampla e detalhada na seção 4.3.3, inclusive com uma discussão a respeito da capacidade adaptativa da máquina de estados.

3.2.6 Flocking

A técnica de flocking foi inicialmente apresentada por Reynolds, durante a SIGGRAPH de 1987 (REYNOLDS, 1987), cujo principal objetivo era simular o comportamento de grupo, como de cardumes, por exemplo.

O trabalho original previa três tipos de comportamento para cada membro do grupo: separação (evitar ficar muito perto dos demais membros), alinhamento (mover em direção à média das direções de movimento dos membros vizinhos) e coesão (mover em direção à média das posições dos membros vizinhos).

O próprio Reynolds, posteriormente, criou uma quarta regra, para que fossem evitadas colisões com obstáculos ou inimigos.

Flocking é uma técnica sem memória, pois num determinado estado um membro não tem qualquer informação sobre o passado, ele apenas age de acordo com a situação atual.

Essa forma simples de ação produz resultados interessantes e é uma excelente opção para simular comportamentos de grupo de forma realista e tem sido usada com sucesso em diversos jogos comerciais (WOODCOCK, 2000a).

3.2.7 A*

Fazer com que os agentes se movimentem dentro do ambiente do jogo e mesmo encontrem as melhores rotas para determinados destinos é um dos assuntos mais "quentes" entre desenvolvedores de jogos e, inclusive, entre aqueles que estão apenas aprendendo.

A importância disso se comprova com a quantidade de artigos que tratam exclusivamente do assunto ou abordando-o de forma indireta. A presença de discussão desse tema em livros como (RABIN, 2002a) confirma essa importância.

Entre todos os algoritmos de busca de caminhos, um deles se sobressai como a grande vedete: o famoso A*. Este algoritmo, e suas variações, pode ser considerado de longe o mais utilizado para a busca do melhor caminho de agentes dentro de jogos (MATTHEWS, 2002; STOUT, 1996; STOUT, 2000) apesar de ser uma técnica relativamente velha, já que vem sendo usada há mais de trinta anos pela comunidade acadêmica.

De uma forma simples, o A* nada mais é do que uma busca heurística. Caso não haja uma solução, o A* puro funcionará exatamente como uma busca exaustiva. A idéia básica é, com a ajuda de uma função heurística, diminuir a quantidade de nós a serem explorados. Embora tenha um desempenho médio bem superior a uma busca em largura tradicional, por exemplo, o uso deste método não reduz assintoticamente a complexidade do problema e, por isso, no pior caso será necessário visitar todos os nós da busca como num método tradicional.

Os desenvolvedores, para resolver esse problema, têm de lançar mão de alguns truques a fim de evitar que a busca se prolongue por muito tempo, ou mesmo para, de alguma forma (provavelmente também heurística) descobrir se não há solução ainda antes de aplicar o método (STOUT, 2000). Isso pode introduzir outros problemas, como imprecisão na busca, por exemplo.

Com o espaço de busca muito grande, mesmo usando heurísticas, muitas vezes é necessário cortar nós, ou, de alguma forma, diminuir o tamanho da entrada para o algoritmo. Com essas aproximações, muitas imperfeições acabam aparecendo e situações estranhas e no mínimo constrangedoras para os desenvolvedores podem surgir, como um agente tentando atravessar uma parede, batendo com a cabeça na mesma.

É possível que outros problemas aconteçam se o A* for usado "ao pé da letra", ou seja, após calculado o caminho, o agente seguir o caminho cegamente. Assim, qualquer obstáculo em que apareça dinamicamente é o suficiente para desorientá-lo. Sem contar que, na maioria das vezes, o agente parecerá pouco natural fazendo viradas bruscas de ângulos altos, ou então andando numa espécie de *zigue-zague*. Soluções para esse tipo de problemas também são amplamente propostas e implementadas, como se pode ver em (RABIN, 2002a), na tentativa de evitar essas situações inusi-

tadas e pouco desejáveis dentro de um jogo.

3.2.8 Minimax

Minimax é um método de busca cuja natureza faz dele um modelo muito propício para ser usado em jogos do tipo tabuleiro, ou que sejam baseados em turno. Apesar de ser possível discretizar as jogadas em um jogo de ação de maneira a torná-lo, de certa forma, semelhante àqueles baseados em turnos, permitindo, assim, o uso de algoritmos de busca (NELLER, 2003), este método não é muito interessante para o escopo deste trabalho e, por isso, será apenas descrito brevemente.

De forma simples e resumida, a idéia é construir uma árvore de jogo em que cada nível representa as possíveis jogadas do agente ou do adversário, de forma intercalada. No último nível da árvore, nos nós folha, quando não há mais como descer ou porque aquele é o nível máximo estipulado, calcula-se um determinado valor, usando-se uma função heurística para qualificar aquele estado de jogo. De baixo para cima, vai-se escolhendo os nós, supondo-se que o adversário sempre fará a melhor jogada possível. Pretende-se, dessa forma, escolher a jogada que minimize as chances de perder o jogo. Entre um caminho no qual seja possível ganhar mas também perder e um outro em que só haja possibilidades de empate, o minimax vai preferir a segurança do empate.

O chamado "corte alfa-beta", usado em conjunto com o minimax, é uma forma de tentar diminuir a quantidade de nós visitados pela busca. A idéia é evitar que sejam explorados caminhos que, dado o conhecimento atual, não irão mudar em nada o resultado final.

No fundo, porém, o minimax é uma busca em profundidade exaustiva refinada, com alguns possíveis cortes (como o alfa-beta) e uma pontuação heurística para os nós.

3.3 Inteligência Desonesta

"No intuito de anunciar que tinha uma erudição superior, desovava nos jornais longos artigos sobre contabilidade pública. Eram meras compilações salpicadas aqui e ali com citações. Interessante é que os companheiros o respeitavam, tinham em grande conta o seu saber e ele vivia na seção cercado do respeito de um gênio."

Lima Barreto, *Triste Fim de Policarpo Quaresma*

Os desenvolvedores, como já foi dito, infelizmente não costumam dar muita atenção à elaboração da inteligência de um jogo. Esse comportamento acaba gerando um problema, uma vez que alguma inteligência precisa ser implementada (na maioria dos casos) e, provavelmente, ela terá importância dentro do contexto do jogo. Esse dilema, em geral, é resolvido implementando-se algum dos métodos tradicionais de forma meio improvisada ou até, no pior caso, usando-se uma "inteligência desonesta". Ou, em poucas palavras, o jogo trapaceia.

Para que os agentes do jogo possam proporcionar algum desafio a um jogador, eles devem ser razoavelmente elaborados, mas, em não havendo tempo, disposição ou recursos para tal, basta, secretamente, fazê-lo superior ao adversário humano. Isto se consegue de diversas maneiras. A inteligência desonesta, de uma forma geral, manifesta-se como o excesso de informações e/ou habilidades ao agente, ou seja, conferir a ele recursos que o jogador não tem.

Um exemplo clássico é o agente "onisciente", que simplesmente sabe tudo o que acontece no mundo. Se ele está num labirinto, tem acesso ao mapa. Se está perseguindo o jogador, sabe exatamente onde ele está e o que está fazendo. E assim por diante. Claro que, numa perseguição, essas informações conferem uma vantagem enorme a quem delas dispõe. Para uma competição justa, o jogador teria de ter acesso a essas mesmas informações.

Outra trapaça é a falta de "ruído" na percepção do agente. Num jogo de corrida, por exemplo, ao se colocar atrás de um carro adversário, tem-se a visão atrapalhada por fumaça ou até poeira (dependendo do solo). Pode-se até, independente de posição, ter a visão confusa por causa de alguma condição climática, como chuva, neblina, neve etc. O agente, entretanto, "enxergando" tudo perfeitamente como numa estrada pavimentada num dia ensolarado estará contando com uma vantagem extra, já que não está enfrentando o mesmo tipo de condições adversas que o jogador.

Outra forma de trapacear é fazer o agente ter a agilidade instantânea. Na verdade, mais do que isso, ele praticamente "prevê" a ação do jogador. Num jogo de luta, por exemplo, basta o programa identificar o que o adversário está fazendo num determinado turno de jogo, mesmo que a ação ainda esteja visualmente óbvia (ou seja, dentro do universo de jogo) e já preparar sua ação baseada naquilo que foi observado. Se o adversário está preparando algum tipo de golpe, o programa já sabe que isso está acontecendo e, de antemão, prepara alguma forma de defesa.

Uma das formas mais simples de se fazer uma inteligência desonesta é num jogo

de cartas. Basta informar ao agente as cartas de que os jogadores dispõem e executar a jogada ação baseada nessas informações. Que jogador de pôquer não gostaria de saber a mão do adversário? Claro que há várias outras formas de "roubar" num jogo de cartas, como, por exemplo, distribuir as cartas de forma viciada.

É comum, de uma forma geral, os jogadores "sentirem" quando a inteligência do jogo é desonesta, se esta for evidente. Até em resenhas de jogos encontram-se reclamações nesse sentido e é comum ouvir algo como "esse jogo está roubando" e expressões do gênero. Em alguns casos, contudo, a trapaça não é tão evidente (como no exemplo da corrida), chegando até a não ser considerada como tal pelos próprios desenvolvedores. Este é um problema razoavelmente bem conhecido (WOODCOCK, 2003a; WOODCOCK, 2003b; CHOWN, 2003), embora talvez falte uma discussão bem mais ampla a respeito. Talvez os próprios desenvolvedores não se interessem em tocar no assunto e, muitas vezes, os debates a respeito ficam restritos a pequenos nichos pela Internet, como fóruns por exemplo (CHAMPANDARD, 2003). O assunto chega a ser discutido levemente em (SCOTT, 2002) do ponto de vista do desenvolvedor, porém de forma muito superficial. Um debate mais amplo é necessário.

Claro que a fronteira entre o que é trapaça e o que não é pode ser tênue. Enquanto algumas formas de implementar a inteligência se classificam como desonestas sem muitas dúvidas (como no caso das cartas), em outras isso será discutível. Um bom exemplo é a questão da percepção do agente, no que diz respeito a seu campo de visão. Quando pessoas estão jogando, ao avistarem objetos distantes elas podem ficar em dúvida quanto à natureza dos mesmos. No caso de um agente, dificilmente um jogo simulará esse tipo de comportamento. Ainda que, em teoria, os desenvolvedores tenham todo o cuidado para evitar um agente "onisciente", o simples fato de passarem informações visuais que estão fora do campo de visão qualifica essa implementação como desonesta? Esta é uma questão discutível, afinal, seria necessário desenvolver algum método que simulasse de alguma forma essa questão da visão. O problema é que esse tipo de questão às vezes simplesmente nem passa pela cabeça de quem está fazendo o jogo, mas não deixa de ser um desafio interessante para a elaboração da inteligência do agente.

Alguns casos, todavia, podem se tornar extremamente irritantes para os jogadores. Em jogos do gênero FPS é bastante comum verificar que os agentes têm mira perfeita! Por mais incapaz e ineficiente que seja o algoritmo A* implementado

e, muitas vezes, o agente pareça pouco inteligente ou hábil batendo a cabeça na parede freqüentemente, sempre que ele atirar no jogador acertará o tiro. Isso não passa despercebido, irritando aqueles que estão jogando pois, afinal, fica difícil de acreditar que uma criatura que não consegue achar um caminho decente para se movimentar consiga acertar um tiro com precisão cirúrgica. E, dos problemas de realismo, esse não é nem de perto o mais difícil de ser resolvido. Uma forma simples de contornar tal problema é acrescentar um intervalo de "margem de erro" em cada eixo da mira, escolhendo aleatoriamente um valor nesse intervalo para ser somado à "mira perfeita".

Neste ponto, porém, entra em cena o debate ético. Embora seja plausível afirmar que, muitas vezes, os desenvolvedores não se dão conta desse tipo de questão, em outras vezes isso não faz muito sentido. Quem criaria um agente "onisciente" sem querer? A questão, então, passa a ser, questionar se o desenvolvedor prefere ignorar esse tipo de problema e simplesmente manter a inteligência desonesta. Conferir maiores poderes ao agente, afinal, é torná-lo mais difícil de ser derrotado e, portanto, aparentemente mais inteligente. Eticamente falando, será que isso não é condenável? Caso o jogador seja, de alguma forma, alertado para esse caso (dentro do contexto do jogo, por exemplo), o debate não faz sentido, uma vez que a pessoa sabe que o agente tem mais poderes do que ela. Não avisar o jogador, contudo, de que estará duelando com um adversário que tem muitos mais poderes do que se imagina, não seria uma forma de enganá-lo? Até porque é comum encontrar em caixas de jogos frases de efeito como *improved AI* (IA melhorada). Isso na se caracterizaria como propaganda enganosa? Mesmo sem a propaganda, o fato de o jogador não ter consciência da verdadeira natureza do agente não é induzi-lo, de algum modo, a acreditar que a inteligência do jogo é superior? E, no caso, não se trata de uma atitude eticamente condenável?

Toda essa questão se torna ainda mais complexa quando pensamos no fato de a maioria dos jogos, como propriedade industrial que são, ter seu código fonte guardado em segredo a sete chaves e que muito pouco (quase nada) é publicado pelas empresas que revele, de alguma forma clara, como a implementação do jogo foi realmente feita. Qual garantia temos, então, ao ler na caixa de um determinado jogo que o mesmo usa "técnicas de redes neurais", "computação evolutiva e algoritmos genéticos", "inteligência revolucionária baseada em métodos adaptativos" etc, de que **realmente** tais técnicas foram implementadas? Afinal de contas, esses nomes

"bonitos" e desconhecidos do público leigo em geral podem soar como boa estratégia de marketing para vender o produto, mas, no final das contas, qual a garantia que o jogador, como consumidor, tem de que aquilo que está comprando contém o que vistosamente anuncia? Este é mais um problema ético que faz parte de todo esse complexo contexto do desenvolvimento (e da comercialização) de jogos e que mereceria destaque e discussão maiores inclusive por parte do meio acadêmico.

Este é um assunto que gera inúmeras questões e debates interessantes. Não é, entretanto, o objetivo principal deste trabalho abordá-lo de forma mais aprofundada e, por isso, encerra-se aqui a exposição do tema. Não deixou de ser, apesar disso, interessante mencioná-lo e divagar um pouco a respeito, visto que essa sombra que paira sobre a inteligência desenvolvida em jogos comerciais (ou, de uma forma geral, os que não têm código aberto) dificulta bastante qualquer termo comparativo entre o que é desenvolvido na academia e o que é desenvolvido no mercado. Até porque pouco se faz, na academia, na área de jogos, se comparado a tantas outras áreas prolíferas da computação em geral.

3.4 Estado da Arte

É muito difícil descrever com precisão o estado da arte de inteligência em jogos, uma vez que se fala muito a respeito, mas pouco se vê. O problema, já citado, do segredo industrial atrapalha, e muito, esse tipo de consideração. A (ainda) pequena quantidade de conferências específicas sobre o assunto não é o suficiente para que se possa ter um conjunto mais bem definido acerca daquilo que de mais avançado está se fazendo no setor.

E, realmente, muito se fala mesmo. Muitas vezes podemos ler entrevistas ou mesmo ver apresentações e palestras em conferências em que representantes de empresas desenvolvedoras cantam todas as maravilhas do que têm sido feito em seus jogos. Falam de conceitos como aprendizado, evolução, imersão, *storytelling* etc. mas pouco ou nada revelam sobre como isso é feito e implementado.

Desconsiderando as técnicas mais tradicionais já descritas aqui, nota-se uma crescente preocupação da indústria com relação ao aprendizado dos agentes, que se vem tornando algo cada vez mais freqüente em conferências e mesmo em fóruns de discussão, pelo mundo, sobre o desenvolvimento de inteligência para jogos. Como observa Rabin, os próprios desenvolvedores consideram o aprendizado como a nova

estrela dos jogos (RABIN, 2002a). O número de trabalhos presentes nos livros especializados a esse respeito realmente parece dar crédito à teoria de Rabin. Começam a pipocar artigos a respeito de redes neurais, lógica nebulosa e algoritmos genéticos que até alguns anos atrás pareciam simplesmente ser ignorados da comunidade de jogos.

Em recente matéria, Ward descreve que, entre muitos desenvolvedores, a grande coqueluche do momento é a criação de ambientes completamente interativos, nos quais o mundo possa se adaptar a qualquer ação desempenhada pelo jogador. Dessa forma, não haveria restrições a respeito da interação de personagens com o ambiente de jogo, desde que este pudesse adaptar-se perfeitamente a qualquer conseqüência das ações. Não haveria, portanto, qualquer enredo linear e final definido, pois tudo dependeria de uma série de atitudes dos personagens, de como eles iriam interagir entre si e com os elementos do mundo, permitindo inúmeras situações diferentes e diversos eventos possíveis (WARD, 2003). Alguns testes já estão sendo conduzidos neste sentido, ainda segundo a reportagem.

Woodcock mantém em sua página na Internet uma lista muito útil de "jogos que utilizam técnicas interessantes de inteligência artificial" (WOODCOCK, 2003d). Há uma seção comentando sobre o badalado jogo *Black & White*, inclusive com palavras do seu principal desenvolvedor. A descrição do comportamento é semelhante à idéia colocada no último parágrafo, e a promessa e propaganda do jogo de "IA revolucionária" também faz acreditar nisso. Mas é difícil concluir qualquer coisa através do que é exposto, uma vez que a descrição do método usado no jogo nos leva a crer que não passe de uma série de regras estabelecidas de forma esperta. Pelas próprias palavras do autor, muito parece se basear em cópia de comportamento, e o aprendizado se faz através de uma série de recompensas ou castigos ao personagem, sem especificar exatamente como isso é feito.

O jogo *The Sims* também é descrito, na mesma página, como fazendo uso de máquinas de estado nebulosas. Além disso, são associadas ações e comportamentos a objetos. Como exemplo, apresenta-se uma TV, que conteria instruções sobre como assistir, ligar e desligar, bem como as condições sob as quais alguém desejaria assistir, como reagiria ao fazê-lo etc. O desenvolvedor descreve esta técnica como *smart terrain*, ou seja, algo como "terreno esperto". Woodcock comenta que, conversando com os desenvolvedores, ficou sabendo que o *smart terrain* é uma espécie de técnica de orientação a objetos com uso de máquinas de estado. Não temos, infelizmente,

mais informações técnicas a respeito da implementação em si, tendo de nos contentar com essas superficiais. Ainda assim, não deixa de ser um ponto de vista interessante para implementação em jogos.

Outro famoso jogo, *Half-Life*, que chega a ser considerado por Tozour como tendo "excelente inteligência artificial tática" (TOZOUR, 2002), segundo Woodcock é apenas uma demonstração do bom uso possível de técnicas convencionais, como flocking, máquinas de estado etc. (WOODCOCK, 2003d). Como ele mesmo conclui, excelente inteligência, mas nada de revolucionário.

Enfim, apesar do desenvolvimento de diversas técnicas adaptativas como as já citadas (redes neurais, algoritmos genéticos e lógica nebulosa), aparentemente a indústria de jogos tem se concentrado mais em refinamentos de técnicas tradicionais. O problema do segredo de desenvolvimento, por causa da ferrenha competição, aliado a uma questão de necessidade de marketing, esperneando o uso de várias técnicas com nomes bonitos para o leigo consumidor, tornam muito mais complicado o trabalho de avaliação do estado da arte do desenvolvimento de inteligência na área de jogos. Acreditamos, apesar disso, que há indícios suficientes, como maior interesse no desenvolvimento da inteligência e uso de técnicas não convencionais, para supormos que, nos próximos anos, o campo de IA em jogos testemunhará um salto qualitativo, semelhante ao vivido na última década com relação à qualidade gráfica.

Capítulo 4

Modelos Propostos

"A mente criativa brinca com o objeto que ama"
- Carl Jung

Neste capítulo, apresentamos os métodos desenvolvidos como forma de alcançar os objetivos propostos e, embora alguns possam ser usados juntamente com outros, eles são descritos separadamente.

A idéia, como já foi exposto, é tentar criar maneiras para que a dificuldade de um jogo cresça de acordo com o aprendizado do jogador. Os modelos e técnicas aqui apresentados, portanto, foram desenvolvidos com o objetivo de, através de sua aplicação, permitirem que o ambiente de jogo evolua de forma condicionada ao progresso do jogador.

4.1 Coevolução em Tempo Real

Como já foi discutido na seção 2.3, os algoritmos coevolucionários vêm sendo utilizados com muito sucesso em diversas áreas, até mesmo em jogos. O que nos interessa, porém, é o uso desses algoritmos dentro do contexto de jogos de ação e, portanto, em tempo real.

Fazer os agentes evoluírem em tempo real, de acordo com o adversário, implica ter poucos dados disponíveis e uma restrição de tempo muito forte. Se num jogo de xadrez é perfeitamente aceitável que o adversário demore alguns minutos (ou mesmo vários) para fazer a sua jogada, no nosso caso isso pode representar a derrota do agente, afinal, enquanto ele "pensa" no que vai fazer, o jogador continua tomando suas ações. Qualquer demora extra é, geralmente, fatal. Além disso, ainda não

podemos nos dar o luxo de precisar de várias gerações para evoluir, afinal, se demorar muito o jogador provavelmente vai ficar entediado com a facilidade de derrotar o agente muito antes de alcançarmos todas as gerações necessárias.

O método aqui proposto é uma forma de tentar vencer todas essas dificuldades e permitir que os agentes possam evoluir de acordo com a habilidade do jogador e junto com o mesmo. A principal forma de coevolução aqui é competitiva, entre jogador e agentes, porém também há a possibilidade de cooperação entre subespécies de agentes. Este método tem como base o trabalho previamente publicado sobre o assunto em (DEMASI, CRUZ, 2002c), como parte deste trabalho de pesquisa, e foi desenvolvido originalmente para fazer parte do conjunto de estratégias apresentadas neste capítulo, constituindo uma contribuição original.

A idéia básica por trás deste método e suas variantes é aproveitar qualquer conhecimento externo que possuamos a respeito do ambiente do jogo para, de alguma maneira, moldar a evolução dos agentes. De alguma forma procuramos comprimir todo o processo de evolução, que pode ocorrer durante horas, para alguns minutos, da forma mais suave possível.

A aplicação deste método pressupõe jogos com uma quantidade razoável de agentes, distribuídos ou não em subpopulações. Considera-se, também, que os agentes são destruídos (ou mortos) e repostos por novos, mas isso não é necessário, até porque o próprio método prevê o contrário.

4.1.1 Definições

Chamaremos de *objetivo* o indivíduo "ideal", isto é, aquele que deve ser alcançado em determinado estágio de evolução, seja ele final ou intermediário. Ou seja, se há um indivíduo considerado o melhor contra aquele adversário, este será o objetivo que os indivíduos da população terão de "alcançar" através das etapas evolutivas.

Precisamos, também, de uma função de *facilidade*, que seja capaz de medir o quanto o jogo em um determinado instante de tempo t está sendo fácil para o jogador. Ou seja, quão bem ele está indo num determinado momento. Essa função será representada por $ef(.)$.

Representaremos por l_f um limite de facilidade. Este é o limite acima do qual o jogo poderá ser considerado fácil demais naquele instante.

Definimos como TTL um limite de tempo (*time to live*) utilizado para forçar a evolução de agentes (como será visto posteriormente). Atribuindo $TTL = +\infty$

estamos desconsiderando esse limite. Cada agente i do jogo terá um contador ttl_i associado para verificar se o limite foi alcançado.

4.1.2 Descrição Geral

Apresentaremos, posteriormente, as quatro variantes desenvolvidas do método, em relação ao modo como os indivíduos evoluem e no uso de objetivos (intermediários e finais). A idéia básica é permitir que, a partir de determinadas informações, os agentes possam evoluir, resultando num aumento progressivo do nível de dificuldade do jogo de acordo com o quão bem o jogador está jogando.

A forma geral do funcionamento das variantes, contudo, é dada pelo algoritmo 4.1.2.

Algoritmo 4.1 Forma Geral da Coevolução em Tempo Real

```
gerar as subpopulações iniciais
enquanto não terminar o jogo faça
    a cada intervalo Δt calcule ef(.)
    para cada agente i faça
        se i está morto então
            se ef(.) > l_f então
                evolua i
            senão
                reponha i
            fim se
        senão se ttl_i > TLL então
            ttl_i ← 0
            se ef(.) > l_f então evolua i
        fim se
    fim para
fim enquanto
```

De forma resumida, quando um agente é morto ele é reposto por um mais evoluído, se a facilidade do jogo ultrapassar l_f, ou por um outro agente do mesmo nível, no contrário. Se o agente não morreu mas esgotou-se seu ttl, então ele evolui. Mais detalhes do funcionamento geral deste algoritmo serão vistos dentro das variantes.

4.1.3 Indivíduo Objetivo

O indivíduo objetivo é o "ideal" que um agente deve alcançar em determinado instante. Esse ideal pode ser final ou intermediário. Um indivíduo objetivo final pode ser considerado como o equivalente ao melhor agente possível para aquela espécie (ou subespécie), uma espécie de "deus".

Sob o ponto de vista de algoritmos genéticos, os indivíduos objetivos serão considerados como "reprodutores", cujo código genético é desejado que seja obtido na população, e são usados para melhorar a mesma.

Objetivos intermediários podem ser definidos desempenhando o papel de marcadores da evolução. Ou seja, a evolução das subpopulações deve passar por esses indivíduos objetivos e, portanto, eles são uma espécie de guia para a evolução.

Não há muitas mudanças no método geral para o caso de se usarem esses objetivos intermediários. A principal diferença é que a evolução ocorre em relação ao próximo objetivo. Podemos encarar o uso de apenas um objetivo final como um caso particular, usando apenas um único intermediário.

Outra diferença está em adaptar o método para escolher qual indivíduo objetivo será usado como base para a evolução. Neste caso podemos iniciar como o primeiro dos intermediários e passar adiante quando todos os indivíduos o tiverem alcançado. Pode-se fazer de outra forma, controlando qual o objetivo a ser usado para cada agente. Sendo assim, quando um determinado agente chegasse ao ideal atual, ele passaria a evoluir em relação ao próximo intermediário. Usando essa forma, diferentes agentes, mesmo que de uma mesma subpopulação, podem ter indivíduos objetivos diferentes num mesmo instante de tempo. Da forma anterior, ao contrário, todos têm o mesmo objetivo num mesmo instante de tempo.

A opção de usar objetivos intermediários tem a vantagem de permitir uma evolução mais controlada em direção ao ideal desejado, enquanto que o uso de apenas um objetivo tende a tornar esse processo mais caótico, dependendo também da forma de evolução usada. Por outro lado, há mais trabalho para a determinação dos objetivos intermediários. Dependendo da variante a ser usada, isso pode tornar as coisas bem mais complicadas. Enfim, é uma decisão que depende daquilo que se pretende e se pode realizar. Quando os agentes têm uma codificação mais complexa e genótipo grande, a necessidade do uso de mais objetivos intermediários cresce. A razão disso é simples, pois quanto maior a quantidade de bits no código genético

do agente, maior é o espaço de busca (que cresce exponencialmente no número de bits) e, portanto, muito mais "espalhadas" nesse espaço as soluções tendem a ficar. Sendo assim, usando-se poucos objetivos intermediários (ou mesmo nenhum), os indivíduos novos criados tendem a ter uma evolução muito mais caótica em direção ao objetivo.

4.1.4 Função de Facilidade

Colocando de uma maneira simples, a função de facilidade pode ser vista como a função de avaliação do algoritmo genético em relação ao adversário. Funciona, portanto, como avaliação do jogador, quão bem ele está em determinado momento. Quanto mais "adaptado" ele estiver aos agentes, mais fácil consideramos que o jogo está. Claro que não iremos evoluir o jogador, porém usar inversamente a função de avaliação nos traz informações importantes.

A idéia geral do uso da função é avaliar, a cada determinado instante (ou intervalo de tempo), quão fácil está o jogo para o jogador. A partir desta informação, o jogo pode determinar se os agentes devem ou não evoluir (e, portanto, tornar o jogo mais difícil). A forma como esses passos é feita depende de qual das variantes do método se estiver usando, e que serão descritas posteriormente. A idéia geral, entretanto, no uso da função é justamente prover uma avaliação instantânea a respeito do estado do jogo em relação à facilidade encontrada pelo jogador. Como a função é aplicada em tempo real, ela deve ser usada em intervalos de tempo que podem ser ou não regulares. Assim, portanto, a função é definida num intervalo Δt (possivelmente variável) usando, para isso, informações do jogo.

A função pode ser definida de inúmeras maneiras e, assim como no caso de algoritmos genéticos, ela é essencialmente heurística. Quanto melhor a definição puder descrever a facilidade que o jogador está encontrando para jogar, melhor tenderá a ser o funcionamento do método, uma vez que o mesmo terá uma estimativa mais confiável da relação do jogador com os agentes. De uma forma geral, quanto maior o valor retornado por $ef(.)$, mais fácil será o jogo. Para facilitar o trabalho de avaliação dos resultados da função, sugere-se que o valor esteja sempre no intervalo real $[0, 1]$, atuando como se fosse o grau de pertinência do jogo num hipotético conjunto nebuloso de facilidade. Neste caso, quanto mais próximo de 1 mais fácil será considerado o jogo, enquanto que quanto mais próximo de 0 mais difícil o será.

Fatores importantes de cada jogo particular que influenciem na facilidade en-

contrada por um jogador devem ser considerados na função. Por exemplo, vidas perdidas em geral são um sinal de dificuldade do jogador, assim como energia, se for o caso. Em jogos em que é necessário atirar, o aproveitamento desses tiros (ou seja, tiros acertados divididos por tiros disparados) pode ser um indicador de facilidade. Pontos conquistados (ou aproveitamento dos mesmos), bem como inimigos derrotados e itens coletados também costumam ser bons indicadores. Como já foi dito, tudo depende muito do jogo em si, e a escolha em geral é baseada em dados empíricos e, muitas vezes, na base da tentativa e erro.

Através das avaliações da função de facilidade, pode-se ajustá-la de forma a torná-la mais ou menos sensível. Ou seja, é possível fazer com que a função tenda a retornar valores mais ou menos próximos de um determinado patamar. A conseqüência disso é uma espécie de "sintonia fina" na avaliação da dificuldade do jogo.

As características desejáveis de $ef(.)$ são confiabilidade e eficiência. Ou seja, é imprescindível que ela retorne valores confiáveis e que, além disso, seja rápida o suficiente, não gastando muitos recursos computacionais. Em poucas palavras, deve ser simples. Os jogos com os quais estaremos lidando são jogos de tempo real e, como estaremos avaliando constantemente o estado do jogo através da função de facilidade, será extremamente desagradável se o cálculo da função não for eficiente, podendo arruinar toda a experiência de jogo. Dessa forma, pode ser mais interessante usar uma função simples que não seja tão confiável, porém eficiente, do que uma extremamente confiável e lenta. A restrição do tempo, portanto, tem um peso muito maior do que a necessidade de precisão.

4.1.5 TTL

O TTL é usado para evitar que agentes fiquem estagnados durante o tempo, ou seja, não evoluam. Como a evolução, a princípio, seria feita sempre que um determinado agente morresse e a facilidade ultrapassasse o limite l_f, então apenas os agentes mortos evoluiriam. Se um jogador, portanto, matasse sempre o mesmo agente, então apenas este evoluiria, o que não acrescentaria muito ao jogo (considerando, obviamente, que haja um número razoavelmente alto de agentes).

Pode ser, além disso, que o jogo em questão não tenha o conceito de "destruição" de agentes, que estariam sempre "vivos" e, portanto, é necessária uma forma de permitir a evolução dos mesmos nesse caso.

Por outro lado, se for desejável que os agentes evoluam apenas quando forem

mortos, então basta desconsiderar o TTL. Para fazer isso numa aplicação genérica do método, como já foi dito, basta atribuir $TTL = +\infty$.

4.1.6 Formas de Evolução

Dadas as devidas condições, como já foi exposto, os agentes devem evoluir com o propósito de alcançar seu respectivo indivíduo objetivo. Nesta seção discutiremos as formas de fazer isso.

Considerando o objetivo e o agente como os pais, podemos criar um novo indivíduo através do cruzamento (*cross-over*) de ambos. As várias formas de cruzamento (um ponto, dois pontos, máscara etc.) podem ser aplicadas aqui. Da mesma forma, a mutação pode ser aplicada após a criação do novo indivíduo.

Uma outra forma de evoluir o indivíduo, que pode ser particularmente vantajosa de acordo com a estrutura dos genes, é trocar um bit diferente entre o agente e o objetivo. Seja a distância de Hamming entre duas seqüências binárias b_1 e b_2 o total de bits diferentes entre ambas. Ou seja, fazendo b_1 xor b_2 e contando os bits diferentes de zero. Essa distância entre o agente e o objetivo indica o quanto ele deve evoluir para alcançá-lo. Ao invés de fazer o cruzamento como comumente definido em algoritmos genéticos, podemos apenas trocar um bit (possivelmente escolhido de forma aleatória) do agente que seja diferente do objetivo. Sendo h a distância de Hamming entre ambos, então serão necessários h passos evolutivos para alcançar o indivíduo objetivo. Essa forma de evolução é particularmente atraente para genes relativamente simples, de forma que se pode controlar melhor (ou pelo menos ter uma expectativa melhor) de como os agentes vão evoluir.

Ambas as formas de realizar a evolução possibilitam a criação de indivíduos indesejados. Cruzando o agente com o objetivo pode-se ter um novo indivíduo que seja superior ao objetivo. O mesmo pode acontecer trocando-se os bits. A forma como o genótipo é definido e interpretado tem uma influência, possivelmente grande, nesse aspecto.

Por exemplo, digamos que o indivíduo seja definido por 4 bits. Se temos dois objetivos, digamos 1011 e 1111, sendo o segundo melhor que o primeiro, e o agente atual é 0100, ao cruzarmos 0100 com 1011 podemos obter 1111 que, teoricamente, é mais evoluído que o objetivo atual (1011). Da mesma forma, se temos um agente 1110, cuja distância de Hamming para 1011 é 2, ao sortearmos um bit para mudar corremos o risco de obter 1111, que também é teoricamente mais evoluído que o

objetivo atual (1011). Mais dificuldades podem surgir. Não necessariamente um novo indivíduo pode ter um código genético igual ao de um estágio superior de evolução daquele que ele deveria alcançar e, mesmo assim, pode ser mais evoluído que o objetivo atual.

Claro que este foi um exemplo simples, mas é uma amostra clara dos problemas que podem ocorrer. Quanto maior for o código genético do indivíduo, porém, mais difícilmente isso tende a acontecer, afinal, é muito mais provável uma combinação intermediária não ser boa do que o contrário. De preferência, a combinação de genes e sua interpretação devem ser determinadas de forma a minimizar a possibilidade desse tipo de ocorrência. É possível que seja necessário mesmo fazê-lo após observações experimentais, como foi, inclusive, o caso deste trabalho e será visto na seção 5.3.4.

Uma outra questão relativa à evolução é como fazer a reposição dos agentes mortos quando a facilidade não ultrapassar o limite estipulado. Uma solução simples seria simplesmente replicar o mesmo agente, copiando seu código genético, sem usar qualquer tipo de evolução. Outra forma seria fazer uma espécie de rodízio entre os agentes já mortos. Ou seja, cada vez que um indivíduo de uma subespécie fosse morto, ele seria adicionado a um conjunto (relativo a sua subespécie) e seria reposto por um agente escolhido aleatoriamente dentro desse conjunto (e, possivelmente, ele próprio). O conjunto necessitaria ser esvaziado quando o objetivo fosse atualizado, de forma a evitar que novos indivíduos defasados fossem criados.

Por outro lado, pode-se desejar criar agentes novos, dentro dos limites do objetivo atual (isto é, não melhores que o mesmo). Para isso, um conjunto de indivíduos da mesma espécie (ou subespécie, se for o caso), pode ser guardado para reprodução. Como os novos agentes criados têm apenas o código genético daqueles selecionados, sem usar o próximo objetivo, evita-se que a popualação evolua demais, e, ainda assim, permite-se que haja certa diversidade. Esse conjunto seria formado por agentes já mortos, ou até mesmo pelos que estão em atividade naquele instante. Quando um agente morrer, escolhe-se os pais dentro desse conjunto (possivelmente usando métodos convencionais como o da roleta) e, pelo cruzamento de ambos, cria-se o novo indivíduo. Isso pode acarretar problemas parecidos com os que já foram citados, e o mesmo cuidado aconselhado anteriormente aqui se aplica. Para códigos genéticos maiores, da mesma maneira, a possibilidade de esses problemas acontecerem diminui consideravelmente. Afinal, o obtenção de um indivíduo mais apto aproveitando

pouca diversidade (como é o caso, apenas alguns agentes num conjunto) é um dos grandes problemas para a evolução em tempo real. Se a probabilidade de isso acontecer fosse alta, então todo o nosso trabalho estaria imensamente facilitado!

4.1.7 As Variantes

Definidos os termos e os detalhes que compõem o método de forma geral, podemos descrever as variantes de aplicação do mesmo. Elas se referem, basicamente, à determinação dos indivíduos objetivos. Cada variante é uma forma diferente de tentar resolver o mesmo problema, ou seja, evoluir de acordo com o progresso relativo do jogador.

O uso de objetivos intermediários, o uso de apenas uma espécie ao invés de subespécies, ou o uso da cooperação entre subespécies é igualmente aplicável a todas as variantes. O mesmo se verifica para os demais conceitos desenvolvidos até aqui, como a forma de evolução com relação aos objetivos e também para os agentes mortos, o uso do TTL, definição da função de facilidade etc.

Quaisquer características particulares que apenas se aplicam em uma determinada variante ou que sejam diferentes das demais serão devidamente observadas e descritas nas respectivas seções.

4.1.8 Variante 1: Usando Informações do Jogo

Suponhamos que conheçamos o jogo suficientemente bem para que possamos criar os indivíduos objetivos diretamente. Ou seja, somos capazes de definir precisamente os agentes ideais para o jogo.

Isso aconteceria, por exemplo, pelo fato de o genótipo ser simples o suficiente para que possamos prever, sem problemas, quais combinações são as melhores. Ou, por outro lado, essa informação pode ser obtida de forma empírica. Após jogar diversas vezes, podemos concluir que uma determinada estratégia (ou um conjunto delas) é a melhor.

Para o caso de haver n subespécies coexistindo, deve-se definir n indivíduos objetivos, um para cada subespécie. É possível até que haja indivíduos iguais, ou mesmo que o objetivo final seja igual para todos. O mesmo deve ser observado se houver objetivos intermediários sendo usados.

Esta variante tem a grande vantagem de ser simples e permitir um grande con-

trole sobre os objetivos definidos. Por outro lado, ela requer um bom conhecimento prévio do jogo a que se vai aplicar, bem como a capacidade de codificar as estratégias desejadas para os objetivos nos genes dos indivíduos. Essas desvantagens podem inviabilizar que se use uma quantidade alta de objetivos intermediários, ainda mais quando a codificação dos genes é mais complexa e maior. Isso inviabilizaria o uso desta variante, uma vez que nesses casos é bastante desejável ter vários objetivos intermediários para um melhor controle da evolução, como já foi discutido.

De uma forma geral, esta variante tende a ser mais apropriada para jogos com estratégias mais simples e cujas melhores estratégias para os agentes podem ser deduzidas satisfatoriamente. Para agentes mais complexos e espaços de busca maiores, bem como necessidade de alto número de agentes intermediários, como visto, não é recomendada.

É possível usar qualquer tipo de evolução dos indivíduos, tanto no caso de a facilidade ultrapassar o limite l_f como no de não ultrapassá-lo. Uma vez que a tendência desta variante é o uso em situações mais simples, o aproveitamento da distância de Hamming tende a ser melhor, até por permitir um controle maior na quantidade de operações de evolução como forma de alcançar o objetivo. Isso não exclui, contudo, o uso do cruzamento como forma de evolução.

4.1.9 Variante 2: Usando Dados Evolutivos Pré-Gravados

Se não quisermos definir os objetivos manualmente, podemos fazer simulações evolutivas e aproveitar os dados nelas obtidos para definir os objetivos que serão usados durante o jogo em tempo real.

A idéia é basicamente a mesma da variante anterior. A grande diferença, neste caso, é que, ao invés de definirmos os objetivos com base no nosso conhecimento do jogo e da estrutura genética dos agentes, aproveitamos dados coletados durante simulações de jogo pré-gravadas.

Isso é feito como se fosse uma simulação tradicional de algoritmos genéticos, sem ser em tempo real. Após um número estipulado de gerações, podemos recolher os dados obtidos com a simulação para definir os indivíduos objetivos.

A grande vantagem desta variante em relação à anterior é não requerer um conhecimento empírico do jogo para determinar as melhores estratégias, bem como as intermediárias. Além disso, os objetivos intermediários podem ser retirados da mesma simulação de determinadas gerações. Ou seja, a quantidade de objetivos

intermediários não aumenta a dificuldade de uso da variante.

O grande problema é como realizar a simulação. Sim, porque para fazê-la é necessário substituir, ou de alguma forma simular, o adversário dos agentes, isto é, o jogador. Isso pode ser feito de algumas maneiras diferentes.

Criando um outro agente para desempenhar o papel do jogador, podemos realizar a simulação tantas vezes quantas forem necessárias. A vantagem, neste caso, está na possibilidade de se fazerem testes exaustivos, podendo-se ter dados suficientes para escolher os objetivos da forma mais apropriada possível. A grande desvantagem é razoavelmente óbvia, uma vez que é necessário criar um outro agente, o qual terá de desempenhar o papel do jogador. Este fator é quase igual à característica da variante anterior de requerer um conhecimento empírico do jogo, que também será necessário para a criação desse novo agente. Um problema a mais é a questão da competição entre jogador e agentes. Caso a simulação do jogador seja estática, não haverá todo o ambiente de coevolução entre os agentes e o adversário, o que pode ou não ser um problema. Por outro lado, se a simulação for adaptativa, sua evolução pode influenciar no resultado da determinação dos objetivos, devido ao efeito da rainha vermelha, como visto na seção 2.3.1, e, neste caso, a forma como a evolução do novo agente acontece pode modificar consideravelmente os dados que serão aproveitados para a criação dos indivíduos objetivos, o que pode atrapalhar seu aproveitamento final.

Ao invés de criar um agente, pode-se aproveitar uma quantidade suficientemente grande de jogadores para desempenhar o papel evolutivo. Através de uma arquitetura cliente / servidor, pode-se substituir o agente que seria criado por diversos jogadores e aproveitar os dados das jogadas desses vários jogadores para realizar a simulação. A evolução não aconteceria em tempo real, enquanto um determinado jogador estivesse jogando. Ao contrário, os dados da partida seriam usados para a função de avaliação do indivíduo. Toda a evolução aconteceria como num algoritmo genético tradicional, a única diferença seria justamente a avaliação dos indivíduos da população, que seria feita usando-se, como foi dito, os dados de uma ou mais partidas jogadas.

Esse tipo de arquitetura para evolução aproveitando dados de diversos jogadores pode ser feita através da Internet, por exemplo, como proposto e descrito em (FUNES et al, 1997; SKLAR et al, 1999). A desvantagem, no caso, é a necessidade de recursos computacionais, como um servidor, por exemplo, e a dependência

da participação dos jogadores. Superadas essas desvantagens, porém, os resultados podem ser animadores. E, como demonstrado em (FUNES et al, 1998; FUNES, POLLACK, 2000), pelo aproveitamento da Internet, é possível obter evolução de agentes através da interação com seres humanos. Essa evolução é justamente o que desejamos, pois usando diversas pessoas diferentes podemos, na simulação, obter os indivíduos objetivos que precisamos para utilizar no método.

No caso de se conseguir vencer essas barreiras técnicas descritas, o uso da Internet torna esta variante particularmente atraente, mesmo para jogos com estratégias um pouco mais complexas. Caso não seja possível aproveitar a interação humana, a criação de um agente para simular o comportamento do jogador seria vantajosa para os casos em que isso não fosse extremamente complicado. Para situações em que o grande desafio do jogo, e portanto para o jogador, for aprimorar sua perícia no mesmo, não é nem um pouco complicado simular esse comportamento. Se, por outro lado, o desafio for criar estratégias elaboradas, a criação do agente estaria seriamente comprometida.

O aproveitamento de objetivos intermediários, nesta variante, tende a ser o maior possível, independente da forma de realizar a simulação. Tendo os dados obtidos com a evolução, pode-se determinar um número bem alto de indivíduos objetivos. Claro que, quanto maior a quantidade de objetivos intermediários, mais lenta tenderá a ser a evolução em direção ao final, o que será ou não o que se deseja.

Assim como no caso anterior, o uso de qualquer forma de evolução é normalmente aplicável nesta variante. A observação dos dados obtidos na simulação, inclusive, ajudaria na determinação de como evoluir os agentes.

4.1.10 Variante 3: Coevolução Pura em Tempo Real

Nesta variante simplesmente não é usada qualquer informação externa ao jogo e, portanto, não há indivíduos objetivos. A evolução é feita somente com base nos dados do jogo atual. As dificuldades neste caso são, obviamente, enormes, e a maior delas é justamente a pouca quantidade de informação disponível.

Como não há o cruzamento com os objetivos, a evolução tem de ser feita usando-se a forma convencional de algoritmos genéticos. Para isso é necessário definir uma função de avaliação para os indivíduos. Esta avaliação, por ser coevolução, deve se basear na interação entre os agentes e o jogador (competição) bem como em relação às demais subpopulações (cooperação) da forma geral como já foi visto na seção 2.3.

Para criar os novos agentes, mantém-se um conjunto com os melhores indivíduos, para cada subespécie. Sempre que for necessário repor um determinado elemento, escolhem-se os pais de dentro desse conjunto (através do método da roleta, por exemplo) e fazendo o cruzamento entre eles. Além disso, mutação, elitismo etc. também podem estar presentes.

O conjunto dos melhores indivíduos é modificado sempre que um determinado agente for mais apto que o pior do conjunto. Ou seja, quando tem a função de avaliação maior que a menor de todos. Esse pior é retirado para dar lugar ao mais adaptado.

Para jogos que têm uma vazão alta de agentes (há uma reposição freqüente dos mesmos), esta variante é especialmente atrativa. Com uma grande quantidade de agentes novos em pequenos intervalos de tempo, a velocidade da evolução seria, possivelmente, sensivelmente melhorada.

Se o espaço de busca for relativamente extenso, o uso desta variante pode se tornar pouco útil, até porque há poucos dados para que a busca nesse espaço seja feita de forma satisfatória pelo algoritmo genético. Apesar disso, para aplicações em tempo real nas quais esse espaço de busca é extremamente imprevisível e bastante dinâmica e para as quais a possibilidade de generalização de uma determinada estratégia é muito pequena, a evolução em tempo real pode até mesmo conseguir resultados melhores do que a evolução convencional (AGOGINO, STANLEY, MIIKKULAINEN, 2000) e, portanto, torna-se uma boa opção a ser experimentada e testada. Como os jogos de ação caem, em geral, nessa categoria de problemas, o uso desta variante pode ser particularmente tentador.

4.1.11 Variante 4: Hibridismo

Esta variante é uma tentativa de juntar as principais vantagens das anteriores numa só. A idéia básica é realizar a evolução em dois "estágios" distintos. No primeiro, tudo funciona como na primeira (ou segunda) variante. Depois de alcançado um determinado objetivo final, ou possivelmente após um intervalo de tempo limite pré-determinado, a evolução passar a ser feita como na terceira variante.

Podemos encarar esta variante como tendo uma fase "introdutória", que seria a primeira, e uma outra de "especialização", que seria a segunda.

O uso desses dois passos permitiria que os agentes partam de uma estratégia basicamente aleatória evoluindo até um estágio pré-determinado. A partir de então,

eles passariam a evoluir "livremente", ou seja, sem objetivos.

Esta variante pode ser aplicada a problemas em que algumas estratégias básicas são conhecidas e que funcionam bem, mas que para derrotar jogadores mais avançados são necessárias técnicas mais refinadas. O primeiro passo conduziria os agentes até esse estágio de táticas básicas e, a partir dali, eles evoluiriam para mais avançadas. Também seria bem aproveitada para, no caso do uso da primeira variante, só sermos capazes de deduzir empiricamente algumas estratégias básicas, deixando para a coevolução pura tentar encontrar as mais complexas. Uma outra aplicação interessante é para os casos em que algumas formas básicas de jogo tendem a se sair bem, mas em que determinadas estratégias podem ser muito boas ou muito ruins dependendo do adversário. Neste caso os agentes evoluiriam até um patamar básico de jogo e, a partir de então, teriam liberdade de evoluir para estratégias que melhor se adaptassem contra o adversário em questão, o que, em geral, tende a variar consideravelmente entre jogadores diferentes.

4.1.12 Interpretação do Método

As primeiras variantes deste método, utilizando indivíduos objetivos, podem ser encaradas como uma aproximação para uma função já conhecida.

Suponhamos que esteja sendo usada a segunda variante e que uma quantidade enorme de dados a respeito da evolução dos agentes tenham sido obtidos. O estágio mais avançado de evolução possivelmente demorou horas, ou mesmo dias, para ter sido alcançado. Durante um jogo não podemos nos dar o luxo de gastar tanto tempo. Se quisermos usar a mesma evolução obtida fora das condições de tempo real dentro de um jogo de ação teremos de comprimir um processo evolucionário de muitas horas para alguns minutos. Seria como se estivéssemos compactando a função de avaliação das gerações ao longo do tempo, ou seja, aproximando-a.

Queremos aproveitar a curva de aprendizado lenta para um período de tempo muito menor. Encarando a interpretação como se tivéssemos o gráfico de uma função $y = f(x)$, o que precisamos é comprimir o domínio dos x. Para isso, escolhemos uma quantidade finita n de pontos, discretizando a função. Usando esses n pontos é possível representar a mesma função discretamente. Como perdemos a informação a respeito dos pontos de $f(x)$ entre dois pontos consecutivos, podemos aproximá-los através de uma função linear, por exemplo.

A figura 4.1 mostra o gráfico hipotético do crescimento de uma função qualquer.

Figura 4.1: Gráfico de uma função qualquer

Figura 4.2: Nove pontos marcados na função

Figura 4.3: Aproximação da função usando os pontos marcados

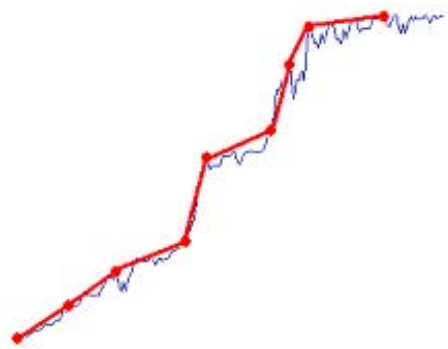

Figura 4.4: Sobreposição dos gráficos da função original e sua aproximação

Escolhemos nove pontos distribuídos pela curva (não necessariamente eqüidistantes), como mostra a figura 4.2. Supondo que perdemos a informação de $f(x)$ e que só dispomos dos pontos selecionados, precisamos reconstruir a função com essa informação. Usando uma aproximação linear, ligamos os pontos adjacentes através de retas e obtemos uma aproximação da $f(x)$, como mostra a figura 4.3. Também comparamos nossa aproximação com o gráfico da função original, como mostra a figura 4.4.

Quanto mais pontos forem usados, mais precisa será a aproximação. No limite, com infinitos pontos teremos a aproximação perfeita da função. A escolha dos pontos também influencia nesse sentido.

Os agentes poderiam ser considerados os x, enquanto que $y = f(x)$ seria a sua aptidão. Os n pontos escolhidos para aproximar a função são os indivíduos objetivos e, finalmente, a aproximação entre dois pontos consecutivos, que no exemplo foi feita por uma função linear, é desempenhada pela forma de evolução escolhida. Claro que é difícil garantir que a transição entre dois objetivos seja tão controlada como no exemplo gráfico, e, como já foi discutido, a forma de realizar essa fase transitória pode ter uma influência na evolução dos agentes. Quanto mais suave for, melhor será o resultado final.

O uso da quarta variante, então, é interpretado como sendo uma função que, para um determinado intervalo $[x_1, x_2]$, comporta-se de forma mais ou menos constante independente do jogador e que para o intervalo $(x_2, x_3]$ tende a ser extremamente dependente da estratégia adversária e, portanto, bem mais imprevisível, ou até mesmo desconhecida. Neste caso, aproximaríamos a função para o primeiro intervalo e, ao chegar no segundo, liberaríamos a coevolução para ocorrer de acordo com o adversário.

4.2 Padrões Adaptativos

Como foi visto na seção 3.2.3, programar a inteligência de determinados agentes dentro de um jogo usando, para isso, o método de padrões é algo bastante comum, mesmo nos dias de hoje. A analogia com um programa em código de máquina mostra que há uma grande versatilidade e que há várias alternativas a serem exploradas.

Nesta seção propomos um método novo baseado na estratégia de padrões já descrita, porém tornando-a dinâmica e adaptativa. Este método é uma contribuição original deste trabalho.

O nosso primeiro passo será abstrair um pouco a definição (e a concepção) dos padrões. Procuremos, antes de mais nada, não nos preocupar com a implementação em si, abstraindo um pouco essa questão. Vamos supor que os padrões são determinadas entidades, e que essa definição é hierarquicamente relaxada. Em outras palavras, podemos tomar como um padrão uma linha de código, um trecho de código, uma função ou mesmo um programa inteiro. Não vai nos interessar, a princípio, o que representa aquele padrão, apenas que se trata de um padrão.

É possível, dessa forma, agrupar trechos de código (arbitrariamente grandes ou pequenos) como padrões. Claro que, na prática, ao fazer isso, alguns cuidados

deverão ser tomados, principalmente no que tange a alterações no fluxo de execução do programa, sejam elas absolutas ou relativas.

O uso desses padrões, porém, não precisa ficar restrito à esfera do comportamento de agentes. Boa parte dos jogos, como visto na seção 3.1, são subdivididos em fases ou estágios, através dos quais o jogador passa gradativamente. É possível pensar nos padrões como trechos de fases, ou mesmo como fases inteiras (a granularidade, neste caso, é tão flexível quanto em relação ao código). Supomos, portanto, que a especificação de uma determinada parte de uma fase (ou de uma fase inteira), pode ser encarada como um padrão e, da mesma forma, o método proposto aqui também se aplica a fases.

O objetivo de tornar todo o esquema de padrões como adaptativos é permitir que os mesmos se modifiquem de acordo com o nível de quem joga. Ou seja, de maneira simplificada, o objetivo é fazer com que a dificuldade do jogo cresça ou diminua, de acordo com o desempenho do jogador. Quanto melhor ele estiver jogando, mais difícil o jogo deverá ficar (e vice-versa).

De uma forma geral, jogos que possuem a figura do "chefe de fase" até hoje se utilizam do método de padrões. Esses agentes realizam uma série de tarefas e, para vencê-los, em geral o jogador tem de realizar uma outra determinada série de tarefas (que normalmente exigem um determinado nível de perícia) para poder derrotá-lo. Após vencer o agente algumas vezes, o jogador passa a se acostumar com aquela situação, "pega o jeito" de jogar contra aquele adversário, e logo o jogo começa a ficar bem mais fácil do que de início. Se, após ser derrotado por algumas vezes, o agente pudesse mudar o seu comportamento de forma a explorar algumas possíveis fraquezas detectadas no jogador, o jogo poderia não ficar fácil naquele estágio tão rapidamente. Além disso, pensando na fase como uma série de padrões, temos uma situação bastante parecida. Após jogar naquele ambiente por várias vezes, o jogador começa a se acostumar com seus desafios e tende a passar pela fase com muito mais facilidade. Se, da mesma forma que com os adversários, a fase pudesse ser dinâmica a ponto de explorar as fraquezas percebidas no jogador, aquela parte do jogo deixaria de ficar mais fácil durante o tempo (ou pelo menos poderia retardar em muito esse processo), tornando-se bem mais desafiadora a longo prazo. Por outro lado, uma determinada fase (ou adversário), que esteja sendo muito difícil para o jogador, talvez até o desencorajando a jogar por causa disso, poderia ficar mais fácil Sendo assim, o método permitiria facilitar um pouco as coisas, tornando uma

vitória possível. Obviamente, depois disso, a fase seguiria normalmente seu rumo de aumentar a dificuldade baseada nas informações obtidas das limitações do jogador.

A aplicação deste método não precisa ficar restrita a formas distintas de padrões dentro de um jogo (como adversários e fases). Outras aplicações que possam tirar vantagem de diferentes permutações de padrões com evolução de dificuldade podem utilizá-lo. Principalmente em questões de aprendizado, os padrões poderiam ser encarados como determinadas formas de conteúdo a serem exploradas, por exemplo.

4.2.1 Definições

Seja P o conjunto finito dos padrões distintos definidos dentro de um determinado contexto. Seja n a cardinalidade de P e, portanto, o número de padrões distintos disponíveis ($\|P\| = n$).

Uma permutação de padrões é uma seqüência $w = w_1 w_2 ... w_m$, tal que $w_i \in P, 1 \leq i \leq m$. Seja W o conjunto finito de seqüências de padrões de P. A cardinalidade de W, ou seja, a quantidade de seqüências do conjunto, é igual a k e, portanto $\|W\| = k$.

Para evitar ambigüidades, sendo w_i a i-ésima seqüência de W, então representamos w_i como $w_i = w_i^1 w_i^2 ... w_i^m$, onde m é o tamanho da seqüência ($\|w_i\| = m$). Logo, por w_i indicamos a i-ésima seqüência de W e por w_i^j o j-ésimo padrão pertencente à i-ésima seqüência de W. Referências por "seqüência" ou "permutação" são variantes da mesma nomenclatura, ou seja, uma $w_i \in W$.

Qualquer seqüência w_i não precisa ter tamanho n (ou seja, não necessariamente $n = m$). Dessa forma, um determinado padrão $p \in P$ pode aparecer um número arbitrário de vezes em w_i (inclusive nenhuma vez). De maneira mais genérica, as seqüências $w_i \in W$ não precisam ter o mesmo tamanho, embora isso possa acontecer.

Uma permutação representa a ordem com que os padrões serão executados por um determinado agente, ou como uma sucessão de eventos numa fase, por exemplo. Note-se que, dependendo da maneira com que os padrões forem agrupados, talvez não tenhamos como garantir que a execução dos padrões seguirá a ordem especificada em w_i. Um exemplo simples é haver um comando de mudança de fluxo em w_i^a para w_i^b, com $b \neq a + 1$.

Teoricamente nada impede que w_i possa ser uma seqüência infinita, mas para os propósitos deste trabalho isso não faz muito sentido. Iremos supor, portanto, que w_i é sempre finita. Vale a pena notar que isso não significa que a execução de w_i

será finita (pelo menos em teoria), uma vez que, chegando ao final de w_i^m, pode-se voltar a w_i^1. Em outras palavras, a execução da permutação w_i pode ser feita como um laço infinito (até que o agente seja vencido, por exemplo) ou simplesmente de forma finita (como uma fase, por exemplo).

Seja W um conjunto finito de seqüências. Para os propósitos deste trabalho iremos considerar as seguintes propriedades que devem ser observadas em W:

(1) $w_i \in W \Rightarrow \|w_i\| > 0$. Ou seja, não há seqüências vazias, sem padrões.

(2) $w_i, w_j \in W \Rightarrow \|w_i\| \neq \|w_j\|$ ou $\exists a$ tal que $w_i^a \neq w_j^a$. Ou seja, não há seqüências idênticas, o que podemos indicar simplesmente por $w_i = w_j \Leftrightarrow i = j$.

A primeira propriedade nos garante que não haverá padrões vazios para se jogar, por exemplo, um agente que não faz nada ou uma fase vazia, o que, certamente, não é nada desejável. Já a segunda nos garante que não há seqüências idênticas em W, o que é dispensável, já que não acrescenta nada ao conjunto de possibilidades.

Além disso, pelo exposto anteriormente, temos que os elementos de W não necessariamente possuem cardinalidade fixa, ou seja, podem ter tamanhos distintos entre si (embora nada impeça que também possam ser todos do mesmo tamanho).

Seja $f(w_i)$ uma função qualquer de avaliação da seqüência $w_i \in W$. Ao aplicar $f(.)$ para todas os elementos de W, podemos criar uma ordenação dessas seqüências baseada nos valores retornados por $f(.)$. Seja $X = \{x_1, x_2, ...x_k\}$ o conjunto das seqüências de W em ordem não crescente usando-se a função de avaliação $f(.)$. Temos, portanto, que $f(x_i) > f(x_j) \Leftrightarrow i < j$. Obviamente, cada $x_i \in X$ equivale a exatamente um $w_j \in W$.

Se considerarmos a função $f(.)$ como uma forma ideal de avaliação de todas as seqüências de W, então X será um conjunto ideal que conterá todas as permutações de padrões ordenadas, começando pela mais difícil e terminando com a mais fácil. O conjunto X é criado cada vez que $f(.)$ é aplicada em W, afinal X nada mais é do que W ordenado, usando-se $f(.)$ para isso. Como os valores de $f(.)$ podem mudar ao longo do tempo, então X possivelmente estará constantemente mudando sua ordem.

Por uma questão prática, vamos designar por $\emptyset$ o padrão nulo, ou seja, o padrão w_i tal que $\|w_i\| = 0$. Muito embora não admitamos um padrão nulo como membro do conjunto W, a definição de tal padrão nos será útil no futuro.

Temos, agora, todas as definições de que precisamos para que possamos desenvolver o método proposto. Supondo ser a forma mais fácil de expor seu funcionamento, seguiremos de cima para baixo. Ou seja, iniciaremos explicando como o método

em si funciona, supondo já resolvidos problemas específicos para, em seguida, nos concentrarmos em tais detalhes.

4.2.2 Descrição Genérica

Dadas as definições de que faremos uso e supondo que todos os elementos foram devidamente definidos (como isso é feito será visto nas seções seguintes), podemos descrever, de uma maneira amplamente genérica, o funcionamento do método da seguinte forma:

Algoritmo 4.2 Forma Geral dos Padrões Adaptativos

```
criar os padrões a serem usados (conjunto P)
criar as permutações de padrões a serem usadas (conjunto W)
eleger uma permutação qualquer w_i ∈ W como a inicial
enquanto não terminar o jogo faça
    após uma partida completa aplicar f(.) para toda permutação w_i ∈ W
    formar o conjunto ordenado X a partir dos f(.) calculados
    a partir de X, escolher a nova permutação a ser usada na próxima partida
fim enquanto
```

Esta é a idéia geral de aplicação do método. É possível que haja, obviamente, algumas variações e não necessariamente o método será traduzido exatamente como exposto. Mas o pseudo-algoritmo descrito acima serve para especificar uma espécie de linha de raciocínio para sua aplicação. Nas seções seguintes, iremos detalhar os passos necessários e como preencher as lacunas, bem como estender o uso de padrões de forma a flexibilizá-los um pouco mais e adaptá-los a diferentes situações de uso.

4.2.3 Utilizando as Permutações

Vamos supor, primeiramente, que já temos o conjunto W definido e que este será constante (ou seja, ele não será alterado durante todo o curso do método). Supomos, além disso, que a função de avaliação $f(.)$ já foi definida e iremos apenas aplicá-la. Nas seções seguintes, veremos formas de criar W a partir dos padrões disponíveis, como fazer e aplicar a função de avaliação e também como estender o método para permitir um W dinâmico, entre outros detalhes, mas por enquanto isso não é o mais importante.

Tendo W e $f(.)$, podemos, em um determinado momento, aplicar $f(.)$ a todos os elementos de W e, assim, obter X. Uma forma simples de definir a estratégia seria utilizar como padrão, sempre, x_1, ou seja, o mais difícil de todos. Como já foi discutido anteriormente, porém, isso pode não ser exatamente o objetivo, já que, em algumas situações, podemos também querer facilitar as coisas para o jogador. Além disso, mesmo que não seja esse o caso, usar a permutação mais difícil de todas possivelmente estaria exagerando em muito a dificuldade do jogo, enquanto que o ideal seria usar uma seqüência com um incremento de dificuldade mais suave com relação à atual. Apesar dessas ponderações todas e das desvantagens apresentadas, talvez se queira, por algum motivo, utilizar essa estratégia, que é extremamente simples, elegendo-se x_1 sempre como a próxima seqüência a ser usada pelo jogo.

Uma outra estratégia igualmente simples é usar como próxima permutação a que estiver uma posição acima da atual na nova arrumação. Ou seja, digamos que $w_i \in W$ seja a permutação atual. Após a jogada, aplica-se $f(.)$ a todos os elementos de W e gera-se X. Assim, se $w_i = x_j$, então a próxima seqüência a ser utilizada será $w_i = x_{j-1}$ (se $j > 1$) ou $w_i = x_j$ (se $j = 1$). O incremento de dificuldade, portanto, será teoricamente o menor possível, tornando mais suave a gradação. Essa estratégia tem as desvantagens de não prever a questão de facilitar o jogo quando muito difícil e de estar sempre tornando o jogo mais difícil, não importa quão bem (ou mal) o jogador estiver se saindo.

Uma forma um pouco mais flexível de encarar essa estratégia seria definir um limite de dificuldade, fazendo com que a seqüência só fosse incrementada em dificuldade caso esse limite fosse quebrado. Suponhamos que d seja um limite tal que se $f(w_i) < d$ (sendo $w_i \in W$ a permutação atual que está sendo usada) então concluímos que a fase ficou fácil demais para o jogador e, conseqüentemente, a estratégia descrita no parágrafo anterior passa a ser utilizada. A definição de d depende, obviamente, de $f(.)$, além de ser um valor de natureza heurística (assim como a própria $f(.)$, conforme veremos adiante).

Essa última estratégia, da forma como foi descrita, não prevê o caso de se tornar o jogo mais fácil, caso se queira. Se considerar essa possibilidade for realmente desejado, então pode-se estender a definição de d separando-a em d_{sup} e d_{inf}. Se $f(w_i) < d_{inf}$ então segue-se como definido anteriormente. Se, por outro lado, $f(w_i) > d_{sup}$, então concluímos que o jogo está difícil demais e podemos agir de forma análoga ao caso anterior, porém usando a permutação imediatamente pos-

terior (mais fácil) à atual. Ou seja, se $w_i = x_j$, então atribuimos $w_i = x_{j+1}$ (se $j < \|X\|$) ou $w_i = x_j$ (se $j = \|X\|$). Finalmente, se $d_{inf} \leq f(w_i) \leq d_{sup}$, então mantém-se a mesma seqüência. A estratégia utilizada no parágrafo anterior pode ser considerada como uma variação desta, bastando, para isso, fazer $d_{inf} = d$ e $d_{sup} = +\infty$.

O uso desses limites é flexível o suficiente para permitir algumas particularidades, se estas forem desejadas. É possível usar, por exemplo, valores de limites diferentes para as fases. Sendo assim, atribuindo-se de forma pertinente os valores, tornamos as fases iniciais menos suscetíveis ao aumento do nível de dificuldade (tornando-o mais lento) e fazendo o contrário com as últimas (o mesmo se aplicaria a possíveis chefes de fase). Outra possibilidade seria evitar qualquer mudança do padrão inicial antes de um número arbitrário de jogadas. Uma fase, por exemplo, seria sempre a mesma nas primeiras vezes em que o jogador jogasse, só havendo possibilidade de mudança após ter jogado um determinado número de vezes. Para isso, basta atribuir $d_{inf} = -\infty$ e $d_{sup} = +\infty$ para essas jogadas iniciais.

4.2.4 Avaliando as Permutações

Até este momento, supomos que a função de avaliação $f(.)$ estava definida e que ela conseguia, de forma ideal, avaliar os elementos de W. Uma função perfeita que avalie idealmente a dificuldade apresentada por cada permutação de W está longe da realidade, então iremos descrever algumas formas de tentar defini-la da melhor forma possível.

Vamos, antes de mais nada, estabelecer alguns critérios possíveis de avaliação da dificuldade das permutações de W. O que queremos é, a partir das partidas jogadas, estimar uma avaliação para todas os elementos de W. Isso, provavelmente, será uma tarefa especialmente ingrata, pois se já é difícil avaliar a dificuldade apresentada pela própria seqüência que foi jogada, extrapolar essa avaliação para seqüências ainda não jogadas será uma tarefa ainda mais árdua.

A função $f(.)$, assim como a função de avaliação usada normalmente em algoritmos genéticos, tem uma natureza essencialmente heurística. Defini-la, portanto, é uma tarefa que difere muito de uma determinada aplicação para outra. É possível, todavia, ressaltar alguns métodos gerais de defini-la que podem, ou não, ser usados em conjunto, como veremos mais adiante.

Antes de mais nada, como já foi dito, pretendemos, a partir dos resultados das

partidas disponíveis, estimar a dificuldade que todas as seqüências apresentariam àquele determinado jogador (e naquele determinado instante), e o fato de todos os padrões disponíveis nas seqüências de W serem conhecidos é uma vantagem. Sim, pois $\forall w_i^j \in w_i, w_i \in W \Rightarrow w_i^j \in P$, ou seja, qualquer padrão que apareça em alguma seqüência de W é um padrão de P e, portanto, conhecido. Ora, se estimarmos a avaliação da dificuldade de uma determinada seqüência baseada nos padrões que ela contém, poderemos aproveitar essa avaliação para as demais seqüências. De um forma geral, é isso o que iremos fazer.

Independente da forma de avaliar os padrões, que veremos a seguir, os valores obtidos serão guardados em memória. Isso é razoavelmente óbvio, pois após uma jogada, com os padrões avaliados, iremos utilizar esses valores para estimar o nível de dificuldade das demais seqüências usando, para isso, os padrões que as formam. Temos, portanto, de manter os valores calculados em memória para podermos utilizá-los nos cálculos futuros.

A forma de avaliação mais direta possível seria, durante a partida, avaliar a dificuldade de cada padrão. Suponhamos, então, que um determinado padrão w_i^j tenha um valor dado por uma função $\phi(w_i^j)$. Teremos, então, uma estimativa para todas as seqüências baseada na avaliação dos padrões obtida através da aplicação de $\phi(.)$. Assim, definimos $f(.)$ em função de $\phi(.)$ como:

$$f(w_i) = \phi(w_i^1) + \phi(w_i^2) + ... + \phi(w_i^m) = \sum_{j=1}^{m} \phi(w_i^j) \tag{4.1}$$

Uma outra forma de avaliar os padrões seria fazê-lo de acordo com sua posição na seqüência. Sendo assim, cada padrão precisaria ter até m avaliações, supondo ser m o tamanho máximo das permutações em W. Seja $\varphi_j(w_i^j)$ a função que avalia o padrão w_i^j na posição j. Então poderíamos atribuir:

$$f(w_i) = \varphi_1(w_i^1) + \varphi_2(w_i^2) + ... + \varphi_m(w_i^m) = \sum_{j=1}^{m} \varphi_j(w_i^j) \tag{4.2}$$

Uma maneira diferente de encarar a questão da posição do padrão na seqüência é levar em conta os padrões vizinhos a ele. Ou seja, se $w_i \in W$ é uma seqüência, então $w_i = w_i^1 w_i^2 ... w_i^m$ e os vizinhos do padrão w_i^j em w_i são w_i^{j-1} e w_i^{j+1}, com exceção de w_i^1 e w_i^m, que só têm um vizinho. Atribuindo, porém, o padrão nulo como seus vizinhos, podemos considerar cada padrão como tendo dois vizinhos. Sendo assim, e considerando n a quantidade de padrões ($n + 1$, se levarmos em conta o padrão

nulo), são $(n+1)^2$ combinações de vizinhos para cada padrão. Seja, então, a função $\xi_{w_i^{j-1}w_i^{j+1}}(w_i^j)$ que avalie o padrão w_i^j quando tem como vizinhos os padrões w_i^{j-1} (à esquerda) e w_i^{j+1} (à direita). Considerando $w_i^0 = w_i^{m+1} = \emptyset$, podemos atribuir:

$$f(w_i) = \xi_{w_i^0 w_i^2}(w_i^1) + \xi_{w_i^1 w_i^3}(w_i^2) + ... + \xi_{w_i^{m-1} w_i^{m+1}}(w_i^m) = \sum_{j=1}^{m} \xi_{w_i^{j-1} w_i^{j+1}}(w_i^j) \quad (4.3)$$

Vale destacar que, para o caso de a seqüência estar sendo usada em forma de laço (ou seja, ao terminar o último padrão volta ao primeiro), podemos encarar a permutação como sendo circular e, portanto, passamos a ter $w_i^0 = w_i^m$ e $w_i^{m+1} = w_i^1$.

Pode-se variar a interpretação da função $\xi(.)$ atribuindo-a apenas em relação ao padrão anterior. Ou seja, considera-se que o padrão seguinte não influencia em nada na dificuldade do atual, embora se admita que o anterior o faça. Logo, a função seria definida como

$$f(w_i) = \xi_{w_i^0}(w_i^1) + \xi_{w_i^1}(w_i^2) + ... + \xi_{w_i^{m-1}}(w_i^m) = \sum_{j=1}^{m} \xi_{w_i^{j-1}}(w_i^j) \quad (4.4)$$

Além de usar essas funções separadamente, é igualmente possível utilizá-las em conjunto, somando-as. Por exemplo, podemos fazer

$$f(w_i) = \alpha_1 \sum_{j=1}^{m} \phi(w_i^j) + \alpha_2 \sum_{j=1}^{m} \varphi_j(w_i^j) + \alpha_3 \sum_{j=1}^{m} \xi_{w_i^{j-1} w_i^{j+1}}(w_i^j) \quad (4.5)$$

Sendo $\alpha_1, \alpha_2, \alpha_3 \in \Re$ pesos a serem atribuídos a cada tipo de avaliação. Determiná-los também constitui uma tarefa puramente empírica, sendo difícil avaliar com precisão os impactos de cada função *a priori*. Podemos, porém, analisar, de uma forma geral, as vantagens e desvantagens inerentes a cada uma delas, o que pode ajudar numa decisão a viabilidade ou não de seu uso e sua importância.

A função $\phi(.)$ tem a grande desvantagem de, sendo aplicada sozinha, poder ser simplesmente inútil. Se tivermos um conjunto W composto apenas por permutações dos padrões em P tais que um determinado padrão $p \in P$ ocorre uma (e apenas uma) vez em cada seqüência $w_i \in W$, então o valor de $\sum \phi(w_i^j)$ será constante, o que não é de muita utilidade. Além disso, essa avaliação simplesmente considera que a posição e a ordem em que os padrões ocorrem em nada influenciam sua dificuldade para o jogador, o que nem sempre pode ser ignorado. A grande vantagem, porém, é sua simplicidade e o baixo consumo de memória, afinal, é necessário apenas uma quantidade de memória proporcional a $O(n)$ para guardar os valores (onde $n = \|P\|$).

A função $\varphi(.)$ não apresenta o mesmo risco que a anterior de ter o valor de seu somatório constante para todos os padrões de W. Por precisar, contudo, levar em conta a posição do padrão na seqüência, requer mais espaço de memória. Sendo m o tamanho da maior seqüência de W, então a memória requerida para a avaliação é proporcional a $O(nm)$, já que é necessário guardar os valores retornados pela função para cada possível posição na seqüência de cada padrão de P.

Finalmente, a função $\xi(.)$ também tem um acréscimo considerável na quantidade necessária de espaço de memória para armazenar seus valores. Como já foi visto, cada padrão possui $(n+1)^2$ possíveis vizinhos, sendo n a quantidade de padrões. Se cada um dos n padrões requer $(n+1)^2$ vizinhos a serem armazenados, então a complexidade de espaço de memória é proporcional a $O(n^3)$. Essa análise, obviamente, se modifica caso se decida usar a variante discutida anteriormente, definindo a função em relação apenas ao padrão anterior. Neste caso, para cada um dos n padrões é necessário guardar $n+1$ valores e, portanto, a complexidade de memória passa a ser proporcional a $O(n^2)$.

Se as três avaliações forem utilizadas em conjunto, levando em conta (4.3) ao invés de (4.4), temos então um gasto de memória proporcional a $O(n^3+nm+n) = O(n^3+nm)$. Para um número alto de padrões e tamanhos de seqüência longos, esse valor pode vir a ser proibitivo. Para a maioria das aplicações, entretanto, muito provavelmente o gasto não vai ser dos maiores. Se tivermos, por exemplo, 50 padrões (um número relativamente alto), com um tamanho máximo de seqüência de 1000 (provavelmente mais do que qualquer jogo poderá precisar), o gasto de memória será perto da casa dos 200 Kb, o que não chega a assustar.

Uma outra preocupação natural diz respeito à eficiência do cálculo de $f(.)$, e seu custo computacional. O fator de maior influência, neste caso, é $\|W\|$. Afinal, cada padrão presente na seqüência sendo jogada vai ser avaliado um número constante de vezes, logo teremos um gasto de tempo linear para o cálculo de cada padrão. Terminada a partida, os valores calculados serão guardados nas respectivas matrizes e, a partir de então, a função $f(.)$ deverá ser calculada para cada permutação de W. Sendo $\|W\| = k$, supondo $f(.)$ definida como em (4.5), com três somas (número constante) de somatórios de valores de padrões, e sendo m o maior tamanho de seqüência em W, então teremos $O(3m) = O(m)$ somas para cada aplicação de $f(.)$. Como são necessários k cálculos de $f(.)$ (cada um requerendo $O(m)$ somas), então temos $O(km)$ somas necessárias para o cálculo de $f(.)$ para todas as seqüências.

Além disso, é preciso ordenar o conjunto X, como já explicado, o que requer tempo $O(k \log k)$, já que $\|X\| = \|W\| = k$. Como temos, inicialmente, um tempo gasto proporcional a $O(m)$ para calcular os valores dos padrões da seqüência sendo jogada (como dito de início), então a complexidade total do método é proporcional a $O(m + km + k \log k) = O(km + k \log k)$. Como geralmente (mas não necessariamente) $k >> m$, então a grande influência no custo computacional do método será k, o número de seqüências disponíveis (o que é mais ou menos intuitivo). Para a grande maioria das aplicações, esse custo não será um grande empecilho para a aplicação do método.

Por fim, uma última preocupação diz respeito à atualização dos valores calculados para os padrões. Com o decorrer do tempo, eventualmente um determinado valor já calculado (em qualquer um dos três casos discutidos) para um certo padrão será novamente calculado. O que fazer com o valor previamente definido? Há várias formas de se lidar com esse tipo de problema. A mais simples de todas seria simplesmente sobrescrever o valor. Afinal, por ser um cálculo mais recente a respeito da mesma situação, supostamente o novo valor deve ser mais preciso com relação à situação atual do jogador do que o anterior. Não deixa de ser verdade, de certa forma. Como não podemos, porém, esperar uma curva de aprendizado do jogador bem linear, sem altos e baixos, simplesmente desprezar o valor antigo pode não ser uma boa idéia. Até porque é possível que o novo valor tenha vindo numa jogada seguinte, ou bem próxima, à última. Uma outra forma simples de se lidar com esse problema seria fazer uma média (possivelmente com pesos diferentes) entre os últimos x valores, para algum valor constante de x. Outras estratégias diferentes também podem ser utilizadas. Por exemplo, usar a média, porém descartar o valor anterior e recomeçar com o novo, caso um número arbitrário de jogadas tenha passado (ou seja, supostamente a média estaria defasada). Um outro caso seria descartar a média, se a diferença entre o valor atual e o novo fosse maior que um determinado limite. É possível, inclusive, usar uma estratégia híbrida. Por exemplo, para a avaliação absoluta do padrão usa-se sempre o último valor calculado, enquanto que para as demais (relativas à posição e aos vizinhos), usa-se a média. Enfim, há inúmeras possibilidades.

4.2.5 Como Avaliar os Padrões

Na seção anterior, descreveu-se como calcular a função de avaliação baseando-se, para isso, no uso de outras funções para avaliar cada padrão dentro de uma determinada partida de diferentes maneiras. Supôs-se, no entanto, que esses valores já estavam calculados sem entrar muito em detalhes.

Avaliar a dificuldade de um determinado padrão é uma tarefa puramente empírica. É impossível aferir com precisão esse valor, ainda mais de uma forma universal, que valha para todos os jogadores. Quanto mais justa, porém, for essa aproximação, melhor o método, como um todo, tenderá a funcionar.

A definição da avaliação dos padrões é muito semelhante à função de *facilidade* discutida na seção 4.1.4. Alguns parâmetros de jogo podem ser usados de forma a fazer a avaliação da dificuldade que aquele trecho (padrão) de jogo apresentou ao jogador. Dados como vidas perdidas e pontos conquistados podem ter um peso grande no cálculo final do valor, por exemplo. O conhecimento do jogo em questão no qual o método está sendo aplicado é fundamental, pois com este é possível definir com maior segurança quais parâmetros são indicadores confiáveis de quão fácil ou difícil aquele padrão foi para o jogador.

Uma outra questão complicada é o cálculo repetido de um mesmo padrão. Se o método está sendo usado para o comportamento de um agente, é muito provável que haja um laço que fique repetindo a seqüência atual e, portanto, um mesmo padrão vai ser jogado por diversas vezes na mesma jogada. Isso pode igualmente ocorrer mesmo com os padrões sendo usados como trechos de fase, afinal, é comum acontecer de o jogador voltar a um mesmo lugar em que esteve anteriormente. Lidar com esse problema vai depender de como a função para avaliar o padrão foi definida e das próprias características. Se ela depender do tempo gasto, por exemplo, soma-se o valor de todas as "visitas". Se depender do número de vidas perdidas ou pontos ganhos, idem. Em alguns casos, contudo, esses mesmos parâmetros somados talvez não façam muito sentido. No caso do tempo, por exemplo, talvez apenas nos interesse saber quanto tempo levou para o personagem conseguir sair de um determinado trecho do jogo (representado pelo padrão), de forma que somar os tempos vai levar a uma avaliação errônea. Neste caso seria mais acertado considerar apenas a última avaliação, ou até uma média entre as avaliações (em caso análogo ao discutido na última seção).

O cálculo diferenciado ou não das funções também pode ser um problema. Por exemplo, é possível retornar o mesmo valor (ou seja, usar a mesma função de avaliação) para um determinado padrão tanto na forma absoluta quanto, digamos, em relação ao vizinhos. Ou seja, não estamos usando funções diferentes, estamos apenas aproveitando o mesmo valor retornado em diferentes contextos (no caso mais específico, guardando o mesmo valor obtido em matrizes diferentes). Ao invés disso, podemos, de fato, definir funções diferentes para as diferentes situações. Como um exemplo simples, determinamos:

$$\phi(w_i^j) = t_{w_i^j} \tag{4.6}$$

$$\varphi_j(w_i^j) = en_{w_i^j} \tag{4.7}$$

$$\xi_{w_{j-1}^j w_{j+1}^j}(w_i^j) = K * \frac{pts_{w_i^j}}{tot_{w_i^j}} + rec_{w_i^j} \tag{4.8}$$

Onde $t_{w_i^j}$ é o tempo que o jogador levou em w_i^j, $en_{w_i^j}$ é a quantidade de energia perdida pelo jogador em w_i^j, K é uma constante qualquer, $pts_{w_i^j}$ é a quantidade de pontos conquistada em w_i^j pelo jogador, $tot_{w_i^j}$ é a quantidade total de pontos em disputa em w_i^j e $rec_{w_i^j}$ é uma quantidade qualquer de recursos (por exemplo, tiros) que foram desperdiçadas durante a passagem por w_i^j.

4.2.6 Determinando as Permutações

Falta-nos, ainda, especificar de que forma podemos determinar o conjunto W de permutações para que possa ser utilizado pelo método. De forma geral, existem várias maneiras de se chegar a esse conjunto, e descreveremos aquelas que consideramos mais convenientes.

A forma mais "segura", digamos assim, de criar W seria determinar todas as permutações desejadas, uma por uma. A grande vantagem dessa estratégia é ter um controle completo sobre todas as possíveis permutações a serem utilizadas. A grande desvantagem, porém, é justamente o imenso trabalho de determinar todos os elementos de W um a um, o que pode ser particularmente inviável se a intenção for ter uma grande quantidade de seqüências possíveis.

Uma outra maneira muito simples de criar W seria simplesmente gerar todas as permutações de padrões de P. Além de ser muito simples, temos todas as com-

binações possíveis de padrões em W. A desvantagem óbvia é a quantidade extremamente grande de seqüências, o que talvez torne o seu uso impraticável.

Outro grande problema é o fato de não se ter controle sobre as permutações geradas, de forma que alguns elementos de W podem acabar sendo muito pouco úteis quando usados realmente num jogo, ou mesmo nem funcionar direito. Afinal, ao se criar os padrões, é possível que haja determinadas restrições quanto ao seu aparecimento numa seqüência. Por exemplo, um padrão $p \in P$ pode não fazer sentido (por algum motivo qualquer) se aparecer depois de um outro padrão $q \in P$ numa permutação.

Vamos supor, para refinar um pouco o problema, que cada padrão de P seja um nó de um grafo orientado $G = \{V, E\}$, logo $V = P$. Podemos representar uma restrição como uma aresta nesse grafo. Por exemplo, se desejamos que $p \in P$ necessariamente ocorra antes de $q \in P$, então adicionamos uma aresta (p, q) a E. Ao final do processo, o grafo G terá $V = P$ e E contendo todas as restrições desejadas. Qualquer ordenação topológica em G vai retornar uma ordem que satisfaça a todas as restrições (se houver tal ordem).

Mais formalmente, uma ordenação topológica de um grafo orientado $G = \{V, E\}$ é uma ordenação linear de todos os vértices de V tal que se $(u, v) \in E$ então u aparece antes de v na ordenação (CORMEN et al, 2002).

Essa ordenação, porém, não necessariamente é única, pois pode haver (e provavelmente haverá) diversas possibilidades de ordenação que satisfaçam as restrições. Por exemplo, se $V = \{a, b, c\}$ e $E = \{(a, b), (c, b)\}$, então acb e cab são seqüências válidas, pois em ambas a ocorre antes de b, assim como c antes de b.

Construído, então, o grafo G tal como descrito, podemos aplicar o algoritmo proposto em (KNUTH, SZWARCFITER, 1974) para retornar todas as ordenações topológicas de G, que formarão justamente o conjunto W desejado. Esse algoritmo tem complexidade linear no número de ordenações, mas é claro que essa quantidade pode ser exponencial no número de nós.

Porém, mesmo que a quantidade de elementos a ser gerada para W seja particularmente grande, a aplicação do algoritmo só será feita uma vez, com seu resultado sendo gravado em disco. A grande preocupação, portanto, com o tamanho de W é no que diz respeito à aplicação da função de avaliação, como já foi analisado na seção 4.2.4.

A principal desvantagem de usar as ordenações topológicas, tal como foi pro-

posto, é o fato de que todas as seqüências geradas terão, necessariamente, o mesmo tamanho. Isso, contudo, dependendo da aplicação, talvez não seja nenhum problema. Pode-se, por outro lado, desejar que as seqüências tenham comprimentos variados, e, então, é preciso que algumas mudanças sejam feitas.

Uma solução simples (e trabalhosa) seria criar diferentes grafos G, aplicar o algoritmo de geração de ordenações topológicas para cada um deles e fazer a união dos conjuntos gerados. Uma outra solução seria usar um tipo de padrão especial, de "corte" (representado por x), que seria interpretado como o fim da seqüência, ignorando o que vier a seguir. Por exemplo, sejam $G = \{V, E\}, P = \{a, b, c, d\}, V = P \cup \{x\}$ e $E = \{(a, x), (a, c), (a, d), (b, x), (b, c), (b, d), (c, d)\}$. Aplicando o algoritmo para encontrar as ordenações topológicas, teremos $W = \{abxcd, baxcd, abcxd, bacxd, abcdx, bacdx\}$. Se interpretarmos o x como "corte" (como se fosse o caractere NULL em C), passamos a ter $W = \{ab, ba, abc, bac, abcd, bacd\}$.

4.2.7 Considerações

Após verificar os principais elementos do método de padrões adaptativos, cabe aqui uma pequena discussão a respeito de alguns de seus aspectos, bem como de extensões do mesmo.

Podemos, antes de mais nada, encará-lo como uma espécie de metáfora, ou analogia, de algoritmos genéticos. Se pensarmos nos padrões como um bit de uma string de bits, uma determinada seqüência pode ser encarada como o código de um indivíduo, e o conjunto W como toda a população. Da mesma forma a função de avaliação tem um papel muito semelhante nos dois casos. Há, entretanto, algumas diferenças marcantes. No caso dos padrões, o principal objetivo é manter o nível de dificuldade (ou, de forma geral, a função de avaliação) dentro de um determinado intervalo, ou seja, nem tão fácil, nem tão difícil. Já os algoritmos genéticos têm essencialmente a otimização como objetivo, buscando sempre o melhor indivíduo (ou grupo de indivíduos) possível. Como diferença prática, salta aos olhos o fato de não haver reprodução nos padrões, já que a população inteira já está definida em W. Embora esse conjunto não precise ser estático (como veremos em seguida), não ocorre seleção de indivíduos para reprodução, nem outros operadores (como mutação). Usando a metáfora dos algoritmos genéticos, é como se já tivéssemos a população inteira definida, selecionássemos, através de algum critério, um indivíduo (que seria a seqüência de padrões a ser utilizada) e, a partir da avaliação deste, ge-

neralizássemos a avaliação para toda a população. Não há novas gerações e a seleção que é feita diz respeito ao indivíduo que será "jogado".

A noção de população faz muita diferença neste caso. Quando descrevemos a coevolução em tempo real na seção 4.1, supusemos que havia uma quantidade razoável de agentes jogando simultaneamente, possivelmente em cooperação. No caso dos padrões, há apenas um indivíduo (sendo assim entendida uma permutação) jogando em um determinado momento e, portanto, não há diversidade suficiente que justifique uma abordagem plenamente evolucionista. Além disso, normalmente há mais restrições a respeito de quais combinações de genes (sendo metáforas para padrões) são válidas ou não e, sendo apenas um indivíduo, criar uma seqüência inviável pode resultar em conseqüências desastrosas. Por exemplo, uma fase sem saída seria algo desastroso de ocorrer. No caso evolucionista, tendo vários agentes simultâneos, se acontecer de um deles simplesmente ficar parado sem fazer nada (e ser facilmente destruído em decorrência disso) não é um acontecimento grave que prejudique consideravelmente o jogo.

Essa comparação com a abordagem evolucionista traz à luz uma questão interessante, que é o fato de o conjunto W ter sido considerado, até então, como estático. De fato, não há nada, teoricamente, que o impeça de ser dinâmico. Uma alternativa para evitar que W fosse sempre estático seria criar diferentes conjuntos (digamos W_1, W_2 etc.) e acrescentar os diferentes grupos a W no decorrer das partidas. Por exemplo, inicia-se $W = W_1$ e sempre que uma determinada condição for satisfeita (por exemplo, passar um número certo de rodadas) fazer $W = W \cup W_i$. Esta técnica pode ser particularmente útil quando se deseja evitar, por alguma razão, que determinadas seqüências possam ser jogadas antes de alguma condição acontecer. Outra forma, que seria útil, de fazer W dinâmico seria apagar as seqüências que estiverem com o valor de $f(.)$ muito abaixo do limite inferior (ou seja, que já ficaram muito fácil), para economizar memória e, principalmente, melhorar o desempenho (menos seqüências = menos avaliações). Usada em conjunto com a tática anterior de definir diversos W, pode manter o conjunto atual muito menor do que a quantidade total possível de seqüências, melhorando consideravelmente o desempenho. Uma desvantagem é o fato de inviabilizar o uso das seqüências descartadas como, por exemplo, no caso de um novo jogador inexperiente começar a jogar o mesmo jogo. Para contornar esse problema, uma solução seria usar perfis, de forma que um novo jogador teria de criar um novo perfil que seria iniciado do zero.

Outra questão importante é a instabilidade previsível no início de aplicação do método. Por exemplo, após uma única partida, poucas serão as avaliações feitas, afinal, a quantidade de dados disponível será apenas a da partida em questão. Logo, muitas seqüências terão sua avaliação incompleta. Uma solução para isso é fazer um preenchimento "default" dos valores tabelados das funções com base em jogos realizados, por exemplo, por *beta-testers*. É possível usar uma média dos cálculos de cada valor tabelado em suas primeiras avaliações, ou mesmo fazer uma estimativa empírica dos mesmos, apenas para que tenham um valor inicial preenchido. Esse tipo de procedimento é semelhante, lembrando a analogia com o caso coevolucionário, à idéia de se usar os indivíduos objetivos descritos na seção 4.1.3. A diferença é que, neste caso, não se estimam fases da evolução do jogo, apenas os valores iniciais para que o método possa funcionar a partir dos mesmos. É claro que, não necessariamente, os valores iniciais obtidos de alguma forma em uma fase anterior vão ser exatamente os mesmos que o jogador encontrará. Em outras palavras, é possível que as dificuldades do jogador sejam diferentes das estimadas inicialmente. Isso não chega a ser um grande obstáculo, contudo, uma vez que os valores tabelados vão tender a se ajustar, ao longo das partidas, às particularidades do jogador.

Alguns problemas também podem ser previstos no uso deste método. Um deles é o fato de que um determinado padrão que tenha baixa dificuldade mas que apareça muitas vezes em uma seqüência w_i pode fazer com que ela seja avaliada como mais difícil do que uma w_j que tenha padrões fáceis e um padrão extremamente difícil. É provável que seja mais fácil o contrário, ou seja, conseguir passar por w_i, que simplesmente tem vários padrões com pouca dificuldade. Uma forma de tentar contornar isso seria evitar que esses tipos de seqüências tão diferentes internamente sejam criadas. Se isso não for possível, ou previsível, uma forma seria aplicar algum modificador de peso, ou uma média, aos padrões da seqüência para minimizar o efeito de repetidos padrões de pouca influência, amplificando a contribuição de um único padrão que tenha uma avalição muito acima ou abaixo dos demais. Esse problema pode assumir uma outra forma para seqüências de tamanhos diferentes, supondo que uma w_i tenha muitos padrões fáceis e uma w_j tenha exatamente os mesmo padrões, porém em menor quantidade. É bem possível que, idealmente, seja desejado que a avaliação de ambas seja igual ou pelo menos bem próxima, o que não vai acontecer, uma vez que w_i acabará avaliada como sendo muito mais fácil que w_j. Esses modificadores de peso ou média também podem ser úteis nesse caso,

auxiliando a manter valores mais ou menos próximos.

De uma forma geral, não é muito complicado de prever que tamanhos muito diferentes de seqüências podem trazer problemas para a avaliação das mesmas, pelo menos usando as mesmas funções. Essa desproporcionalidade torna uma avaliação igual para todas as seqüências uma tarefa extremamente árdua, pois os próprios disparates de tamanho já são uma enorme influência no cálculo que podem ou não se refletir na prática.

Outro problema é a possibilidade de ocorrer um padrão muito difícil logo no início de uma seqüência, fazendo com que a partida termine logo. Quando isso acontece, os demais padrões não serão avaliados, evitando uma estimativa desses para todos os casos. Essa pode ser uma questão mais complicada principalmente no início de jogo, pelo fato de se ter tão poucas informações a respeito dos demais padrões, como já foi exposto.

Um problema sério que pode acontecer é quando não há seqüências com avaliação dentro dos limites. Há várias causas possíveis para tal. É possível que a própria avaliação esteja incoerente e não seja bem feita. Outra possibilidade é uma quantidade imprópria de seqüências. Numa situação limite exagerada, se houver apenas uma seqüência, a tendência é haver nenhuma dentro dos limites. Uma outra causa, essa mais óbvia, seria os limites estarem definidos de maneira incoerente. Essa situação de fugir dos limites, a princípio, nunca deveria acontecer pois, afinal, o objetivo do método é justamente sempre manter a dificuldade dentro de um determinado intervalo. Não é possível, entretanto, prever que não vá ocorrer e, assim, é necessário que se preveja um tratamento para o caso de a situação ocorrer. É possível resolver isso de três formas simples: pegando a primeira seqüência com $f(w_i) > d_{sup}$, com $f(w_j) < d_{inf}$ ou então, sendo w_i e w_j exatamente essas seqüências anteriores, escolher a de menor diferença em relação ao limite ($w_i - d_{sup}$ ou $w_j - d_{inf}$). Em outras palavras, escolhe-se a mais difícil acima do limite superior, a mais fácil abaixo do limite inferior ou a que estiver mais próxima de seu respectivo limite.

Mais um problema que talvez surja é um padrão ser avaliado como muito difícil e, apesar disso, suas seqüências serem avaliadas como muito fáceis. Baixando do limite inferior, elas não voltarão a ser utilizadas. Se esse padrão não tiver muitas ocorrências e for usado algum método de atualização como média, pode não haver muitos exemplos que baixem suficientemente sua avaliação de maneira a torná-la mais real. Ou seja, um padrão que nas primeiras jogadas estava muito difícil e

que depois se tornou fácil, por ter poucas ocorrências talvez tenha sua avaliação supervalorizada. De uma forma geral, o uso de médias para a atualização dos valores das avaliações têm esse risco de haver poucos exemplares, comprometendo o cálculo de valores mais justos para os mesmos. Uma solução seria usar mais vezes o padrão ou, então, aumentar, de alguma forma, o peso das avaliações mais recentes na atualização dos valores para o padrão.

Mais extensões ao modelo podem ser facilmente aplicadas, uma vez que ele é flexível o suficiente para acomodar diversas adaptações e modificações. A idéia principal, entretanto, é que se deve adaptar um conjunto de padrões tal que ele seja o estimado para, da melhor forma possível, manter-se dentro de um intervalo de dificuldade estipulado para o jogador.

Algumas outras aplicações podem ser pensadas como forma de aproveitar este método, principalmente no campo de ensino e treinamento, que têm uma estreita relação com a questão de avaliação de dificuldade. Outras possibilidades seriam sistemas para configurações auto-ajustáveis e interfaces adaptativas, por exemplo.

4.3 Lógica Nebulosa

A lógica nebulosa (ou fuzzy) vem sendo usada com bastante êxito em diversas aplicações, o que fez crescer bastante a sua disseminação e popularidade na indústria e entre a comunidade acadêmica. Sua versatilidade faz dela uma excelente opção para aplicações que têm um certo grau de incerteza e/ou que precisam de grande flexibilidade e capacidade de adaptação. Sendo assim, jogos eletrônicos constituem um campo potencial animador para tal aplicação.

Num artigo para a Game Developer Magazine, Larry O'Brien fez uma breve análise do uso de lógica nebulosa em jogos (O'BRIEN, 1996). Apesar de um pouco superficial, o artigo teve o mérito de chamar a atenção dos desenvolvedores para o imenso potencial do uso dessa abordagem. Há outros artigos interessantes sobre o assunto como (MCCUSKEY, 2000) que, na verdade, é uma espécie de introdução muito resumida e simples da teoria cujo mérito é a possibilidade de gerar, nos desenvolvedores, interesse sobre o assunto, tornando-o mais comentado e menos obscuro para o público em geral (não acadêmico).

O uso de regras lingüísticas torna bastante intuitiva a aplicação de lógica nebulosa mesmo em casos complexos, como a estratégia de um agente dentro de um jogo,

por exemplo. Pode ser difícil, mesmo assim, conseguir determinar todas as regras de forma satisfatória. Isso pode ser causado tanto pela complexidade do problema como por um desconhecimento de determinadas situações. O sistema, nesses casos, tende a ter um grau muito alto de incerteza, o que talvez não seja adequado.

Como já foi discutido na seção 4.1, há casos em que temos alguma idéia de uma melhor estratégia inicial para um determinado jogo, porém depois é necessário alguma evolução sobre ela. Além disso, como também foi comentado, pode acontecer de existirem estratégias que funcionam melhor ou pior dependendo daquela contra a qual elas estão disputando e, sendo assim, não temos apenas uma estratégia vencedora (ou um conjunto finito estático das mesmas). Nesses casos, mesmo que a modelagem do agente tenha sido feita de forma bem sucedida através de um sistema nebuloso, a dinâmica do jogo talvez acabe fazendo com que o agente seja derrotado facilmente. Isso aconteceria se, por exemplo, o jogador descobrisse uma estratégia que para conseguir derrotar a do agente e passasse a usá-la com freqüência. Num ambiente dinâmico e complexo, como é o caso, isso acontece com facilidade.

Outro problema possível é criar um agente difícil demais de ser derrotado. Preocupando-se com as questões que acabaram de ser levantadas, pode-se simplesmente tentar modelar a melhor estratégia possível, depois de diversos testes e resultados. Sendo assim, é possível que o agente seja tão difícil que o jogador nem mesmo tenha chance de aprender a jogar suficientemente bem de forma a descobrir formas de derrotar o agente. Ou então, mesmo que isso venha a ser possível, talvez seja tão complicado e pouco provável que desestimule o jogador, fazendo-o abandonar o jogo prematuramente.

Fica claro que, mesmo conseguindo modelar de forma satisfatória uma estratégia para o agente, o fato de o mesmo ser estático pode ser um problema sério para o sucesso do jogo. Nesses casos, então, faz-se necessário tornar, de alguma maneira, dinâmico o sistema de regras e seu uso.

É possível aproveitar o método descrito na seção 4.1 com um sistema nebuloso. Na verdade, o uso de algoritmos genéticos em conjunto com sistemas nebulosos é algo razoavelmente comum e bem documentado na literatura, existindo várias formas diferentes de aproveitá-los em conjunto (ALANDER, 1997; OMAIFAR, MCCORMICK, 1995; BELARBI, TITEL, 2000). Pode-se encontrar, inclusive, abordagens de algoritmos coevolucionários para sistemas nebulosos (PEñA-REYES, SIPPER, 2001), encarando regras e funções como subespécies em cooperação. Também é

possível encarar a cooperação entre subespécies de agentes da mesma forma, apenas modificando seus genótipos para descreverem regras nebulosas, por exemplo. Ou seja, há inúmeras possibilidades de aproveitar o método já descrito dessa maneira. Exploraremos, porém, nas seções seguintes, outras possibilidades do uso de lógica nebulosa para modelar a inteligência de um agente com o objetivo de torná-la adaptativa.

4.3.1 Aprendizado de Regras

O método aqui descrito foi proposto em (WANG, MENDEL, 1992). Como vamos utilizá-lo com mínimas alterações, o descrevemos de forma bem genérica e superficial, apenas para contextualizar seu uso neste trabalho. Apresentaremos, na seção 4.3.2, uma discussão e uma proposta de adaptá-lo de forma a aproveitá-lo de maneira dinâmica. Para mais detalhes a respeito do método original, deve-se consultar a obra citada.

De maneira geral, a idéia do método é aproveitar uma série de medições práticas realizadas e, com esses dados, construir o conjunto de regras. Ou seja, dados as variáveis e os conjuntos nebulosos, através de uma série de resultados experimentais do problema, as regras são definidas.

Cada observação feita é um conjunto de dados numéricos correspondentes às entrada e saída. Os valores obtidos são, então, convertidos nas variáveis nebulosas correspondentes para formar a regra original. Como um determinado valor pode pertencer a mais de um conjunto, considera-se aquele com o maior grau de pertinência.

Então, por exemplo, ao serem obtidos os dados numéricos x_1 e x_2 de entrada e y de saída, digamos que os mesmos sejam convertidos nos respectivos conjuntos nebulosos com os seguintes graus de pertinência x_1: [$0,25$ em A_1, $0,75$ em A_2, $0,0$ em A_3], x_2: [$0,0$ em B_1, $0,33$ em B_2, $0,66$ em B_3] e y: [$0,8$ em C_1, $0,2$ em C_2, $0,0$ em C_3]. A regra construída, neste caso, seria **if** x_1 **is** A_2 **and** x_2 **is** B_3 **then** y **is** C_1.

Com grande quantidade de dados (e possivelmente até com pequena) é provável que haja regras conflitantes, ou seja, que tenham os mesmos antecedentes, porém com saídas diferentes. Um exemplo seria uma regra **if** x_1 **is** A_2 **and** x_2 **is** B_3 **then** y **is** C_3 que seria conflitante com a do exemplo anterior.

Para resolver esse tipo de conflito, é associado um grau a cada regra obtida pelo método. Sendo $\mu_X(x)$ o grau de pertinência de x no conjunto X, o grau D de uma

regra, para n variáveis de entrada e uma saída, é definido como:

$$D(regra) = \mu_{X_1}(x_1)\mu_{X_2}(x_2)...\mu_{X_n}(x_n)\mu_Y(y) = \mu_Y(y)\prod_{i=1}^{n}\mu_{X_i}(x_i) \qquad (4.9)$$

Que é, justamente, o produto de todos os graus de pertinência encontrados. Assim, quanto maiores esses forem, mais *forte* será uma regra. Quando há duas regras com os mesmos antecedentes, a mais forte (ou seja, com maior grau D) é aproveitada e a outra, descartada. Pode-se, também, aproveitar o grau D com um fator de corte α. Nesse caso, se $D(regra) < \alpha$ então a regra é automaticamente descartada, mesmo que não haja conflitos. Assim estaremos ignorando regras geradas que não sejam fortes o suficiente.

O método pode ser aplicado tanto sobre um conjunto vazio de regras quanto como num conjunto pré-determinado, acrescentando regras ao mesmo ou, inclusive, alterando as que nele se encontram. Regras criadas antes da aplicação do método devem ter, também, os respectivos graus D definidos.

As grandes vantagens deste método são sua simplicidade e eficiência. Não é muito complicado implementar os passos descritos, nem esses consomem muito tempo para ser executados. Na verdade, a eficiência vai depender da quantidade de variáveis e conjuntos, mas, mesmo assim, a complexidade total do método é linear no número de variáveis pois, afinal, a criação de uma regra depende apenas da avaliação dos graus de pertinência de cada variável de entrada e saída.

Há basicamente duas possibilidades de aplicação deste método no contexto de jogos: aprendizado e predição. No caso de aprendizado, encaramos os dados obtidos como realizados por um "tutor" e como sendo as regras que representam a melhor forma de jogar. Sendo assim, o agente vai aprender por imitação daquilo que lhe foi exposto e, nesse caso, quanto mais exemplos ele tiver à disposição, mais completo estará seu sistema de regras. Já o uso de predição, baseia-se no conhecimento do adversário. Construindo um sistema que descreve satisfatoriamente o comportamento do jogador, podemos aproveitá-lo para tentar prever seu movimento em determinado cenário. Ou seja, é como se considerássemos que temos em mãos o próprio sistema de inferência do adversário e, ao usá-lo, estamos sabendo de antemão sua próxima ação. Há várias formas possíveis de se aproveitar essa informação. No caso de um jogo de luta, por exemplo, poderíamos prever que espécie de golpe o adversário está para desferir e, assim, tentar uma defesa correspondente se houver tal possibilidade.

Outros métodos de previsão serão discutidos na seção 4.5.

4.3.2 Aprendizado de Regras em Tempo Real

O método apresentado na seção 4.3.1 permite que o conjunto de regras de um sistema nebuloso seja montado a partir de dados numéricos obtidos em exemplos práticos.

Nada impede, a princípio, que o método seja integralmente aproveitado num contexto dinâmico, ou seja, dentro de um jogo em andamento. O sucesso do método, porém, está intimamente ligado à disposição de informação suficiente para sua construção (WANG, MENDEL, 1992), o que significa que é necessário um número satisfatório de observações para que o sistema de regras gerado seja, no mínimo, consistente. O que é "satisfatório" vai depender de diversos fatores, como a complexidade do problema, a quantidade e a capacidade de descrever bem o contexto das variáveis e dos conjuntos entre outros.

Temos, dessa forma, um período de instabilidade do sistema enquanto não forem jogadas partidas suficientes contra um determinado jogador. Mas há um outro problema que é a própria dinâmica de estratégias, tantas vezes já citadas ao longo deste trabalho.

O agente pode levar um determinado tempo até aprender as regras do adversário que, por sua vez, pode usar várias diferentes e, assim, dificultar em muito o aproveitamento do método. Os graus não ajudam neste caso, pois uma determinada regra talvez tenha um D bem alto num determinado momento e, depois, ser abandonada. Uma regra subseqüente, com um D menor, não será aproveitada, mesmo que a anterior esteja defasada, e isso atrapalharia ou até mesmo inviabilizaria o aproveitamento do conjunto de regras gerado.

Para tentar contornar esse problema, definiremos um *fator de desuso* d para cada regra, o qual desempenha a função de um peso no grau D baseado no "tempo de vida" da regra. Sendo assim, portanto, quanto mais antiga for a regra, menor será o seu D. Sendo K uma constante, então uma das possíveis formas de definir o fator de desuso d de uma regra em função do tempo t é:

$$d_{regra}(t) = e^{-Kt} \tag{4.10}$$

Quanto maior for o valor de K, obviamente, mais rápido será o decaimento. A equação (4.9) deve ser modificada de forma a comportar o fator de desuso, sendo

redefinida como:

$$D(regra) = d_{regra}(t)\mu_Y(y)\prod_{i=1}^{n}\mu_{X_i}(x_i) = e^{-Kt}\mu_Y(y)\prod_{i=1}^{n}\mu_{X_i}(x_i) \tag{4.11}$$

Podemos obter uma estimativa simples da velocidade com que uma determinada regra se torna obsoleta fazendo o cálculo de "meia-vida", ou seja, quanto tempo é necessário para que d atinja o valor $0,5$. Ou seja:

$$\begin{aligned} d_{regra}(t) &= e^{-Kt} \\ 0,5 &= e^{-Kt} \\ \frac{1}{2} &= \frac{1}{e^{Kt}} \\ e^{Kt} &= 2 \\ t &= \frac{\ln 2}{K} \end{aligned} \tag{4.12}$$

Assim, para $K = 1$ teremos $t \approx 0,69315$, o que é muito pouco se considerarmos t em turnos de jogo, ou mesmo em segundos. Se levarmos em conta que um jogo normalmente tem de 30 a 60 quadros por segundo, então usar turnos de jogo como referencial tornaria o decaimento de 30 a 60 vezes mais rápido do que em segundos.

Podemos considerar t em minutos e, sendo assim, uma regra levaria, com $K = 1$, aproximadamente 42 segundos para ficar com fator de desuso $0,5$, o que já é mais razoável. Por outro lado, considerando t relativo a turnos de jogo e fazendo $K = 10^{-4}$, obtemos $t \approx 6931,5$ turnos, o que equivaleria, com 60 quadros por segundo, a quase dois minutos.

Mesmo esses valores podem ser pequenos, dependendo do jogo, e, talvez, até mesmo muito altos. O valor de K e o referencial para t devem ser determinados levando-se em conta justamente as características do jogo e é possível que seja necessário ajustá-los após alguns resultados experimentais.

Como o decaimento é exponencial, a fórmula (4.10) tende a zero, mas não alcança esse valor. Uma determinada regra, por causa desse fator, pode se tornar muito fraca, com grau próximo de zero, após um certo tempo. Se nenhuma outra regra com os mesmos antecedentes for inferida pelo sistema, ela prosseguirá cada vez mais fraca e próxima de zero, mas nunca com valor zero. Mesmo assim, contudo, é possível que

se queira mantê-la e não descartá-la. Neste caso, não há nada de diferente a ser feito. Se, porém, considerarmos que após um intervalo conhecido uma regra fica obsoleta e, portanto, deixa de ser confiável, podemos simplesmente descartá-la. Para isso, basta que apliquemos o fator de corte α definido na seção 4.3.1, que passaria, então, a ser aplicado dinamicamente.

Outra dificuldade da aplicação do fator de desuso é justamente o cálculo da exponencial. Considerando um volume alto de regras, calcular d para cada uma pode ser muito custoso computacionalmente e tornar ineficiente o método. Uma forma de contornar esse problema é simplesmente tabelar os resultados da função e e aproveitá-los durante o jogo. Para $K = 10^{-4}$ e $\alpha = 0,1$, são necessárias pouco mais de 23.000 entradas numa tabela, e considerando cada entrada ocupando 8 bytes (o equivalente ao tipo **double** da linguagem C), isso tudo ocuparia menos de $200Kb$ de memória, o que não é nada demais para a capacidade de armazenamento dos computadores de hoje, sendo, portanto, bastante vantajoso trocar essa quantidade quase desprezível de memória por maior eficiência na execução.

Um outro modificador que pode ser aplicado à equação (4.11) é a *credibilidade* c de uma regra, que exprime o quão confiável ela é. Suponhamos que uma regra r_1 tenha sido inferida e tenha grau $D(r_1)$. Uma outra regra r_2, com os mesmos antecedentes, porém com outra saída, é posteriormente verificada. Se $D(r_1) > D(r_2)$, então r_1 não será substituída, mas isso é um possível indicador de que ela não é tão confiável como a princípio seria. Podemos, ainda, supor que, quanto mais próximo de $D(r_1)$ for $D(r_2)$, menos confiável a regra será.

Sendo assim, para cada regra r, temos inicialmente $c(r) = 1,0$ e esse valor será decrementado sempre que uma outra regra com os mesmo antecedentes for verificada. Esse decremento pode ser feito a uma taxa constante, ou seja, fazendo $c(r) \leftarrow c(r) - const$ toda vez que for verificada uma nova regra com os mesmos antecedentes. É possível, por outro lado, tornar esse decremento proporcional à razão $D(r_2)$ / $D(r_1)$, atribuindo $c(r) \leftarrow c(r) - const$ ($D(r_2)$ / $D(r_1)$).

De uma forma ou de outra, a equação (4.11) seria modificada de maneira a acomodar esse novo fator, tornando-se:

$$D(regra) = c(regra) d_{regra}(t) \mu_Y(y) \prod_{i=1}^{n} \mu_{X_i}(x_i) \tag{4.13}$$

É possível considerar a fórmula (4.13) como genérica para a aplicação deste

método. Se não se desejar usar a credibilidade, considera-se $c(regra) = 1,0$ para cada regra em todos os instantes, o mesmo podendo ser feito com o fator de desuso.

4.3.3 Máquinas de Estado Nebulosas

O assunto abordado nesta seção foi brevemente introduzido na seção 3.2.5 e será desenvolvido aqui com mais detalhes. Há várias maneiras diferentes de se definir uma máquina de estado usando lógica nebulosa, embora o objetivo geral seja consideravelmente relacionado. Uma extensa lista de referências e sobre os trabalhos e modelos considerados pode ser encontrada em (DUBOIS, PRADE, 1980). Nesta seção definiremos e descreveremos o modelo que foi utilizado ao longo do trabalho, bem como uma discussão a seu respeito.

Como já foi dito, basicamente a diferença entre uma máquina de estados tradicional (FSM) e uma nebulosa (FuSM) é o fato de as transições serem feitas com variáveis e conjuntos nebulosos. Uma característica interessante, e a princípio curiosa, das FuSMs é o fato de poderem estar em mais de um estado ao mesmo tempo com graus de pertinência maiores que zero.

Há vários outros pequenos detalhes pertinentes às FuSMs, que muitas vezes dependem do modelo que é adotado. Há várias maneiras diferentes de se modelar e implementar uma FuSM, como já foi dito, e a que descreveremos aqui é muito semelhante à descrita em (REYNERI, 1997).

Seja S o conjunto de n estados e S_i cada um dos n estados de S. Representamos por $\mu(S_i)$ o grau de pertinência do estado S_i. Esse grau representa o peso daquele estado num determinado momento. Numa FSM tradicional, um (e apenas um) estado tem grau $1,0$ enquanto que os demais têm $0,0$. Numa FuSM isso nem sempre é verdade, embora seja possível.

Um dos problemas comuns a modelos de máquinas de estado nebuloso é o fato de elas poderem se degenerar ao longo do tempo, ou seja, o somatório das pertinências de todos os estados vai tendendo a zero à medida que o tempo passa. Por outro lado, dependendo das regras associadas, é possível também que esse mesmo somatório, em determinados momentos, ultrapasse o limite de $1,0$. Como não queremos que nenhum desses casos ocorra, iremos definir a seguinte restrição para a nossa FuSM:

$$\sum_{i=1}^{n} \mu(S_i) = 1,0 \tag{4.14}$$

Logo, em qualquer momento a soma dos graus de pertinência será sempre igual a $1,0$. Se pensarmos bem, essa restrição é natural para as FSM tradicionais, afinal, se apenas um estado tem pertinência $1,0$ e os demais todos $0,0$, então é trivial verificar que a equação (4.14) é verdadeira para toda FSM. Ao definir tal restrição, portanto, estamos mantendo uma certa correlação entre as FuSMs e as FSMs.

Para cada estado S_i podem estar associadas até $n-1$ transições para cada um dos demais estados. Seja $\mathcal{T}_\mathrm{i}$ o conjunto das transições saindo do estado i e $t_j \in \mathcal{T}_\mathrm{i}$ cada uma das transições de $\mathcal{T}_\mathrm{i}$. Neste caso, uma transição representa uma regra nebulosa cuja saída é, na verdade, a transição para o novo estado. Sendo A e B conjuntos nebulosos e a e b variáveis, um exemplo de transição seria $t_j =$ **if** a **is** A **and** b **is** B **then** *state* **is** S_i. Representando o conjunto de todas as transições como $\mathcal{T}$, então $t_i^j \in \mathcal{T}$ é a transição do estado S_i para o S_j.

Toda transição t_i^j terá um grau de pertinência $\mu(t_i^j)$ associado, que será justamente a avaliação dos antecedentes de sua regra. A contribuição dessa transição para o estado destino S_j, será $\mu(S_i)\mu(t_i^j)$, o produto do grau de pertinência de S_i pelo calculado para t_i^j.

Dessa forma, o novo grau de pertinência de um estado num determinado instante será igual à soma de todas as contribuições que o mesmo receber dos demais estados no instante anterior. Logo:

$$\mu'(S_j) = \sum_{t_i^j \in \mathcal{T}} \mu(S_i)\mu(t_i^j) \tag{4.15}$$

Uma observação importante que devemos fazer é que o somatório dos graus de pertinência das transições que saem de um determinado estado deve ser sempre igual a $1,0$. Ou seja:

$$\sum_{t_j \in \mathcal{T}_\mathrm{i}} \mu(t_j) = 1,0 \tag{4.16}$$

A importância de (4.16) é justamente permitir que (4.14) sempre se verifique. O que nos leva ao seguinte lema:

Lema 4.1 *Se (4.16) for sempre verificada e os novos graus de pertinência dos estados forem calculados segundo (4.15), então basta que (4.14) seja verificada num instante inicial (ou seja, de inicialização da FuSM) para que seja sempre verdadeira.*

Demonstração Sendo (4.16) verdadeira, como a contribuição de uma determinada transição é $\mu(S_i)\mu(t_i^j)$, então é fácil verificar que o total de contribuição que sai de um estado S_i é sempre igual a $\sum_{t_j \in \mathcal{T}_\text{i}} \mu(S_i)\mu(t_j) = \mu(S_i) \sum_{t_j \in \mathcal{T}_\text{i}} \mu(t_j) = \mu(S_i)$. Logo, a contribuição de um estado é sempre igual ao seu grau de pertinência. Sendo assim, o total de contribuição em cada instante na máquina de estados é igual a $\sum_{S_i \in S} \mu(S_i)$. Como os novos graus são calculados segundo (4.15), então a soma dos graus de pertinência dos estados num determinado instante equivale à da contribuição dos mesmos no instante anterior e, portanto, à própria soma dos graus de pertinência que, assim, permanece sempre constante ao longo do tempo. Logo, se (4.14) se verificar no instante inicial da FuSM, então permanecerá constante e, portanto, verdadeira para todos os instantes seguintes. ■

Agora nos falta, apenas, nos certificar de que (4.16) será sempre verdadeira. Se permitirmos que as transições sejam feitas por regras nebulosas livremente, temos dois possíveis (e óbvios) problemas: a soma dos graus das transições ser menor que $1,0$ ou ser maior. Para evitá-los, vamos definir algumas novas propriedades para nosso modelo de FuSM.

Para contornar a possibilidade de a soma dos graus de pertinência das transições ser maior do que $1,0$, vamos aplicar uma normalização que será calculada sempre que isso ocorrer. Isso pode ser feito de uma forma bem simples, dividindo o grau de pertinência de cada transição t_j de um estado S_i pela soma dos graus de todas as transições de $\mathcal{T}_\text{i}$. Ou seja:

$$\mu(t_j) = \frac{\mu(t_j)}{\sum_{t \in \mathcal{T}_\text{i}} \mu(t)} \tag{4.17}$$

Já para a possibilidade da soma ser menor do que $1,0$, vamos recorrer a um esquema simples. Sempre que isso acontecer, o que faltar para alcançar $1,0$ "permanecerá" no estado, e, portanto, será uma contribuição para o mesmo no instante seguinte. Seria uma espécie de transição do estado para ele mesmo e sua contribuição seria:

$$\mu(S_i)(1,0 \; - \sum_{t_j \in \mathcal{T}_\text{i}} \mu(t_j)) \tag{4.18}$$

A "auto-contribuição" de um estado é sempre o máximo entre (4.18) e 0. Dessa forma, (4.18) é apenas aplicada quando a soma dos graus de pertinência das demais transições é menor que 1,0, ou seja, quando (4.18) é positiva.

Um dos grandes perigos do uso de FuSMs é que há a possibilidade de haver vários estados com graus de pertinência diferentes de zero e, assim, todas as regras de cada transição desses estados devem ser avaliadas. Se a FuSM tiver muitos estados e muitas transições, a eficiência do método pode ficar seriamente comprometida. Há algumas formas de tentar contornar esse problema, mas nenhuma pode ser considerada plenamente satisfatória, pois alteram o funcionamento normal da FuSM e podem introduzir novos problemas.

Uma saída seria aplicar um fator de corte aos graus de inclusão das transições que, quando fossem menores que um determinado patamar, seriam automaticamente considerados iguais a zero, o que evitaria que contribuições de valores muito baixos fossem passadas adiante. Pode acontecer, entretanto, de um determinado estado receber diversas contribuições de valores baixos e a soma dos mesmos ser considerável. Usando esse corte, esse tipo de estado simplesmente deixaria de receber tais contribuições.

Uma outra forma de lidar com essa questão seria utilizar o fator de corte diretamente nos graus de pertinência dos estados. Sendo assim, se um determinado estado tivesse um grau de pertinência menor que um determinado patamar, ele seria simplesmente zerado. Para manter a estabilidade da FuSM, satisfazendo (4.14), seria necessário criar um estado especial que receberia, como contribuição, justamente esses graus cortados dos estados, como se fosse um "coletor de lixo" que "reciclasse" tais graus. Desse estado poderiam sair transições para outros estados normalmente, a sua diferença seria apenas na forma de receber as contribuições (como já descrito). O problema, neste caso, é que estaríamos introduzindo um estado que foge às características do modelo aqui descrito, o que pode ocasionar instabilidades na FuSM, como acabar acumulando um grau de pertinência alto demais, "esvaziando" o resto dos estados. Outro fator seria privilegiar determinados estados com esses "graus reciclados" que, a princípio, não teriam possibilidade de receber contribuições dos estados cortados, o que pode ir contra o que a FuSM representa. Isso pode ser atenuado criando mais estados especiais, cada um específico para um grupo de estados. Deve-se ter cuidado ao fazer isso, contudo, porque criar um número muito alto vai justamente contra toda a idéia de seu uso que é evitar que haja muitos estados a ser avaliados.

Apesar de este ser um modelo basicamente estático, ele pode ser considerado como adaptativo, principalmente por sua flexibilidade e facilidade de extensão. Di-

ferentemente das máquinas de estado tradicionais, nas quais apenas há possibilidade de se estar em um estado por vez, a FuSM se adapta de maneira bem mais natural ao contexto no qual está inserido.

Uma forma interessante de se estender ainda mais essa adaptação é usar FuSMs hierárquicas, ou seja, FuSMs cujos estados representam outras FuSMs. A idéia é extremamente semelhante àquela vista na seção 3.2.4 para FSMs comuns. No caso das FuSMs, a única modificação significativa seria considerar os graus de pertinência de cada estado como sendo o produto dele com o do estado da FuSM hierarquicamente superior na qual ele está inserido. É possível, inclusive, mesclar FuSMs com FSMs. Se forem usados dois níveis hierárquicos e o mais externo for uma FSM, cada estado representará uma FuSM e as transições desempenharão o papel de seleção de qual FuSM mais adequada para ser usada naquele momento.

Além disso tudo, obviamente, ainda há a possibilidade de se aproveitar modelos como algoritmos genéticos para evoluir as FuSMs, inclusive podendo aproveitar o método de coevolução em tempo real, já apresentado neste capítulo, para tal fim.

4.4 Navegação e Percepção de Ambiente

Embora pareça um pouco deslocado do objetivo deste trabalho, afinal não há adversários contra quem se adaptar, o tópico de navegação pode ser considerado dentro deste contexto, principalmente se considerarmos o ambiente como um adversário para o agente e, assim, deve adaptar-se ao mesmo.

O principal objetivo de considerar este tópico brevemente é buscar uma forma mais justa de realizar a navegação dos agentes. Há muito escrito sobre o assunto para jogos, como visto na seção 3.2.7, mas muitas são trapaceadas, mesmo não sendo a intenção, como já foi, inclusive, discutido na seção 3.3.

Não queremos que o agente saiba, pelo menos *a priori*, exatamente o mapa do ambiente onde ele se encontra, que tenha um milagroso sistema de GPS que lhe permita localizar todos os inimigos nem saber, mesmo sem ver, o que se encontra atrás de portas e paredes.

Para evitar todas as tentações, vamos considerar esse problema como sendo praticamente uma aplicação simulada de navegação com robôs. Supondo que estamos controlando um robô, com todas as restrições decorrentes, vamos tentar fazer com que nosso agente seja capaz de se movimentar, explorar o mundo e, em algum mo-

mento, ser capaz de lembrar e visitar lugares por onde já passou.

Muito já foi pesquisado e publicado sobre este assunto em robótica (SOUZA, KAK, 2002) e não é o nosso objetivo fazer uma pesquisa profunda neste campo, mesmo porque já foge um pouco mais da nossa intenção inicial. O que queremos é trazer essas dificuldades que se encontram em aplicações reais em navegação de robôs para jogos, tentando criar uma inteligência mais honesta que possa navegar dentro dos ambientes.

Para isso, as características do agente serão:

1. Não haverá mapas prontos do mundo nem muito menos sistemas de radar que permitam localizar qualquer alvo desejado, a não ser que haja bons motivos para tal.

2. O agente será equipado de sensores de distância em volta do corpo, de maneira que ele possa sentir, naturalmente, objetos muito próximos.

3. A visão será feita através de uma câmera, com um campo de visão realista (algo em torno de 60^o a 90^o). O agente só verá aquilo que estiver em seu campo de visão.

4. As distâncias para objetos, obstáculos, paredes etc. terão de ser calculadas com base na visão, com exceção daquelas que podem ser sentidas muito perto pelos sensores. Essas distâncias, ao contrário de serem precisas, devem estar sujeitas a erro devendo, dessa forma, ser representadas como distâncias nebulosas, por exemplo.

5. O reconhecimento de objetos não precisará ser feito por reconhecimento de padrões. Apesar disso, não deve ser possível para o agente reconhecer elementos que estejam a uma distância considerável que, na visão dele, apareceriam como pontos ou entidades muito pequenas e indistingüíveis.

6. O erro aleatório deve ser aplicado ao movimento (ver seção 4.6). Isso evitará que o agente saiba exatamente quantos graus girou ou quantos metros percorreu, adicionando uma componente de incerteza à sua memória de ação.

Em geral, essas características visam a adicionar incerteza à percepção de ambiente e navegação do agente. A movimentação propriamente dita dentro de um

ambiente é feita, por exemplo, como variação de métodos simples e fáceis de implementar como, por exemplo, seguir a parede direita, que pode ter resultados muito bons para ambientes com poucas "ilhas", ou seja, regiões afastadas. Esta forma de andar por um ambiente é, inclusive, facilmente implementada usando-se a incerteza, baseando-se em valores de sensores e nas distâncias nebulosas de visão.

Quanto à memória, ela deve ser apenas representada por locais onde o agente esteve. As distâncias entre os locais, devido à imprecisão descrita no último item, devem ser representadas de forma nebulosa.

Para tornar as coisas ainda mais realistas, evita-se que o agente se lembre de locais muito distantes, a não ser que ele os tenha visitado um número alto de vezes. Quanto mais distante o local, mais difícil seria de lembrá-lo, por exemplo.

Uma forma ainda mais simples de fazer uma memória do agente seria implementar um algoritmo do tipo "João e Maria", aqueles personagens da literatura que, para fugir da floresta da bruxa malvada, foram marcando o caminho com migalhas de pão para achar o caminho de volta. A memória do agente seria algo do tipo "caminhando muito para a direita eu volto ao local A; caminhando pouco para trás eu volto ao local B; girando poucos graus no sentido horário e caminhando médio para a frente eu volto ao local C" etc. o que seriam apenas representações puramente lingüísticas de regras nebulosas. Em cada local guardado em memória, o agente saberia os demais locais àquele ligados, ou seja, em quais direções e distâncias (propriamente nebulosas) ele colocou as migalhas de pão.

Esta seção não descreveu exatamente um método proposto, mas apenas uma série de medidas para tentar a questão de navegação de forma mais justa, pelo menos do ponto de vista do jogador, e mais desafiadora, apesar de simples, do ponto de vista da inteligência. Apesar disso tudo, no próximo capítulo veremos aplicações de outros métodos deste capítulo em conjunto com a questão da navegação para torná-a adaptativa dentro de todo o contexto deste trabalho.

4.5 Previsão

Poder prever o que o adversário vai fazer, quais serão seus próximos movimentos, qual estratégia ele está adotando etc. com sucesso é uma enorme vantagem que qualquer jogador gostaria de ter. No caso deste trabalho, ter condições de prever a estratégia do adversário pode nos ajudar a desenvolver uma evolução ainda mais

condicionada, uma vez que, sabendo, com certa confiança, quais os próximos passos do jogador, é possível gerar uma resposta apropriada, dependendo do nível de dificuldade que se queira aplicar. A idéia do estudo de métodos de previsão, portanto, é permitir que o jogo possa, a partir dela, responder de forma adequada à capacidade do jogador (ou seja, quanto melhor ele for, melhor deverá ser a resposta).

Como controla a simulação do ambiente, é claro que qualquer programa pode permitir a seu agente a capacidade de "prever" ações. Num jogo de luta, por exemplo, se o jogador desfere um soco para atingir o rosto do agente, o jogo pode passar essa informação no exato momento do golpe e, com "reflexos perfeitos", o golpe é defendido com sucesso pelo agente. Como visto na seção 3.3, isso está muito mais para trapaça do que para uma previsão honesta.

Desconsiderando essa forma de "previsão", vamos procurar, nesta seção, propor algumas formas de prever as ações do adversário baseando-se em seu passado. Uma dessas formas já foi vista nas seções 4.3.1 e 4.3.2, aproveitando o método de aprendizado de regras nebulosas como previsor.

O que desejamos é muito parecido com os bem conhecidos paradigmas de classificação e clusterização. As técnicas convencionais dessas áreas, porém, nos são de pouco uso, uma vez que precisamos agir de acordo com o adversário e, fundamentalmente, de forma eficiente, em tempo real. Os métodos bem estudados dessas áreas requerem esforço computacional muito fora da realidade deste trabalho, que visa a implementações que funcionem em tempo real, e, portanto, teremos de desconsiderá-los.

Se as técnicas convencionais disponíveis, com tempo suficiente à disposição, já não são exatamente perfeitas, o que dirá as que teremos de usar, com severas restrições de tempo de execução e, portanto, conseqüentemente uma drástica (e óbvia) perda de precisão?

Bem, não podemos nos iludir a ponto de acharmos que vamos conseguir fazer em alguns décimos de segundo o que décadas de estudo mal conseguem em horas. Nossos objetivos são mais modestos. Ter um método eficiente que funcione bem em determinadas situações já nos basta e não podemos ter a pretensão de imaginar que conseguiremos prever com sucesso todas as ações do jogador, ainda mais em se tratando de uma entidade tão imprevisível e criativa como é o ser humano.

Veremos, nas próximas duas seções, três formas simples de predição (sendo que duas suas essencialmente variações da mesma idéia) baseadas em seqüências de co-

mandos de forma a se juntar com o referido uso do aprendizado de regras nebulosas para montar nosso leque de opções de métodos de previsão para implementação em jogos. Vamos supor que podemos representar ações do adversário através de símbolos pertencentes a um determinado alfabeto, que seria um conjunto de todas as possíveis ações em um dado momento. Uma seqüência desses símbolos, como se fosse uma string, representa a série de ações tomadas pelo jogadas. A partir dos elementos dessa seqüência é que vamos tentar prever quais serão os próximos. Os símbolos, porém, são meras abstrações e podem ter outros significados, como, por exemplo, uma determinada regra que foi satisfeita naquele instante.

4.5.1 Previsão Seqüencial

O método aqui descrito é inspirado pelo apresentado em (MOMMERSTEEG, 2002), com algumas diferenças no algoritmo final e sua forma de implementação. Dada uma seqüência s qualquer sobre um alfabeto Σ, queremos achar o maior sufixo de s tal que haja uma subseqüência s_{ij} de s (ou seja, que inicia na i-ésima posição e termina na j-ésima) exatamente igual (que não seja o próprio sufixo). Entendemos por *subseqüência* de s um subconjunto *contínuo* de símbolos de s. Por sufixo entendemos uma subseqüência cujo último símbolo coincide com o último de s.

Por exemplo, dados $\Sigma = \{a, b, c\}$ e $s = abbacbba$, existem oito sufixos diferentes para s (a, ba, bba, $cbba$, $acbba$, $bacbba$, $bbacbba$ e $abbacbba$). Para o sufixo a existem duas subseqüências de s iguais. Para ba, também; Para bba, apenas uma. Para as demais, não há nenhuma. Logo, o maior sufixo igual a uma outra subseqüência de s é bba.

Depois de descoberto o maior sufixo, pegamos a subseqüência correspondente e consideramos o símbolo seguinte como sendo a previsão do sucessor da seqüência. Logo, dada a s do último parágrafo, o método iria prever c como o próximo símbolo da seqüência. Claro que é possível que haja mais de uma seqüência idêntica ao maior sufixo e, neste caso, é necessário usar alguma forma de "desempate". Uma solução simples (afinal, a idéia aqui é permanecer simples) é considerar a primeira ou a última (a mais antiga ou a mais nova). Outro problema é se simplesmente não houver nenhuma solução (por exemplo, para $s = abc$ não há). Essa situação só vai ocorrer se o último símbolo de s tiver aparecido pela primeira vez. Neste caso, uma forma simples de contornar o problema seria usar o símbolo que mais aparece em s como solução e, em caso de empate, usar uma forma semelhante à anterior como

desempate (mais novo ou mais antigo).

Supondo que n seja o tamanho da seqüência s e que t_i guarde o tamanho da maior subseqüência terminada em s_i (isto é, na i-ésima posição) igual a um sufixo de s, um algoritmo simples para encontrar a subseqüência desejada seria o seguinte:

Algoritmo 4.3 Algoritmo Simples para Previsão Seqüencial

para i `de` $n-1$ `até` 1 **faça**
    $j \leftarrow 0$
    enquanto $s_{i-j} = s_{n-j}$ **faça**
        $j \leftarrow j+1$
    fim enquanto
    $t_i \leftarrow j$
fim para

Depois de aplicado o algoritmo, basta varrer o vetor t e procurar o maior valor, usando um critério qualquer para possíveis desempates. A complexidade desse algoritmo é muito simples de ser calculada. O laço i é claramente $O(n)$. A cada iteração, entretanto, é possível percorrer até i vezes o vetor. No pior caso (quando s for formado apenas por repetição de um mesmo símbolo), o algoritmo vai realizar $n + (n-1) + (n-2) + ... + 1$ passos e, portanto, sua complexidade é proporcional a $O(n^2)$. A varredura final para encontrar o maior valor toma tempo $O(n)$ e como é feita fora de qualquer laço a complexidade final do algoritmo é $O(n^2 + n) = O(n^2)$.

Nada mal, afinal, é um algoritmo de tempo de execução polinomial. Ele tende a ser muito ruim, porém, quando há muitas repetições na seqüência, por causa do segundo laço. O problema é que muitas repetições é justamente o que desejamos, pois a falta delas torna a previsão completamente aleatória. Logo, ele vai mal justamente nas situações mais vantajosas para seu uso, o que não é algo muito bom.

Usando uma abordagem de programação dinâmica podemos melhorar consideravelmente o tempo de execução do algoritmo. O caso base, quando $n = 1$, é trivial, pois a previsão pode ser o próprio símbolo solitário da seqüência. Podemos, também, definir o caso base como sendo $n = 2$, e então $t_1 = 1$ se $s_1 = s_2$ ou $t_1 = 0$ caso contrário. Vamos, então, supor que já realizamos o teste para os $n-1$ símbolos da seqüência e que, agora, queremos fazê-lo para os $n-1+1 = n$ símbolos. Qualquer sufixo de s vai ser igual ao caso anterior apenas com a concatenação de s_n ao seu final. Ora, neste caso, qualquer seqüência que, anteriormente, tenha sido reconhecida como igual a qualquer sufixo, pode ser igualmente reconhecida se o símbolo

seguinte a ela for igual a s_n. Podemos aproveitar este fato da seguinte forma: qualquer ocorrência de um símbolo igual a s_n em s numa posição i terá uma seqüência terminando em i idêntica ao sufixo cujo tamanho será a soma do tamanho de $i-1$ no passo anterior mais um. As demais serão todas iguais a zero.

Um pouco mais formalmente, seja $s' = s - \{s_n\}$ e t' os valores de t (como definido para o algoritmo anterior) em relação a s'. Ora, se $s_i \neq s_n$ então $t_i = 0$. E se $s_i = s_n$, então $t_i = t'_{i-1} + 1$. O raciocínio é razoavelmente simples, pois se um determinado símbolo de s é diferente do último de s (presente obrigatoriamente em todos os sufixos), então não pode haver seqüência terminada em s_i igual a algum sufixo e, portanto, $t_i = 0$. Por outro lado, se s_i é igual ao último símbolo do sufixo, então há uma seqüência terminando em s_i igual a algum sufixo que, no pior caso, será o próprio s_n e, portanto, o menor t_i possível será 1. Mas como já realizamos o cálculo de t', então sabemos qual era o valor de t'_{i-1}. Qualquer seqüência em s' terminada em s_{i-1} que fosse igual a algum sufixo de s' permanecerá como tal em s se s_i for igual ao s_n acrescentado (como é o caso que estamos considerando). Portanto, o valor de t_i será o valor de t_{i-1} mais 1, que é o próprio s_i.

A princípio precisaríamos guardar n vetores t diferentes, para poder realizar todos os passos. Isso não é realmente necessário, uma vez que só precisamos das informações do passo anterior. Então, na verdade, só precisamos de dois vetores t. Bem, nem isso é verdade, pois uma vez utilizado o valor t'_{i-1} para o cálculo de t, este não será mais necessário. Podemos, portanto, fazer tudo em um único vetor t, bastando, para isso, ter o cuidado de varrer o vetor da esquerda para a direita, ou seja, de n para 1.

De forma algorítmica, temos:

Algoritmo 4.4 Algoritmo de Programação Dinâmica para Previsão Seqüencial

para i `de` $n-1$ `até` 1 **faça**
  se $s_i = s_n$ **então**
    $t_i \leftarrow t_{i-1} + 1$
  senão
    $t_i \leftarrow 0$
  fim se
fim para

Deve-se apenas lembrar de cuidar do caso base (já descrito), bem como, na hora da implementação, fazer um caso especial para t_1, que não tem t_0 para comparar.

O caso de t_1, entretanto, trata-se da mesma forma como descrito para o caso base de $n = 2$ em qualquer valor de n.

Analisando o algoritmo, é fácil verificar que sua complexidade é apenas pelo laço de i e, portanto, é proporcional a $O(n)$, sendo linear. O algoritmo aqui descrito difere do de (MOMMERSTEEG, 2002) em que é usado uma matriz de ponteiros, a qual ele chama de histograma, de dimensão igual ao tamanho do alfabeto, para retornar os valores do passo anterior. Embora o algoritmo também seja $O(n)$ de execução, sua complexidade de espaço passa a ser $O(n)$ (para guardar s) somada a $O(mn)$ da matriz e, portanto, igual a $O(n + mn)$, sendo m o tamanho de Σ. A complexidade de espaço do algoritmo aqui proposto é $O(n+n) = O(n)$, além de ser também de implementação mais simples.

Este é um método extremamente simples, de implementação rápida e tempo de execução linear, bem como espaço de armazenamento. Em suma, é eficiente e compacto, porém assume que a repetição se dá a partir de repetição de padrões seqüenciais de ação, sem considerar o contexto que levou o adversário a tomar aquelas ações, o que pode diferir bastante de um caso para outro e ser um fator determinante.

4.5.2 Distância de Edição

Distância de Edição (*edit steps* ou *edit distance*) é uma técnica bem conhecida e documentada. Em (MANBER, 1989) há uma excelente descrição do problema bem como uma explicação detalhada do algoritmo e seu funcionamento, além das aplicações do mesmo.

Posto de forma resumida, dadas duas seqüências p e q de tamanhos n e m respectivamente, sobre um alfabeto Σ qualquer. Desejamos encontrar a menor quantidade de operações de edição que levam p a q. Essas operações de edição são adicionar símbolo, apagar símbolo e trocar símbolo. Por exemplo, suponhamos que $p = abbc$ e $q = babb$. Se apagarmos o primeiro a, teremos $p = bbc$. Depois, podemos inserir um a entre os dois b, obtendo $p = babc$. Finalmente, trocamos o c por um b obtendo $p = babb = q$, tendo realizado três operações. Por outro lado poderíamos simplesmente inserir um b no início de p, obtendo $p = babbc$ e depois apagar o c, obtendo $p = babb = q$, com apenas duas operações.

O algoritmo 4.5 é a forma conhecida para resolver esse problema, baseado em programação dinâmica, tem complexidade de execução e espaço iguais a $O(nm)$. Além de retornar a menor quantidade de operações de edição necessárias para transformar

p em q, também é possível, com algumas adaptações, retornar ainda *quais* operações devem ser realizadas (MANBER, 1989). Este último detalhe, porém, não é relevante para a nossa aplicação. Consideramos, como entrada, duas *strings* a e b com tamanhos de, respectivamente, n e m caracteres, a matriz C é usada para encontrar o resultado final e os inteiros x, y e z são variáveis temporárias.

Algoritmo 4.5 Distância de Edição

```
para i de 0 até n faça C[i,0] ← i
para j de 1 até m faça C[0,j] ← j
para i de 1 até n faça
    para j de 1 até m faça
        x ← C[i-1,j] + 1
        y ← C[i,j-1] + 1
        se a_i = b_j então
            z ← C[i-1,j-1]
        senão
            z ← C[i-1,j-1] + 1
        fim se
        C[i,j] = min(x,y,z)
    fim para
fim para
retorna C[n,m]
```

Suponhamos que temos k seqüências p de jogadas realizadas durante o jogo pelo jogador. Temos, também, uma seqüência q atual. Aplicando o algoritmo 4.5 a cada p_i em relação a q, encontraremos uma seqüência com a menor quantidade de edições e podemos elegê-la como a mais próxima da atual e aproveitá-la para a previsão.

Isso pode funcionar da seguinte forma. Vamos supor, por simplicidade, que todas as seqüências p_i têm o mesmo comprimento n. A seqüência atual, p, tem comprimento m. Então, em determinado momento, quando formos aplicar o algoritmo, truncamos cada p_i para tamanho m e o utilizamos normalmente. Após elegermos a melhor p_i, podemos aproveitar o próximo símbolo ($p_{i,m+1}$) como a previsão. Se considerarmos *ed* como uma variável que guarda o valor da seqüência atual p testada, *pp* como uma variável usada para receber p truncada para os m primeiros símbolos (através de uma função *trunc*) e *maior* como sendo o símbolo escolhido para previsão, então o algoritmo 4.6 ilustra o método que acabamos de descrever. Apenas um detalhe de nomenclatura do algoritmo para evitar ambigüidades na listagem do algoritmo: por $p_{i,j}$ indicaremos o j-ésimo símbolo da i-ésima seqüência p (lembrando

que há k seqüências p). Quando só aparecer um índice subscrito, como p_i, estaremos nos referindo à i-ésima seqüência p.

Algoritmo 4.6 Previsão por Distância de Edição

$temp \leftarrow -\infty$
para i `de` 1 `até` k **faça**
    $pp \leftarrow trunc(p_i, m)$
    $ed \leftarrow distancia\_\_edicao(pp, q)$
    se $ed > temp$ **então**
        $temp \leftarrow ed$
        $maior \leftarrow p_{i,m+1}$
    fim se
fim para

O retorno da função seria, obviamente, *maior*. O laço principal tem complexidade $O(k)$. Este, todavia, faz sempre uma chamada a *distancia__edicao* que tem complexidade $O(nm)$, como visto. Neste caso, como $n = m$, então consideramos $O(m^2)$. Assim, portanto,a complexidade total do algoritmo 4.6 será $O(km^2)$.

Como no caso da seção 4.5.1, há necessidade de se estipular um critério de desempate, caso haja duas seqüências com a mesma quantidade de edições. Há várias maneiras de fazer isso, mas de preferência que sejam simples. Uma solução seria atribuir pesos diferentes às formas de edição (apagar, modificar, incluir). Isso, porém, iria requerer que além da quantidade de operações também computássemos quais foram, o que seria mais gasto de tempo apenas para um critério de desempate. Os mesmos critérios descritos na seção 4.5.1 podem também ser aproveitados aqui, bem como quaisquer outros que, como dito, sejam eficientes. No caso do algoritmo 4.6, a seqüência de menor índice vence o desempate.

Um dado interessante é que, como estaremos avaliando a quantidade de edições para todas as k seqüências p, podemos aproveitar essa informação fazendo uma espécie de previsão diferente. Suponhamos que seja estipulado um determinado corte de distância máxima α tolerada, então qualquer p cuja quantidade de edições seja menor que α será considerada. A previsão final, assim, poderia ser dada de uma forma nebulosa, o que, ainda por cima, poderia ajudar a solucionar o problema do desempate já descrito. Claro que ainda seria possível haver empate (em graus de pertinência), mas, nesse caso, a decisão não caberia mais ao algoritmo em si, posto que seria externo ao mesmo. O cálculo do grau de pertinência de cada símbolo da previsão poderia ser feito como um valor proporcional ao somatório da quantidade

de edições de todas as seqüências. O algoritmo 4.6 ilustra a aplicação dessa variação.

O vetor E guarda, no final, o grau de pertinência dos símbolos. Logo, $E[p_{i,j}]$ indica o grau de pertinência do símbolo presente na j-ésima posição da i-ésima seqüência p. Além disso, *total* é usado para o cálculo dos graus de pertinência e *temp* não requer maiores esclarecimentos. As demais variáveis têm o mesmo significado já mencionado anteriormente.

Algoritmo 4.7 Previsão Nebulosa por Distância de Edição

para i `de` 1 `até` k **faça**
    $pp \leftarrow trunc(p_i, m)$
    $temp \leftarrow distancia\_\_edicao(pp, q)$
    se $temp < \alpha$ **então**
        $E[p_{i,m+1}] \leftarrow E[p_{i,m+1}] + (\alpha - temp)$
        $total \leftarrow total + (\alpha - temp)$
    fim se
fim para
para `cada` $i \in \Sigma$ **faça**
    $E[i] \leftarrow E[i] \ / \ total$
fim para

O algoritmo 4.7 retorna o vetor E, cujos graus serão usados pela rotina que chamou para decidir qual será a previsão. Pode-se, entre outras possibilidades, eleger o de maior grau de pertinência, só considerar como segura uma previsão que tenha grau de pertinência acima de um determinado fator de corte, utilizar os valores dentro de algum sistema de regras nebulosas, aproveitar esse método como contribuição dentro de um sistema maior e mais amplo etc. Enfim, há várias maneiras de aproveitar o resultado cuja escolha vai depender da aplicação que se deseja fazer.

Uma outra observação importante diz respeito ao cálculo de E. Claro que não é a única maneira de fazê-lo e, da forma como é feito no algoritmo 4.7, o somatório de todos os graus de pertinência será sempre $1,0$. Outro possível problema é que um determinado símbolo que só seja levado em conta uma vez, porém com $temp = 0$ pode ter um grau de pertinência menor do que outro que apareceu com $0 < temp < \alpha$ várias vezes. Por exemplo, se $\alpha = 4$, suponhamos que um símbolo x tenha aparecido apenas uma vez com $temp = 0$. Logo $E[x] = (4 - 0) = 4$ (antes do cálculo do grau de pertinência). Digamos que um símbolo y tenha aparecido três vezes com *temp* valendo, respectivamente, 2, 1 e 3. Logo, $E[y] = (4 - 2) + (4 - 1) + (4 - 3) = 6$.

Nesse caso, então, $total = 4+6 = 10$ e, portanto, os valores finais serão $E[x] = 0,4$ e $E[y] = 0,6$. Como y apareceu mais, terá um grau de pertinência maior, mesmo não tendo conseguido $temp = 0$ em nenhuma oportunidade, como foi o caso de x. Isso pode ser muito conveniente e desejado em alguns casos, mas em outros, dependendo do problema, não. Uma alternativa simples para contornar essas particularidades seria redefinir o cálculo de E para:

$$E[p_{i,m+1}] \;=\; \frac{\alpha - temp}{\alpha} \;=\; 1 - \frac{temp}{\alpha} \tag{4.19}$$

Em (4.19) estamos considerando um valor único para *temp*. Ou seja, não estamos levando em conta que um determinado símbolo pode aparecer em mais de uma previsão e, portanto, ter mais de um valor de *temp*. Neste caso teríamos de modificar o cálculo para algo do tipo:

$$E[p_{i,m+1}] \;=\; \frac{\sum(\alpha - temp)}{\sum \alpha} \;=\; 1 - \frac{\sum temp}{\sum \alpha} \tag{4.20}$$

De certa forma (4.19) é equivalente a (4.20), apenas levando em conta a possibilidade citada de mais de uma previsão para um mesmo símbolo e, portanto, somando esses diferentes valores e dividindo pela respectiva soma do fator α. Aproveitando o exemplo anterior, teríamos, neste caso, os valores calculados como $E[x] = 4/4 = 1,0$ e $E[y] = (2+3+1)/(4+4+4) = 6/12 = 0,5$.

Claro que pode haver outras formas de se determinar E que não apenas as duas sugeridas aqui, inclusive maneiras mais elaboradas ou refinadas. As aqui apresentadas, todavia, são simples de se calcular e eficientes, além de se adequarem ao espírito que se deseja conferir a essa variação do método.

No que diz respeito à variação nebulosa, a complexidade do algoritmo 4.7 é igual à do 4.6, ou seja $O(km^2)$. Só há um acréscimo, fora do laço principal, para calcular o vetor E. Se considerarmos como sendo e a quantidade de símbolos em E, então, teoricamente, a complexidade total seria $O(km^2 + e)$. O valor de e, no entanto, provavelmente vai ser desprezível em relação a km^2, e é quase impossível que tenha mais do que dois dígitos sendo, portanto, perfeitamente lícito desprezá-lo e considerar a complexidade final do algoritmo 4.7 como sendo $O(km^2)$ mesmo.

O grande problema deste método (em suas duas formas) é justamente sua eficiência. Para seqüências muito grandes (ou seja, com valores altos de m), o seu aproveitamento pode ficar muito comprometido. Isso sem contar o próprio valor de k, que é

mais um fator multiplicador e embora não seja quadrático como m, é ainda assim importante.

Um detalhe interessante sobre o aproveitamento deste método é o fato de que, ao contrário do descrito na seção 4.5.1, pode ser útil para situações em que se deseja mapear o contexto. Por exemplo, podemos supor que haja m variáveis que sejam levadas em conta para uma determinada ação. Grava-se esses valores em cada p_i, e atribui-se a $p_{i,m+1}$ justamente a ação tomada. Portanto, os primeiros m valores correspondem às condições de entrada e o $m+1$ à ação realizada. Este aproveitamento pode tornar este método "compatível" com aquele descrito na seção 4.3.2, podendo ser usados em conjunto. Uma outra vantagem é o fato de permitir diferenças entre as seqüências já gravadas (refinável através de α), podendo detectar padrões semelhantes que não sejam completamente idênticos e, no caso do algoritmo 4.7, ainda estipular um grau de pertinência.

4.6 Erro Artificial

"Errare humanum est."
Ditado latino

4.6.1 Conceito

Todos os esforços realizados nas áreas de sistemas inteligentes e correlatas sempre foram concentrados em tentar, mesmo que às vezes de forma caricata, se aproximar da inteligência humana. O fato de desejarmos que os agentes sejam inteligentes o suficiente de forma a desempenhar determinadas tarefas como seres humanos as fariam, é algo extremamente árdua que acaba, muitas vezes, nos desviando um pouco do objetivo original. Se errar é humano, e sabemos muito bem disso, por que, então, muitas vezes simplesmente ignoramos este fato, ou mesmo tentamos eliminá-lo, ao tentar criar um modelo de inteligência que, supostamente, deveria simular o de um humano?

A concepção do erro artificial é basicamente um conceito, e muito pouco algo específico a ser implementado. Esta é uma seção mais abstrata que as demais e busca muito mais sugerir possíveis aplicações do que propriamente propor um método concreto.

O erro artificial é basicamente o que o nome faz supor. A idéia é introduzir, de forma artificial, possíveis erros nas ações dos agentes. A idéia, a princípio, pode parecer um pouco absurda. Pretendemos mostrar, porém, que em determinados contextos ela pode fazer muito sentido.

4.6.2 Erro como Modificador

Uma questão que já foi levantada anteriormente diz respeito à possível desonestidade da inteligência implementada nos agentes. Mesmo que não seja proposital, em alguns casos essa característica pode ser particularmente irritante. Um exemplo simples é o caso dos jogos do gênero FPS. Em boa parte dos casos os agentes têm mira perfeita, ou seja, não erram um tiro (a não ser que o alvo se esquive de alguma maneira), o que, certamente, não é muito realista e, muitas vezes, pode provocar irritação nos jogadores, os quais, muito provavelmente, irão perceber isso em algum momento.

Para casos como o que foi citado, uma ação simples evita todos os transtornos causados pela "perfeição" do agente. Basta estipular uma determinado intervalo de erro para esse tipo de ação e calcular, de forma aleatória, algum desvio para o valor original. Se o valor utilizado para o tiro for, por exemplo, o ângulo θ do agente em relação a um determinado eixo, estipula-se uma variação desse ângulo dentro de um intervalo $[-\beta, \beta]$, sortear um valor aleatório α dentro deste intervalo e fazer $\theta' = \theta + \alpha$. É uma solução tão simples que é difícil de acreditar que muitos desenvolvedores não tenham a cogitado antes. De fato, parece mais fácil acreditar que a dificuldade para o jogador decorrente da perfeição do agente seja considerada vantajosa, e, portanto, fazendo parte da "inteligência" (que está mais para "malandragem") do jogo.

O erro não precisa ser introduzido apenas como um modificador aleatório para alguma variável determinada, como forma de evitar que o agente tenha características super-humanas, podendo ser usado, também, como um modificador casual para qualquer elemento do agente. Ou seja, de forma simples, seria uma probabilidade universal de o agente falhar em determinada tomada de decisão (ou ação).

Há várias formas diferentes de se implementar isso dentro da arquitetura de inteligência de um jogo (ou de um sistema qualquer). Um exemplo simples num controlador nebuloso seria aplicar uma probabilidade, a cada desnebulização de se errar no cálculo, o que seria semelhante ao caso do intervalo de erro sugerido anteriormente. Outra maneira diferente de fazer isso, no mesmo controlador, seria aplicar

uma chance de erro na avaliação das regras, fazendo, digamos, com que algumas fossem ignoradas (não avaliadas) ou até mesmo avaliadas de forma errada (aplicando um not na avaliação, por exemplo). Mais um jeito de realizar esse erro seria nos valores de entrada do sistema, aplicando um determinado ruído aos mesmos.

Esse tipo de aplicação de erro faz com que o agente pareça muito mais verossímil. Claro que trabalhar com dados incorretos ou imprecisos traz uma dificuldade muito maior ao sistema, mas isso é um desafio sedutor a ser encarado. Mais do que isso, trabalhar com esse tipo de variação dentro de um ambiente que pode ser considerado ideal (afinal, é uma simulação controlada pelo mesmo programa que controla o agente) é possibilitar uma generalização muito maior dos métodos aplicados, já que a sua validação será feita sob condições muito mais extremas, trazendo benefícios a uma futura aplicação em ambientes reais.

Até aqui estamos tratando a aplicação do erro apenas como uma espécie de modificação de valores. Tomando um jogo como uma série de ações discretizadas seqüenciais, as aplicações do erro nos instantes t_1 e t_2 são independentes. Claro que se pode extrapolar o erro calculado para um determinado intervalo Δt, de forma que o erro seria o mesmo entre t_1 e $t_1 + \Delta t$. De um jeito ou de outro, a aplicação continuaria desempenhando um papel de modificador.

4.6.3 Erro como Busca

Uma maneira diferente de encarar o erro artificial seria vê-lo como uma espécie de mecanismo de busca. Seria, de forma análoga, como a mutação em algoritmos genéticos. O erro tem, neste caso, a função de buscar estratégias diferentes que, em caso de serem bem sucedidas, podem ser incorporadas ao leque de opções do agente.

Como aplicar isso depende em muito daquilo que se está usando. Agentes baseados em computação evolucionária já têm esse tipo de característica naturalmente. Já um controlador nebuloso poderia simplesmente gerar aleatoriamente novas regras (e/ou mudar as atuais) e, após um determinado intervalo de tempo, avaliar, de alguma forma, se valeu a pena aquela modificação e, em caso positivo, mantê-la.

Em espaços de busca relativamente grandes e com poucos agentes, a chance de um erro ser útil será particularmente pequena. Ainda assim, mesmo que não adicione nada à estratégia do agente, pelo menos pode servir para torná-lo mais realista, suscetível de errar.

Quanto menor for a probabilidade de acontecer o erro menor, será a freqüência

do mesmo e, conseqüentemente, menor será a chance de se encontrar um desvio de estratégia útil. Por outro lado, quanto maior for essa probabilidade, mais aleatório o agente tenderá a ser, o que pode não ser muito desejável. Um valor adequado deve ser encontrado, o que, em alguns casos, é difícil de se determinar.

4.6.4 Erro como Estratégia

"As histórias mais errôneas são aquelas as quais julgamos conhecer melhor e, conseqüentemente, nunca as questionamos."

Stephen Jay Gould

Uma face menos óbvia do erro artificial é a estratégia. Usar o erro propositalmente como uma estratégia parece ser algo muito pouco explorado, ou pelo menos não se tem notícia de que seja usado.

Muitas vezes o sentimento de superioridade óbvia pode levar uma pessoa a menosprezar o seu adversário. Quando um computador é esse adversário, o sentimento pode ser ainda mais forte.

Testemunhar o adversário fazendo algo extremamente estúpido (pelo menos aparentemente) pode causar no jogador uma certeza de vitória e um menosprezo pelo adversário que podem causar nele falta de atenção e pouca concentração no jogo pela simples certeza da vitória. Tomar propositalmente atitudes extremamente estúpidas por um período de tempo de forma a causar no jogador essa sensação talvez seja uma estratégia interessante a ser explorada.

Podemos desenvolver um pouco mais esse conceito. Mesmo que o relaxamento não seja suficiente para se manter a longo prazo (ou seja, mais cedo ou mais tarde o jogador voltaria ao estágio normal de atenção), é possível aproveitar um lapso momentâneo decorrente de uma ação inusitada para conseguir alguma vantagem dentro do contexto do jogo.

Por exemplo, o jogador vendo o seu adversário tentando passar através de uma parede (o que muitas vezes é um problema comum de agentes sem um sistema adequado de navegação) ou até mesmo no chão (fingindo-se de morto, possivelmente), o que talvez lhe cause surpresa e até mesmo um momento de contemplação. Muitas vezes é comum que o jogador jogue com mais calma quando percebe que sua vitória (ou uma determinada ação que lhe traga alguma vantagem) está garantida. É natural que se aja, nesses casos, de forma mais relaxada e, de repente, o agente pode sair

de seu transe e aproveitar o momento de surpresa do jogador para atacá-lo, tirando proveito disso.

Ações inusitadas programadas podem, portanto, ser usadas como estratégia para surpreender o jogador. Uma forma interessante de fazer isso seria aproveitar as próprias limitações dos agentes e simulá-las nesses casos. Por exemplo, se, de alguma forma, o sistema de navegação não for adequado e, em alguns casos, o agente realmente tenta atravessar a parede por algum motivo, aproveita-se isso como estratégia. Esse aproveitamento do erro também pode desempenhar a tarefa de atenuar um pouco as falhas mais evidentes do projeto, uma vez que o próprio erro do jogo passa a ser explorado e confundido como uma estratégia do mesmo!

Mas é óbvio que, ao longo do tempo, o jogador passará a ser mais cuidadoso nas ocasiões especiais em que determinadas ações aparentemente estúpidas são utilizadas como estratégia, de forma que, mais cedo ou mais tarde, ele supostamente deixará de ser enganado por tal artifício. Para prolongar o máximo possível o "tempo de vida" das estratégias de erro, deve-se evitar usá-las com excessivas repetições em pequenos intervalos de tempo e, também, se possível, ter várias estratégias de erro diferentes, de maneira a poder alternar entre as mesmas mais freqüentemente.

Capítulo 5

Implementação, Aplicações, Resultados e Análise

"O artista é o criador de coisas belas."
Oscar Wilde, *O Retrato de Dorian Gray*

Neste capítulo são descritas e analisadas as diversas implementações diferentes feitas para se poder observar o comportamento de cada um dos métodos e estratégias propostos neste trabalho.

As seções 5.1 e 5.2 tratam das experiências iniciais com evolução em tempo real e coevolução, cujos resultados obtidos motivaram e inspiraram o desenvolvimento do método descrito na seção 4.1, o qual, por sua vez, tem sua implementação descrita na seção 5.3, com os respectivos resultados. Os testes e resultados dos métodos de previsão descritos nas seções 4.3.2 e 4.5 são discutidos na seção 5.7, que também traz algumas considerações a respeito do uso de lógica nebulosa na modelagem da inteligência em jogos. A estratégia de padrões adaptativos descrita na seção 4.2 é testada e discutida nas seções 5.4 e 5.5. Finalmente, os métodos de máquina de estados nebulosa e navegação descritos nas seções 4.3.3 e 4.4 são apresentados na seção 5.6.

Todos os programas, a não ser quando explicitamente indicado o contrário, são originais, desenvolvidos durante a elaboração deste trabalho. Com exceção do jogo da seção 5.1 e do controlador usado na seção 5.6, todos os demais programas foram implementados em C usando o compilador gcc, rodando tanto no Windows quanto no Linux. A biblioteca de jogos Allegro foi utilizada em todos os jogos feitos em C e uma *engine* para jogos 2D chegou a ser criada para auxiliar na implementação dos

jogos utilizados nas seções 5.3 e 5.7.

Boa parte dos desenhos usados nos jogos das seções 5.1, 5.3 e 5.7 são oriundos da *SpriteLib*, de propriedade de Ari Feldman e utilizados aqui, dentro de seus termos de uso, para fins não lucrativos, como parte do desenvolvimento deste trabalho acadêmico.

Como já foi levantado nas seções 1.1.2 e 1.2, há algumas dificuldades fortes no uso de jogos como base de um projeto de pesquisa para que seja possível aproveitar todo o potencial que existe nessa área. Muitas foram essas desvantagens encaradas ao longo deste trabalho, como falta e inconsistência de dados, participação errante de voluntários etc. Juntando isso à dificuldade de interpretação dos dados, tivemos de lutar contra um número considerável de adversidades para poder levar adiante o projeto, a fim de alcançar os objetivos propostos.

Apesar de todos esses problemas, foi possível obter resultados animadores, os quais serão analisados levando-se em consideração todas essas dificuldades, dentro do contexto deste trabalho. As seções a seguir, portanto, descrevem, detalhadamente, as implementações usadas para testar os métodos propostos no capítulo anterior.

5.1 Genetic Invaders

Este foi o primeiro teste realizado para avaliar o aproveitamento dos métodos tradicionais de algoritmos genéticos dentro de um contexto em tempo real. Foi implementado um jogo simples em Java (um *applet* para rodar dentro de um navegador) cujo objetivo era destruir os alienígenas que ficavam rondando a tela. Cada fase era composta por dez inimigos que se movimentavam tentando fugir do ponteiro do *mouse* e o jogador os destruía justamente clicando em cima deles. Quando todos eram derrotados, iniciava-se uma nova fase. A figura 5.1 mostra a tela inicial de jogo.

A idéia era aproveitar cada fase como uma geração do algoritmo genético e, conseqüentemente, cada alienígena como um indivíduo. Logo, em cada geração haveria dez indivíduos, e a evolução aconteceria entre as fases. O objetivo, portanto, era que a cada fase passada pelo jogador, os agentes ficassem mais "inteligentes" e o jogo, dessa maneira, progressivamente mais difícil. Com isso seria possível testar não só o aproveitamento de algoritmos genéticos tradicionais em tempo real como, também, de coevolução, já que, supostamente, haveria competição entre os agentes

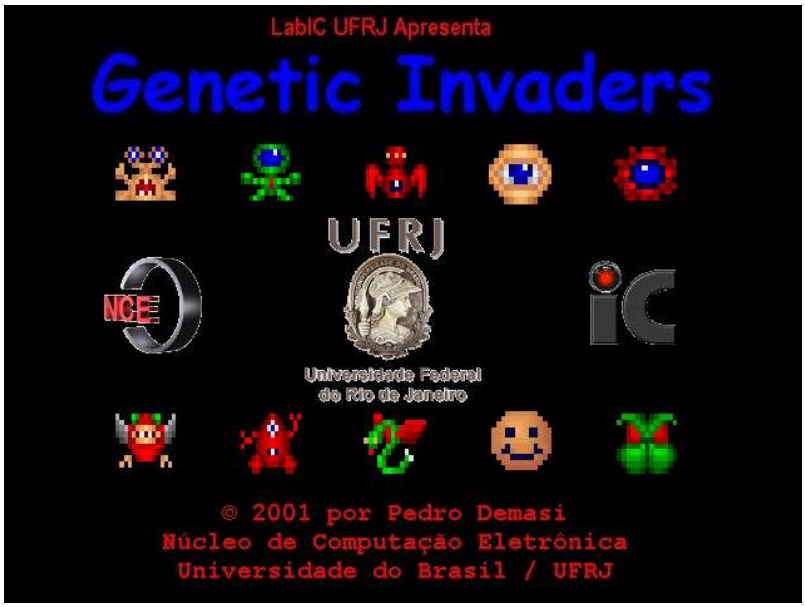

Figura 5.1: Tela inicial do jogo "Genetic Invaders"

e o jogador.

Embora teoricamente simples, a realização desse objetivo provou ser extremamente complicada e, em última instância, simplesmente impraticável. Foram feitas diversas tentativas e mudanças no algoritmo básico de forma a conseguir uma configuração cujos resultados fossem pelo menos animadores. Essas tentativas, todavia, acabaram sendo infrutíferas, como veremos a seguir.

5.1.1 Função de Avalição

A primeira função de avaliação idealizada foi extremamente simples e diretamente relacionada à competição entre as espécies (agentes e jogador). O *fitness* de cada indivíduo era proporcional ao seu tempo de vida na fase. Assim, aqueles que fossem mortos logo de início seriam os menos aptos, ao contrário daqueles que fossem os últimos.

Apesar de nos parecer, pelo menos a princípio, intuitivamente bem adequada ao problema, já nos primeiros testes essa definição da função de avaliação mostrou-se falha. O grande problema, pelo que foi observado, é que ela partia da premissa que os últimos alienígenas a serem caçados e, conseqüentemente, mortos pelo jogador, seriam justamente os mais "inteligentes" da população e, portanto, os mais

difíceis da fase. O que foi observado, na prática, contudo, foi que muitas vezes os jogadores preferiam deixar para perseguir agentes que não ofereciam muito desafio justamente por último. Era comum acontecer de indivíduos que simplesmente não se movimentavam serem os últimos a morrer, o que os deixava com alto grau de aptidão dentro da geração. Um outro fator, de certa forma relacionado com este último, que pudemos observar foi a disposição dos jogadores em não desistir de caçar um determinado agente. Em outras palavras, se logo o primeiro indivíduo a ser perseguido pelo jogador calhasse de ser o mais evoluído (e, portanto, o mais difícil) daquela geração, normalmente ele não desistia de persegui-lo enquanto não o destruísse, mesmo havendo outros bem mais fáceis para serem mortos. Outro comportamento comum observado nos jogadores era uma certa tendência a tentar matar o agente que estivesse mais perto, o qual não necessariamente era o mais fácil a ser destruído.

A conseqüência desses comportamentos todos foi muito simples: havia uma proporção muito grande de indivíduos com alta aptidão que não necessariamente eram os mais difíceis de ser derrotados, invalidando qualquer tentativa de análise mais profunda dos resultados. Era necessário, portanto, uma nova função de avaliação e o reinício dos testes.

A grande dificuldade era definir, após a fracassada experiência anterior, uma função que descrevesse da melhor forma possível a aptidão de um indivíduo. Já que a definição mais intuitiva e direta da competição não funcionou, foi preciso modificar o próprio código do jogo para incluir uma nova característica.

A idéia baseou-se justamente no fato, já descrito, de, muitas vezes, o jogador tender a perseguir os agentes até conseguir destruí-los, independetemente de haver outros mais fáceis ou não. Ora, para alienígenas que se movimentassem mais, era natural que a quantidade de tentativas para matá-los fosse consideravelmente maior do que para aqueles que simplesmente ficassem parados. O problema, no caso, seria como saber a qual agente "pertencia" um determinado *clique* errado. Para isso foi criada uma espécie de fronteira para cada indivíduo. Quando o jogador errava o alienígena mas acertava sua fronteira, então era computado um erro induzido para o agente. Se o erro pertencesse a mais de uma fronteira, então todas essas eram levadas em conta.

A função de avaliação, portanto, estava definida como sendo um valor proporcional à quantidade de erros induzidos de cada agente. Logo, o agente que fosse o

mais apto seria aquele que tivesse gerado a maior quantidade de erros no jogador. Uma outra definição paralela a essa também foi usada, considerando tanto o tempo quanto os erros induzidos, mas com um peso bem maior a estes do que àquele. As duas definições, na prática, não mostraram guardar muitas diferenças entre si, de forma que foi indiferente o uso de uma ou de outra para a obtenção dos resultados finais.

O uso dessa segunda definição ajudou a contornar os problemas gerados pela primeira, mas devido a diversos outros fatores, que serão analisados com detalhes na seção 5.1.4, os resultados acabaram não sendo dos mais animadores.

5.1.2 Codificação dos Indivíduos

Foram usadas duas formas básicas de codificar os genes dos indivíduos. A primeira foi aproveitada apenas nos testes iniciais e logo foi substituída pela segunda, que mostrou ser mais adequada ao problema.

O modelo inicial pretendido seria codificar nos genes uma seqüência determinada de movimentos que o alienígena iria fazer. Isso funcionaria de certa forma como os padrões descritos na seção 3.2.3, aproveitados em algoritmos genéticos. O problema é que o comportamento dos agentes seria independente dos movimentos do jogador e, portanto, como se estivessem "cegos". Na verdade o que teríamos, idealmente, numa geração plenamente evoluída, seria vários agentes se movimentando pela tela de forma desordenada sem qualquer preocupação com a perseguição realizada pelo jogador.

A segunda forma de codificar os indivíduos tinha a principal preocupação de realizar as ações de acordo com a posição do *mouse* e, portanto, do jogador. Cada grupo de três bits representa uma ação como conseqüência de uma regra. Cada combinação desses bits indica o movimento em uma das oito direções: cima, baixo, esquerda, direita, cima e esquerda, baixo e direita, cima e direita e baixo e esquerda. Foram usadas seis regras, como veremos mais adiante. Logo, como cada regra precisa de três bits (oito direções) para indicar a conseqüência, cada indivíduo tem $6 \times 3 = 18$ bits de código genético. Das seis regras, três são relativas ao eixo x e as outras ao eixo y. Cada regra leva em consideração o Δx ou Δy, ou seja, a distância entre o x do jogador e o do agente (análogo ao y). As seis regras, portanto, representam $\Delta x = 0$, $\Delta x > 0$, $\Delta x < 0$, $\Delta y = 0$, $\Delta y > 0$ e $\Delta y < 0$. Dessa maneira, há sempre duas regras sendo disparadas (uma em relação a Δx, outra em relação a Δy), e, por isso, se as

ações forem opostas (por exemplo, direita e esquerda), o agente simplesmente fica parado.

Cada três bits do código genético dos agentes, portanto, representam uma regra como, por exemplo, `se` $\Delta x > 0$ `então movimento = esquerda`.

5.1.3 Parâmetros Genéticos

Vários parâmetros foram utilizados, ao longo dos testes, para o algoritmo genético. Diferentes valores percentuais para elitismo, mutação, indivíduos aleatórios e cruzamento, assim como métodos de cruzamento de um ponto e dois pontos. Para a seleção foi sempre usado o método da roleta.

Nenhuma configuração se mostrou muito superior às demais. Os parâmetros finais utilizados foram 30% de indivíduos gerados aleatoriamente, 20% por elitismo e 50% por cruzamento, realizado com o método da roleta em um ponto. A taxa de mutação por bit ficou em 1%, e quando acontecia o bit era invertido (*bit flip*). A população, como já foi dito anteriormente, era formada por dez indivíduos e a primeira geração era formada por agentes com código completamente aleatório.

A quantidade de gerações e, portanto, de fases, também variaram. A princípio a idéia era usar vinte fases, mas logo se percebeu que ninguém agüentava jogar por tanto tempo. Passou-se a usar dez e cinco fases que, no final, acabaram não fazendo muita diferença.

5.1.4 Resultados e Discussão

Foram usadas, para poder avaliar o desempenho dos agentes, duas medidas: o tempo gasto pelo jogador em cada fase e a quantidade de tiros (cliques) dados. Quanto menos tempo gasto numa fase, mais rápido o jogador derrotou os alienígenas. E quanto menor a quantidade de tiros dada, mais fácil foi destruir os agentes.

Os resultados iniciais foram decepcionantes. Com a primeira função de avaliação descrita em 5.1.1 a única tendência do jogo era ficar mais fácil, isso quando havia alguma tendência. Após a mudança para a segunda função de avaliação, desapareceu qualquer tendência e tudo o que restou foram dados sem qualquer indicação concreta.

Quanto aos dados obtidos, não foi possível observar qualquer coerência. Em geral as fases alternavam momentos de maior facilidade e dificuldade sem o menor critério aparente. A figura 5.2 mostra um exemplo para cinco gerações.

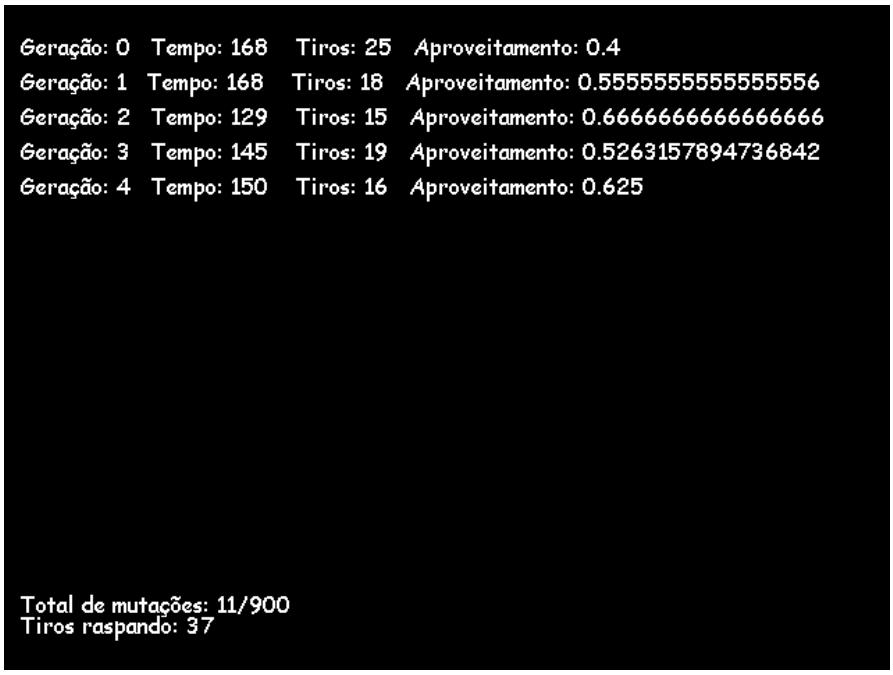

Figura 5.2: Exemplo de tela de resultados do jogo "Genetic Invaders"

O que se nota, nesse exemplo, é que o jogo vai ficando mais fácil progressivamente nas três primeiras gerações, fica um pouco mais difícil na quarta para, na quinta, novamente ficar mais fácil. Esse tipo de comportamento errático dos dados foi extremamente comum, e nenhuma tendência concreta de qualquer forma pôde ser observada.

Um fator que poderia explicar esse facilidade crescente seria a evolução do próprio jogador, a qual seria muito superior à dos agentes. Essa teoria, porém, não se sustenta pois não havia uma tendência do jogo a ficar mais fácil, mesmo após o mesmo jogador jogar diversas vezes. Os resultados eram sempre semelhantes aos das primeiras vezes, igualmente caóticos, independente da experiência do jogador.

Na verdade, se fosse verificado um comportamento errático (com altos e baixos), mantendo essa tendência, seria um bom resultado, pois mostraria um comportamento competitivo entre a dificuldade do jogo e o aprendizado do jogador (ainda que fosse difícil para um caso com tão poucas possibilidades como este). Não foi, infelizmente, o caso, pois isso se verificou, como foi dito, mesmo com o jogador bem mais experiente, sem nenhuma indicação de ganho de perícia ou aumento de dificuldade dos agentes. Na verdade, esses resultados indicam muito mais as dife-

renças entre as jogadas do próprio jogador do que mudanças no comportamento dos agentes.

Entrevistando as pessoas que jogaram o "Genetic Invaders", a opinião geral era de que não haviam sentido diferença significativa entre uma fase e outra, nada que pudesse ser percebido, pelo menos. Observando o comportamento do jogo, porém, percebemos alguns detalhes que, a princípio, analisando apenas os números, não podiam ser descobertos.

Foi possível notar que, muitas vezes, determinados padrões de comportamento dos agentes passavam a se repetir com freqüência. Na verdade, havia muitos agentes com ações iguais ou semelhantes. Isso, certamente, tornava mais previsível seu comportamento e, portanto, mais fácil sua derrota. A diversidade ficava por conta dos alienígenas gerados aleatoriamente. Eliminando essa proporção, ou seja, criando a nova geração apenas a partir de cruzamentos e elitismo, o jogo tendia a ficar completamente previsível e, aí sim, sua única tendência seria ficar cada vez mais fácil.

É interessante notar que o espaço de busca do problema é relativamente pequeno, afinal, com apenas 18 bits, são $2^{18} = 262.144$ combinações. A quantidade de indivíduos por geração, entretanto, também é muito pequena (dez) e a pouca diversidade presente em cada geração acabou tornando a evolução muito viciada. Aumentando a componente aleatória, simplesmente colaboramos para tornar a evolução do jogo caótica e imprevisível, enquanto que eliminando essa componente tornamos os agentes viciados e fáceis de serem derrotados.

No final das contas, com relação aos resultados, este teste foi um fracasso absoluto, nada pôde ser aproveitado e logo a idéia do jogo foi abandonada. Por outro lado, a experiência serviu para mostrar que um determinado caminho não se mostrava o mais apropriado para servir de abordagem para evolução em tempo real. A quantidade de agentes era realmente insuficiente para que qualquer evolução fosse possível por pelo menos um número razoável de gerações, o que significa que, provavelmente, para que o jogo pudesse evoluir, seriam necessárias tantas gerações que nenhum jogador teria paciência de jogar.

Esses questionamentos possibilitaram, enfim, que o método descrito na seção 4.1 pudesse ser moldado de maneira mais direcionada. De alguma forma, portanto, o fracasso desta experiência possibilitou que outra estratégia pudesse emergir e ter resultados bem mais animadores.

5.2 Os Três Porquinhos

Após o fracasso inicial com os testes de evolução em tempo real, era necessário verificar a aplicação das idéias de coevolução para seu futuro aproveitamento. Como a competição entre jogador e agente já é natural do próprio contexto do jogo, afinal são adversários, o objetivo era verificar a cooperação entre agentes como forma de conseguir uma estratégia conjunta melhor para derrotar o jogador.

Para realizar o teste, foi implementada uma simulação de um jogo simples. Os agentes são representados por três porcos, enquanto que o jogador pelo "lobo mau". Como a idéia era isolar a cooperação e avaliar seu aproveitamento, o jogador era, também, um agente que não evoluía. Assim, com apenas os porcos evoluindo, foi possível verificar a atuação da cooperação entre eles sem que houvesse intereferência por causa de uma possível evolução do adversário. O objetivo foi comparar os resultados das evoluções dos porcos usando algoritmos genéticos convencionais e usando coevolução cooperativa. Os resultados deste trabalho foram, inicialmente, apresentados em (DEMASI, CRUZ, 2002a) e estão aqui desenvolvidos.

O objetivo dos porcos no jogo é se salvar, enquanto que o do lobo é capturá-los (e comê-los). Há exatamente três porcos em cada simulação e um igual número de casas. Sempre que um porco alcança uma casa, ele está automaticamente salvo, ao passo que se ele for alcançado pelo lobo, está automaticamente capturado. Cada casa só pode comportar um porco e, portanto, uma vez que esteja ocupada, não poderá mais ser aproveitada.

O mundo do jogo é um espaço discretizado representado por uma matriz de 201 x 201 pontos. Cada elemento (lobo, porcos e casas) ocupa exatamente um ponto no espaço e não pode haver mais de um num mesmo ponto simultaneamente. A posição inicial do lobo é $x = 100$ e $y = 200$. A tabela 5.1 mostra as posições inicias de cada porco e cada casa.

Porco	**X**	**Y**	**Casa**	**X**	**Y**
Porco #1	90	100	Casa #1	0	0
Porco #2	100	100	Casa #2	100	0
Porco #3	110	100	Casa #3	200	0

Tabela 5.1: Posições inicias dos porcos e das casas

Por se tratar de um mundo discretizado com movimetos igualmente discretos,

todas as distâncias são calculadas e consideradas usando a Distância de Manhattan, ou seja, para dois pontos $p(x, y)$ e $p'(x', y')$, temos que:

$$d(p, p') = |x - x'| + |y - y'| \tag{5.1}$$

Então, tomando por base a tabela 5.1, podemos calcular as distâncias iniciais entre os porcos e as casas usando a fórmula (5.1), que são ilustradas na tabela 5.2. Pode-se perceber claramente que a casa mais próxima de todos os porcos é a de número 2 e isso, como veremos posteriormente, terá um grande peso no resultado final das estratégias usadas pelos porcos.

Porco	**Casa**	**Distância**	**Porco**	**Casa**	**Distância**	**Porco**	**Casa**	**Distância**
1	1	190	2	1	200	3	1	210
1	2	110	2	2	100	3	2	110
1	3	210	2	3	200	3	3	190

Tabela 5.2: Distâncias iniciais entre porcos e casas

O lobo e os porcos só podem se mover em uma das quatro direções: cima, baixo, esquerda e direita. As casas têm posição constante. A cada passo do jogo, as presas movimentam-se um ponto e o lobo dois (nessa ordem). Logo, o lobo tem o dobro da velocidade dos porcos.

O algoritmo do lobo é fixo: ele sempre vai na direção do porco mais próximo. A razão disso, como já foi dito, é isolar a parte da coevolução cooperativa entre os porcos, pois caso o lobo também evoluísse, haveria, então, uma competição implícita entre lobo e porcos, o que dificultaria bastante o acompanhamento dos resultados.

Cada porco tem à disposição quatro algoritmos possíveis de movimentação, sendo que a cada instante deve escolher um dentre eles, os quais são os mostrados pela tabela 5.3.

Número	**Algoritmo**
Algoritmo 0	Fica parado
Algoritmo 1	Vai em direção à casa mais próxima
Algoritmo 2	Foge do lobo
Algoritmo 3	Movimento aleatório

Tabela 5.3: Algoritmos de movimentação dos porcos

Os porcos usam regras através das quais escolhem qual algoritmo utilizar em cada passo. Essas regras, por sua vez, são baseadas em duas variáveis de entrada: distância para a casa mais próxima (d_c) e distância para o outro porco mais próximo (d_p). Uma variável, em função de uma distância d, assume um dos valores descritos pela tabela 5.4.

Distância	**Valor**
$0 \leq d \leq 100$	Perto
$100 < d \leq 200$	Médio
$d > 200$	Longe

Tabela 5.4: Valores de acordo com a distância

Como são duas variáveis de distância (d_c e d_p) usadas pelas regras e cada uma pode assumir três valores, podemos concluir que são $3^2 = 9$ o total de regras para cada porco, cobrindo todas as combinações de valores entre as variáveis. Cada uma dessas regras tem, como conseqüência, um dos quatro algoritmos já apresentados, que requerem 2 bits para sua representação. Logo, o cromossomo de cada porco precisa ter $9 \times 2 = 18$ bits.

Foram realizadas dez simulações nas duas abordagens diferentes para o problema: uma utilizando algoritmos genéticos convencionais e outra com coevolução cooperativa. Os parâmetros em comum usados em ambas as abordagens estão descritos na tabela 5.5.

Parâmetro	**Valor**	**Parâmetro**	**Valor**
% de elitismo	10	Gerações	100
% de aleatórios	15	Probabilidade de Mutação	0,5%
% de cruzamentos	75	Probabilidade de cruzamento	85%

Tabela 5.5: Parâmetros genéticos comuns nas simulações

Assim, a cada geração, 10% dos indivíduos são gerados por elitismo (escolhendo os 10% mais aptos da geração anterior), 15% de forma aleatória e os demais 75% através de cruzamento entre pares de indivíduos da geração anterior, selecionados pelo método da roleta. A cada par selecionado há 85% de chance de ocorrer o *crossover* (em um ponto aleatório), caso contrário os pais simplesmente são replicados para a geração seguinte. A probabilidade de mutação é calculada por bit e, em ocor-

rendo, este é trocado. A primeira geração é toda formada por indvíduos aleatórios. A função de avaliação foi definida como:

$$f = s \times (K - p) + (1 - s) \times (k \times p - dc) \tag{5.2}$$

Onde $s = 1$ se o porco se salvou ou $s = 0$ caso contrário; K é uma constante que premia o porco salvo (usamos $K = 5000$); k é uma outra constante que premia o tempo vivido pelo porco morto (usamos $k = 5$); p é o tempo durante o qual o porco viveu (quantos passos deu antes de morrer ou se salvar); d_c tem o mesmo significado já explicado. Essa função premia os porcos que conseguiram se salvar (e, dentre esses, os que o fizeram em menos tempo) e, entre os capturados, aqueles que viveram mais tempo, ficando o mais perto possível de uma casa vazia.

5.2.1 Simulações e Resultados

Para a simulação usando algoritmo genético convencional, em cada geração das simulações foram realizadas 75 partidas entre o lobo e os porcos. Como cada partida conta com 3 porcos, o total de indivíduos por geração foi de $3 \times 75 = 225$. A figura 5.3 mostra os resultados obtidos de porcos sobreviventes por geração em cada uma das 10 simulações feitas.

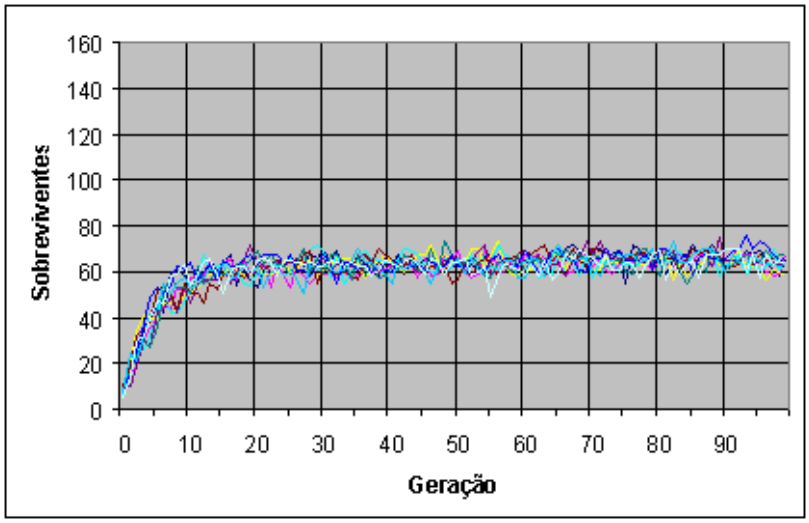

Figura 5.3: Resultados obtidos com o algoritmo genético convencional

O processo utilizado, no caso do coevolução cooperativa, foi o seguinte: para

cada indivíduo de cada subespécie a ser avaliado, sorteia-se um de cada outra das duas subespécies entre os 50% mais aptos na geração anterior. Estes servem apenas para completar o cenário da partida, como já dito sobre coevolução cooperativa na seção 2.3.2. Na primeira geração, como não há geração anterior, esses indivíduos complementares são totalmente aleatórios. Cada porco da simulação representa uma subespécie.

Para preservar o mesmo esquema utilizado para as simulações do algoritmo genético convencional, foram feitas 75 partidas por geração também para a coevolução cooperativa. Como cada porco representa uma subespécie, foram feitas 25 partidas para cada uma. Logo, ao invés de 225 indivíduos por geração, temos 75, sendo 25 de cada subespécie. Os outros 150 vêm da geração anterior e usados apenas para avaliação (50 para a avaliação de cada subespécie).

A função de avaliação sofre uma pequena alteração, passando a ser a soma do fitness de cada indivíduo na partida. Ou seja, a avaliação de um indivíduo de uma subespécie é a soma do *fitness* dele com o de cada um dos outros dois indivíduos complementares, todos calculados na partida em questão. Logo, os indivíduos sorteados para compor a partida não "trazem" o *fitness* da geração anterior nem "levam" esse novo *fitness* para o cálculo em sua subespécie. A figura 5.4 mostra os resultados obtidos de porcos sobreviventes por geração em cada uma das 10 simulações feitas.

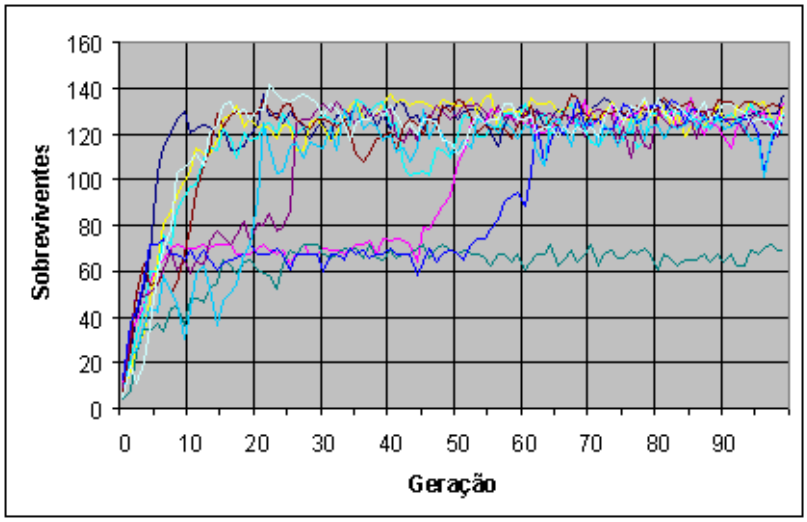

Figura 5.4: Resultados obtidos com coevolução cooperativa

5.2.2 Análise e Discussão

Analisando os resultados mostrados pela figura 5.3, notamos uma forte e clara tendência das simulações feitas usando algoritmos genéticos convencionais a se estabilizarem entre 60 e 70 sobreviventes a partir da 20^a geração. Isso corresponde a quase um terço de sobreviventes, ou seja, praticamente um porco salvo por partida.

O resultado é previsível, afinal, o algoritmo genético convencional, funcionando de forma egoísta, tende a encontrar, como melhor solução, ir para a casa mais próxima. Ou seja, numa configuração estável obtida pelo método, todos os porcos buscam a casa mais próxima, a qual, como pode ser visto na tabela 5.2, é a mesma para todos. Como apenas um pode ocupá-la, somente esse se salva. Quando a casa é ocupada e os outros dois porcos devem ir em direção às outras, o lobo já está perto demais para que eles consigam escapar.

A melhor estratégia para um porco, individualmente, é justamente correr para a casa mais próxima. Para o grupo, entretanto, notamos que essa não é a melhor solução. Observando os resultados da figura 5.4, com os resultados da cooperação, notamos uma tendência a se estabilizar (com exceção de uma simulação) entre 120 e 140 sobreviventes (o dobro do caso anterior). Isso acontece porque quando as subespécies da ponta geram uma solução em que tendem a fugir da posição central e buscam as casas laterais, em geral dois porcos sobrevivem.

Resumidamente, quando os porcos da ponta buscam as outras casas, o lobo, após perseguir o porco do meio (que consegue se salvar), vai buscar um dos outros dois. Fatalmente um deles será capturado, mas o outro, estando já bem distante, consegue se salvar. Se, no entanto, um desses porcos da ponta resolver ir pro meio, dois serão capturados (o da ponta que não foi pro meio e um dos que disputam a casa central).

Isso significa que, para o sucesso maior do grupo, deve haver uma coordenação entre as estratégias da ponta e do centro, sendo que um porco será sacrificado. As diferenças apresentadas no salto da faixa dos 60 a 70 para a dos 120 a 140 (inclusive com uma simulação não conseguindo esse salto) indicam que, a partir do momento em que esse tipo de solução coordenada é gerada, ela acaba sendo passada com sucesso para as gerações seguintes. Uma das 10 simulações, contudo, não "conseguiu" gerar essa solução e ficou estagnada no mesmo padrão de solução do AG convencional, não dando o "salto evolutivo".

Este resultado, por sinal, lembra um pouco a teoria de equilíbrios pontuados, de Stephen Jay Gould, que argumenta que a evolução, ao longo da história, alterna momentos de certa estagnação com outros de grande aceleração, não ocorrendo, portando, de forma suave e, sim, por saltos (GOULD, 1991).

Analisando a situação, levando-se em conta apenas a preocupação de um porco, tanto faz, para ele, se nenhum outro porco foi capturado além dele ou se houve outros porcos que também foram alcançados. Para ele apenas interessa se ele fugiu ou não e, individualmente, a chance de escapar é bem maior se for para a casa que está mais perto. Se todos fizerem isso, contudo, apenas um vai sobreviver. Se os porcos da ponta, ao contrário, tentarem buscar as casas mais distantes, há uma boa chance de um deles ser capturado, mas pelo menos os outros dois, se esse for o caso, escaparão.

Esses resultados mostram que o uso de métodos baseados em algoritmos genéticos cooperativos podem auxiliar no desenvolvimento de estratégias coordenadas entre grupos de agentes de forma a melhorar seu desempenho contra o jogador dentro de um determinado jogo, permitindo que trabalhem em conjunto.

5.3 Olhos Assassinos

Após obter os resultados descritos nas seções 5.1 e 5.2 e a criação do método apresentado na seção 4.1, era necessário fazer a implementação de um jogo para poder testar seu funcionamento na prática. Foi, então, criado o jogo "Olhos Assassinos", cujo nome se deve ao fato de os agentes serem representados por desenhos de olhos.

O jogo se passa numa sala quadrada cujas dimensões são 480×480 pixels e onde um personagem, controlado pelo jogador, deve sobreviver. Ele é perseguido por 16 pequenos monstros (os olhos assassinos), cujo objetivo é, justamente, matar o jogador, e, para conseguir isso, basta tocarem-no. O personagem conta, para se proteger, com uma pistola e pode disparar tiros contra os olhos para destruí-los. Sempre que um tiro disparado atige um agente, este morre e um ponto é contabilizado. Logo, a pontuação final equivale ao número de olhos destruídos durante o jogo. Para cada agente morto, porém, um novo entra na sala, pelas bordas da mesma. A arma tem, inicialmente, 20 tiros para serem disparados e a cada 15 segundos um novo cartucho aparece numa posição aleatória. Se conseguir pegar a munição, o jogador ganha mais 20 tiros.

Um outro movimento que pode ser feito é o teletransporte. Para cada 30 segundos o jogador pode usar esse recurso uma única vez. Sempre que o fizer, ele desaparece da posição que ocupar para aparecer numa nova, aleatória. Isso pode ser perigoso, obviamente, pois a nova posição pode estar cercada de agentes. Por outro lado, numa situação de desespero em que tenha pouca munição e esteja cercado de olhos, esse pode ser um recurso salvador.

A velocidade dos agentes e do jogador é a mesma, ou seja, eles podem se locomover o mesmo número de pixels a cada turno. O jogador, contudo, pode andar em oito direções (cima, baixo, esquerda, direita e diagonais), enquanto que os olhos podem apenas fazê-lo em quatro direções. O tiro dado pelo jogador segue a trajetória de sua posição atual. Ou seja, se ele estiver andando na direção da diagonal inferior esquerda, o tiro dado seguirá restritamente essa direção.

O jogador tem, inicialmente, três vidas. Ao perder as três vidas, o jogo termina e a pontuação final, como já foi dito, é a quantidade de agentes mortos. Sempre que é tocado por um olho, ele perde uma vida e o jogo volta à posição inicial, a qual está ilustrada na figura 5.5. Pode-se perceber que os 16 agentes estão distribuídos pelos quatro cantos da sala de maneira razoavelmente simétrica, com quatro olhos ocupando cada canto. Ao voltar para a posição inicial, a quantidade de tiros restantes na pistola permanece a mesma de antes de o jogador morrer. É possível, também, ver na figura 5.5 um cartucho um pouco acima da posição central da sala, local ocupado justamente pelo jogador.

5.3.1 Descrição dos Agentes

Os olhos foram divididos em dois grupos de oito em cada. Logo, há duas diferentes subespécies as quais terão de cooperar entre si de forma a derrotar o jogador. Ambas têm o mesmo código genético, o que significa que a grande diferença entre elas será a maneira como se comportam durante o jogo.

Cada agente conhece dois algoritmos básicos: fugir e perseguir. Portanto, dados um alvo e um desses algoritmos, o olho age perseguindo ou fugindo. Os alvos possíveis são o jogador, um cartucho ou um tiro. Em cada turno, todos os agentes têm a informação precisa a respeito das distâncias em relação ao jogador, ao cartucho e ao tiro mais próximo. Essas distâncias, por sua vez, são calculadas em pixels pela distância de Manhattan, já definida em (5.1).

Assim como no jogo discutido na seção 5.2, neste caso também há variáveis

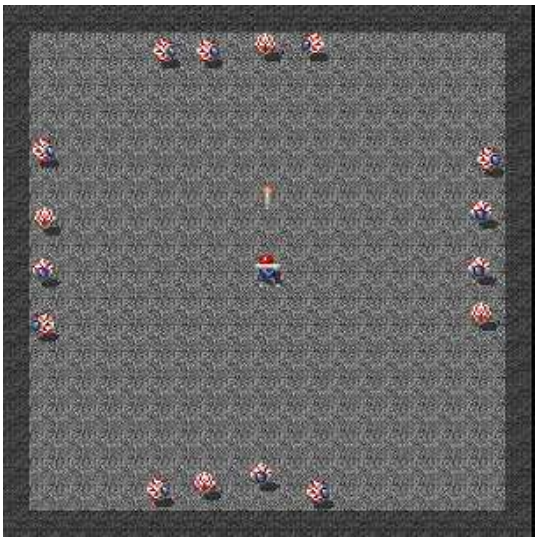

Figura 5.5: Posição inicial do jogo “Olhos Assassinos”

que assumem determinados valores em função dessa distância, descritos na tabela 5.6. Sendo assim, os agentes percebem as posições dos alvos em três graus discretos distintos. Ou seja, para eles um alvo pode apenas estar “perto”, “médio” ou “longe”. Como há três alvos (já mencionados) e três graus de distância, há um total de $3^3 = 27$ combinações alvo-distância.

Distância	**Valor**
$0 \leq d < 150$	Perto
$150 \leq d < 300$	Médio
$d \geq 300$	Longe

Tabela 5.6: Valores de acordo com a distância

Essas combinações formam, justamente, regras da forma `se` $\texttt{alvo}_1$ `está a uma distância` $\texttt{d}_1$ `e` $\texttt{alvo}_2$ `está a uma distância` $\texttt{d}_2$ `e` $\texttt{alvo}_3$ `está a uma distância` $\texttt{d}_3$ `então faça algo`, sendo os alvos o jogador, o cartucho e o tiro mais próximo. Muito embora isso se assemelhe muito a uma regra nebulosa, neste caso não é, embora nada impeça que elas sejam usadas.

O código genético dos agentes, portanto, contém a informação para que uma ação seja tomada de acordo com cada uma das 27 combinações de regras. Se levarmos em conta que há dois algoritmos básicos (fugir e perseguir) e três alvos, há seis ações possíveis de serem tomadas. Vamos, todavia, eliminar as possibilidades de

o agente perseguir um tiro (seria suicídio), fugir do jogador (seria contraditório) e fugir de um cartucho (não faz o menor sentido). Restam três ações: perseguir o jogador, perseguir o cartucho e fugir do tiro. Adicionaremos uma outra a essas que é o movimento aleatório. Há, portanto, quatro conseqüências para as regras, que podem ser codificadas em dois bits. Como são 27 regras ao todo, o código genético dos olhos tem $27 \times 2 = 54$ bits.

A função de facilidade ef do jogo é calculada a cada dois segundos e é dada por:

$$ef = \frac{r + d}{2} \tag{5.3}$$

Onde r é razão entre a quantidade de agentes mortos nesse intervalo de tempo e o número de tiros disparados, com $r = 1$ se nenhum tiro foi disparado (para evitar divisão por zero), e $d = 0$ se o jogador perdeu uma vida durante esse intervalo de tempo, com $d = 1$ caso contrário. Então, por exemplo, para 3 agentes mortos com 4 tiros, sem que o jogador tenha perdido uma vida, temos, por (5.3), $r = 3 \ / \ 4 = 0,75$ e, portanto, $ef = (0,75 + 1) \ / \ 2 = 0,875$. Se, entretanto, o jogador tivesse perdido uma vida durante esse tempo, teríamos $ef = (0,75 + 0) \ / \ 2 = 0,375$.

É fácil perceber que $0 \leq ef \leq 1$ e que 0 indica a situação mais difícil, ao passo que 1, a mais fácil. Embora seja uma função muito simples e esteja sujeita a imperfeições, correndo o risco de estimar de forma errônea o nível de facilidade atual do jogo, ela tem a vantagem de ser extremamente fácil de ser calculada e, por isso, é praticamente desprezível o tempo gasto com ela. O limite de facilidade usado foi $l_f = 0,6$ e, portanto, sempre que $ef \leq 0,6$ os agentes não evoluem mais, com exceção do caso do TTL que, por sua vez, foi definido como $TTL = 5$ (em segundos).

A função de avaliação f dos agentes é dada por:

$$f = tl + dist + d \times K \tag{5.4}$$

Onde tl é o total de segundos que o agente viveu, $dist$ é o quão longe ele conseguiu chegar em relação a sua posição inicial, d tem o signficado semelhante ao usado em (5.3) e K é uma constante que, no caso, foi usada como $K = 30$. Essa função, obviamente, premia os olhos que se movimentam mais, que vivem por mais tempo e, acima de tudo, que conseguem tirar um vida do jogador. A definição de f dada por (5.4) foi usada tanto para a parte em tempo real (para o critério de escolha dos

pais na reprodução) como na evolução "off-line". As primeiras gerações eram sempre formadas por indivíduos cujas regras eram todas preenchidas pela ação aleatória. A tabela 5.7 descreve os principais parâmetros genéticos usados.

Parâmetro	**Valor**	**Parâmetro**	**Valor**
% de elitismo	10	Cruzamento	Um ponto
% de aleatórios	10	Probabilidade de Mutação	0,1%
% de cruzamentos	80	Probabilidade de cruzamento	85%

Tabela 5.7: Parâmetros genéticos usados

5.3.2 Implementação e Resultados

Foram testadas todas as variantes, descritas nas seções 4.1.8, 4.1.9, 4.1.10 e 4.1.11. Foram usadas algumas formas simples de inteligência para simular o jogador e obter os indivíduos objetivos, porém não houve diferença significativa entre eles. A idéia principal era simular um comportamento defensivo (matar os agentes que chegassem perto) e buscar os cartuchos. No caso da determinação empírica (primeira variante), foi determinado que os objetivos finais de cada subpopulação eram, respectivamente, perseguir sempre o jogador e perseguir o cartucho.

A primeira variante foi implementada tanto usando distância de Hamming como cruzamento como forma de evolução. Já para a segunda foi apenas usado o cruzamento. Não foi testado o uso de dados colhidos pela Internet para determinar os indivíduos objetivos, visto que além de não dispormos de infra-estrutura necessária para tal, também não era necessário já que os resultados não iam diferir muito dos objetivos obtidos empiricamente e por uso de agentes no lugar do jogador, dada a simplicidade das estratégias possíveis. A quarta variante foi usada aproveitando os indivíduos obtidos pela segunda e, sendo híbrida, ela começava pela segunda e prosseguia pela terceira.

Não foram usados indivíduos objetivos intermediários para os testes da primeira variante e, obviamente, para a terceira, sendo usados apenas na segunda e quarta a qual, na verdade, iniciava pela segunda, como já citado anteriormente. Nos casos em que eles foram aproveitados, definiram-se quatro indivíduos intermediários.

A tabela 5.8 mostra os resultados obtidos pelo método de coevolução em tempo real, em suas variantes testadas, no jogo "Olhos Assassinos". OI indica a quantidade

de indivíduos objetivos utilizados (nenhum significa que apenas o indivíduo objetivo final foi usado), T_i representa quantos segundos, em média, foram jogados antes de o jogador perder a i-ésima vida, AM é a média de agentes mortos e FM é a facilidade média.

Variante	**OI**	T_1	T_2	T_3	**AM**	**FM**
1 (Hamming)	Nenhum	75,7	101,5	124,3	144,5	0,8055
1 (Cruzamento)	Nenhum	45,8	96,9	116,9	149,1	0,7905
2 (Cruzamento)	4	84,1	106,8	130,3	146,5	0,8199
3	Nenhum	36,2	51,3	60,5	80,1	0,7968
4 (2 + 3)	4	79,4	102,9	126,2	143,2	0,8076

Tabela 5.8: Resultados do método de coevolução em tempo real

Esses dados foram obtidos pela contribuição de 25 jogadores. Nem todos os dados, porém, puderam ser aproveitados. Muitos, por exemplo, estavam incompletos. Em alguns casos podíamos notar, pelo arquivo de *log* gerado, que o jogador tinha simplesmente abandonado o jogo após perder uma vida, reiniciando-o. Em outros casos, após perder uma vida, as demais eram perdidas logo em seguida. Outro problema que aconteceu foi que muitos jogadores perdiam vidas muito rápido quando começavam a jogar, por não estarem acostumados com os comandos nem com o jogo em si. Como o objetivo era estudar a evolução dos agentes dentro de um mesmo jogo e já que eles não têm qualquer memória entre um jogo e outro, foram descartados esses resultados e foram apenas consideradas as médias das melhores partidas de cada jogador.

Uma outra questão quanto aos resultados é que o valor absoluto dos dados obtidos não tem a menor importância, afinal, saber que os jogadores perderam a primeira vida, em média, com $75,7$ segundos de jogo para a primeira variante não nos diz muita coisa. Um resultado mais interessante, neste caso, é a diferença entre os valores subseqüentes dos T_i que mostram, justamente, a progressão do tempo durante o qual o jogador ficou sem perder vidas. A tabela 5.9 apresenta exatamente esse cálculo.

As figuras 5.6, 5.7 e 5.8 mostram alguns dos comportamento de diversos níveis de evolução dos agentes. A figura 5.6 mostra uma situação ainda inicial de jogo em que há alguns olhos cercando um cartucho e poucos indo em direção ao jogador. Boa parte ainda está com movimentos quase sempre aleatórios. A explosões que podem ser vistas são apenas um efeito do jogo para quando um tiro destrói um

Variante	T_1	T_2 - T_1	T_3 - T_2
1 (Hamming)	75,7	25,8	22,8
1 (Cruzamento)	45,8	51,1	32,2
2 (Cruzamento)	84,1	22,7	23,5
3	36,2	15,1	9,2
4 (2 + 3)	79,4	23,5	23,3

Tabela 5.9: Diferenças entre os tempos de perda de vida

Figura 5.6: Agentes pouco evoluídos

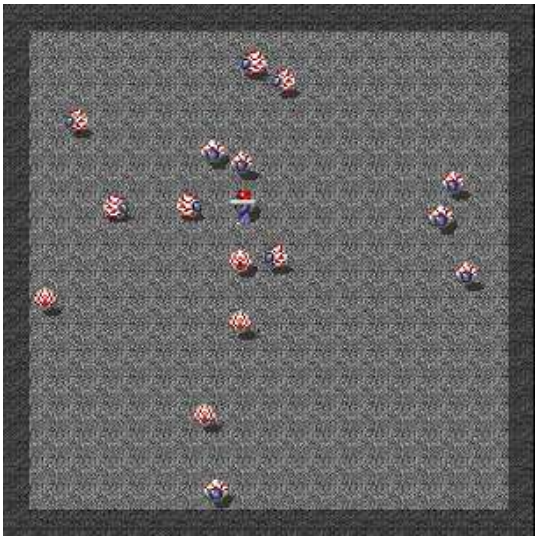

Figura 5.7: Agentes cercando o jogador

olho. A figura 5.7 já mostra os agentes mais evoluídos e cercando o jogador, ficando bem mais difícil conseguir sair de tal situação. Já a figura 5.8 mostra um estágio avançado de evolução no qual boa parte dos agentes cerca o cartucho para evitar que o jogador o pegue, enquanto que a outra parte persegue o jogador na tentativa de matá-lo.

Com esse tipo de comportamento (metade dos agentes tentando cercar o cartucho e a outra metade perseguindo o jogador), fica realmente bastante difícil sobreviver por muito tempo. Na verdade, o grande problema é que, enquanto se tenta matar os olhos que estão chegando perto, os demais estão cercando o cartucho. Chega um momento em que os tiros começam a acabar e conseguir passar pelos agentes que estão cercando a munição torna-se quase impossível, por mais reflexos que se tenha.

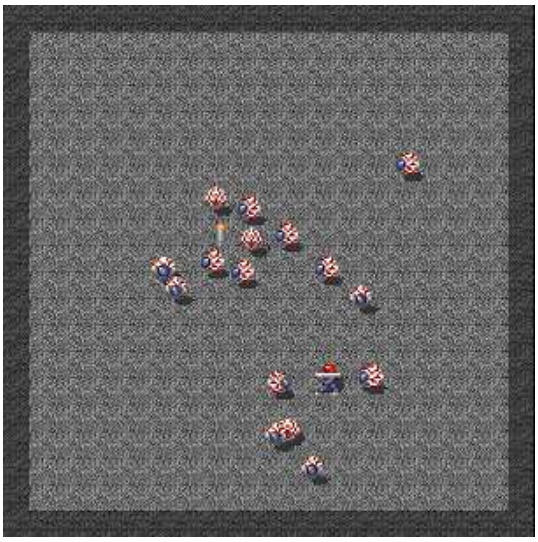

Figura 5.8: Estágio avançado de evolução dos agentes

5.3.3 Análise e Discussão

Pelos resultados apresentados na tabela 5.9, podemos perceber que há uma tendência a diminuir o tempo entre as perdas de vidas do jogador nos métodos. A primeira variante com cruzamento foi a única em que a diferença entre a perda da segunda vida foi maior do que a da primeira. Por outro lado, a perda da primeira vida foi muito mais rápida que com o uso da distância de Hamming. Isso indica uma evolução inicial mais veloz e, pela tabela 5.8 é possível notar que, realmente, o cruzamento obteve uma evolução mais rápida que a distância de Hamming, levando-se em conta os dados da primeira variante.

A segunda variante mostrou uma variação muito pequena entre os tempos de perda da segunda vida e da terceira. Teve, por outro lado, o maior valor de T_1 entre os testes realizados, o mesmo acontecendo com T_3 e, portanto, a maior média de tempo total de jogo, assim como a média de facilidade. O fato de usar os indivíduos objetivos intermediários contribuiu bastante para isso, uma vez que ajudou a diminuir e controlar mais a velocidade de evolução dos agentes. O fato de os valores de $T_2 - T1$ e $T_3 - T_2$ serem praticamente iguais sugere que a perda da primeira vida aconteceu num estágio mais avançado de evolução, com as perdas subseqüentes acontecendo com poucas mudanças em relação aos agentes, ou pelo menos mudanças que não influenciaram significativamente o resultado.

A terceira variante foi a que obteve, claramente, os menores tempos de perdas de

vida, de jogo e de diferenças entre perdas de vida. O que se observou foi que, sendo o espaço de busca relativamente pequeno para o problema, além de haver diversas "soluções" que implicassem uma dificuldade grande para o jogador (por exemplo, há várias combinações possíveis que fazem com que os agentes persigam o jogador) e havendo uma reposição considerável de agentes (média de 80 mortos por partida mais aqueles evoluídos por causa do TTL), os olhos tendiam a perseguir o jogador muito mais rapidamente do que nos demais testes. Verificou-se, também, que não houve muita possibilidade de as subpopulações alcançarem uma cooperação como a determinada para os objetivos da primeira variante. Embora não tenha sido possível analisar todos os casos individualmente, pois para boa parte só dispúnhamos dos arquivos de *log* gerados com os resultados (ou seja, não pudemos testemunhar as partidas sendo jogadas), nos casos em que foi possível testemunhar o que acontecia, notamos que dificilmente se verificava uma organização bem dividida entre as subpopulações, e a tendência dos agentes era se dirigirem ao jogador diretamente.

Os dados obtidos na evolução usada para determinar os agentes da segunda variante, embora não fossem exatamente os mesmos definidos empiricamente, seguiram mais ou menos a mesma tendência, dividindo uma subpopulação para proteger o cartucho e outra para perseguir o jogador. Já no caso da coevolução em tempo real isso não se verificou, como acabamos de observar, apesar de ser uma implementação muito semelhante. Como a grande diferença era o agente que simulava o comportamento do jogador, o disparate é provavelmente causado por essa influência. De fato, como o agente implementado tem uma agilidade muito superior à de um ser humano, para ele tende a ser mais fácil se livrar dos olhos que o perseguem sem, no entanto, ter de se preocupar com o cartucho sendo guardado. Quando há um grupo cercando o cartucho, além de ter de lidar com os que o perseguem, o jogador também tem de matar aqueles que impedem que ele se aproxime da munição, dificultando as coisas. No caso do jogador humano, porém, o fato de ter um número razoável de agentes o perseguindo já pode ser o suficiente para determinar sua derrota. Essa questão nos faz lembrar do que foi discutido na seção 4.6 a respeito do erro artificial. Implementando um agente que simulasse o jogador inclusive com um erro determinado de tiro e também com uma forma de "segurar" sua destreza, provavelmente teríamos chegado a resultados bem mais próximos, no caso pré-determinado, àquele obtido em tempo real.

No caso da quarta variante, os resultados mostram pouca coisa. Assim como

no caso da segunda, os valores de $T_2 - T_1$ e $T_3 - T_2$ são praticamente iguais. Isso não é nem um pouco surpreendente, pelo contrário, é plenamente esperado, visto que a quarta variante é justamente a segunda seguida pela terceira. A influência da terceira, por seu lado, é quase imperceptível, uma vez que os jogadores já encontram bastantes dificuldades com os últimos indivíduos objetivos, de forma que as evoluções subseqüentes pouco influenciam no resultado final.

Como avaliação geral, a primeira variante, apesar de sua simplicidade, mostrou que pode desempenhar um excelente papel em situações para as quais, de alguma forma, temos uma boa informação a respeito das estratégias, mesmo que empiricamente, ou até se for possível deduzi-las de alguma maneira. Muitos jogos têm "níveis de dificuldade", como "fácil", "médio", "difícil" etc; o uso dessa variante pode tornar a transição entre esses níveis de dificuldade muito mais suave e voltada ao desempenho do jogador.

Já a segunda variante pode ser bastante últil nos casos em que há alguma impossibilidade de determinar empiricamente os indivíduos objetivos, podendo aproveitar dados obtidos através de simulações realizadas com tempo suficiente para abreviar essa evolução, permitindo que a mesma aconteça em tempo real e, como dito acima, dependente da habilidade do jogador. O uso de dados obtidos em servidores pela Internet, com a possibilidade de vários jogadores serem responsáveis por essa evolução a ser aproveitada na versão em tempo real, é especialmente tentadora, porém requer maiores recursos e talvez não seja possível contar com eles, como foi o caso deste trabalho.

A terceira variante, por sua vez, é mais aplicável a casos em que haja muita liberdade na escolha de estratégias, podendo adaptar-se melhor a mudanças imprevisíveis. É importante, porém, que haja uma grande vazão de agentes, ou seja, uma reposição constante, de forma a evitar, a qualquer custo, os efeitos observados na seção 5.1. Como observado em (AGOGINO, STANLEY, MIIKKULAINEN, 2000), quando o grau de incerteza tende a ser muito grande com respeito a estratégias e deve-se encontrar qual se sai melhor em determinado contexto desconhecido, usar o coevolução pura em tempo real pode ter resultados até melhores do que o contrário. O fato de o agente criado para simular o jogador ter induzido uma estratégia entre as subpopulações diferente da encontrada em tempo real é um indicador disso. Afinal, apesar de menos refinada, a coevolução em tempo real encontrou uma estratégia que funcionava bem melhor de acordo com as características do adversário, menos

ágil do que no caso do agente simulado.

A quarta variante, finalmente, acabou mostrando pouca diferença em relação à segunda, pelos motivos já relacionados. Sua escolha, apesar disso, é especialmente tentadora em casos em que haja algum conhecimento prévio, ainda que preliminar, a respeito de estratégias, ou mesmo quando há alguma forma básica de comportamento que se deva alcançar antes de passar para outras mais refinadas, permitindo que, após isso, a coevolução possa ocorrer de forma livre. Apressa-se, assim, os passos iniciais evolutivos que teriam de ser desempenhados pelos agentes, fazendo com que a busca de estratégias mais elaboradas torne-se, a princípio, menos trabalhosa. Para casos em que as melhores estratégias já são razoavelmente bem conhecidas e previsíveis, como foi o caso deste jogo, seu uso certamente vai ter pouca influência, uma vez que o primeiro passo já vai cuidar de alcançar os mais altos níveis de evolução.

5.3.4 Distância de Hamming *versus* Cruzamento

Mesmo após a análise dos resultados obtidos e a discussão a respeito dos mesmos, consideramos que vale a pena discutir, mesmo que brevemente, alguns aspectos a respeito da diferença do uso de distância de Hamming e cruzamento que foram observados durante a implementação do método.

É interessante, antes de mais nada, descrever o principal problema que tivemos inicialmente no uso da distância de Hamming. Nos primeiros testes do jogo, usando a primeira variante com essa forma de evoluir os agentes, misteriosamente apenas uma subpopulação parecia evoluir. Na verdade, os olhos que deveriam perseguir o jogador chegavam a esse estado evolutivo, enquanto que os que deveriam cercar o cartucho, em geral, tinham comportamentos diversos ou, até mesmo, tendiam a perseguir o jogador.

Após de algum tempo de *debug*, tanto no código do jogo em si quanto no que tratava especificamente da implementação do método, conseguimos detectar o que estava causando aquilo. Como imaginávamos que fosse algo sério, algum problema no código que tratava dos indivíduos objetivos ou até mesmo alguma falha inerente do método, fomos surpreendidos ao finalmente perceber que o vilão da história era a forma como os bits estavam definidos para representar as ações das regras. Resumindo o caso, a subpopulação que deveria perseguir o cartucho tinha uma distância de Hamming entre os indivíduos iniciais e o objetivo final duas vezes maior que a dos que tinham de perseguir o jogador que, conseqüentemente, tinha uma

evolução duas vezes mais rápida. Ao trocar a definição dos bits das regras, fazendo com que ambos os tipos de agente tivessem a mesma quantidade de bits diferentes dos objetivos, a variante do método passou a funcionar normalmente.

Esse tipo de preocupação, portanto, é extremamente importante. Dependendo da forma como forem codificados os genes dos indivíduos, o uso da distância de Hamming pode ficar comprometido ou, simplesmente, não funcionar.

Por outro lado, um problema observado com o cruzamento é o fato de a evolução em direção ao objetivo acontecer de maneira muito rápida. Por exemplo, usando o cruzamento de um ponto, a chance de se mudar apenas 1 bit em relação ao objetivo (que é o quanto se muda com a distância de Hamming) é de apenas 1 / n, onde n é o total de bits. Claro que essa chance leva em conta que todos os bits dos genes entre os indivíduos são diferentes, o que provavelmente não será verdade. Mesmo assim, com esse valor aumentando um pouco, ainda vai ser muito menor que no caso da distância de Hamming, que sempre muda um único bit com probabilidade 1,0.

Claro que esse efeito pode ser exatamente o que se deseja obter, enquanto que, em outros casos, pode não ser. Algumas formas de tentar contornar essa questão, se for um problema, é usar outras formas de cruzamento. Há diversas maneiras diferentes de se fazer isso, plenamente disponíveis na literatura do assunto, e mesmo algumas simples como cruzamento de dois pontos ou usando uma máscara podem ajudar a solucionar ou, pelo menos, atenuar essa influência. Além disso, quando se usam muitos indivíduos objetivos intermediários, a tendência é que essa questão simplesmente passe desapercebida, afinal, assim há um controle bem maior da evolução em direção ao objetivo final.

É difícil, de qualquer maneira, poder precisar qual forma de evolução é melhor. Dependendo do caso, uma pode funcionar melhor que a outra. Há diversas formas diferentes de fazer o cruzamento e o uso da distância de Hamming funciona como mais uma opção. Então, de acordo com a aplicação em questão e o uso que é feito do método, uma determinada escolha pode ter um peso maior sobre a outra e, assim, prevalecer como opção final.

5.4 Simulador de RPG

Para poder testar o comportamento do método de padrões adaptativos, apresentado na seção 4.2, foi desenvolvido um programa que simulava um jogo simples de RPG.

A escolha baseou-se no fato de esse tipo de jogo apresentar uma forma natural de desenvolvimento em níveis de experiência, bem como apresentar componentes aleatórias que são um fator importante para simular o comportamento às vezes imprevisível dos jogadores e suas reações aos elementos do jogo.

5.4.1 Descrição do Jogo

O personagem principal do jogo é um aventureiro, que representa o que seria jogador humano, o qual tem como objetivo principal passar por diversos perigos numa caverna e resgatar tesouros. Durante sua jornada, ele vai ganhando experiência devido aos combates travados contra diversos tipos de monstros e às riquezas encontradas. Além disso, em alguns pontos, ele pode encontrar poções mágicas que lhe restituem energia.

O aventureiro tem algumas características básicas que o descrevem. São elas: energia (hp), ataque (ob), defesa (db), dano (dmg), experiência (xp) e nível (lvl). A energia é um valor que indica sua saúde. Cada vez que ele sofre um ataque, um valor é subtraído desse total e, quando o mesmo chega a zero, ele morre. O ataque indica sua capacidade de acertar os adversários, enquanto que a defesa indica justamente sua capacidade de proteger de ataques. O dano representa a quantidade de dano que ele inflige a um inimigo se o acertar. Os pontos de experiência indicam quanta experiência foi acumulada durante sua aventura e, dependendo desse total, avança-se ou não de nível. A tabela 5.10 mostra os valores iniciais para cada nível dessas características. No caso dos pontos de experiência, a tabela indica quantos são necessários obter para alcançar o respectivo nível.

Característica/Nível	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Energia (hp)	10	20	30	50	75	100	250	500
Ataque (ob)	25	40	50	75	100	125	150	200
Defesa (db)	5	15	25	40	60	80	90	100
Dano (dmg)	1-4	1-6	2-8	3-12	5-20	5-30	10-40	10-60
Pontos de Experiência (xp)	0	500	1000	1500	2500	5000	35000	75000

Tabela 5.10: Valores iniciais das características por nível do jogador

O combate acontece da seguinte maneira: primeiro o jogador ataca, depois os monstros atacam, e assim sucessivamente. Quando um ataque é feito, toma-se o ob e subtrai-se do db do adversário. O resultado obtido é a probabilidade de acerto.

Sorteia-se, então, um número entre 1 e 100 e se este for menor ou igual ao valor $ob - db$ calculado, então o atacante acertou. A quantidade de dano é calculada e subtraída do total de energia do adversário. Quando a energia fica menor ou igual a zero, o personagem morre. Se for o jogador a morrer, a partida termina. Se for um monstro, o jogador ganha pontos de experiência relativos ao adversário derrotado.

Os monstros contra os quais o jogador pode lutar são goblin, zumbi, orc, ogro, troll, aranha gigante e dragão. A tabela 5.11 mostra as características dessas criaturas, sendo que o valor de pontos de experiência indica quanto o jogador recebe, caso derrote o respectivo adversário. O aventureiro só pode lutar contra um tipo de monstro por vez, mas podem aparecer até três criaturas iguais ao mesmo tempo, com exceção do dragão, que sempre aparece sozinho.

Característica/Monstro	**Goblin**	**Zumbi**	**Orc**	**Ogro**	**Troll**	**Aranha**	**Dragão**
Energia (hp)	5	10	25	40	50	75	250
Ataque (ob)	20	25	35	50	75	110	150
Defesa (db)	0	5	10	20	30	60	100
Dano (dmg)	1-4	1-4	2-8	2-12	3-12	5-30	10-60
Pontos de Experiência (xp)	10	15	25	35	50	75	250

Tabela 5.11: Características dos monstros

Além de monstros, como já foi dito, podem aparecer, também, poções e tesouros. Há dois tipos de poção: uma recupera toda a energia do jogador até o máximo, enquanto que a outra restaura apenas 50% do máximo. Então, se o aventureiro estiver no segundo nível, com $hp = 6$, e encontrar uma poção do primeiro tipo fica com $hp = 20$ e, do segundo, com $hp = 6 + (20 \times 0.5) = 16$. Obviamente que não se pode ficar com hp maior que o máximo do nível e, portanto, se no caso fosse $hp = 13$, a hp iria apenas até 20, e não 23. Quanto aos tesouros, há três tipos do mesmo que valem, respectivamente, 10, 50 e 100 pontos de experiência.

5.4.2 Modelando os Padrões

Dada a descrição do jogo, descreveremos, a patir de agora, como a simulação foi modelada de forma a implementar o método de padrões adaptativos e testar seu comportamento.

Os padrões são, naturalmente, monstros, tesouros ou poções. São sete tipos de criaturas que, com exceção do dragão, como já exposto, podem aparecer em grupos

de 1 a 3. São, portanto, $3 \times 6 + 1 = 19$ padrões contando com os monstros. Há, além disso, três tipos de tesouro e dois de poção e, assim, há um total de 24 padrões possíveis.

Uma seqüência, neste contexto, representa uma série de encontros do aventureiro, sejam esses com adversários, tesouros ou poções mágicas, e seria o equivalente a uma fase de um jogo comum. Ao terminar uma seqüência, completando-a ou morrendo no meio, os cálculos de $f(.)$ são feitos e um novo conjunto X gerado. A seguir, começa uma nova seqüência para o aventureiro. O valor de hp sempre inicia com o máximo do seu nível, enquanto que os demais valores continuam iguais aos que estavam antes de terminada a seqüência anterior (mesmo que o jogador tenha morrido). Após cada padrão completado, o jogo testa se os pontos de experiência são suficientes para passar de nível e, em caso positivo, os valores das características são atualizados de acordo. Não são somados pontos ao xp de padrões completados parcialmente (ou seja, sem que todos os adversários tenham sido derrotados).

Falta ainda descrever a função $f(.)$ de avaliação dos padrões. Antes de mais nada, podemos observar alguns fatos. É obviamente mais fácil enfrentar um dragão depois de ter combatido dois goblins do que três aranhas gigantes, o que mostra que, realmente, a ordem com que os padrões de monstros aparecem tem influência na dificuldade da fase. Por sua vez, é muito mais fácil lutar contra um dragão após encontrar uma poção mágica do que depois de lutar contra qualquer adversário, de maneira que os padrões de poção também têm uma influência grande na ordem. No caso do padrão do tesouro, não há muita influência com relação ao padrão seguinte. Afinal, se o aventureiro lutou contra três aranhas gigantes, depois encontrou um tesouro de $100xp$ e, finalmente, encontra um dragão, a não ser que ele tenha passado de nível nesse meio tempo, o efeito na dificuldade, para a ordem dos padrões, é idêntico a como seria se não houvesse o tesouro entre as aranhas e o dragão.

Ainda temos de lembrar que há uma dificuldade absoluta de cada padrão. Afinal, é bem mais difícil lutar contra um dragão do que contra um goblin, bem como não há dificuldade nenhuma em encontrar uma poção ou tesouro.

Essas observações nos levam à conclusão de que é necessário utilizar uma função de avaliação do padrão propriamente dito, bem como uma que o avalie de acordo com aquele que o antecedeu. Adicionemos a estas avaliações uma que verifique o todo da seqüência, ou seja, que avalie de forma geral como aqueles padrões em conjunto apresentaram dificuldades ao jogador. A avaliação, neste caso, ocorre em

relação à seqüência como um todo e não a um padrão em particular.

Como referência, consideraremos o valor nulo (zero) como sendo a avaliação de uma fase de dificuldade média, ou seja, nem fácil nem difícil e, a partir deste referencial, é que determinamos os valores das funções e as constantes usadas. Quanto mais acima de zero estiver a avaliação, mais difícil será a seqüência, enquanto que o contrário se verifica, quanto mais abaixo de zero estiver.

Para avaliar a dificuldade absoluta que um determinado padrão apresentou, aplicamos a função $\phi(.)$, definida para este jogo da seguinte forma:

$$\phi(w_i^j) = (hpa - hpd) + (K \times mt) - xpt \tag{5.5}$$

Onde *hpa* e *hpd* indicam a quantidade de energia (*hp*) do jogador antes de iniciar o padrão e depois de terminado, respectivamente. Ou seja, quanto mais *hp* for perdida no padrão, maior vai ser o valor adicionado à sua dificuldade. A variável *mt* vale 1 se o aventureiro morreu naquele padrão e 0 caso contrário. Finalmente, K é uma constante (foi usado $k = 25$) e *xpt* indica o quanto de *xp* foi ganha sem a necessidade de luta. É fácil verificar que, no caso dos monstros teremos $\phi(w_i^j) = (hpa - hpd) + (K \times mt)$, para os tesouros $\phi(w_i^j) = -xpt$ e, paras as poções, $\phi(w_i^j) = 0$.

Quanto mais energia for perdida num padrão, mais difícil o mesmo será avaliado pela função (5.5). Se, além disso, o aventureiro ainda for morto, ainda maior será essa dificuldade avaliada. No caso da poção, a função retornará sempre zero, indicando que, tomada sozinha, ela não é fácil nem difícil. No caso do tesouro, o retorno será um valor negativo proporcional à quantidade de *xp* obtida.

Para avaliar a dificuldade de um padrão em relação ao anterior, aplicamos a função $\xi(.)$, definida para este jogo da seguinte forma:

$$\xi_{w_i^{j-1}}(w_i^j) = K \times (mt + 0,7 - \frac{hpa}{hp}) \tag{5.6}$$

Onde *hpa*, *mt* e K têm o mesmo significado já mencionado, enquanto que *hp* indica o valor máximo de *hp* possível para o nível do aventureiro. Uma observação importante: a função (5.6) somente se aplica a padrões de monstros, sendo sempre nula nos demais casos.

A avaliação retornada por (5.6) tende a considerar mais fácil o padrão atual, em relação ao que o antecedeu, quanto mais próximo da energia máxima o aventureiro

estiver e mais difícil quanto mais perto de zero estiver. Se, por exemplo, a *hp* estiver em seu valor máximo e o aventureiro não morrer, teremos $\xi_{w_i^{j-1}}(w_i^j) = K \times (0 + 0,7 - 1) = -0,3K = -7,5$ (para $K = 25$).

Para poder avaliar a seqüência como um todo, usamos uma função $g(.)$ aplicada diretamente a w_i, definida da sequinte maneira:

$$g(w_i) = C \times (1,0 - \frac{percorrido}{total}) - (K \times (1 - mt)) \tag{5.7}$$

Onde K é o mesmo das outras fórmulas e mt tem significado semelhante, aplicando-se, desta vez, à seqüência e não a um padrão apenas. O termo *percorrido / total* indica a proporção da seqüência que foi completada, ou seja, a razão entre os padrões terminados sobre o total. Finalmente, C é uma outra constante que, neste caso, foi usada como $C = 200$. É fácil verificar que, caso o aventureiro tenha passado por todos os padrões, o primeiro termo da fórmula será nulo e, portanto, teremos $g(w_i) = -K = -25$. Se isto não acontecer, $g(.)$ será positiva, considerando a fase mais difícil quanto menos padrões do total tiverem sido superados.

Juntando as fórmulas (5.5), (5.6) e (5.7), podemos definir, afinal, a função $f(.)$ de avaliação de uma seqüência w_i como sendo:

$$f(w_i) = \sum_{j=1}^{m} \phi(w_i^j) + \sum_{j=1}^{m} \xi_{w_i^{j-1}}(w_i^j) + g(w_i) \tag{5.8}$$

Lembrando que m é o tamanho da seqüência w_i, ou seja, quantos padrões ela têm. Observe-se que não utilizamos a função $\varphi(.)$ e, além disso, não definimos pesos, o que equivale a dizer que consideramos todas as funções usadas com pesos iguais a 1.

O limite superior não foi utilizado, e definiu-se $d_{inf} = -50$. Sendo assim, consideramos que uma avaliação em torno -50 representa uma fase fácil, em torno de 0, média (como já foi citado) e em torno de 50, difícil. Podemos dizer que de -100 para baixo é muito fácil e de 100 para cima muito difícil.

Para escolher a próxima seqüência a ser utilizada, após o cálculo de todos os $f(w_i)$ e a ordenação do conjunto X, escolhia-se aquela que estivesse mais próxima de d_{inf}.

Quando uma determinada função sobre um padrão ($\phi(.)$ ou $\xi(.)$) era calculada para um padrão, o valor guardado na memória era igual à média entre o valor já existente para aquele padrão e o novo. Assim, por exemplo, se um certo padrão w_i^j

tem gravado $\phi(w_i^j) = a$ e um novo valor b acaba de ser calculado, então o novo valor a ser guardado será $(a + b)$ / 2.

Outro detalhe importante é que todos os padrões tiveram seus valores de $\phi(.)$ iniciados com 10 e de $\xi(.)$ com zero. Isso significa, de certa forma, que um padrão é um pouco difícil até que se prove o contrário.

5.4.3 Resultados e Análise

Nesta seção apresentamos os resultados obtidos nas simulações realizadas, bem como uma análise dos dados obtidos e do comportamento do método de padrões adaptativos.

Vários tamanhos distintos de seqüências foram usados, e chegou-se a um total de dez padrões por fase para as simulações finais e um total de 30 seqüências. Ou seja, $\|W\| = 30$, $m = 10$ e lembrando que $\|P\| = 24$, como já foi visto. As seqüências foram criadas uma a uma, sem o uso de métodos automáticos, como permutação ou ordenação topológica.

Durante as simulações, arquivos de *log* enormes foram gerados, detalhando informações como o resultado de cada combate passo a passo (todos os ataques, danos calculados etc.), o cálculo da dificuldade dos padrões e das seqüências, a progressão de níveis do aventureiro etc. Através desses dados pudemos observar bem o comportamento do método e também verificar possíveis problemas com o programa simulador em si (busca por *bugs*) e verificar o desempenho do jogador em cada fase.

As variáveis aproveitadas aqui para resultados e sua análise foram a avaliação da seqüência atual (ou seja, $f(.)$ aplicada à fase que acabara de ser jogada), a porcentagem de padrões completados pelo jogador e o nível do mesmo. Assim, a cada partida realizada, guardamos o resultado de $f(.)$ aplicado à fase jogada, quantos padrões da mesma foram completados e o nível do aventureiro ao fim da mesma.

A figura 5.9 mostra o resultado médio de $f(.)$ por partida em 100 simulações distintas, cada qual com um total de 150 partidas realizadas. Os dados equivalem, portanto, à média de $f(.)$ para 100 jogadores distintos simulados, cada qual jogando 150 vezes seguidas.

Percebe-se, claramente, uma tendência de a dificuldade se estabilizar em torno do zero. Essa característica "dentada" do gráfico, com altos e baixos, porém, indica que o jogo alterna constantemente períodos em que é mais fácil e mais difícil. Isso é plenamente coerente, uma vez que o à medida que o aventureiro vai passando de

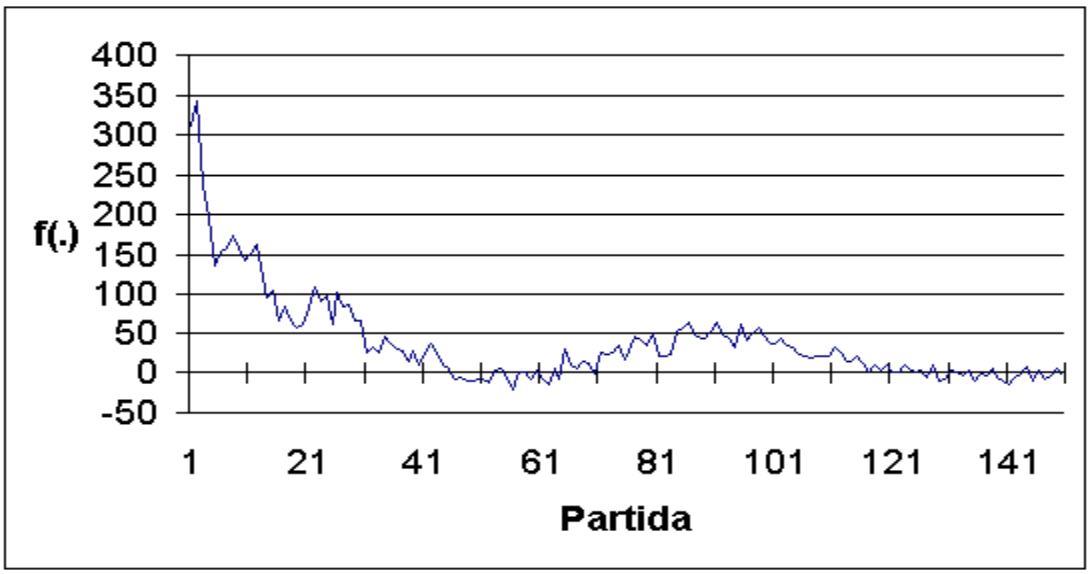

Figura 5.9: Valor médio de $f(.)$ por partida realizada

nível, seus adversários vão ficando mais fáceis, mas, então, a fase modifica-se de forma a apresentar novamente um desafio a ele. Sempre que uma determinada fase fica fácil demais, o método escolhe uma outra que seja mais difícil para ser jogada, e essa marcante característica pode ser bem observada com os dados da figura 5.9.

Para auxiliar na discussão dos dados da figura 5.9, podemos observar, também, outros valores medidos. As figuras 5.10 e 5.11 mostram, respectivamente, a média de padrões que foram passados pelo aventureiro, por partida, e o nível de experiência médio do mesmo, em cada partida. Percebe-se, pela figura 5.10, que perto da 80^a partida já há uma tendência a se estabilizar em 100% dos padrões vencidos, e depois da 100^a esse valor realmente se estabiliza. Verificando na figura 5.11 essas duas marcas, vemos que o nível de experiência está, respectivamente, entre 6 e 7 e próximo de 7. Ou seja, quando atinge em média 100% de padrões vencidos por seqüência, o jogador ainda não está em seu nível máximo de experiência (8). Ora, observando a figura 5.9, vemos que os valores de $f(.)$ perto da 80^a e 100^a partidas ainda estão acima de zero, embora abaixo de 50, o que indica que ainda há uma certa dificuldade enfrentada pelo aventureiro. Em outras palavras, mesmo terminando as fases, o jogo ainda não ficou fácil para o jogador, mantendo um nível razoável de desafio, o que é um resultado bastante interessante. De fato, o jogo tende a estabilizar em zero, ou um pouco abaixo, depois de 120^a partida, quando o nível médio de experiência

já está muito perto do máximo.

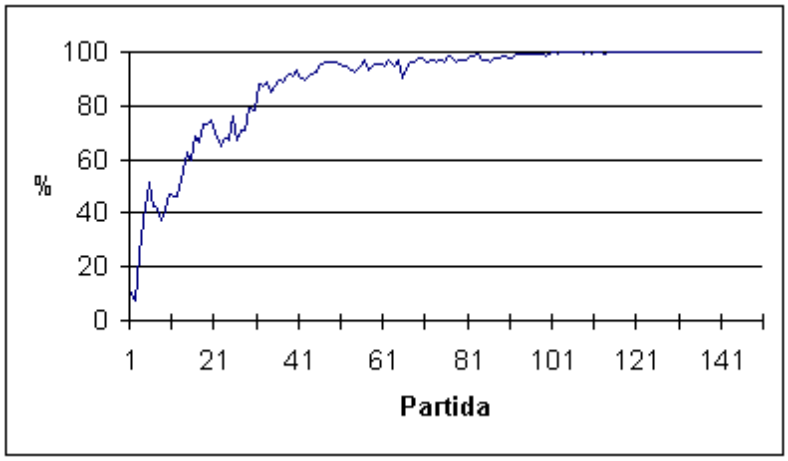

Figura 5.10: Porcentagem média de padrões passados por partida realizada

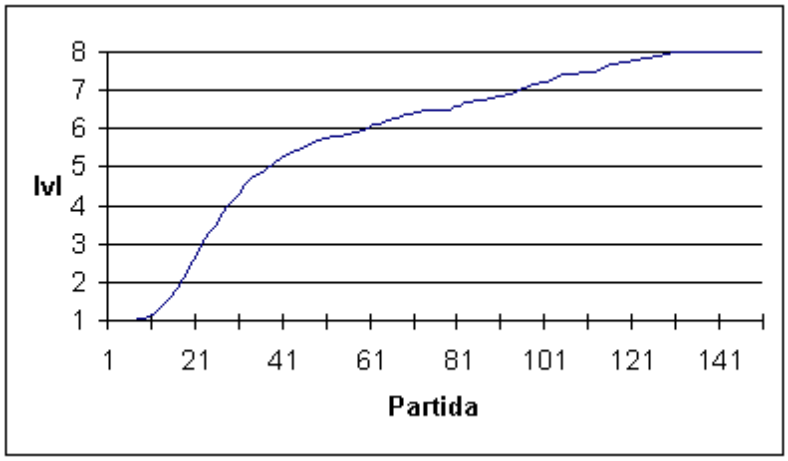

Figura 5.11: Nível de experiência médio por partida realizada

A forma amortizada da figura 5.9 sugere um desenvolvimento interessante da dificuldade do jogo, que, à medida que se torna mais fácil (porque o jogador ganhou experiência), se adapta à nova situação, dificultando novamente as partidas. No final das contas, o jogador tende a ganhar esse "cabo de guerra", mas não de uma forma bem direta, sem que haja "resistência" por parte do jogo, o que constitui um resultado bastante interessante e animador, mostrando que os objetivos da aplicação do método foram alcançados.

Já os gráficos das figuras 5.10 e 5.11 sugerem uma forma logarítmica, o que também é coerente, dado que a passagem dos níveis de experiência do aventureiro tem uma tendência exponencial, o que pode ser verificado pela tabela 5.10. Ainda assim a figura 5.10 mostra oscilações, o que é natural que ocorra, uma vez que há um constante aumento na dificuldade global do jogo, e estas oscilações acompanham aquelas de dificuldade demonstradas pela figura 5.9. A média dos níveis de experiência não apresenta esses "altos e baixos", o que é plenamente previsível, visto que é impossível, dentro do contexto deste jogo, um jogador perder experiência e "cair" de nível.

Depois de obter esses resultados, resolvemos estender o teste do método modificando a simulação de forma a comportar duas novas "raças" de aventureiros. Consideramos que os testes feitos e apresentados até este momento usaram sempre a raça humana. As demais raças são elfo e anão.

Os elfos são quase iguais aos homens, mas como são seres mágicos, eles têm uma grande vantagem: avançam de nível mais rápido. Dessa maneira, eles precisam de metade dos pontos de experiência que um ser humano normal precisa para passar para o mesmo nível. Logo, a única mudança é que os valores para xp na tabela 5.10 ficam divididos por dois. A inclusão dos elfos no teste tem o objetivo de verificar o comportamento do método com jogadores que aprendem mais rápido que o normal, para os quais, portanto, o jogo tende a ficar fácil mais rapidamente.

Os anões, por sua vez, têm as mesmas características dos homens. A diferença é que eles têm mais dificuldades em lutar contra zumbis do que orcs e contra ogros do que trolls. Isso significa trocar, na tabela 5.11, as características de zumbis com orcs e de ogros com trolls. O objetivo da inclusão dos anões é verificar o comportamento do método com jogadores que têm dificuldades diferentes, uma vez que, para anões, orcs são mais fáceis do que zumbis e trolls são mais fáceis do que ogros, enquanto que para os homens é o contrário.

As simulações realizadas para elfos e anões seguiram exatamente o mesmo esquema usado para os homens, fazendo-se apenas as mudanças citadas. Foram 100 simulações, cada qual com 150 partidas realizadas e os mesmos dados apresentados para os homens foram recolhidos para análise e discussão.

A figura 5.12 mostra o gráfico da média de $f(.)$ por partida realizada para os elfos. É possível observar o comportamento oscilatório verificado na figura 5.9. Comparando as duas, contudo, podemos verificar que o comportamento entre a 70^a

e 120^a partidas na figura 5.9 é "comprimido" entre as partidas 40 e 70 na figura 5.12. Dessa forma, a $f(.)$ tende a se estabilizar com valores um pouco abaixo de zero a partir, mais ou menos, da partida 70, em torno de 50 partidas mais cedo do que no caso dos humanos. O que observamos é que, para os elfos, a oscilação inicial tende a ser mais acentuada e mais breve, estabilizando perto do zero bem mais cedo. O efeito disso é natural, uma vez que, aprendendo mais rápido, as partidas tendem a ficar fáceis rapidamente, acentuando as quedas e, também, as subidas de dificuldade. Sim, porque tendo essa rapidez em aprender, o valor de $f(.)$ para os padrões tende a diminuir depressa, o que acaba fazendo com que fases que nem foram jogadas fiquem com avaliações baixas, o que resulta em seqüências mais difíceis sendo usadas bem antes do que o normal.

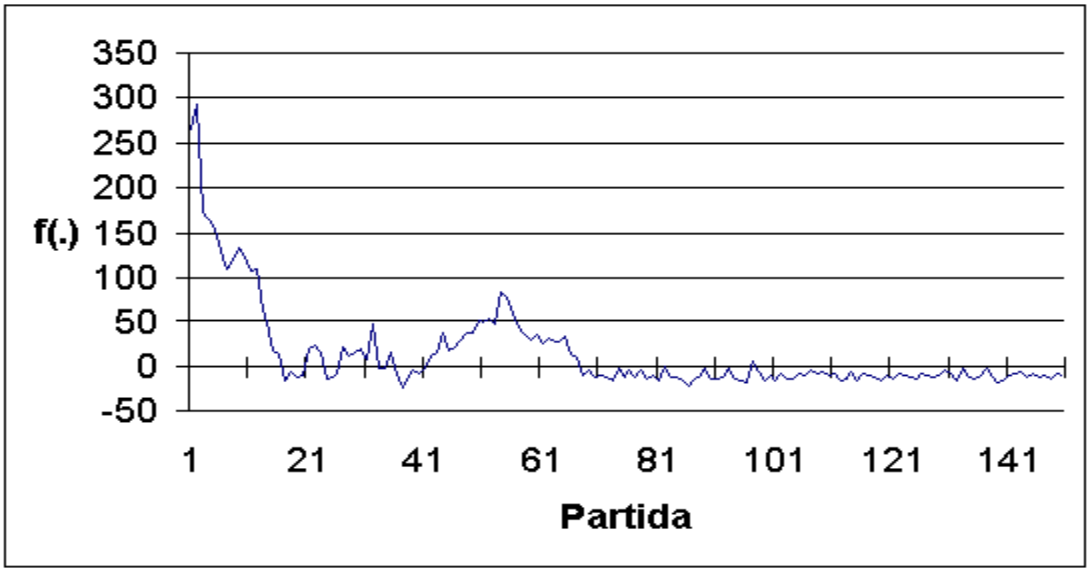

Figura 5.12: Valor médio de $f(.)$ por partida realizada para elfos

A figura 5.13 mostra o resultado da média de $f(.)$ por partida realizada para as simulações com os anões. Apesar de algumas pequenas diferenças, a forma geral do gráfico é extremamente parecida com a do gráfico dos homens da figura 5.9. As maiores oscilações ocorrem quase que nos mesmos pontos. Uma diferença que pode ser notada é que, no caso dos anões, a tendência a estabilizar um pouco abaixo de zero começa a se verificar por volta da 130^a partida, cerca de 10 mais tarde do que com os humanos, o que não chega a ser muito significativo. De fato, ao comparar ambos os gráficos, as diferenças são tão pequenas que até parecem apenas duas

simulações diferentes das mesmas configurações. É óbvio, porém, que se tivéssemos trocado o dragão com o goblin, por exemplo, certamente o resultado seria muito diferente e pouco aproveitável. Mas isso não chega a ser realmente um problema, porque dificilmente, dentro de um jogo ou contexto semelhante, padrões tão opostos terão esse tipo de resposta do jogador. Mesmo se isso ocorrer, uma solução seria criar um conjunto $\|W\|$ maior, uma vez que quanto maior a quantidade de seqüências, mais possibilidades estaremos cobrindo e, portanto, maior será a chance de o método se adaptar bem a esse tipo de situação esdrúxula. Deve-se lembrar, entretanto, que quanto maior for $\|W\|$, mais tempo de computação será necessário.

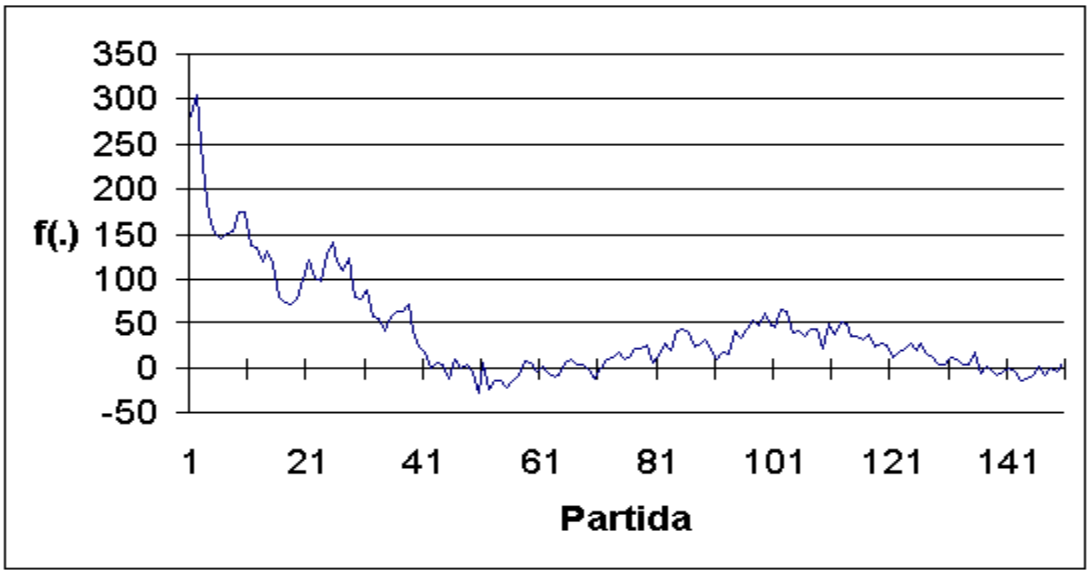

Figura 5.13: Valor médio de $f(.)$ por partida realizada para anões

De uma maneira geral, o fato de os resultados apresentados pelas figuras 5.9 e 5.13 terem sido tão parecidos demonstra que o método se saiu bem com indivíduos que têm facilidades e dificuldades distintas, conseguindo se adaptar bem aos dois casos testados.

Os demais resultados (porcentagem de padrões completados e nível de experiência médios) de ambas as raças (elfos e anões) apresentam as mesmas características da função $f(.)$ já analisadas em cada caso, com os elfos obtendo maior rapidez para apresentar o mesmo comportamento dos homens e com os anões apresentando as mesmas tendências, com diferenças mínimas ou pouco significativas.

Os resultados obtidos com as simulações demonstram o bom aproveitamento

obtido pelo método e um comportamento animador do mesmo. Apesar de terem sido realizados sem a interação humana, os testes foram feitos de forma a simular, da melhor maneira possível, o ganho de experiência de jogadores, bem como as diferentes situações que podem ser encaradas com variados níveis de dificuldade.

Os testes realizados com elfos e anões, para verificar o comportamento do método com diferentes tipos de jogadores, mostra que ele consegue se adaptar bem a essas situações, permitindo uma certa generalização.

As oscilações de dificuldade apresentadas são uma clara demonstração do método agindo e o jogador, por sua vez, reagindo. As mudanças dos níveis de dificuldade em um contexto dinâmico, como é o caso de jogos por causa do aprendizado natural do jogador, requerem um método que se adapte bem à situação para avaliar qual a melhor opção a ser apresentada como desafio. O método de padrões adaptativos, numa avaliação geral, saiu-se muito bem nas simulações realizadas e mostrou-se uma boa alternativa para aplicações que necessitem desse refinamento e gradação de níveis de dificuldade.

5.5 Hiper Mario

Para observar o comportamento do método de padrões adaptativos, descrito na seção 4.2 e testado numa simulação como mostrado na seção 5.4, foi criado o jogo "Hiper Mario", baseado na famosa série *Super Mario Bros* da fabricante japonesa Nintendo.

Assim como os jogos originais que o inspiraram, o "Hiper Mario" é do gênero plataforma e o principal requisito para o sucesso do jogador é sua perícia. Como, em muitos casos, é preciso que se repita diversas vezes as jogadas para que se consiga aprender a realizar satisfatoriamente os movimentos necessários a fim de vencer os obstáculos, o jogo torna-se especialmente convidadtivo para a aplicação do método. Ainda há, além disso, a influência da seqüência de elementos dentro de uma mesma fase na dificuldade final da mesma, o que também sugere que o método dos padrões adaptativos possa ser bem aproveitado.

O jogador controla um personagem (Mario) que deve vencer inimigos, pisando nos mesmos, e coletar moedas, além de se alimentar de cogumelos, ficando maior e mais poderoso. Em seu caminho para completar a fase, o jogador deve passar por diversos obstáculos além de inimigos, como poços de lava, estacas mortais, abismos

e canos, além de poder encontrar passagens secretas e ter de atravessar verdadeiros labirintos.

Um padrão, neste contexto, é uma série de elementos de jogo agrupados, como blocos, inimigos, moedas etc., de uma determinada maneira. O conjunto desses objetos e a forma como eles são agrupados dentro de um padrão podem determinar o quão difícil ele será e, ao mesmo tempo, facilitar ou dificultar a passagem pelo padrão seguinte.

A seqüência desses padrões constitui uma fase e o objetivo final do jogador, neste caso, é completá-la com vida. Há duas formas que o personagem pode assumir: o Mario "fraco" e o Mario "forte". O jogador sempre inicia no estado fraco e, ao pegar um cogumelo, ele automaticamente passa para o forte (se ele já estiver no forte, nada acontece). Se for tocado por um inimigo e estiver forte, ele volta ao estado fraco, e se já estiver no fraco, morre. Se cair num poço de lava ou for atingido por estacas mortais, ele morre não importa em que estado esteja. Ao perder uma vida, o jogo reinicia do início da fase (possivelmente com outra seqüência). O jogador tem, de início, três vidas e, ao perder todas elas, o jogo se encerra.

A figura 5.14 mostra uma tela de jogo que poderia, inclusive, representar um padrão. Neste caso os padrões podem ter tamanho variável, não necessariamente todos têm a mesma largura ou quantidade de elementos.

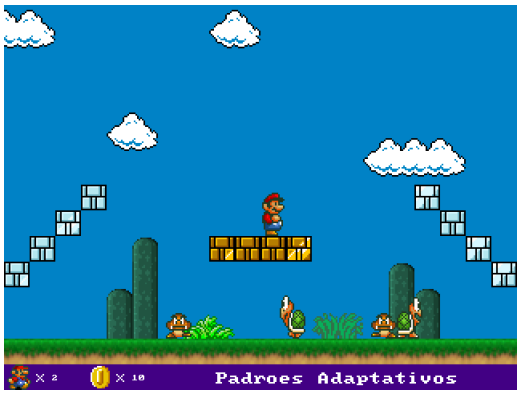

Figura 5.14: Tela do jogo "Hiper Mario"

Uma dificuldade que temos aqui e que não se verificava no caso do simulador de RPG é que, dependendo da forma como forem criados, determinados padrões podem não se "encaixar" em seqüência. Um exemplo simples seria um padrão em que só haja um caminho pelo qual o jogador deve caminhar e, no padrão seguinte,

justamente aquela região por onde ele deveria passar está bloqueada (ou seja, não há possibilidade de se passar de um para outro). Diversas outras situações desse tipo e até mais complicadas podem ocorrer, de maneira que é necessário ter cuidado redobrado na hora de criar as seqüências.

5.5.1 Modelando os Padrões

Foram criados 15 padrões diferentes e 100 seqüências de tamanhos variáveis (desde 3 até 12 padrões). As figuras 5.15, 5.16, 5.17 e 5.18 mostram quatro exemplos de trechos de padrões do jogo (que não aparecem por inteiro na tela).

Figura 5.15: Exemplo de Padrão no "Hiper Mario"

Figura 5.16: Exemplo de Padrão no "Hiper Mario"

Figura 5.17: Exemplo de Padrão no "Hiper Mario"

Figura 5.18: Exemplo de Padrão no "Hiper Mario"

Para gerar as seqüências foram usadas formas diferentes. Algumas foram colocadas diretamente no conjunto, enquanto que outras foram geradas através de

ordenações topológicas.

As avaliações foram feitas baseadas no padrão individualmente (ϕ) e em relação ao padrão anterior (ξ), além de uma função g, que avalia a seqüência como um todo. Alguns padrões também tinham uma pré-avaliação *default*, apenas para separar alguns considerados os mais difíceis (quatro).

De maneira geral, portanto, as definições das avaliações foram semelhantes às usadas para o caso do simulador de RPG visto na seção 5.4. E, assim como no caso anterior, as atualizações dos valores nas funções são feitas usando a média entre o novo resultado calculado e o antigo.

Dessa forma, a função de avaliação f dos padrões para o caso do Hiper Mario foi definida como:

$$f(w_i) = \sum_{j=1}^{m} \phi(w_i^j) + \sum_{j=1}^{m} \xi_{w_i^{j-1}}(w_i^j) + g(w_i) \tag{5.9}$$

E as respectivas funções que a formam foram definidas como:

$$\phi(w_i^j) = K_1 \times mt + \frac{t - 1000}{K_2} - K_3 \times cog \tag{5.10}$$

$$\xi_{w_i^{j-1}}(w_i^j) = -tipo \times K_3 \tag{5.11}$$

$$g(w_i) = K_4 \times (1,0 - \frac{percorrido}{total}) \tag{5.12}$$

Onde mt indica quantas vezes o jogador foi golpeado por inimigos, t indica o tempo gasto no padrão (medido em quadros de jogo), cog é o número de cogumelos pegos no padrão, $tipo$ vale 0 para o Mario fraco e 1 para o Mario forte (após ter pego um cogumelo) e $percorrido$ / $total$ é a proporção da seqüência que foi completada. Os K são constantes, cujos valores usados foram $K_1 = 25$, $K_2 = 50$, $K_3 = 15$ e $K_4 = 200$.

O objetivo das funções é avaliar como sendo difícil um padrão em que o jogador tenha dificuldades com inimigos (morrendo ou perdendo energia) e no qual gaste bastante tempo para passar pelo padrão. Por outro lado, se o jogador demora pouco tempo, consegue pegar cogumelos que lhe dão energia e passa sem ter dificuldades contra inimigos, então o padrão é considerado mais fácil. Os limites foram definidos como $d_{inf} = -25$ e $d_{sup} = +\infty$, com a próxima seqüência w_i a ser escolhida sendo a de menor f tal que $f(w_i) > d_{inf}$.

Ao contrário, porém, do simulador de RPG, neste caso os valores de f que indicam uma facilidade média ficam um pouco acima do zero. Valores negativos indicam seqüências fáceis, e valores de 20 para cima indicam difíceis.

5.5.2 Resultados e Avaliação

Nos testes de comportamento do método implementado no jogo Hiper Mario, realizado com dez voluntários, percebeu-se um comportamento mais errático, com maiores "saltos" de dificuldade, como pode ser visto na figura 5.19, que apresenta a média dos valores obtidos para f ao longo das jogadas.

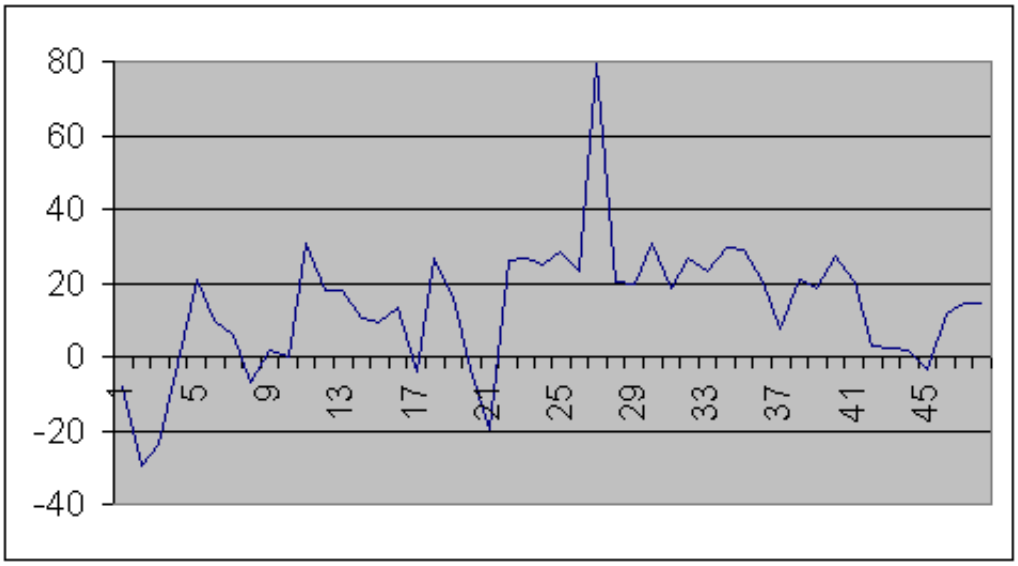

Figura 5.19: Resultados do Hiper Mario

Podemos notar um pico perto da 25ª jogada, o que pode representar um erro de estimativa do algoritmo ou mesmo uma jogada muito infeliz de um jogador, forçando uma avaliação alta de dificuldade para a seqüência. Claro que testes com poucas fontes tendem a sofrer mais com esse tipo de resultado.

Apesar disso, o mais interessante dos testes realizados foi a alternância apresentada, em todos os casos, entre fases fáceis e difíceis. O fato, inclusive, de haver uma rotatividade entre as seqüências, dificulta que o jogador consiga, por repetição exaustiva, passar com facilidade pela fase.

Um outro dado interessante é que, em média, foram jogadas 28 seqüências diferentes (das 100 possíveis) por cada jogador, em cerca de 50 jogadas cada um (ou

seja, 22 foram iguais). O que dá, aproximadamente, 1 fase nova a cada 2 jogadas. Por outro lado, considerando os 10 jogadores, tivemos 78 seqüências diferentes jogadas ao menos uma vez. A figura 5.20 mostra a quantidade de vezes que cada uma das 100 seqüências foi jogada.

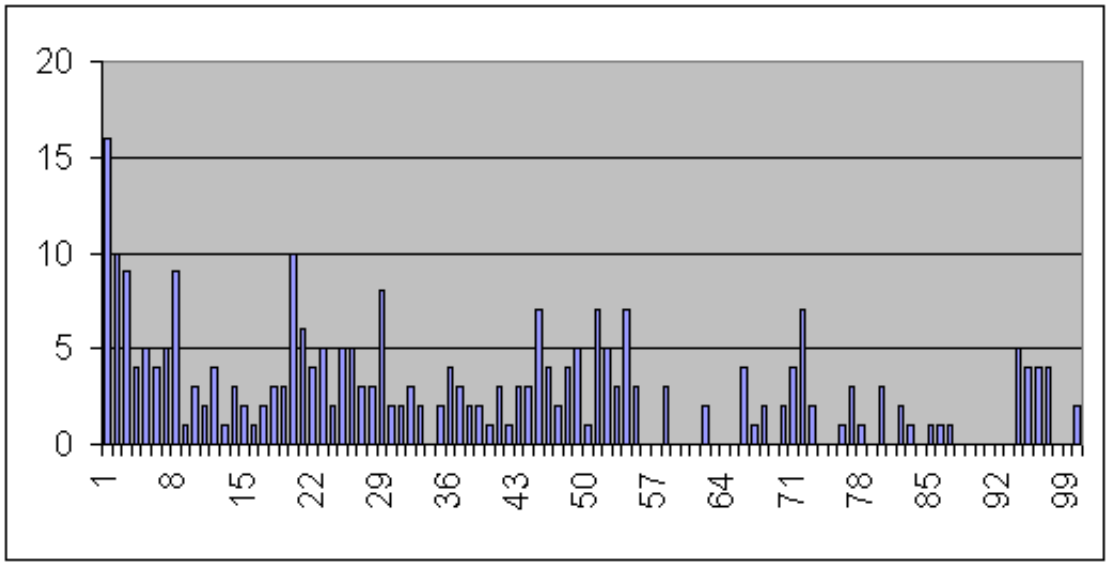

Figura 5.20: Seqüências Jogadas do Hiper Mario

Cada jogador apresentou uma ordem distinta de seqüências jogadas, e algumas que foram jogadas por um nem foram consideradas para outros, por terem sido avaliadas muito fáceis ou muito difíceis de acordo com a estimativa dada pela f.

O que foi possível determinar pela aplicação do método num jogo razoavelmente complexo como este, foi que a dificuldade maior reside, como era de se esperar, na determinação satisfatória de f, e as demais funções que a compõem, para avaliação dos padrões.

O comportamento do método, como um todo, foi considerado bastante promissor, não só por conseguir, satisfatoriamente, alternar entre diferentes seqüências fáceis e difíceis, como, também, por ter sido extremamente eficiente (não houve qualquer mudança na taxa de quadros por segundo do jogo), satisfazendo a principal restrição (de tempo) para sua aplicação.

Certamente mais testes devem ser feitos, e em diversas áreas para, inclusive, averiguar a utilização do método em diferentes formas de aplicação. As simulações realizadas, entretanto, nesta seção e na seção 5.4 mostram que o comportamento do

método é bastante satisfatório e indicam que pode ser muito bem aproveitado em aplicações que necessitam de adaptação condicionada e progressiva em tempo real, como é o caso aqui explorado, dos jogos eletrônicos.

5.6 Cebolinha, Labirinto e ALife

A implementação das máquinas de estado nebulosas foi aproveitada em conjunto com a navegação em uma série de testes, descritos neste capítulo.

O primeiro deles é um ambiente em 2D no qual uma partícula tem de se movimentar sem colidir com as paredes. Ela dispõe de cinco sensores que, na interface do programa, são traçados na tela, como mostra a figura 5.21. Justamente pelo aspecto da partícula e seus sensores é que lhe foi dado o apelido de "Cebolinha", já que a mesma lembra o famoso personagem de quadrinhos.

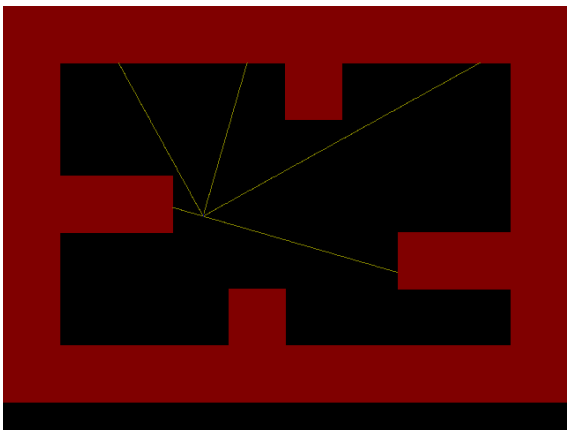

Figura 5.21: Tela do "Cebolinha"

A máquina de estados nebulosa criada foi simples, com três estados, representando seus movimentos (frente, esquerda e direita) e apenas quatro transições. O grau de pertinência num determinado estado representa o quanto, num passo de jogo, aquele estado contribui no movimento total da partícula. Assim, se ela pode se mover até 5 pontos e girar até 10^o por turno, estar $0,5$ no estado "frente", $0,35$ no "direita" e $0,15$ no "esquerda" significa que serão dados $0,5 \times 5 = 2,5$ passos para frente, além de girar $0,35 \times 10 - 0,15 \times 10 = 3,5 - 1,5 = 2,0$ graus para a direita. Claro que, na representação discreta do monitor só é possível andar passos completos, de maneira que esses valores acabam sendo arredondados.

Essa FuSM, extremamente simples, foi capaz de simular o comportamento de seguir paredes satisfatoriamente, evitando que a partícula colidisse.

O passo seguinte foi aplicar a máquina de estados dentro de um simulador mais complexo, o Labirinto, simulando um ambiente em 3D, desenvolvido com a técnica de *Ray Casting* (LAMOTHE et al, 1994; PERMADI, 2003), mostrado na figura 5.22. Nele, os sensores foram substituídos por um processamento simples do campo de visão do agente. Na verdade ele reconhecia apenas a "quantidade de chão" adiante dele, isto é, quanto havia de chão, em cada coluna de sua visão, antes de haver um obstáculo. Os dois sensores laterais da partícula foram substituídos por uma informação do jogo que permitia ao agente "sentir" os obstáculos laterais muito próximos, mesmo fora do campo de visão. Os sensores diagonais e o dianteiro foram substituídos pelo processamento, já descrito, da imagem.

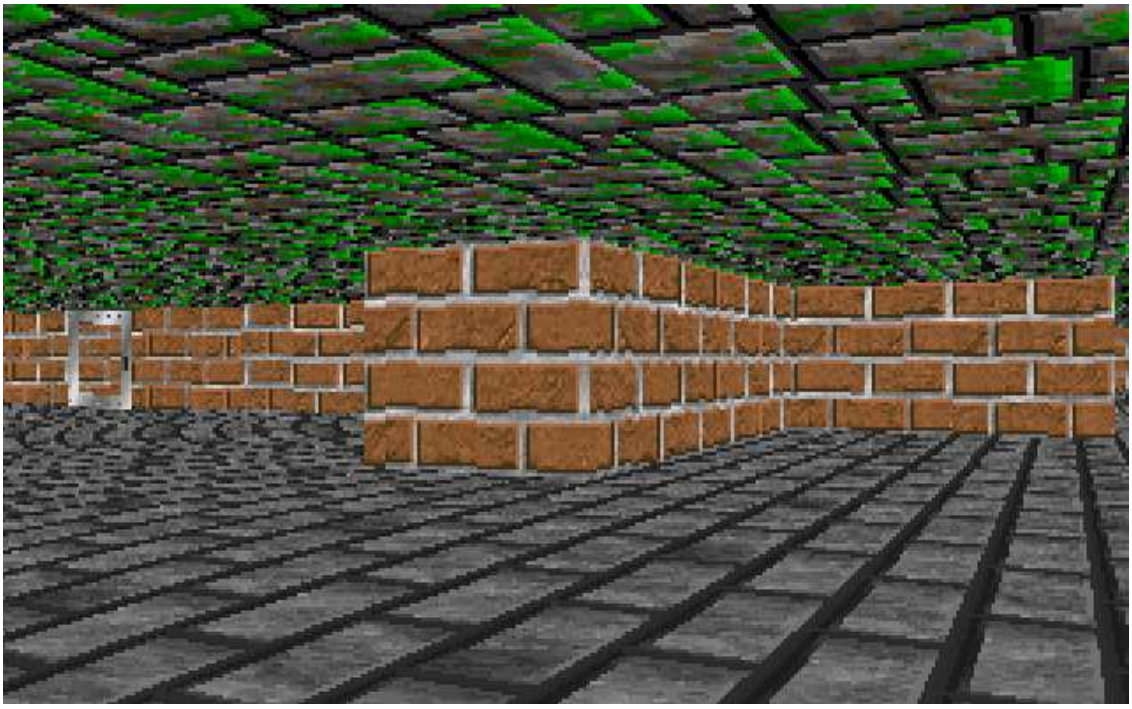

Figura 5.22: Tela do *Ray Casting*

A exemplo do caso do Cebolinha, no Labirinto o agente também conseguiu se movimentar satisfatoriamente com a FuSM simples implementada, sem colidir, simulando um comportamento de seguir a parede.

Todo esse desenvolvimento foi, finalmente, implementado num simulador de

robôs, entrando num concurso chamado *ALife*. Neste torneio, os robôs devem competir entre si dentro de um ambiente, dois de cada vez. Cada um tem uma bateria cuja carga é decrementada ao longo do tempo. No lugar onde se encontram, com vários obstáculos, há quatro carregadores e, uma vez que um robô fique em frente ao mesmo, ele o descarrega e recarrega sua bateria. Após certo tempo, os recarregadores voltam ficar disponíveis.

Cada robô dispõe de um câmera com resolução de 80×60 pixels e que representa um ângulo de visão de, aproximadamente, 70^o. Além disso, há oito sensores dispostos a sua volta que detectam obstáculos muito próximos. A movimentação é feita aplicando-se rotação nas rodas direita e esquerda.

O objetivo é simplesmente sobreviver. Aquele que tiver sua energia esgotada primeiro, perde. As disputas são feitas em melhor de cinco partidas, vencendo aquele que ganhar três. A cada dia é disputado um *round*, e o resultado final do torneio sairá no dia 1^o de maio de 2003. A página na Internet do mesmo é http://cyberboticspc1.epfl.ch/contest/index.html.

O grande problema foi implementar o controlador do robô, o que tinha de ser feito em Java, requerendo uma certa tradução de todas as rotinas feitas em C.

Foram implementadas as técnicas descritas na seção 4.4, bem como a máquina de estados nebulosa usada nas simulações do Cebolinha e do Labirinto. Além disso, criou-se, hierarquicamente, uma FuSM, superior a essa, de dois estados, para determinar se o robô deveria explorar o ambiente ou voltar para algum carregador que ele já tivesse encontrado, usando a técnica de "João e Maria". As distâncias para a FuSM foram retiradas processando-se a imagem da câmera (como antes, "contando o chão"), para obstáculos distantes ou os sensores para aqueles muito pertos. Todas essas distâncias foram representadas, internamente, de forma nebulosa (ou seja, esses dados foram nebulizados).

O resultado final tem sido satisfatório. No dia 25 de março, o robô implementado (Minerva) ocupava a quinta posição entre 40 competidores (figura 5.23), sua melhor colocação. Esse resultado é expressivo, considerando que muitos controladores foram feitos por estudantes e pesquisadores de centros de robótica, que trabalham exclusivamente com esse tipo de problema. Ou seja, mesmo sendo um método simples de navegação proposto, ele se adequa bem a simulações realistas, o que nos leva a concluir que seu uso em jogos pode ser muito bem aproveitado. Além disso, sua simplicidade e eficiência permite que seja usado na situação do torneio, em que

os cálculos devem ser feitos rapidamente, já que a taxa de *refresh* do robô (sensores, câmera etc.) é da ordem de $64ms$. Sendo assim, se o processamento for muito demorado, o controlador perderá informação, o que certamente irá prejudicá-lo na competição. Além disso, essa também é uma amostra do potencial do uso das FuSMs, pois mesmo uma construção hierárquica simples é capaz de controlar satisfatoriamente o robô num ambiente de simulação complexo onde há, inclusive, competição com outro agente.

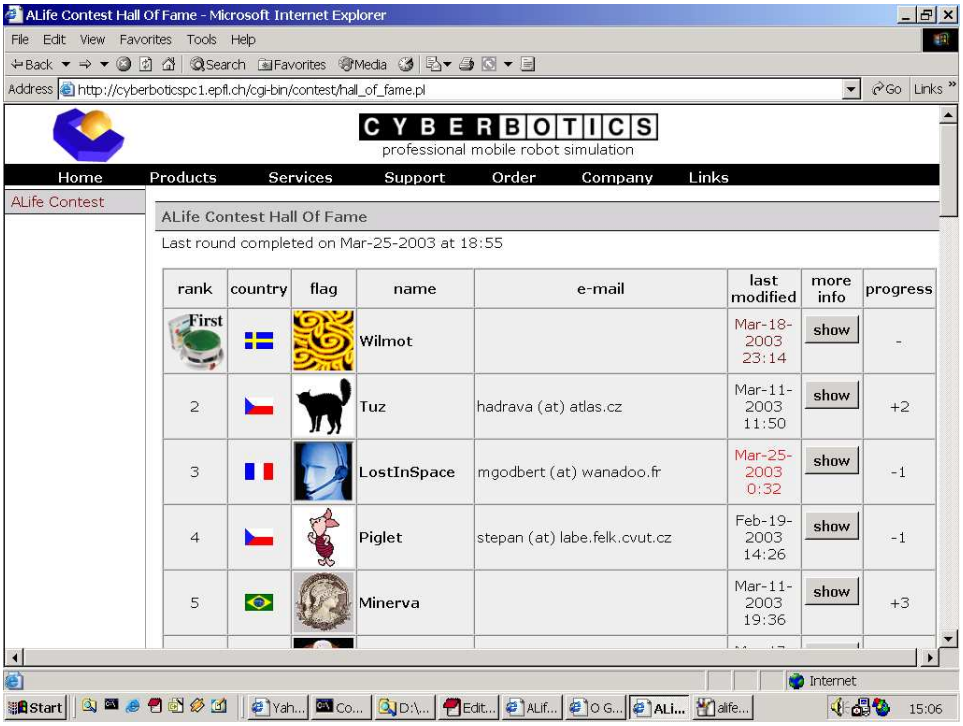

Figura 5.23: Tela do *Ranking* do Torneio *Alife*

5.7 I-Juca-Pirama

"E à noite nas tabas, se alguém duvidava / Do que ele contava, / Tornava prudente: Meninos, eu vi!"

Gonçalves Dias, *I-Juca-Pirama*

Para poder testar o comportamento das técnicas de previsão, bem como o método de aprendizado de regras nebulosas em tempo real no contexto de "tutor" (e não

apenas de previsor), foi criado o jogo "I-Juca-Pirama", cujo nome foi inspirado no poema épico de Gonçalves Dias que, em Tupi, significa "aquele que deve morrer".

Criou-se, inicialmente, um agente baseado exclusivamente em regras nebulosas para o jogo. Depois, alguns testes foram feitos para observar o comportamento dos métodos. Primeiro, em partidas normais entre agente e jogador, verificou-se a porcentagem de acerto de cada método de previsão em relação tanto ao jogador quanto ao agente. Além disso, com os dados dessa partida, mais um agente foi criado usando o método de aprendizado de regras nebulosas em tempo real.

5.7.1 Descricão do Jogo

O jogo é constituído por duas naves cujo objetivo é destruir a adversária. Cada uma possui três tipos de ação: movimento, ataque e defesa (os dois últimos são mutuamente exclusivos). As naves têm recursos limitados, estando estes associados às ações (cada ação implica um gasto do respectivo recurso): combustível, mísseis e escudo.

As naves ficam uma de frente para a outra (uma na parte inferior da tela, outra na superior). Só há duas direções de movimento possíveis: esquerda ou direita. O comando para a nave se mover funciona como um acelerador (ou freio), aumentando em 1 pixel a velocidade da nave por laço de jogo. Ou seja, no primeiro laço ela se movimenta 1 pixel, depois 2 pixels, 3 pixels etc. Cada pixel que a nave se movimenta implica uma perda proporcional do combustível. Esse tipo de movimento faz com que as naves tenham uma certa inércia, pois quando o jogador libera a tecla de movimentação, a nave não pára automaticamente (ela vai desacelerando, também à taxa de 1 pixel por laço de jogo). Acelerar na direção contrária ao movimento atual equivale a uma ação de freio.

O ataque é feito disparando-se mísseis, os quais têm velocidade vertical constante e horizontal nula. Ou seja, a nave de baixo atira para cima e vice-versa. Se um míssil colidir com a nave adversária, esta é destruída. Por outro lado, se os mísseis adversários se cruzarem, nenhum dos dois é destruído. Esta última característica do jogo foi assim implementada para que os mísseis não constituíssem, também, uma estratégia de defesa.

A defesa é feita acionando-se um escudo que fica posicionado em frente à nave. Se um míssil colidir com o escudo, ele é destruído e a nave nada sofre. Cada laço de jogo em que o escudo está ativo implica um gasto proporcional do recurso. As

laterais não ficam protegidas (ou seja, se a nave colidir "de lado" com um míssil, ela é destruída, mesmo com o escudo acionado).

Um elemento complicador foi adicionado ao jogo: duas paredes de fogo colocadas nas laterais da tela. Se as naves colidirem com elas (independente de estarem ou não com o escudo acionado), são destruídas. A figura 5.24 mostra um exemplo de tela do jogo.

Figura 5.24: Exemplo de tela do jogo "I-Juca-Pirama"

Neste exemplo, vemos as duas naves (uma de frente para a outra), as duas paredes de fogo laterais, 5 mísseis (3 lançados pela nave de cima, sendo que 1 já está quase fora da tela, e 2 pela de baixo), o escudo da nave de cima acionado e a barra inferior de informações. Esta última indica a quantidade de recursos restantes de cada jogador, além de informações como o tempo de jogo (esgotado o tempo máximo, a partida é considerada empatada; o *default* é de 60 segundos), o placar (vitórias, empates e derrotas do jogador) e a quantidade de ciclos de jogo e quadros por segundo. Não aparecem, no modo normal de jogo, as informações relativas ao computador, apenas a do próprio jogador (a tela mostrada na figura 5.24 é do modo de testes do sistema).

5.7.2 Modelagem do Agente

O sistema que controla a inteligência do jogo é composto, basicamente, por 8 variáveis de entrada (quantidade relativamente alta) e 2 de saída.

Algumas observações: nos conjuntos nebulosos a seguir, não se deve confundir expressões como "pouco perigo" com modificadores do tipo *hedge* (como *very*, *little* etc.) pois são apenas parte do nome do conjunto. Deve-se, também, observar que são consideradas positivas as posições à direita da nave e negativas as à esquerda (a nave é o referencial). Logo, -30 pixels, por exemplo, significa 30 pixels à esquerda da nave.

As variáveis de entrada dividem-se em 4 grupos (entre parênteses a quantidade de variáveis de cada um): adversário (2), fatores externos (1), perigo (2) e recursos (3).

As variáveis do tipo adversário são "delta x" e "delta v". A primeira indica a distância da nave adversária (em pixels) no eixo x, tendo 5 conjuntos (muito esquerda, esquerda, perto, direita e muito direita). A segunda indica a velocidade relativa da nave adversária, tendo também 5 conjuntos (distanciando rápido, distanciando lento, constante, aproximando lento, aproximando rápido).

A única variável do tipo fator externo é "parede de fogo", que indica quão perigosa é a posição da nave em relação a alguma parede de fogo (também 5 conjuntos: muito perigo à esquerda, perigo à esquerda, seguro, perigo à direita, muito perigo à direita).

As variáveis do tipo perigo são "defesa" e "esquiva". A primeira, que tem 3 conjuntos (pouco perigo, médio perigo, muito perigo), indica quão perigosa é a posição dos mísseis adversários (se estão muito próximos de colidir etc.). Já a segunda, que também tem 3 conjuntos (esquerda, centro, direita), indica uma posição segura para a nave esquivar (ou seja, que não colida com mísseis adversários).

Finalmente, as variáveis do tipo recursos são "mísseis", "escudo" e "combustível". Todas têm 3 conjuntos (pouco, médio, muito) e servem para indicar a quantidade restante dos respectivos recursos de que a nave dispõe.

As variáveis de saída são "ação" e "direção". A primeira tem apenas 3 conjuntos (defende, neutro, ataca) e serve para escolher a ação da nave. Há um nível de tolerância (que pode ser definido no código do jogo) acima do qual o resultado desnebulizado (*defuzzyfied*) do sistema indica um lançamento de míssil ou ativação do

escudo. A segunda variável, que tem 5 conjuntos (muito esquerda, pouco esquerda, centro, pouco direita e muito direita) serve para indicar qual a direção de movimento que a nave deve tomar: o valor desnebulizado indica quantos pixels a nave deveria se movimentar. Assim, se o resultado for, por exemplo, 17, significa que a melhor posição, no momento, para a nave, seria 17 pixels para a direita. Baseado na velocidade atual, o sistema escolhe acelerar (ir na mesma direção), frear (ir na direção contrária) ou, simplesmente, desacelerar (não ir em nenhuma direção).

Para evitar que o processamento fosse crítico, foram usadas apenas funções trapézio e triângulo para modelar os conjuntos nebulosos (em todas as variáveis), afinal, a eficiência dos métodos usados é extremamente importante, como já cansamos de frisar, em jogos de ação em tempo real.

Foi utilizado, a princípio, um conjunto que variou de 30 a 40 regras para descrever o comportamento da nave, número extremamente pequeno se considerarmos a quantidade de variáveis e conjuntos.

As regras, de uma forma geral, definiram um comportamento agressivo da nave, que ataca desde o início do jogo e persegue o adversário com o objetivo de decidir logo. Esse tipo de estratégia, é interessante notar, dá um excelente resultado nas primeiras partidas contra adversários humanos que estão "frios", que ainda não se acostumaram o suficiente com o jogo.

Outra característica é o "conservadorismo" em relação às paredes de fogo. A nave tende a usar uma faixa do espaço de jogo perto da metade do total da tela, evitando ao máximo se aproximar das paredes. Essas regras, inclusive, entram em conflito com a perseguição ao adversário. Como o "medo" de colisão, porém, é maior que a vontade de perseguição, geralmente chega um limite em que o agente espera o adversário sair de muito perto da parede para persegui-lo e atacá-lo. Algumas das regras que foram usadas no teste do sistema são dadas a seguir para ilustrar essas estratégias:

---

se parede de fogo é muito perigo à esquerda então direção é muito direita
se delta v é aproximando rápido e defesa é pouco perigo e míssil é muito então ação é ataca
se delta x é muito esquerda e delta v é constante então direção é muito esquerda
se defesa é muito perigo e escudo é muito então ação é defende
se defesa é muito perigo e escudo é pouco e esquiva é esquerda então direção é muito esquerda

---

O primeiro exemplo mostra o "medo" da nave em colidir com as paredes de fogo e representa as regras que se distanciam o máximo possível das mesmas. O segundo mostra a preocupação em atacar: se o adversário está se aproximando rápido e não há perigo (ou seja, não há necessidade de ativar escudos), então ele ataca. O terceiro mostra a perseguição, seguindo na direção do adversário. O quarto mostra uma ação defensiva acionando o escudo, quando este ainda não foi muito gasto e há muito perigo. O último exemplo, finalmente, mostra uma ação defensiva de esquiva, para o caso de haver muito perigo e de o escudo estar quase acabando.

5.7.3 Erro Artificial

Algumas das ponderações feitas na seção 4.6 foram aproveitadas neste jogo. Um problema surgiu, inicialmente, porque era impossível derrotar o computador. Com sua precisão matemática, o agente era capaz de apenas acionar o escudo no exato momento em que o míssil estava para colidir com ele, gastando a menor quantidade possível do recurso, tendo o suficiente para bloquear todos os ataques do adversário. O jogador, teoricamente, também poderia conseguir isso, mas seriam necessárias perícia e precisão muito incomuns para um ser humano.

Para evitar isso, foram colocadas imperfeições no comportamento do agente. Para solucionar a questão dos mísseis, foi acrescentado um tempo extra ao acionamento do escudo para que o computador perca mais esse recurso, equilibrando as ações e o impossibilitando de defender todos os ataques.

Outro problema que surgiu foi a perfeição com que o agente conseguia detectar a necessidade de acionar o escudo. Ou seja, não só ele o acionava pelo menor tempo possível, como, também, sempre sabia quando isso era necessário. Para solucionar essa outra questão, bastou usar o próprio conjunto nebuloso definido, relaxando o intervalo no qual as regras determinavam a necessidade do escudo. Sendo assim, quando um míssil passa "raspando" a nave, isto é, perto, mas não o suficiente para atingi-la, o escudo é acionado da mesma forma. Algumas imperfeições menores na quantidade de movimento também foram acionadas, embora não se tenha notado uma mudança muito significativa do resultado.

Mesmo este exemplo simples de jogo mostra a necessidade de se usar esse tipo de recurso para poder criar um ambiente mais desafiador e uma atmosfera mais realista para o jogador.

5.7.4 Aprendizado de Regras

O método descrito na seção 4.3.2 foi usado como forma de ensinar um novo agente a jogar e, também, para fazer previsão de jogadas, como será visto na seção 5.7.5.

O resultado final, apesar de todos os esforços, não foi dos melhores. Tanto usando o agente como tutor, quanto jogadores humanos, por tempos de até 15 minutos consecutivos de jogo, acabou não representando a criação de um agente decente pelas regras aprendidas, a maioria quase não se movimentava e acionava o escudo quase que aleatoriamente.

A grande dificuldade, neste caso, foi a própria maneira como o jogo foi modelado. O uso de oito variáveis de entrada e duas de saída criou um problema muito sério para o método, pois, para ele, todas as regras têm a mesma quantidade de variáveis, o que acaba gerando uma quantidade enorme de regras. Para as cerca de 40 regras do agente, o método aprendeu mais de 300 em 1 minuto, 700 em 5 minutos e pouco mais de 800 em 10 minutos. Mesmo apresentando uma queda significativa da quantidade de regras acrescentadas, ainda assim são números muito altos. Comparando, porém, com a quantidade total de regras possíveis, levando-se em conta todas as variáveis, esse número mostra-se relativamente baixo. Sendo 4 variáveis com 5 conjuntos e 6 com 3, o número de regras seria $3^6 5^4 = 455.625$.

O total de regras foi consideravelmente menor alterando o fator de desuso para um decaimento mais rápido, mas os agentes criados dessa forma praticamente nem se mexiam. Sem o fator de desuso, ou com o decaimento muito lento, a quantidade de regras aumentou, mas os agentes também passaram pelo menos a se mover, ainda que não muito adequadamente.

Apesar do pouco sucesso da aplicação do método, este não deixa de ser uma boa opção para complementar, em tempo real, algum conjunto de regras estático previamente estipulado. Como verificamos, também, na prática, a forma de modelar o problema pode ser determinante, contribuindo para o sucesso ou fracasso da sua aplicação.

5.7.5 Testes de Previsão

Os métodos descritos na seção 4.5, assim como o aprendizado de regras nebulosas em tempo real apresentado na seção 4.3.2, foram testados neste jogo para verificar o aproveitamento de previsão dos movimentos do adversário. Usou-se, também, pre-

visão aleatória para comparar o aproveitamento dos demais. O método de distância de edição usado foi sem previsão nebulosa, pois neste jogo em particular não há muita diferença entre o uso de um e outro.

A tabela 5.12 apresenta o aproveitamento médio (de acertos) dos métodos após 5 minutos de partida, e a figura 5.25 mostra os resultados obtidos (em porcentagem) com relação ao tempo (em segundos) de cada método, os quais, do melhor para o pior, respectivamente, foram: previsão seqüencial, aprendizado de regras nebulosas, aleatório e distância de edição.

Método	**Aproveitamento**
Previsão Seqüencial	83,56%
Aprendizando de Regras	53,77%
Aleatório	34,02%
Distância de Edição	23,95%

Tabela 5.12: Aproveitamento médio dos métodos de previsão após 5 minutos de jogo

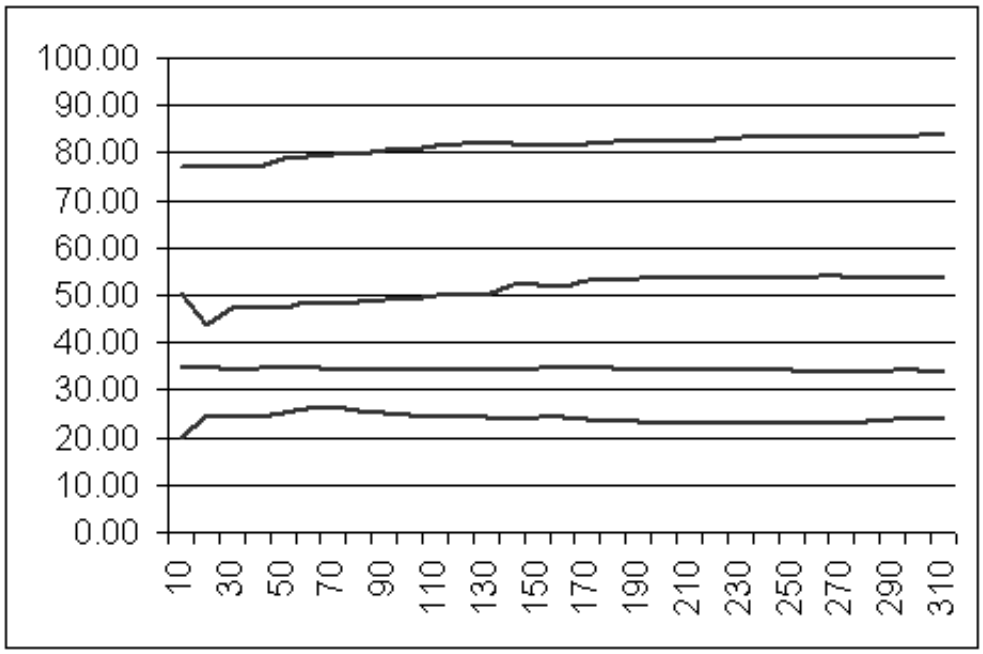

Figura 5.25: Resultados dos Métodos de Previsão

Apesar da surpresa inicial, o sucesso do método de previsão seqüencial não chega a ser muito misterioso. Tratando-se, neste caso, de prever os comandos do ad-

versário, não é muito difícil que haja seqüências longas de comandos repetidos. Por exemplo, direcionar a nave por 1 segundo para a esquerda, no jogo com 30 quadros por segundo, significa 30 movimentos repetidos, idênticos. Se o algoritmo, por exemplo, erra os 5 primeiros, acerta os próximos 25 e erra o 31^o, seu aproveitamento, ainda assim, fica excelente.

Já o método de previsão por distância de edição teve um aproveitamento abaixo do aleatório, o que é um resultado muito ruim. Além de ser o mais ineficiente de todos, tendo complexidade quadrática, ainda se saiu pior. É possível, realmente, verificar que, no final das contas, seu aproveitamento em aplicações de tempo real é pouco prático, não só pelo tempo como também pela memória requerida (além do aproveitamento).

O aprendizado de regras nebulosas como previsor, porém, mostrou-se especialmente vantajoso, estando acima do aleatório, mesmo estando abaixo da previsão seqüencial. Essa taxa de acerto representa uma boa generalização das regras aprendidas apenas por exemplos com pouco tempo jogo (5 minutos). Ao contrário da previsão seqüencial, entretanto, a previsão por aprendizado de regras tende a ser mais genérica, e, como já foi discutido anteriormente, leva em conta o contexto em que as ações foram tomadas para prever o que o adversário fará.

O aproveitamento do aprendizado de regras como previsão, já apresentado na tabela 5.12, sugere que ele conseguiu imitar o comportamento aprendido na metade dos casos, o que pode explicar o porquê de, ao ser usado como agente idependente, não ter conseguido um bom aproveitamento, como já foi discutido na seção 5.7.4.

De qualquer maneira, os métodos de previsão seqüencial e por aprendizado de regras nebulosas mostraram-se vantajosos para aplicações em tempo real, pelos seus resultados e sua eficiência, podendo ser aproveitados em outras aplicações do gênero.

Capítulo 6

Conclusões

"Fiz esta carta mais longa porque não tinha tempo de fazê-la mais curta."
Blaise Pascal

Este trabalho apresentou diversos métodos propostos para aproveitamento em tempo real, muitos deles originais ou modificações sobre outros já existentes. Todos as estratégias propostas foram implementadas, testadas e avaliadas.

As seções a seguir apresentam um balanço geral do trabalho e suas conclusões, visto que as discussões mais aprofundadas foram desenvolvidas ao longo dos demais capítulos.

6.1 Resultados Obtidos e Contribuições

Os resultados aqui obtidos foram, em alguns casos, ruins mas, em outros, extremamente animadores. Os métodos originais principais, a coevolução em tempo real e os padrões adaptativos, mostraram-se promissores e seus resultados foram realmente interessantes.

Os métodos baseados em lógica nebulosa também tiveram um bom resultado, com especial destaque para as máquinas de estado nebulosas que, apesar de sua simplicidade, se mostraram ferramentas poderosas. O aprendizado de regras em tempo real não obteve um resultado muito bom para a criação de um agente a partir do zero, embora tenha tido um desempenho digno de nota no que diz respeito à previsão.

A questão do erro artificial mostrou-se particularmente importante e, em conjunto com o assunto discutido superficialmente como inteligência desonesta, real-

mente precisaria de uma discussão maior. Afinal, de forma resumida, até que ponto o agente deve ter vantagens dadas pelo jogo sobre o jogador?

Como contribuição, este trabalho apresenta dois métodos totalmente originais (coevlução em tempo real e padrões adaptativos) e mudanças sobre métodos já existentes para uso em tempo real. Os resultados sugerem o bom aproveitamento dessas estratégias e que mais pesquisa deve ser feita neste campo tão pouco aprofundado que é evolução e adaptação em tempo real dirigidas de acordo com o adversário.

Mostramos, além disso, que o contexto de jogos eletrônicos pode ser usado como ponto de partida para pesquisas sérias e desenvolvimento de novos métodos para atacar problemas complexos.

6.2 Dificuldades e Crítica

"Não tive filhos, não transmiti a nenhuma criatura o legado de nossa miséria."
Machado de Assis *Memórias Póstumas de Brás Cubas*

A quantidade de métodos distintos testados impossibilitou, infelizmente, que mais testes fossem feitos e mais detalhados ainda fossem, assim como a própria dificuldade de ser necessário criar os jogos para seu aproveitamento também contribuiu neste sentido.

Muitas e muitas linhas de código foram gastas na criação de jogos e *engines* para possibilitar a aplicação dos métodos implementados. Esse deslocamento do tempo que poderia ter sido aproveitado nos testes dos métodos em si, ou mesmo desenvolvendo outras estratégias, sugere que a criação de um grupo de desenvolvimento de jogos forte pode ser uma alternativa muito importante para futuros projetos de pesquisa que necessitem desse mesmo tipo de infraestrutura.

Outra grande dificuldade foi a falta de equipamentos adequados. Apesar de este trabalho ter sido desenvolvido numa máquina razoável, a falta de uma placa 3D impossibilitou qualquer tentativa de implementar um jogo mais refinado, o que poderia atrair mais jogadores para os testes feitos.

Os testes que, por sinal, na maioria dos casos, só foram possíveis graças à boa vontade de muitos colaboradores que, de forma voluntária, participaram deste projeto. Muitos arquivos de *log* corrompidos ou incompletos tiveram de ser descartados de maneira que, não fosse a ajuda dessas pessoas em testar os métodos propostos, menos resultados seriam apresentados.

A pouca quantidade de trabalhos publicados nesta área específica também se mostrou um problema, uma vez que, com poucas referências, fica mais difícil ter uma idéia concreta de por onde começar e quais caminhos tomar. Isso, porém, confere uma característica meio precursora a este trabalho, o que o torna ainda mais interessante do ponto de vista daqueles que o desenvolvem, incentivando-os a buscar respostas para problemas pouco considerados, mas nem por isso menos importantes.

Finalmente, a maior dificuldade deste trabalho foi sua própria natureza. Tentar evoluir em pouco tempo e com poucos dados, com um espaço de busca extremamente grande, é um desafio que parece quase impossível. Dada cada situação, uma ou outra forma de burlar essa limitação pode ser encontrada, embora às vezes, infelizmente, isso não seja possível. Essas dificuldades e limitações todas, entretanto, muito mais do que desencorajar a pesquisa, nos incentivam, posto que se colocam como um desafio sedutor a ser encarado e, quem sabe, vencido, mesmo que parcialmente.

6.3 Trabalhos Futuros

Este é um assunto que não se esgota, e a quantidade de trabalhos que podem ser feitos a partir deste poderia gerar uma lista tão grande quanto o próprio trabalho.

Algo mais ou menos direto é aprofundar os testes dos métodos propostos, coletando mais resultados e, inclusive, aplicando-os em outras áres que não apenas de jogos, de maneira a verificar, inclusive, sua utilidade em outros contextos.

Outra tarefa interessante seria paralelizar os métodos aqui apresentados, ou pelo menos aqueles para os quais isso for vantajoso, de forma a buscar resultados ainda melhores.

Outro trabalho possível, além disso, seria implementar esses métodos por *hardware*, aproveitando-os em outras máquinas que não apenas computadores convencionais, como dispositivos portáteis, por exemplo.

E, obviamente, ainda há a possibilidade de se desenvolver e implementar novos métodos, com diferentes abordagens sobre este mesmo problema, aumentando, assim, o conjunto de soluções a serem tentadas num determinado contexto, com mais chances de se conseguir lidar com a situação de forma satisfatória.

6.4 Considerações Finais

"O resto é silêncio."
William Shakespeare, *Hamlet*

Apesar de todas as dificuldades encontradas e possíveis falhas do próprio trabalho, não temos medo de afirmar que o saldo foi bastante positivo. Sua natureza precursora, como já foi discutido, em conjunto com o grande leque de possibilidades que podem advir a partir do que foi estudado e desenvolvido aqui, nos leva a crer que a exploração deste tema pode render resultados notáveis posteriormente. Os trabalhos futuros aqui propostos também podem, acreditamos, contribuir muito nesse sentido.

Consideramos, por fim, ter conseguido, apesar de todos os percalços, satisfazer os objetivos iniciais deste trabalho e, por isso, o concluímos com a serena consciência do dever cumprido.

Capítulo 7

Referências Bibliográficas

Referências Bibliográficas

AGOGINO, Adrian, STANLEY, Kenneth, MIIKKULAINEN, Risto. "Online Interactive Neuro-evolution". **Neural Processing Letters**. v.11, n.1, 2000. p.29-37.

ALANDER, Jarmo T. "An Indexed Bibliography of Genetic Algorithms with Fuzzy Logic". In: PEDRYCZ, Witold. **Fuzzy Evolutionary Computation**. Boston: Kluwer Academic, 1997. p.299-318.

ANGELINE, Peter J., POLLACK, Jordan B. "Competitive Environments Evolve Better Solutions for Complex Tasks". In: FIFTH INTERNATIONAL CONFERENCE ON GENETIC ALGORITHMS (GA93), May, 1993. **Proceedings**. San Mateo. p.264-270.

BATTAIOLA, André Luiz. "Jogos por Computador - Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação". In: XIX JORNADA DE ATUALIZAçãO EM INFORMáTICA, 2000. **Anais**. Curitiba. p.83-122.

BELARBI, Khaled, TITEL, Faouzi. "Genetic Algorithms for the Design of a Class of Fuzzy Controllers: An Alternative Approach". **IEEE Transactions on Fuzzy Systems**. v.8, n.4, August, 2000. p.398-405.

BIRK, Andreas, PAUL, Wolfgang J. **Using Context to Scale the On-Line Evolution of Animat-Minds**. Bruxelas: Vrije Universiteit Brussel, 1999. (VUB AI Memo 99-02).

BLAIR, Alan D., LAND, Mark, POLLACK, Jordan B. "Coevolution of a Backgammon Player". In: FIFTH ARTIFICIAL LIFE CONFERENCE, 1997. **Proceedings**. Cambridge. p.92-98.

BURKE, Robert *et al.* "Creature Smarts: The Art and Architecture of a Virtual Brain". In: GAME DEVELOPERS CONFERENCE, March, 2001. **Proceedings**. San Jose. p.147-166.

CHAMPANDARD, Alex J. **Honest AI and Cheating AI?**. Disponível na INTERNET via http://artificialintelligence.ai-depot.com/ArtificialIntelligence/152.html. Arquivo consultado em 2003.

CHOWN, Tim. **C.R.A.I.G: Campaign for Real A.I. in Games**. Disponível na INTERNET via http://www.gamesdomain.com/gdreview/zones/strategy/craig/craig.html. Arquivo consultado em 2003.

CLIFF, Dave, MILLER, Geoffrey F. "Coevolution of Pursuit and Evasion II: Simulation Methods and Results". In: FOURTH INTERNATIONAL CONFERENCE ON SIMULATION OF ADAPTATIVE BEHAVIOR, 1996. **Proceedings**. Cambridge. p.506-515.

_____. "Tracking the Red Queen: Measurements of Adaptive Progress in Coevolutionary Simulations". In: THIRD EUROPEAN CONFERENCE ON ARTIFICIAL LIFE, 1995. **Proceedings**. Granada. p.200-218.

CORMEN, Thomas H. *et al.* **Algoritmos: Teoria e Prática**. Rio de Janeiro: Campus, 2002. 916p.

COX, Earl. **The Fuzzy Systems Handbook**. Massachussets: Academic Press, 1994. .

DELOURA, Mark. **Game Programming Gems**. Rockland: Charles River Media, 2000. 614p.

DEMASI, Pedro, CRUZ, Adriano J. de O. "Algoritmos Cooperativos Coevolucionários em Jogos". In: I WORKSHOP BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL, Outubro, 2002. **Anais**. Fortaleza.

_____. "Modelagem Fuzzy para um Jogo de Naves Espaciais". In: I WORKSHOP BRASILEIRO DE JOGOS E ENTRETENIMENTO DIGITAL, Outubro, 2002. **Anais**. Fortaleza.

_____. "Online Coevolution for Action Games". In: THIRD INTERNATIONAL CONFERENCE ON INTELLIGENT GAMES AND SIMULATION (GAME-ON 2002), November, 2002. **Proceedings**. London. p.113-120.

DICKERSON, Julie A., KIM, Hyun Mun, KOSKO, Bart. "Fuzzy Control for Platoons of Smart Cars". In: KOSKO, Bart. **Fuzzy Engineering**. Prentice Hall, 1997. p.177-195.

DUBOIS, Didier, PRADE, Henri. **Fuzzy Sets and Systems: Theory and Applications**. Massachussets: Academic Press, 1980. .

DYBSAND, Eric. "A Finite-State Machine Class". In: DELOURA, Mark. **Game Programming Gems**. Rockland: Charles River Media, 2000. p.237-248.

FLOREANO, Dario, NOLFI, Stefano, MONDADA, Francesco. "Competitive Coevolutionary Robotics: From Theory to Practice". In: ANIMALS TO ANIMATS V, 1998. **Proceedings**. Cambridge. p.512-524.

FUNES, Pablo *et al.* "Animal-Animat Coevolution: Using the Animal Population as Fitness Function". In: FIFTH INTERNATIONAL CONFERENCE ON SIMULATION OF ADAPTATIVE BEHAVIOR (SAB98), 1998. **Proceedings**. Cambridge. p.525-533.

_____. **The Internet as a Virtual Ecology: Coevolutionary Arms Races Between Human and Artificial Populations**. Cambridge: Brandeis University, 1997. (Computer Science Technical Report CS-97-197).

FUNES, Pablo, POLLACK, Jordan B. "Measuring Progress in Coevolutionary Competition". In: SIXTH INTERNATIONAL CONFERENCE ON SIMULATION OF ADAPTATIVE BEHAVIOR, 2000. **Proceedings**. Paris. p.450-459.

GORDIN, Maria, SEN, Sandip, PUPPALA, Narendra. "Evolving Cooperative Groups: Preliminary Results". In: AAAI WORKSHOP ON MULTIAGENT LEARNING, 1997. **Proceedings**. Providence. p.31-35.

GOULD, Stephen Jay. **Wonderful Life: The Burgess Shale and the Nature of History**. Harmondsworth: Penguin, 1991. .

HADWIGER, Markus. "3D Graphics Technology in Computer Games - Past, Present, and Future". In: 4TH CENTRAL EUROPEAN SEMINAR ON COMPUTER GRAPHICS, 2000. **Proceedings**. Austria. p.7-8.

JANG, Jyh-Shing Roger. "Derivative Free Optimization". In: JANG, Jyh-Shing Roger, SUN, Chuen-Tsai, MIZUTANI, Eiji. **Neuro-Fuzzy and Soft Computing: A Computational Approach**. Prentice Hall, 1997. p.173-195.

KNUTH, Donald E., SZWARCFITER, Jayme L. "A Structured Program to Generate All Topological Sorting Arrangements". **Information Processing Letters**. n.9, 1974. p.153-157.

KOSKO, Bart. **Fuzzy Thinking: The New Science of Fuzzy Logic**. London: Flamingo, 1993. .

KOZA, John R. "Genetic Programming". In: WILLIAMS, James G., KENT, Allen. **Encyclopedia of Computer Science and Technology**. Marcel-Dekker, 1998. p.29-43.

LAMOTHE, Andre *et al.* **Tricks of the Game Programming Gurus**. Indianapolis: Sams Publishing, 1994. 746p.

LAMOTHE, Andre. **Tricks of the Windows Game Programming Gurus**. Indianapolis: Sams Publishing, 1999. 1005p.

LI, Kevin W. *et al.* "Fuzzy Approaches to the Game of Chicken". **IEEE Transactions on Fuzzy Systems**. v.9, n.4, August, 1995. p.608-623.

LUCCHESI, Cláudio Leonardo. "Contributions of Jayme Luiz Szwarcfiter to Graph Theory and Computer Science". **Journal of the Brazilian Computer Society**. v.7, n.3, April, 2002. p.9-22.

MANBER, Udi. **Introduction to Algorithms: A Creative Approach**. Addison-Wesley Publishing, 1989. p.155-158.

MATTHEWS, James. "Basic A* Pathfinding Made Simple". In: RABIN, Steve. **AI Game Programming Wisdom**. Hingham: Charles River Media, 2002. p.105-113.

MCCUSKEY, Mason. "Fuzzy Logic for Video Games". In: DELOURA, Mark. **Game Programming Gems**. Rockland: Charles River Media, 2000. p.319-329.

MOMMERSTEEG, Fri. "Pattern Recognition with Sequential Prediction". In: RABIN, Steve. **AI Game Programming Wisdom**. Hingham: Charles River Media, 2002. p.586-595.

NELLER, Todd W. **Action-Based Discretization for AI Search**. Disponível na INTERNET via http://www.gamasutra.com/features/20021212/neller_01.htm. Arquivo consultado em 2003.

O'BRIEN, Larry. "Fuzzy Logic in Games". **Game Developer Magazine**. v.3, n.2, April, 1996. p.52-55.

OMAIFAR, Abdollah, MCCORMICK, Ed. "Simultaneous Design of Membership Functions and Rule Sets for Fuzzy Controllers Using Genetic Algorithms". **IEEE Transactions on Fuzzy Systems**. v.3, n.2, May, 1995. p.129-139.

PERMADI, F. **Ray-Casting Tutorial**. Disponível na INTERNET via http://www.permadi.com/tutorial/raycast. Arquivo consultado em 2003.

PEñA-REYES, Carlos Andrés, SIPPER, Moshe. "Fuzzy CoCo: A Cooperative-Coevolutionary Approach to Fuzzy Modeling". **IEEE Transactions on Fuzzy Systems**. v.9, n.5, October, 2001. p.727-737.

POTTER, Mitchell A., JONG, Kenneth A. De. "Cooperative Coevolution: An Architecture for Evolving Coadapted Subcomponents". **Evolutionary Computation**. v.8, n.1, 2000. p.1-29.

RABIN, Steve. **AI Game Programming Wisdom**. Hingham: Charles River Media, 2002. 672p.

_____. "Implementing a State Machine Language". In: RABIN, Steve. **AI Game Programming Wisdom**. Hingham: Charles River Media, 2002. p.314-320.

REYNERI, Leonardo Maria. "An Introduction to Fuzzy State Automata". In: INTERNATIONAL WORK-CONFERENCE ON ARTIFICIAL AND NATU-

RAL NEURAL NETWORKS (IWANN '97), 1997. **Proceedings**. Ilhas Canárias. p.273-283.

REYNOLDS, Craig W. "Competition, Coevolution and the Game of Tag". In: FOURTH INTERNATIONAL WORKSHOP ON THE SYNTHESIS AND SIMULATION OF LIVING SYSTEMS, July, 1994. **Proceedings**. Massachusetts. p.59-69.

_____. "Flocks, Herds, and Schools: A Distributed Behavioral Model". In: SIGGRAPH CONFERENCE, 1987. **Proceedings**. p.25-34.

ROLLINGS, Andrew, MORRIS, Dave. **Game Architecture and Design**. Scottsdale: Coriolis, 2000. 742p.

ROUSE, Richard. "Do Computer Games Need to be 3D?". **ACM SIGGRAPH Computer Graphics**. v.32, n.2, May, 1998. p.64-66.

SCOTT, Bob. "The Illusion of Intelligence". In: RABIN, Steve. **AI Game Programming Wisdom**. Hingham: Charles River Media, 2002. p.16-20.

SETNES, Magne, ROUBOS, Hans. "GA-Fuzzy Modeling and Classificantion: Complexity and Performance". **IEEE Transactions on Fuzzy Systems**. v.8, n.5, October, 2000. p.509-521.

SILVA, Eugênio da. **Reconhecimento Inteligente de Caracteres Manuscritos**. Rio de Janeiro: Instituto Militar de Engenharia, 2002. (Dissertação de Mestrado).

SILVEIRA, Sidnei Renato, BARONE, Dante Augusto Couto. "Jogos Educativos Computadorizados Utilizando a Abordagem de Algoritmos Genéticos". In: IV CONGRESSO IBEROAMERICANO DE INFORMáTICA EDUCATIVA (RIBIE 1998), 1998. **Anais**. Brasília.

SKLAR, Elizabeth *et al.* "Training Intelligent Agents Using Human Internet Data". In: FIRST ASIA-PACIFIC CONFERENCE ON INTELLIGENT AGENT TECHNOLOGY, 1999. **Proceedings**. Hong Kong. p.354-363.

SOUZA, Guilherme N. de, KAK, Avinash C. "Vision for Mobile Robot Navigation: A Survey ". **IEEE Transactions on Pattern Analysis and Machine Intelligence**. v.24, n.2, 2002. p.237-267.

STOUT, Bryan. "Smart Moves: Intelligent Path-Finding". **Game Developer Magazine**. v.3, n.4, October, 1996.

_____. "The Basics of A* for Path Planning". In: DELOURA, Mark. **Game Programming Gems**. Rockland: Charles River Media, 2000. p.254-263.

SUN, Chuen-Tsai. "Fuzzy Sets and Genetic Algorithms in Game Playing". In: JANG, Jyh-Shing Roger, SUN, Chuen-Tsai, MIZUTANI, Eiji. **Neuro-Fuzzy and Soft Computing: A Computational Approach**. Prentice Hall, 1997. p.551-567.

TOZOUR, Paul. "The Evolution of Game AI". In: RABIN, Steve. **AI Game Programming Wisdom**. Hingham: Charles River Media, 2002. p.3-15.

VENKATARAMAN, Sripriya. **Survey of results in impartial combinatorial games and an extension to three-player games**. Houston: Rice University, 2000. (Dissertação de Mestrado).

WANG, Li-Xin, MENDEL, Jerry M. "Generating Fuzzy Rules by Learning From Examples". **IEEE Transactions on Systems, Man and Cybernetics**. v.22, n.6, November / December, 1992. p.1414-1427.

WARD, Mark. **Video Games Without Frontiers**. Disponível na INTERNET via http://news.bbc.co.uk/1/hi/technology/2708995.stm. Arquivo consultado em 2003.

WARD, Martin. "Program Analysis by Formal Transformation". **The Computer Journal**. v.39, n.7, 1996. p.598-618.

WIEGAND, R. Paul, LILES, William C., JONG, Kenneth A. De. "An Empirical Analysis of Collaboration Methods in Cooperative Coevolutionary Algorithms". In: GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO-2001), 2001. **Proceedings**. San Francisco. p.1235-1242.

_____. "Analyzing Cooperative Coevolution with Evolutionary Game Theory". In: PROCEEDINGS OF THE 2002 CONGRESS ON EVOLUTIONARY COMPUTATION (CEC2002), 2002. **Proceedings**. Honolulu. p.1600-1605.

_____. "Cooperative Coevolution with Evolutionary Game Theory". In: CONGRESS ON EVOLUTIONARY COMPUTATION (CEC2002), July, 1994. **Proceedings**. Honolulu. p.1600-1605.

WILDE, Philippe de. "How Soft Games can be Played". In: 7TH EUROPEAN CONGRESS ON INTELLIGENT TECHNIQUES & SOFT COMPUTING, September, 1999. **Proceedings**. Aachen. p.FSD-6-12698.

WOODCOCK, Steven M. **1997 CGDC AI Roundtable Moderator's Report**. Disponível na INTERNET via http://www.gameai.com/cgdc97notes.html. Arquivo consultado em 2003.

_____. **Campaign for Real A.I. in Games Round Table Discussion**. Disponível na INTERNET via http://www.gamesdomain.com/gdreview/zones/strategy/craig/letters/lett05.html. Arquivo consultado em 2003.

_____. "Flocking: A Simple Technique for Simulating Group Behavior". In: DELOURA, Mark. **Game Programming Gems**. Rockland: Charles River Media, 2000. p.305-318.

_____. **Game AI Poll #25: What is the goal of game AI?**. Disponível na INTERNET via http://www.gameai.com/polls/100101_110901.html. Arquivo consultado em 2003.

_____. "Game AI: The State of the Industry". **Game Developer Magazine**. v.7, n.8, August, 2000.

_____. **Games Making Interesting Use of Artificial Intelligence Techniques**. Disponível na INTERNET via http://www.gameai.com/games.html. Arquivo consultado em 2003.

YEN, John, LANGARI, Reza. **Fuzzy Logic. Intelligence, Control and Information**. New Jersey: Prentice Hall, 1999. 541p.

ZADEH, Lotfi. “Fuzzy Sets”. **Information and Control**. v.8, n.3, 1965. p.338-353.

ØSTERGåRD, Esben H. **Evolving Complex Robot Behaviour**. Dinamarca: University of Aarhus, 2000. (Dissertação de Mestrado).