UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE MATEMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

VINÍCIUS FERNANDES DOS SANTOS

Aperfeiçoando Algoritmos Genéticos para Ambientes Não-Estacionários

Prof. Adriano Joaquim de Oliveira Cruz,
Ph.D.
Orientador

Rio de Janeiro, Junho de 2009

Ficha Catalográfica

Santos, Vinícius Fernandes dos

Aperfeiçoando Algoritmos Genéticos para Ambientes Não-Estacionários / Vinícius Fernandes dos Santos. – Rio de Janeiro: UFRJ IM, 2009.

100 f.: il.

Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro. Programa de Pós-Graduação em Informática, Rio de Janeiro, BR–RJ, 2009.

Orientador: Adriano Joaquim de Oliveira Cruz, Ph.D..

1. Jogos Eletrônicos. 2. Inteligência Computacional. 3. Algoritmos Evolutivos. 4. Algoritmos Genéticos. 5. Ambientes Não-Estacionários. I. Cruz, Ph.D., Adriano Joaquim de Oliveira. II. Título.

Aperfeiçoando Algoritmos Genéticos para Ambientes Não-Estacionários

Vinícius Fernandes dos Santos

Dissertação de Mestrado submetida ao Corpo Docente do Departamento de Ciência da Computação do Instituto de Matemática, e Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários para obtenção do título de Mestre em Informática.

Aprovado por:

_____

Prof. Adriano Joaquim de Oliveira Cruz, Ph.D. (Orientador)

_____

Prof. Felipe Maia Galvão França, Ph.D.

_____

Prof. Antonio Carlos Gay Thomé, Ph.D.

_____

Prof. Josefino Cabral Melo Lima, Docteur

Rio de Janeiro, Junho de 2009

Aos que acreditam e lutam pela educação.

AGRADECIMENTOS

Agradeço aos meus pais, por terem me dado a vida e por terem investido tanto em minha educação, me dando liberdade de escolher meu próprio caminho, à minha namorada, Debora Silva, pelo incentivo e pelo tempo cedido, inclusive revisando este texto, ao meu orientador, Adriano, pela paciência nos meus momentos de indecisão, insegurança e até mesmo preguiça, aos meus amigos pelas idéias, sugestões e incentivo, em particular à Maria Fernanda Wanderley, por ter encontrado erros de português e de digitação, à UFRJ e seus professores que colaboraram direta ou indiretamente com esta dissertação e à Capes, pela ajuda financeira durante o período de realização deste trabalho.

RESUMO

Este trabalho apresenta um estudo sobre Algoritmos Genéticos aplicados a problemas em Ambientes Não-Estacionários. Uma modificação de Algoritmos Genéticos é proposta, motivadas por Jogos Eletrônicos com essa natureza dinâmica. São apresentados resultados e uma análise da comparação entre o desempenho de Algoritmos Genéticos convencionais e modificados.

Palavras-chave: Jogos Eletrônicos, Inteligência Computacional, Algoritmos Evolutivos, Algoritmos Genéticos, Ambientes Não-Estacionários.

Improving Genetic Algorithms to Nonstationay Environments

ABSTRACT

This work presents a study about Genetic Algorithms applied to problems in Nonstationary Environments. A modified Genetic Algorithm is proposed, motivated by Games of dynamic nature. Results and analysis of the comparison of standard and modified Genetic Algorithms are presented.

Keywords: Video Games, Computational Intelligence, Evolving Algorithms, Genetic Algorithms, Nonstationary Environments.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

AG	Algoritmo Genético
AGSS	Algoritmo Genético Steady State
AGSSA	Algoritmo Genético Steady State Adaptativo
FPS	First Person Shooter
FSM	Finite State Machine
FuSM	Fuzzy State Machine
IA	Inteligência Artifial
MLP	Multi-Layer Perceptron
NPC	Non-Player Character
RNA	Rede Neural Artificial

SUMÁRIO

1 INTRODUÇÃO

Jogos eletrônicos são uma importante área da computação e têm um dos maiores mercados da área de entretenimento tendo movimentado 9,5 bilhões de dólares em 2007 (Entertainment Software Association, 2008). Apesar desta importância econômica, esta, assim como o entretenimento em geral, ainda não é uma área muito valorizada no meio acadêmico mas, lentamente, vem conquistando seu espaço.

Os jogos eletrônicos têm um grande potencial a ser explorado, por utilizarem conhecimentos de diversas áreas da computação, como Computação Gráfica, Compiladores, Redes, Inteligência Artificial (IA) etc, e mesmo de outras áreas de conhecimento, como as artes gráficas e o desenvolvimento de roteiros. Esta natureza interdisciplinar traz desafios e até mesmo oportunidades, desde que bem aproveitadas. Um exemplo disso é o avanço da tecnologia dos processadores gráficos das placas de vídeo. O mercado de jogos propiciou também um mercado cativo para estes dispositivos, fazendo com que a tecnologia avançasse e com que os custos fossem reduzidos por conta da produção em escala, o que trouxe benefícios para a área de computação gráfica aplicada. Estas mesmas placas gráficas, atualmente disponíveis a preços acessíveis, são utilizadas para simulações de arquitetura e poços de petróleo, por exemplo. Além disso, seus avançados processadores suportam uma versatilidade que

possibilita seu uso em outras aplicações vetoriais (THOMPSON; HAHN; OSKIN, 2002).

Com a melhoria da capacidade do hardware nos últimos anos, pôde-se abrir mão de uma fatia de tempo do processador, antes dedicado em sua maior parte aos recursos gráficos, para a inteligência artificial. Assim, a inteligência tem tido uma atenção maior nos jogos, já tendo sido aplicada de diversas formas em muitos jogos comerciais (Games Making Interesting Use of Artificial Intelligence Techniques, 2008). Assim como os jogos trouxeram benefícios para a computação gráfica, o emprego de inteligência artificial neste tipo de aplicação pode trazer benefícios a outras áreas com aplicações semelhantes, como sistema de controle para robôs.

A inteligência artificial aplicada em jogos pode ser utilizada de diversas maneiras, mas, em geral, busca-se um maior realismo ou maior nível de dificuldade nas ações e comportamentos dos agentes controlados pelo computador, conhecidos como NPCs (Non-Player Characters). Neste trabalho em particular, estamos interessados em como a inteligência artificial pode tornar jogos eletrônicos mais divertidos através de oponentes mais desafiadores e dinâmicos, ou seja, oponentes cuja estratégia e habilidade de jogo varie no decorrer do jogo ou entre partidas distintas. Para isso, usamos agentes que possam se adaptar ao jogador, ou seja, que evoluam de acordo com a experiência do jogador, elevando seu nível de dificuldade gradativamente.

Existem diversos métodos de aprendizado e evolução amplamente estudados. Porém, em sua maioria, estes métodos exigem grandes recursos computacionais que normalmente não estarão disponíveis em jogos eletrônicos. Tipicamente, estes métodos necessitam de grandes quantidades de dados e também de bastante tempo de processamento, requisitos não disponíveis em aplicações em tempo real. Jogos são aplicações que requerem uma resposta rápida, de modo que o tempo de processamento seja imperceptível ao jogador. Podemos tratar dados conhecidos e utilizá-los

de maneira eficiente durante o jogo, mas para isso precisaríamos que informações estivessem disponíveis previamente, o que seria um obstáculo limitante, uma vez que teríamos que fazer o usuário jogar contra o computador uma quantidade significativa de vezes antes que fosse possível algum tipo de adaptação. Com isso, métodos convencionais podem se tornar impraticáveis. Alguns métodos disponíveis prevêem aprendizado on-line, porém sua aplicação em jogos eletrônicos pode não ser tão imediata e requerer adaptações.

O estudo de adaptação de agentes aplicados a jogos eletrônicos vem sendo abordado de maneira mais ampla desde de 2003. Tem-se verificado o uso de diversas técnicas de Inteligência Artificial tais como Lógica Nebulosa (DEMASI, 2003), Aprendizado por Reforço (ANDRADE et al., 2004) e Algoritmos Evolucionários (DEMASI, 2003), (PONSEN, 2004) e (YANNAKAKIS; HALLAM, 2004).

Apesar da evolução em jogos ter sido estudada anterioremente, observa-se uma carência de trabalhos preocupados em como essa adaptação deve ocorrer em situações onde o jogador pode alterar sua estratégia no decorrer da partida ou ainda onde o jogador pode evoluir com o passar do tempo. Como discutiremos posteriormente, técnicas evolutivas convencionais podem carecer de mecanismos que possibilitem uma melhor adaptação nesses contextos. Motivados por essa deficiência, propomos uma nova abordagem de algoritmos evolutivos que possam responder melhor a este tipo de comportamento. Assim, temos como objetivo principal deste trabalho apresentar propostas que abordem este tipo de necessidade.

Apesar da motivação deste trabalho ser oriunda do estudo de técnicas de Inteligência Artifical para jogos eletrônicos, nossas propostas não se restringem a este domínio e podem ser utilizadas de forma mais ampla, em outros contextos que apresentem esta característica não-estacionária.

Algumas abordagens para problemas desta natureza, onde o ambiente pode se alterar em tempo real, modificando a função de adaptação do agente, já foram propostas e diversas delas podem ser encontradas em (BRANKE, 2001). Entretanto, a maioria delas faz suposições que não podem ser assumidas no nosso assunto de interesse, os jogos. Por exemplo, muitas destas propostas utilizam artifícios para a manutenção da variabilidade genética, facilitando a localização de novos pontos de máximo quando há alterações no ambiente. Entretanto, quando buscamos bons agentes para jogos, não é admissível agentes com comportamentos muito ruins, ou seja, todos (ou quase todos) os agentes utilizados no jogo devem possuir um desempenho razoável (SPRONCK; SPRINKHUIZEN-KUYPER; POSTMA, 2003). Dessa forma, este tipo de manobra não pode ser aplicada ao nosso problema. Outro tipo de abordagem utilizada é o uso de uma memória auxiliar, interessante em situações onde há alguma periodicidade na alteração do ambiente. No caso de jogos eletrônicos, este expediente não se mostra suficiente, pois em situações onde o jogador pode evoluir, informações armazenadas em instantes de tempo em que o nível de perícia do jogador era baixo não serão úteis em momentos futuros. Além disso, uma restrição adicional é a impossibilidade de reavaliação de agentes, uma vez que para reavaliá-lo seria necessário confrontá-lo com o jogador novamente. Dessa forma, uma consequência natural é que um método que trate deste problema deve, de alguma forma, utilizar os resultados obtidos em iterações anteriores, ainda que não confiáveis.

1.1 Organização da dissertação

Este trabalho está organizado em 7 capítulos. O capítulo 2 apresenta uma revisão sobre o uso de inteligência artificial em jogos, não se restringindo aos assuntos relacionados às propostas deste trabalho, mas apresentando uma visão geral sobre as técnicas utilizadas. O capítulo 3 apresenta os conceitos de Algoritmos Genéticos necessários para a compreensão das propostas deste trabalho. O quarto capítulo

descreve os modelos propostos, discutindo suas qualidades e limitações. O capítulo seguinte trata das formas de implementação de forma eficiente do modelo proposto. O capítulo 6 apresenta alguns resultados experimentais das técnicas propostas no capítulo 4. Por fim, o capítulo 7 apresenta as conclusões deste trabalho, bem como sugestões de trabalhos futuros a serem realizados neste tema.

2 INTELIGÊNCIA EM JOGOS

Jogos eletrônicos têm se popularizado cada vez mais, acompanhando o avanço da computação. Com isso, a variedade de títulos disponíveis ao consumidor aumenta a cada dia, em uma indústria bilionária. Em busca de diferenciais competitivos, a Inteligência Artificial se mostra um caminho interessante. O avanço da capacidade de processamento dos computadores nos últimos anos tornou possível um investimento em métodos mais elaborados e computacionalmente mais custosos e sua disseminação acabou inspirando o surgimento de motores de Inteligência Artificial, como a DirectIA[1] e a Autodesk Kynapse[2] por exemplo, seguindo o que já havia ocorrido com a área de Computação Gráfica e a área de Física para jogos, ambas com diversas opções de motores especializados. Apesar disso, métodos mais elaborados, como os estudados pela academia, ainda não são utilizados em larga escala, havendo predominância de métodos simples (CROCOMO, 2008). As principais razões são:

- Economia de recursos computacionais durante o jogo;
- Resistência a métodos não determinísticos;
- Economia de recursos humanos durante a fase de desenvolvimento, deixando

[1] Disponível em http://www.masagroup.net/.
[2] Disponível em http://www.autodesk.com/.

a inteligência como último componente a ser desenvolvido.

Ainda assim, existem exceções, como Black & White[3], The Sims[4] e Spore[5], por exemplo, que inclusive anunciam o uso de Inteligência Artificial como diferencial na divulgação de seus jogos.

No contexto dos métodos de aprendizados e adaptação em tempo real, Spronck apresenta em (SPRONCK; SPRINKHUIZEN-KUYPER; POSTMA, 2003) um conjunto de pré-requisitos que devem ser satisfeitos para que uma técnica seja aplicável na prática. Segundo ele, uma técnica deve ser:

Rápida Durante um jogo os recursos computacionais são escassos e o tempo de resposta deve ser quase que instantâneo.

Efetiva O agente deve ser desafiador para o jogador. Desafiador, aqui, não significa difícil. O agente deve ter um comportamento aceitável, sem jogadas absurdas e ser pelo menos tão desafiador quanto agentes especificados manualmente, podendo, eventualmente gerar comportamentos indesejados, desde que com pouca frequência.

Robusta O método deve ser capaz de lidar com as mais diferentes situações, se comportando de maneira razoável mesmo quando o jogador agir de maneira incoerente ou em situações inesperadas.

Eficiente A convergência deve ser rápida o suficiente pois o jogador ficará desestimulado se precisar confrontar o mesmo tipo de oponente diversas vezes até que a evolução apresente bons resultados.

[3]http://www.lionhead.com/bw2/Default.aspx
[4]http://www.thesims2.br.ea.com/
[5]http://www.spore.com/

Este capítulo apresenta um resumo do uso de inteligência artificial em jogos. Não nos aprofundaremos em detalhes em nenhuma das técnicas aqui citadas, mas apresentaremos, sempre que possível, referências para maiores informações. Embora a maioria das técnicas revisadas neste capítulo não esteja relacionada às propostas deste trabalho, utilizamos esse espaço para fornecer uma visão geral sobre o uso destes modelos. As técnicas aqui apresentadas são aquelas utilizadas com maior frequência em jogos de tempo real. Para jogos de tabuleiro, quebra-cabeças e lógica é normalmente utilizado o algoritmo Minmax (ou Minimax) bem como suas variações e otimizações, sendo o Corte (ou Poda) Alfa-Beta a variação mais conhecida por melhorar drasticamente o desempenho sem interferir no resultado (RUSSELL; NORVIG, 1995).

É importante ressaltar que muito do que se desenvolve para jogos é considerado segredo industrial, portanto, os dados sobre utilização das diversas técnicas são aqueles fornecidos pelos próprios fabricantes, que se utilizam dessas informações como estratégia de divulgação do seu produto.

Outro aspecto relevante dos jogos mais recentes é que, aproveitando-se da difusão da Internet alcançada nos últimos anos, muitos deles têm se focado nas plataformas multijogadores, negligenciando o desenvolvimento de bons agentes e, em alguns casos, restringindo a possibilidade de jogo sem outros jogadores (DEMASI, 2003).

Como bibliografia adicional sobre as diversas técnicas de Inteligência em jogos, sugerimos (SWEETSER, 2002), (ANDERSON, 2003) e (CHAMPANDARD, 2003).

2.1 Máquinas de Estado

Dentre todas as técnicas utilizadas em jogos, as Máquinas de Estado Finito (Finite State Machine, **FSM**) são as mais populares entre os desenvolvedores (SWEETSER, 2002). Razões para isso não faltam: elas são amplamente conhecidas, simples de se entender, implementar e depurar, além de aplicáveis nas mais diversas situações. Por essas razões, diversos jogos comerciais como Age of Empires[6], Doom[7] e Quake[7] utilizaram Máquinas de Estado na IA de seus agentes.

A idéia básica de uma FSM é que, em cada instante de tempo, o sistema, que pode ser um agente, um jogo etc, deve estar em um **estado**, no qual ele realiza **ações**. Cada um desses estados têm um conjunto de possíveis **transições** que podem ser feitas. A realização das transições entre estados depende de variáveis de **entrada**. Assim, a combinação Estado/Entrada define a transição de estado a ser feita e, neste novo estado (que pode até ser o mesmo), realiza as ações adequadas. Um exemplo de FSM pode ser vista na figura 2.1. Uma variante comum é realizar ações apenas nas transições entre estados.

Uma das vantagens da FSM está na implementação, que pode ser feita facilmente com estruturas de desvio, tipicamente comandos do tipo "switch/case". Além disso, pode-se utilizar estruturas hierárquicas, ou seja, um estado de uma máquina de estados contém outra máquina de estados. Isso facilita a modularização do código implementado, além de facilitar a modelagem do próprio problema.

Por outro lado, FSMs também apresentam desvantagens. Uma delas é a previsibilidade de seu comportamento. Em geral, após um certo tempo, o jogador compreende sua lógica de funcionamento e prevê suas ações. Uma maneira de se evitar este pro-

[6] www.microsoft.com/games/empires/
[7] http://www.idsoftware.com/

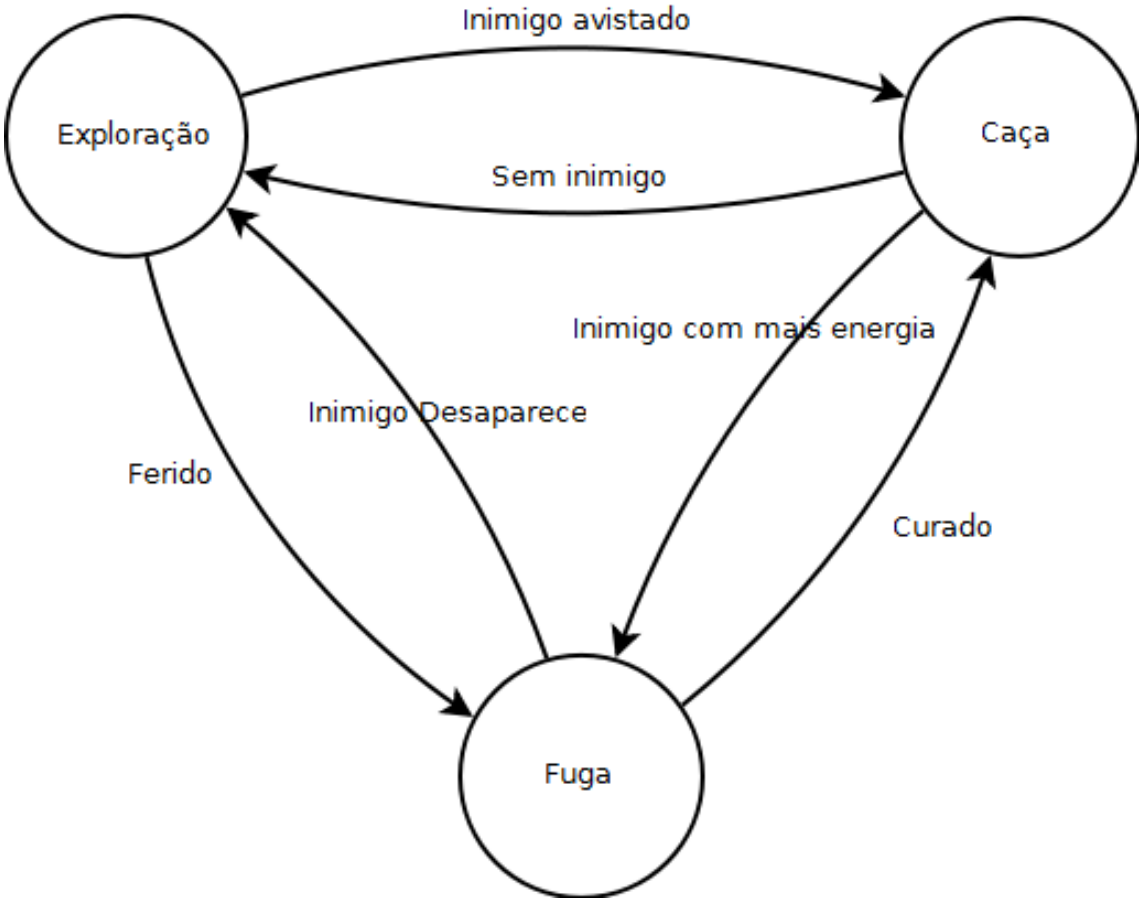

Figura 2.1: Exemplo de máquina de estados simples de um agente.

blema é adicionando uma probabilidade às suas transições, permitindo ao agente um comportamento menos óbvio. Apesar disso, ainda há alguns fatores limitadores no conceito do próprio modelo, como o fato de haver sempre um e apenas um estado ativo. Para resolver esta restrição, um outro modelo, que será abordado na seção 2.3, é utilizado e vem ganhando espaço no cenário de jogos.

Determinados jogos apresentam uma dinâmica muito complexa e diversa, de forma que prever todas as situações possíveis pode se tornar um desafio. Frequentemente os desenvolvedores são surpreendidos por situações criadas pelos jogadores mas não previstas no projeto inicial. Em situações deste tipo, o uso de FSM é desaconselhado,

pois a modelagem do agente torna-se complexa. Outro problema frequentemente abordado na literatura é em relação à sua escalabilidade, pois a manutenção de uma FSM torna-se difícil quando o número de estados cresce, uma vez que a tabela de transições deve ser totalmente preenchida. Porém, apesar de ser custoso montar grandes FSMs, uma versão hierárquica pode ao menos minizar este problema.

Uma característica bastante atraente é a facilidade de se utilizar FSMs juntamente com outras técnicas, fazendo com que as ações de um estado se utilizem de outros métodos mais especializados. Por exemplo, suponha que o nome um dos estados seja "Procurar inimigo" e ela tenha uma ação associada "Escolher caminho", para a tarefa de escolha do melhor caminho a ser percorrido em busca do adversário. Esta escolha pode ser feita por um outro algoritmo como o A* descrito na seção 2.8, por exemplo. Já a ação de atirar em um oponente pode ser feita utilizando-se outra técnica e assim por diante. Assim, pode-se utilizar um sistema híbrido, aproveitando as melhores características de cada modelo.

2.2 Lógica Nebulosa

A Lógica Nebulosa (em inglês "Fuzzy Logic") foi criada por Lotfi Zadeh ainda na década de 60 (ZADEH, 1965) e propõe uma forma de representar matematicamente conceitos que são imprecisos por natureza, como "frio", "lento" ou "perto".

Iniciemos falando sobre a Lógica Clássica de Aristóteles. Na Lógica Clássica, quando analisamos a veracidade de uma sentença, dizemos que ela é verdadeira ou falsa. Pela Lei da Não-Contradição, ela não pode assumir os dois valores simultaneamente e, pela Lei do Terceiro Excluído não há outro valor que a sentença possa assumir. Embora tal definição seja robusta e matematicamente satisfatória, essa forma de tratar os fatos se mostra incompleta quando analisamos diversas situações do mundo

real. O que a lógica nebulosa sugere é que os fatos podem ter um "nível" de verdade.

Considere por exemplo o conjunto das pessoas altas. Na lógica tradicional, uma pessoa pode ser alta ou não, ou seja, pode pertencer ou não ao conjunto das pessoas altas. Mas como definir se uma pessoa é alta ou não? Suponha que as pessoas altas sejam aquelas acima de 1,80m. Sendo assim, pessoas com 1,79m não seriam consideradas altas. Ou seja, dois indivíduos com apenas 1 centímetro de diferença seriam classificados de maneira diferente. A Lógica Nebulosa procura resolver esse problema suavizando o limite entre os conjuntos. Sabemos que o indivíduo com 1,80m é mais alto que o de 1,79m. Desta forma, não seria absurdo dizer que o primeiro "pertence mais" ao conjunto das pessoas altas que o segundo. Um possível gráfico para representar grau de pertinência de um indivíduo no conjunto das pessoas altas em relação à sua altura pode ser visto na figura 2.2.

Desta forma, introduzimos o conceito de **grau de inclusão** de um indivíduo a um conjunto. Chamando o conjunto de A e o indivíduo de x, representamos por $\mu_A(x)$ este grau de inclusão. Nesta nova forma de tratar os conjuntos, torna-se necessária uma nova representação para os elementos de um conjunto. Não podemos apenas listar os elementos de A, pois perderíamos a informação dos graus de inclusão. Seja X uma coleção de objetos. Para listar os elementos de A, um conjunto nebuloso em X, utilizamos os pares ordenados $(x, \mu_A(x)), \forall x \in X$, com $\mu_A(x) \in [0, 1]$, sendo 0 o grau de inclusão mínimo e 1 o grau de inclusão máximo.

A Lógica Nebulosa, através do uso de termos linguísticos, possibilita que sejam construídas bases de regras de maneira intuitiva e, através de mecanismos de nebulização e desnebulização, forneça valores numéricos para problemas dessa natureza. Além disso, sistemas nebulosos são robustos o suficiente para lidar de maneira satisfatória com dados incompletos ou com erros numéricos.

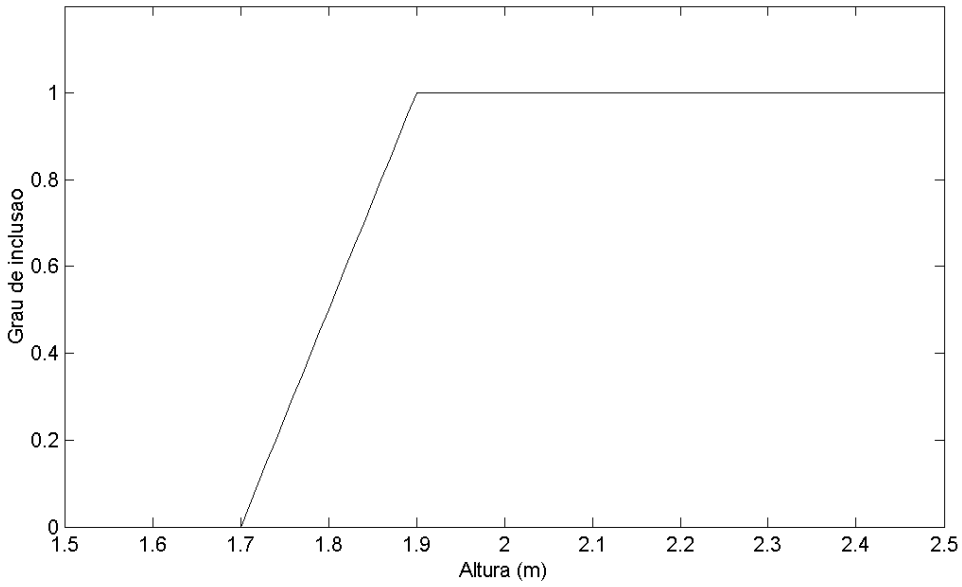

Figura 2.2: Representação gráfica da função de inclusão do conjunto nebuloso dos indivíduos altos

Suas propriedades tornam a Lógica Nebulosa recomendável quando não há um modelo matemático simples para o problema, quando o sistema requer conhecimentos de especialistas e para problemas não-lineares (SWEETSER, 2002). Ainda de acordo com Sweetser, o uso de Lógica Nebulosa em jogos normalmente não vai além de comandos do tipo "se-então".

Alguns jogos comerciais que utilizaram Lógica Nebulosa foram Platoon[8], Battle-Cruiser: 3000AD[9] e SWAT 2[10], dentre outros. Além de jogos, diversas outras áreas já utilizaram Lógica Nebulosa como sistemas de controle ferroviários, sistemas de diagnóstico, controladores de freio etc.

Finalmente, a Lógica Nebulosa é frequentemente utilizada em jogos combinadas com

[8]http://www.digitalreality.hu/
[9]http://www.bc3000ad.com/
[10]http://www.activision.com/

Máquinas de Estado, como veremos a seguir.

2.3 Máquinas de Estado Nebulosas

Como o próprio nome sugere, Máquinas de Estado Nebulosas (Fuzzy State Machines, FuSM ou Finite Fuzzy State Machine, FFSM) combinam os conceitos de Máquinas de Estado Finito com a flexibilidade da Lógica Nebulosa (ALVIM; OLIVEIRA CRUZ, 2008). Este modelo aos poucos tem ficado cada vez mais disseminado, por conta da sua similaridade com as FSMs, aproveitando conhecimento anterior dos desenvolvedores, e de suas vantagens em relação a elas, fornecendo uma maior flexibilidade.

A principal diferença entre FSMs e FuSMs é a possibilidade de, em um determinado instante de tempo, mais de um estado estar ativo, cada um com um determinado grau de ativação. Isso é possível pois, ao invés das transições serem definidas de maneira rígida em relação às entradas, elas passam agora a ser definidas por regras nebulosas com seus determinados graus de ativação. Isso permite que, em um jogo de FPS por exemplo, o NPC esteja em um estado de "Perseguição" e "Busca de munição" ao mesmo tempo. Dessa forma, o comportamento do agente pode se tornar mais realista, pois em um determinado momento ele pode ter mais de um objetivo a ser alcançado. Apesar desta diferença, Máquinas de Estado Nebulosas podem ser utilizadas como FSM tradicionais, sendo consideradas, portanto, uma generalização destas.

Uma vantagem natural desta flexibilidade é a dificuldade de previsão das ações do agente. Para o jogador, torna-se uma tarefa complexa determinar se o adversário está apenas perseguindo-o ou também procurando satisfazer algum outro objetivo, por exemplo.

Outra aplicação na qual FuSMs têm sido utilizadas com frequência é na simulação de comportamento de NPCs e suas emoções, como em (ALVIM; OLIVEIRA CRUZ, 2008). A versatilidade das máquinas de estado nebulosas permite, por exemplo, que os agentes estejam com "muita raiva" ou "pouca raiva" do jogador em um determinado momento, sem a necesidade de estados distintos referentes a cada uma das situações. Dessa forma, com um único estado denominado "raiva" é possível determinar a intensidade desta emoção através do grau de ativação do seu respectivo estado. De acordo com Sweetser (SWEETSER, 2002) esse tipo de comportamento mais flexível pode ainda aumentar o interesse do jogador, por fazê-lo experimentar reações diferentes dos NPCs em cada nova interação com eles.

Jogos comerciais como Close Combat[11], Petz[12] e Civilization: Call to Power[13] utilizaram Máquinas de Estado Nebulosas para definição de prioridades de ações a serem tomadas e também para obtenção probabilidades de realização de cada uma das ações possíveis.

2.4 Redes Neurais

Redes Neurais Artificiais, RNAs, também conhecidas como Redes Neuronais, são uma tentativa de simular o comportamento de neurônios e de suas interligações através do computador. Este modelo possibilita que certas tarefas desempenhadas com facilidade por seres humanos, porém de difícil modelagem computacional, como reconhecimento de caracteres manuscritos (SILVA; THOME, 2007), sejam executadas através do uso de computadores. RNAs são também utilizadas em diversas outras tarefas de classificação, aprendizado e predição. Um dos grandes argumentos para o uso deste modelo é sua capacidade de generalização, fornecendo respostas

[11] http://www.closecombatseries.net/
[12] http://www.petz.com/
[13] http://www.civilization.com/

semelhantes para entradas similares, dispensando a necessidade de apresentação de todos os cenários possíveis possibilitando, assim, atuação satisfatória mesmo em cenários que não foram vistos anteriormente.

O modelo de Redes Neurais mais difundido é o Multi-Layer Perceptron, MLP, que se utiliza de diversas unidades simples, os Perceptrons, dispostos em camadas interligadas, tendo suas ligações entre diferentes camadas ponderadas por pesos sinápticos. Um exemplo pode ser visto na figura 2.3. Este modelo normalmente utiliza o algoritmo de aprendizado de retropropagação (back-propagation) que possibilita que, a partir de um conjunto de amostras de treinamento, os pesos sinápticos sejam ajustados de forma que a saída fornecida pela rede para cada uma das amotras seja a mais próxima possível da esperada pelos exemplos. Com isso, pode-se treinar agentes na fase de desenvolvimento com amostras de exemplo. Porém, seu uso para aprendizado com o jogador normalmente não é utilizado pois elas apresentam uma séria limitação: o seu aprendizado deve ser feito antecipadamente e requer uma quantidade de amostras razoável. Embora possa-se tentar contornar esta limitação realizando o treinamento paralelamente durante a execução do jogo, a necessidade de uma quantidade considerável de exemplos para treinamento acaba por restringir este treinamento para uma frequência baixa, com uma nova Rede Neural a cada intervalo de tempo.

Outro problema levantado com frequência em relação ao uso de RNAs em jogos refere-se ao seu não determinismo no treinamento. O resultado final do treinamento depende dos pesos iniciais, podendo, assim, convergir para soluções inesperadas. Embora o resultado possa ser checado na fase de desenvolvimento do jogo, esta característica pode gerar comportamentos indesejados caso a rede seja treinada para se ajustar ao usuário posteriormente. Ainda assim, caso essas questões sejam levadas em consideração e a quantidade e qualidade de exemplos de treinamento seja suficiente, pode-se obter bons resultados com o uso de Redes Neurais em jogos. Ou-

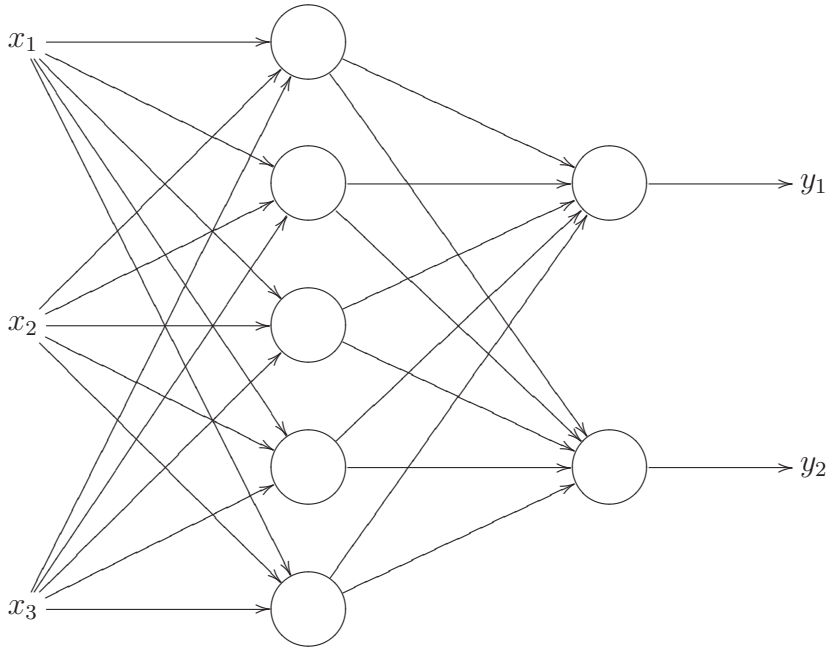

Figura 2.3: Uma rede neural MLP típica, com duas camadas, 3 entradas, 2 saídas e 5 neurônios na camada intermediária.

tro problema levantado é quanto a seu custo computacional. De maneira geral uma rede pode precisar fazer diversos cálculos para fornecer a saída esperada. Entretanto, diversos problemas podem ser resolvidos com redes pequenas e, portanto, que não requerem fatias de tempo muito grandes do processador.

Diversas outras questões como a escolha e tratamento das variáveis de entrada e o tamanho da rede e de suas camadas, por exemplo, interferem diretamente no resultado das Redes Neurais. Apesar disso, não nos alongaremos neste assunto aqui, uma vez que existem melhores referências sobre estes assuntos, como (HAYKIN, 1998), por exemplo.

Seu uso em jogos pode ter diversos enfoques, como interpretação visual e imitação do comportamento de jogadores (tanto para controle de um agente quanto para previsão de futuras jogadas). Apesar disso, pelo receio de problemas como os citados anteriormente, seu uso mais comum é como uma caixa preta construída na fase de desenvolvimento e imutável, embora haja exceções, como o jogo Creatures[14].

2.5 Algoritmos Genéticos

Algoritmos Genéticos, AGs, são métodos que se baseiam no conceito de evolução e aplicam esta mesma idéia em problemas de otimização. Para isso, simula-se de maneira simplificada os principais fenômenos que interferem na evolução das espécies como cruzamento entre indivíduos, mutações e sobrevivência dos mais adaptados. Maiores detalhes sobre esta técnica serão dados no capítulo 3, uma vez que esta técnica é de grande interesse para este trabalho.

Suas principais características são sua capacidade de funcionar bem em espaços de

[14] http://www.gamewaredevelopment.co.uk/

buscas de alta dimensionalidade, complexos e de natureza desconhecida.

No contexto de jogos, diversas críticas são feitas aos AGs, pelo não-determinismo da simulação e também são levantadas diversas questões sobre sua velocidade. Apesar disso, trabalhos anteriores já mostraram que algoritmos genéticos podem ser utilizados com sucessso, como (DEMASI; OLIVEIRA CRUZ, 2003) e (CROCOMO, 2008). Faremos uma breve discussão sobre esses aspectos no capítulo 4.

2.6 Scripts

Esta técnica é muito popular entre desenvolvedores e até mesmo para pessoas sem experiência em programação pela sua facilidade de implementação e compreensão e por se assemelhar com linguagem natural. Por isso, pode ser interessante para equipes heterogêneas, com game designers, artistas e jogadores. Ao mesmo tempo que compartilha da popularidade das FSMs, Scripts também têm limitações em comum, como a alta previsibilidade e necessidade de se conhecer de antemão todas as situações possíveis.

Diversos jogos oferecem suporte à utilização de scripts para controle da Inteligência Artificial de seus agentes, bem como de diversas outras funcionalidades, sendo estes scripts, em muitos casos, linguagens bastante poderosas, suportando diferentes tipos de abstrações. Alguns jogos, como Quake e Unreal desenvolveram suas próprias linguagens de scripts, enquanto outros, como World of Warcraft[15] e MDK2[16] utilizam alguma linguagem existente, como a Lua (IERUSALIMSCHY; FIGUEIREDO; FILHO, 1996). Alguns jogos que utilizam scripts para o controle de IA, dentre muitos

[15] http://www.worldofwarcraft.com/
[16] http://www.bioware.com/games/mdk2/

outros, são Neverwinter Nights[17] e Baldur's Gate[18].

2.7 Flocking

A técnica de Flocking, também conhecida como Swarming, foi inicialmente apresentada por Reynolds em 1987 (REYNOLDS, 1987) e é um modelo bastante específico para o controle de grupos de indivíduos como rebanhos ou cardumes. Este método foi proposto como alternativa aos métodos utilizados na época, como scripts, que se tornavam custosos e trabalhosos para grandes quantidades de indivíduos. Além de sua aplicação em jogos, este modelo foi também utilizado em filmes, simulando o comportamento de animais simulados por computador.

A principal característica desta técnica é a simplicidade (para que possa ser computacionalmente barata). Sendo assim, nenhuma informação é armazenada pelos indivíduos, sendo estes, portanto, sem memória. Assim, os indivíduos se comportam reativamente, baseados apenas nas suas percepções do ambiente.

Apesar de bastante simples, essa técnica apresenta resultados bastante satisfatórios e acaba se mostrando importantíssima para possibilitar o processamento de grupos muito grandes de indivíduos, o qual poderia se tornar excessivamente custoso se realizado com técnicas mais elaboradas.

Os comportamentos mais comuns para membros de grupos utilizando a técnica de flocking são:

- Separação: Cada indivíduo do grupo evita ficar perto de outros membros,

[17] http://nwn.bioware.com/
[18] http://www.planetbaldursgate.com/

aplicável, por exemplo, quando procura-se ocupar uniformemente um espaço;

- Alinhamento: A movimentação de um agente segue a média da movimentação de seu grupo. Comum em grupos migratórios;
- Coesão: Os indivíduos se movimentam na direção da média das posições dos demais membros de seu grupo, um artifício comum para intimidar predadores.

Dentro os jogos que utilizam esta técnica, podemos citar Half-Life[19], Unreal[20] e Enemy Nations[21].

2.8 Algoritmo A*

O algoritmo A* (lê-se "A-estrela") é o algoritmo mais utilizado para fazer um agente se movimentar de um ponto de origem para um ponto de destino. Na maioria dos jogos este algoritmo é utilizado como passo intermediário, em um sistema de Inteligência que utiliza outros métodos em conjunto. Usualmente é utilizado como uma das ações disparadas por uma FSM, FuSM ou por um Script que controla o agente em questão.

Apesar de ter sido apresentado pela primeira vez em 1968 por Peter Hart, Nils Nilsson, e Bertram Raphael (HART; NILSON; RAPHAEL, 1968), este algoritmo costuma ser revisado com enorme frequência e é possível encontrar trabalhos com variações do algoritmo até os dias atuais em congressos especializados.

Para encontrar o caminho entre dois pontos sem a necessidade de se fazer uma busca exaustiva é utilizada uma função heurística, normalmente simples, que provê, em

[19] http://www.steampowered.com/
[20] http://www.unreal.com/
[21] http://www.enemynations.com/

conjunto com as informações já conhecidas, formas de se determinar qual direção seguir.

O A* duas características bastante interessantes: ao mesmo tempo que fornece bons resultados na prática, ele também garante que sempre encontra um caminho, desde que haja um. Por outro lado, apesar de apresentar resultados superiores ao da busca em largura no caso médio, ele não reduz a complexidade assintótica do problema podendo, no pior caso, visitar todos os estados intermediários possíveis.

2.9 Aprendizado em Árvores de Decisão

Este é um método de aprendizado de máquina que é amplamente conhecido e um dos mais utilizados em aplicações práticas, como em Data Mining. Árvores de Decisão são robustas em relação a ruídos e apresentam uma vantagem em relação a outros métodos de aprendizado, como Redes Neurais, que é a fácil interpretação de sua estrutura interna e dos dados que ela armazena. Por exemplo: após treinada, uma Rede Neural apresenta um conjunto de pesos sinápticos que não passam informação de maneira clara, codificando de maneira não intuitiva os parâmetros utilizados na tomada de decisão. Já em uma Árvore de Decisão, a interpretação ocorre de maneira mais natural, com as informações armazenadas em cada nó de maneira similar àquelas fornecidas ao sistema. Além da facilidade de interpretação, Árvores de Decisão são razoavelmente simples de se programar.

Apesar dessas vantagens, esta não é uma técnica amplamente utilizada em jogos, sendo o jogo Black & White 2 um dos poucos casos conhecidos. Neste jogo em particular, uma criatura é capaz de aprender com o jogador ou mesmo com outras criaturas.

3 ALGORITMOS GENÉTICOS

Algoritmos Evolutivos (ou Evolucionários) são uma sub-área da Computação Evolutiva e são baseados na teoria da evolução de Darwin. Algoritmos Genéticos (AGs) são a variação mais conhecida de Algoritmos Evolutivos tendo sido popularizados por J. Holland (HOLLAND, 1975) e vêm sendo utilizados com sucesso em Economia, Química, Matemática, dentre outros campos de conhecimento e também em diversas áreas da computação como Jogos Eletrônicos (GRIECO, 2007) e Biologia Computacional (WANDERLEY et al., 2008).

Esta classe de algoritmos de otimização se baseia no conceito de que um**indivíduo** mais adaptado ao ambiente tem maiores chances de **reprodução** do que outros individuos da mesma **população**, passando com maior probabilidade os **genes**, presentes em seu **cromossomo**, para as **gerações** posteriores. A adaptação do indivíduo ao ambiente é dada por uma **função de avaliação** que é utilizada pelo operador de **seleção** responsável pela escolha de quais indivíduos irão se reproduzir. Além da reprodução dos indivíduos, frequentemente é utilizado também o conceito de **mutação**, de forma a aumentar a **variabilidade genética** e também explorar de forma mais abrangente o espaço de busca do problema em questão.

Algoritmos genéticos realizam uma busca estocástica e por isso são normalmente utilizados em problemas onde é difícil determinar uma solução ótima, seja pela complexidade do problema, impedindo o uso de outras técnicas, seja por um grande espaço de busca.

3.1 Um algoritmo genético básico

Como exposto acima, os algoritmos genéticos, bem como os demais algoritmos evolutivos, se baseiam na idéia de simular a evolução de uma espécie. Para facilitar a compreensão de seu funcionamento, apresentamos no algoritmo 3.1 um modelo de algoritmo genético que, embora simples, apresenta as principais características utilizadas.

Algoritmo 3.1 Algoritmo Genético Básico

```
1:  gerar população inicial
2:  enquanto não satisfizer condição de parada faça
3:      avalia indivíduos da população
4:      enquanto nova população não estiver completa faça
5:          seleciona par de pais de acordo com sua aptidão
6:          gera filhos a partir de cruzamento
7:          efetua possíveis mutações
8:          adiciona novos individuos à nova população
9:      fim de enquanto
10:     substitui população atual pela nova população
11: fim de enquanto
```

A linha 1 é responsável pela geração da população inicial. A população tem um tamanho pré-definido de indivíduos. O cromossomo de cada um dos indivídos é representado por um vetor de bits de tamanho fixo. Porém, para facilitar a convergência do algoritmo e também para melhorar a eficácia dos operadores, outros tipos de dados, diferentes da codificação binária, podem ser utilizados no vetor que representa o cromossomo.

A linha 2 é a responsável pelo laço principal do algoritmo. A condição de parada normalmente é o número máximo de gerações, embora possa ser utilizada alguma outra condição desejada, como um valor de referência para a função de avaliação.

A linha 3 é responsável pela avaliação dos indivíduos. Esta avaliação é, possivelmente, o principal ponto do algoritmo genético. Indivíduos mais adaptados devem receber valores mais altos da função de avaliação, enquanto a indivíduos com uma aptidão inferior devem ser atribuídos valores menores. Esses valores, além de indicar o quão bom um indivíduo é, são utilizados no passo de seleção para reprodução, fazendo com que indivíduos mais adaptados se reproduzam com uma probabilidade maior e, portanto, com maior frequência. Este passo, assim como o laço responsável pela criação da geração posterior, é executado uma vez a cada geração.

O laço das linhas 4-9 é responsável por gerar uma nova geração indivíduos a partir de uma geração atual. São as operações realizadas neste laço que definirão a maneira que o espaço de busca será explorado. Este loop é repetido até que a nova geração seja completamente formada.

A linha 5 utiliza algum operador de seleção para escolher dois pais que se reproduzirão. Os métodos mais comuns são o método da **roleta** e o método do **torneio**. No método da roleta, como o nome sugere, os indivíduos são dispostos em uma roleta com a fatia correspondente a cada individuo tendo tamanho proporcional à sua avaliação. Assim ocorre o sorteio dos dois indivíduos que se reproduzirão. Em termos mais formais, se cada indivíduo I_i tem uma avaliação f_i, a probabilidade p_i deste individuo ser sorteado pode ser dada por

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j},$$

onde n é o total de indivíduos na população. Para que este método funcione, estamos

supondo que $f_i > 0$ para $1 \leq i \leq n$, desprezando o caso onde $f_i = 0$, pois, por pior que seja a avaliação de um indivíduo, é interessante que sua probabilidade de seleção seja maior que zero, para que se mantenha a variabilidade genética. Como pode-se imaginar, este método depende diretamente do valor da função de avaliação. Sendo assim, caso um indivíduo tenha um valor na função de avaliação muito maior que os outros, a probabilidade deste indivíduo ser selecionado tende a 1. Embora este comportamento possa parecer desejável, ele compromete a capacidade de busca do algoritmo genético, não explorando bem o espaço de busca. Caso este indivíduo seja um mínimo local, por exemplo, dificilmente o algoritmo genético conseguirá explorar outros pontos do espaço. Este tipo de comportamento ocorre com frequência em problemas onde a variação da função de avaliação tem natureza exponencial. Para resolver este tipo de problema, podemos utilizar uma outra função de avaliação $f'_i = \log f_i$, suavizando a distribuição de probabilidade. Ainda assim, muitas vezes a natureza da função de avaliação não é totalmente conhecida ou ainda é muito irregular em determinadas regiões do espaço. Para evitar este tipo de problema, utiliza-se um outro método de seleção conhecido como torneio. Como o nome sugere, é realizado um torneio entre um conjunto de indivíduos. Para a seleção dos pais, são selecionados aleatoriamente (com uma distribuição uniforme) k individuos da população, com possíveis repetições. O indivíduo com a melhor avaliação dentre os k indivíduos é selecionado para o cruzamento. O mesmo processo é repetido para a seleção do segundo pai. Como pode-se observar, a ordem de grandeza absoluta da avalição deixa de desempenhar um papel tão importante, deixando o método mais robusto e fazendo a seleção depender apenas da relação de ordem entre os valores de avaliação.

A linha 6 é responsável pelo cruzamento dos filhos. Normalmente há uma probabilidade p_c (entre 50% e 90%) de ocorrer o cruzamento. No caso de não haver o cruzamento, os filhos serão idênticos aos pais. O primeiro passo para o cruzamento é sortear um ponto de corte k entre 1 e $l - 1$. Seja $a_1 a_2 ... a_l$ o cromossomo do

primeiro pai e $b_1b_2...b_l$ o cromossomo do segundo pai. Então, os filhos são dados por $a_1a_2...a_{k-1}a_kb_{k+1}b_{k+2}...b_l$ e $b_1b_2...b_{k-1}b_ka_{k+1}a_{k+2}...a_l$. Em alguns casos utiliza-se diversos ponto de corte, com comportamento análogo ao caso com apenas um ponto. Para vetores com valores numéricos, existem outras estratégias que levam em consideração os valores de cada posição do vetor. Para maiores detalhes, consulte (LINDEN, 2006).

O passo seguinte ao cruzamento é a aplicação da mutação. Assim como na operação de cruzamento, existe um parâmetro p_m, a probabilidade de mutação. Porém, ao contrário da situação anterior, onde a probabilidade decidia se o cruzamento ia ocorrer, neste casos a probabilidade p_m é aplicada a cada gene. Dessa forma, para cada posição do vetor (de cada um dos filhos) existe uma chance de ocorrer uma alteração ao acaso. No caso de vetores binários, uma mutação seria a inversão do bit referente àquela posição. Para casos mais gerais normalmente é sorteado um valor aleatório para substituir o valor atual ou o valor aleatório é utilizado como modificador (somando-se ou multiplicando-se ao valor da posição atual, por exemplo). Em geral o valor de p_m é em torno de 1%, embora possa variar, dependo do tamanho do cromossomo.

Finalmente, na linha 8 os individuos gerados são adicionados à nova população, onde permanecerão até que esta esteja completa, substituindo a população antecessora, e estes novos indivíduos possam gerar seus próprios descendentes.

Além dos operadores de seleção, cruzamento e mutação, uma outra operação comumente utilizada nos algoritmos genéticos é o elitismo. Esta operação se resume a manter na nova população os melhores indivíduos da geração anterior. Dessa forma, além de se manter as melhores soluções do problema (que poderiam se perder por conta de operações de cruzamento e mutação), aumenta-se a presença dos genes desses indivíduos da população. Frequentemente propaga-se apenas o melhor individuo

da população, mas não há uma restrição na quantidade de individuos que pode ser mantida.

Após a criação da nova população, ela substitui a população antecessora, como descrito na linha 10 e a próxima iteração do laço é executada. Ao término da execução do algoritmo, a solução para o problema que estamos resolvendo é dada pelo indivíduo com a melhor avalição dentre os presentes na última geração.

Diversas variações deste algoritmo, como alguns detalhes citados, já foram propostas. Tamanho variárel da população, reprodução entre indivíduos de gerações diferentes, métodos de seleção de reprodutores, de cruzamento e de mutação diferentes, são as alternativas mais conhecidas. Apesar de todas essas possibilidades, não nos aprofundaremos nestes detalhes, pois eles estão além dos objetivos deste trabalho. Para maiores detalhes sobre essas variações, consulte (LINDEN, 2006) ou (GOLDBERG, 1989).

3.2 Reprodução Steady State

O Algoritmo Genético apresentado anteriormente é a forma mais difundida e utilizada na literatura. Um variação menos conhecida e de interesse para este trabalho, é a reprodução Steady State (Estado Estacionário), ou Incremental.

Como visto anteriormente, a cada geração uma nova população era gerada e a anterior descartada. Com isso, um indivíduo pode se reproduzir apenas com um outro da mesma geração. Sabemos que na natureza a reprodução não ocorre em gerações bem definidas, havendo interação de indivíduos de diferentes idades. Portanto, uma tentativa natural é utilizar esta mesma característica nos Algoritmos Genéticos. A técnica de Steady State é uma tentativa de simular esse comportamento.

Da mesma forma que na reprodução tradicional, que chamaremos aqui de**reprodução geracional**, o primeiro passo é gerar a população inicial. A partir daí, a cada iteração do algoritmo são selecionados dois pais para o cruzamento. As técnicas de seleção, cruzamento e mutação são as mesmas utilizadas na versão geracional do algoritmo. Por conveniência, pode-se gerar dois indivíduos por vez, já que a operação de cruzamento gera dois filhos naturalmente, mas é também comum a geração de apenas um filho (descartando um deles ao acaso, por exemplo). Um versão genérica de utilização da técnica Steady State, com geração de apenas um filho por vez, pode ser encontrada no algoritmo 3.2. A única operação nesta versão do algoritmo não presente no algoritmo 3.1 é o **descarte** de um indivíduo para inserção do novo indivíduo gerado na população, na linha 6 do algoritmo.

Algoritmo 3.2 Algoritmo Genético Steady State

```
1: gerar população inicial
2: enquanto não satisfizer condição de parada faça
3:     seleciona par de pais de acordo com sua aptidão
4:     gera um filho a partir de cruzamento
5:     efetua possíveis mutações
6:     descarta um indivíduo da população
7:     adiciona novo individuo à nova população
8:     avalia novo indivíduo
9: fim de enquanto
```

O método de descarte mais intuitivo é o descarte do pior indivíduo. Parece natural que, dentre todos os indivíduos da população, o pior deles seja eliminado. De fato, esta escolha acelera a convergência, porém, acarreta na perda de diversidade genética, aumentando as chances de alcançarmos um mínimo local. Uma vantagem adicional desta estratégia é a manutenção da melhor solução encontrada. Dessa forma, o melhor indivíduo estará presente na população final, de maneira análoga ao elitismo da versão geracional do algoritmo. Uma outra analogia possível com o Algoritmo Geracional é que um Algoritmo Steady State com esta estratégia de descarte pode ser encarado como um Algoritmo Genético Geracional com elitismo mantendo os $n - 1$ melhores indivíduos, onde n é o tamanho da população. Assim,

podemos considerar a versão Steady State do Algoritmo Genético com esta política de descarte como sendo um caso particular do Algoritmo Genético tradicional.

Uma outra forma de descarte comum é o descarte aleatório. Ainda que possa parecer estranho, este método se justifica pelo fato de manter a variabilidade genética, o que pode ser desejável em diversas situações, eliminando, com maior probabilidade, genes muito frequentes na população, melhorando a eficácia da exploração do espaço de busca. Uma estratégia menos agressiva, embora também aleatória, é fazer com que a probabilidade de descarte de um indivíduo seja inversamente proporcional à sua adaptação, de maneira semelhante ao método que utilizamos para a seleção de indivíduos para cruzamento. Conhecida como roleta invertida, esta técnica diminui as chances de descarte de indivíduos muito bons e é, de certa forma, um meio termo entre a estratégia totalmente aleatória e a estratégia gulosa de eliminação do pior indivíduo. Analogamente, outros métodos de seleção podem ser adaptados para a utilização na seleção de indivíduos para descarte.

Finalmente, a última forma de descarte de indivíduos que citaremos é a de eliminação do indivíduo mais antigo. Entre as motivações do uso desta estratégia de descarte podemos destacar seu comportamento em ambientes dinâmicos, foco deste trabalho, evitando que um indivíduo se reproduza demasiadamente acelerando uma convergência genética indesejada para um mínimo local. Além disso, cada indivíduo terá a oportunidade de se reproduzir $2n$ vezes antes de ser descartado, uma vez que são sorteados dois pais a cada reprodução. Supondo um indivíduo mediano, cuja probabilidade de seleção é igual a $1/n$, obtemos uma probabilidade de não ser selecionado igual a

$$\left(1 - \frac{1}{n}\right)^{2n} \leq \left(\frac{1}{e}\right)^{2} \approx 0.1353,$$

valendo a igualdade no limite de n no infinito. Assim, a chance de um indivíduo mediano se reproduzir é superior a 85% e, obviamente, indivíduos mais adaptados terão ainda mais chances de reprodução. Assim, apesar da possibilidade de descarte de um

bom indivíduo, sabemos que esta perda não será muito grande, uma vez que muito provavelmente seus genes já estarão presentes em outros indivíduos da população. Desta forma, um indivíduo não se reproduzirá demasiadamente, por ter no máximo $2n$ oportunidades, e tampouco terá seus genes descartados prematuramente, por ter tido chances de reprodução.

Com a reprodução entre indivíduos de diferentes gerações (incluindo, possivelmente, cruzamentos entre pais e filhos, fenômeno comum em algumas espécies), é facilitada a dominação dos melhores esquemas (conjunto de genes) dentro da população. Com isso acelera-se a convergência. Assim, este tipo de estratégia é recomendada em aplicações onde o tempo de convergência é crucial, como agentes inteligentes em jogos eletrônicos. Diferenças entre as versões geracionais e incrementais de algoritmos genéticos e a robustez de ambos são abordadas em (JONES; SOULE, 2006).

De acordo com (MELANIE, 1998), Algoritmos Genéticos Steady State são comumente utilizados em sistemas baseados em regras, como sistemas classificadores (GEYER-SCHULZ, 1995), onde o aprendizado incremental é importante e os membros da população resolvem o problema coletivamente, de maneira cooperativa. Sendo assim, em jogos onde um time de agentes coopera em função de um objetivo comum, esta variação de algoritmos genéticos tende a ser uma opção interessante para nosso contexto.

Como citado anteriormente, uma convergência muito veloz impede uma boa exploração do espaço de busca. Uma maneira simples de se evitar este tipo de problema é aumentando a taxa de mutação. Esta operação, no contexto de jogos, especificamente, evita ainda que os agentes se comportem de maneira determinística e torna possível a adaptação a situações não vivenciadas anteriormente.

Podemos encarar o trabalho de Demasi publicado em (DEMASI; OLIVEIRA CRUZ,

2003) e em sua dissertação de mestrado (DEMASI, 2003) como sendo uma implementação de um Algoritmo Genético Steady State com a estratégia de descarte do indivíduo menos adaptado com um ajuste de velocidade na velocidade de convergência (de forma que a dificuldade para o jogador não aumente muito rapidamente). Embora esse mecanismo de ajuste de dificuldade evite que o jogo fique difícil demais, ele não garante que a evolução ocorra rápido o suficiente, por conta da dependência da aleatoriedade dos algoritmos genéticos. Assim, se pudermos desenvolver mecanismos que acelerem esta evolução, poderemos combiná-los com outras formas de controle de velocidade, controlando melhor o ritmo das melhorias dos agentes. Da mesma maneira que um adversário impossível de ser vencido (ou difícil o suficiente para o jogador desistir de enfrentá-lo mesmo antes de alcançar a perícia necessária) não é desejável, um agente com ações demasiadamente burras cuja evolução seja lenta demais pode fazer com que o jogador desista de enfrentá-lo antes de o agente ter o tempo necessário para evoluir. Em outras palavras, para tornar um jogo divertido, a evolução do agente não pode ser lenta nem rápida demais.

Embora ambos os problemas sejam interessantes, evitar que o jogo fique difícil pode ser considerado trivial pois podemos simplesmente parar a evolução (como sugerido por Demasi), embora a avaliação do nível de dificuldade em cada instante seja um problema para a tomada de decisão sobre a continuidade da evolução ou não. Por outro lado, a evolução deve ocorrer após uma pequena quantidade de interações entre agentes e jogador (SPRONCK; SPRINKHUIZEN-KUYPER; POSTMA, 2003), o que pode ser uma limitação para o uso de métodos evolutivos. Apesar disso, métodos evolutivos já foram utilizados com sucesso em jogos simples (DEMASI, 2003) e (CROCOMO, 2008), o que nos motiva a realizar novos estudos neste tema. No próximo capítulo, apresentamos uma variação do Algoritmo Genético Steady State que se propõe a melhorar a velocidade de convergência bem como os resultados finais obtidos pelo uso de Algoritmos Evolucionários em ambientes dinâmicos, como aqueles encontrados em jogos eletrônicos.

4 PROPOSTAS

Como visto no início do capítulo 2, Spronck propõe uma lista de pré-requisitos necessários a algoritmos que evoluam ou aprendam em tempo real. Ainda segundo ele, algoritmos evolutivos não satisfazem duas propriedades, a da efetividade, que diz que na maior parte das vezes o comportamento do indivíduo deve ser bom, e da eficiência, que exige uma rápida convergência (SPRONCK; SPRINKHUIZEN-KUYPER; POSTMA, 2003).

Demasi utiliza com sucesso métodos evolutivos, com eficiência superior à ideal para um bom balanço de dificuldade (DEMASI, 2003). Já o trabalho de Crocomo, consistia na realização de experimentos específicos para mostrar que métodos evolucionários podem ser usados com sucesso atendendo aos requisitos citados por Spronck (CROCOMO, 2008), experimentos estes que obtiveram sucesso com resultados superiores aos obtidos pela técnica defendida por Spronck. De acordo com Michalewicz e Fogel, os principais fatores para se obter boa performance de um mecanismo de aprendizado são a eliminação da aleatoriedade do problema em questão, diminuindo variações no ambiente, como variações climáticas ou de iluminação em um sistema de reconhecimento de padrões em imagens por exemplo, e utilização de conhecimentos específicos do domínio em que se está trabalhando (MICHALEWICZ; FOGEL,

2000). A aleatoriedade presente nos jogos está diretamente relacionada à criatividade do jogador, bem como sua liberdade de ação, não havendo formas diretas de melhorar o desempenho dos algoritmos sem comprometer a jogabilidade. Já a escolha da representação utilizada em Algoritmos Genéticos está diretamente relacionada ao conhecimento do domínio em questão e é fundamental para o sucesso do algoritmo (LINDEN, 2006). Assim, deve-se buscar uma codificação satisfatória que possibilite que os Algoritmos Genéticos obtenham bons resultados. Crocomo utiliza uma representação que leva em consideração conhecimentos específicos sobre o jogo em questão e certamente este é um dos fatores fundamentais para garantir a efetividade. Já a eficiência depende do tamanho do espaço de busca, da irregularidade da função objetivo e da representação utilizada e, baseado nos resultados de Demasi e Crocomo, podemos reafirmar que algoritmos evolutivos podem ser utilizados com sucesso em jogos eletrônicos.

Além de serem realmente aplicáveis a jogos eletrônicos, em alguns casos eles se fazem necessários. Os algoritmos evolutivos apresentam a capacidade necessária para atuar em áreas onde as tradicinais técnicas de Inteligência Artificial apresentam deficiências (YANNAKAKIS, 2005).

As propostas apresentadas neste capítulo se propõem a melhorar os métodos atuais de evolução em problemas que requerem evolução em tempo real em ambientes não estacionários, ou seja, problemas que possuem funções objetivo que se alteram ao longo do tempo. Embora a motivação para este trabalho seja a evolução em jogos eletrônicos, a aplicabilidade das propostas apresentadas não se restringe a problemas deste domínio.

4.1 Evolução e Adaptação em tempo real

Uma das dificuldades ao se utilizar métodos evolutivos em jogos eletrônicos é a avaliação do indivíduo. Em problemas de otimização tradicionais, onde algoritmos genéticos são comumente utilizados, a função de avaliação dada por $f(x)$, onde x é um indivíduo qualquer, é conhecida e utilizada diretamente no algoritmo. Já no nosso contexto, essa função, além de inexistente explicitamente, está sujeita à interação com o jogador, dependendo de valores obtidos nesse ambiente. Um exemplo torna esta situação mais clara. Suponha que estamos avaliando o desempenho de agentes de um jogo de ação onde cada jogador têm uma certa quantidade de munição e de energia. O valor da avaliação de um indivíduo pode ser definido em função de diversos parâmetros. A quantidade de munição gasta durante o combate, a quantidade de energia restante do agente (em caso de vitória) ou do adversário (em caso de derrota) e a duração do combate são candidatos naturais a variáveis da função de avaliação.

Por melhor que sejam escolhidos os parâmetros e seu papel na função de avaliação, esta função estará sempre sujeita a anomalias comportamentais do jogador. O jogador pode, por exemplo, se distrair em determinado momento do jogo, fazendo com que o desempenho do agente seja avaliado acima do esperado. Esta situação é muito difícil de ser identificada em ambientes de informação escassa como jogos. Ainda assim, em geral os algoritmos evolutivos são robustos o suficiente para lidar com esse fenômeno de maneira satisfatória, ao contrário de métodos baseados em derivada, por exemplo, por explorarem diversas regiões do espaço de busca simultaneamente e por conta da aleatoriedade presente no algoritmo. Outro problema que não ocorre em problemas de otimização tradicionais, mas que acaba surgindo no contexto dos jogos eletrônicos, é a evolução do próprio jogador. A rigor, isso caracteriza a nossa situação como coevolução, mas, por simplicidade, os termos coevolução e evolução são intercambiáveis no nosso contexto, pois estamos sempre nos referindo à evolu-

ção dos agentes em relação a um jogador específico, a não ser quando especificado o contrário. Em situações onde as populações de gerações distintas não interagem, a evolução do jogador pode ser desconsiderada, uma vez que cada população é gerada em um espaço finito de tempo. Porém, em situações onde indivíduos de diferentes gerações podem pertencer à mesma população ou onde o conceito de geração não se aplica, ou ainda em situações onde o algoritmo genético utiliza elitismo, a evolução do jogador pode causar problemas, como veremos no exemplo a seguir.

Suponha que um agente ruim, no início da interação entre jogador e agentes e, portanto, com o jogador ainda inexperiente, tenha recebido um valor muito alto como avaliação (ou seja, tenha obtido um desempenho bom contra o jogador). Este agente, por ter uma boa avaliação, pode acabar passando seus genes para muitos indivíduos das próximas gerações, mesmo sendo um agente ineficaz. Esta situação pode perdurar atrasando ou mesmo impedindo a evolução. Para tentar resolver este problema, proporemos uma maneira de amenizar essa interferência.

4.1.1 Modelagem do Aprendizado

A primeira observação que podemos fazer é quanto a forma de evolução da perícia do jogador humano. Em geral, todo jogador, ao conhecer as regras e controles do jogo, apresenta uma habilidade inicial, que representaremos aqui por um valor r_i. Além disso, no início da interação com o jogo, o jogador usualmente aprende com uma taxa relativamente elevada. Quanto mais tempo o jogador passa exercitando suas habilidades, mais ele aprende, ou seja, maior sua habilidade fica. Com um tempo de treinamento infinito, podemos supor que o jogador atinge a perfeição. Por outro lado, à medida que o jogador evolui, mais difícil se torna evoluir, por se encontrar mais próximo da perfeição. Dessa forma, a velocidade com que o jogador evolui é proporcional a sua distância da perfeição. Representando a perfeição como

o valor 1 e a habilidade do jogador no instante t como $r(t)$, podemos dizer que a velocidade de aprendizado representada pela derivada $r'(t)$ é proporcional a $1-r(t)$. Desprezando-se constantes multiplicativas, obtemos

$$r'(t) = 1 - r(t),$$

cuja solução pode ser dada por

$$r(t) = 1 - ke^{-t},$$

onde k é uma constante. Para definirmos o valor de k, fazemos

$$r(0) = 1 - ke^0 = 1 - k = r_i,$$

onde r_i é o valor correspondente à perícia inicial do jogador, resultando em $k = 1-r_i$ e obtendo

$$r(t) = 1 - (1 - r_i)e^{-t}.$$

Para ajustar a velocidade com que o jogador aprende, inserimos a variável λ, uma constante multiplicativa desprezada anteriormente, gerando a forma final de nossa equação

$$r(t) = 1 - (1 - r_i)e^{-\lambda t}.$$

Suponha agora, que exista uma função $f(x)$ que nos dá a avaliação de um agente x quando confrontado com um jogador com habilidade máxima. Definimos então a função $g(x,t)$ como sendo

$$g(x,t) = \frac{f(x)}{r(t)}.$$

Dessa forma, o valor que obtemos através da simulação é dado pela função $g(x,t)$, ou seja, o agente, quando confrontado com um oponente inexperiente, receberá uma avaliação alta, uma vez que o denominador será menor.

Observe que

$$\lim_{t\to\infty} r(t) = 1$$

e, portanto,

$$\lim_{t \to \infty} g(x,t) = f(x),$$

ou seja, para valores de t altos, a componente de tempo se torna desprezível.

Embora esta modelagem aproxime a maneira como a evolução ocorre durante o jogo, mesmo que ela fosse infalível ainda precisaríamos estimar os valoresr_i e λ, o que não pode ser feito facilmente. Caso fosse possível este tipo de estimativa, poderíamos, a partir destes valores e de t, obter o valor de $r(t)$ em um determinado instante de tempo e, como consequência, utilizando valores de $g(x,t)$ obtidos experimentalmente, poderíamos estimar o valor real de $f(x)$ dissociando totalmente a avaliação de um agente da habilidade de um oponente.

Considerando estas restrições de informação, a imprecisão do valor de$g(x,t)$ obtido e até mesmo imperfeições do próprio modelo, não podemos utilizar uma solução analítica como a descrita no parágrafo acima. Isso nos motiva a desenvolver um novo método que leve em consideração as características apresentadas. Além disso, esperamos obter um modelo que também se comporte de maneira satisfatória quando o jogador alterar sua estatégia durante o jogo, de forma que o agente consiga se adaptar a essas mudanças. Ainda assim, utilizaremos, nos testes, o modelo descrito acima, para simular a situação de aprendizado do jogador, por sua verossimilhança.

4.1.2 Algoritmo Genético Steady State Adaptativo - AGSSA

O algoritmo aqui proposto, busca levar em consideração o fato de que o jogador pode evoluir com o tempo. Dentro deste contexto, podemos esperar que uma avaliação feita muitas iterações atrás tenha uma avaliação superestimada, ou seja, menos confiável, enquanto uma avaliação feita recentemente terá uma valor bem próximo da avaliação real. É importante ressaltar que estamos considerando contextos onde não

é possível reavaliar um indivíduo, uma vez que isso requereria um novo confronto contra o adversário. Dessa forma, nosso objetivo é aproveitar de alguma forma a informação da avaliação anterior. Como estamos supondo que o jogador está evoluindo, uma avaliação antiga possivelmente teria um valor acima do real, uma vez que o agente teria confrontado um adversário menos experiente. Da mesma forma, em um jogo onde um jogador pode alterar sua estratégia, uma avaliação feita anteriormente pode ter avaliado o jogador em um momento em que outra estratégia estava sendo utilizada e, portanto, seu desempenho pode ser inferior quando confrontado com a estratégia atual. Obviamente, o desempenho do agente contra a estratégia atual poderia até mesmo ser superior ao obtido anteriormente, porém, não temos como prever este comportamento e, por isso, não podemos elevar o valor da avaliação de um agente avaliado previamente. Além disso, em determinadas situações podemos supor que este fato ocorrerá com frequência pequena pois o jogador dificilmente mudará sua estratégia para uma outra que seja inferior do que a atual contra um adversário que ele tenha confrontado recentemente.

Considerando esta defasagem na avaliação dos agentes, propomos uma maneira de compensar este problema. Nosso ponto de partida será a substituição da função de avaliação original $f_0(x_i)$ por uma função que leve em consideração o tempo t_i no qual o indivíduo foi gerado. Para isso, utilizaremos uma função auxiliar $h(t_i, t)$ a ser multiplicada pelo valor da avaliação original. Temos então nossa nova função

$$f(x_i, t) = f_0(x_i)h(t_i, t),$$

onde x_i é o indivíduo que estamos avaliando, t é o instante de tempo atual, t_i é o instante de tempo em que o individuo x_i foi gerado e $f_0(x_i)$ é a avaliação recebida pelo indivíduo x_i no momento de sua geração.

O ponto importante é como definir a função $h(t_i, t)$. Diversas são as possibilidades e esta função desempenha papel importantíssimo na eficácia do Algoritmo Genético. Quando utilizamos $h(t_i, t) = c$, obtemos $f(x_i, t) = cf_0(x_i)$, onde c é uma constante,

e o comportamento do algoritmo será idêntico ao Algoritmo Steady State padrão. Para nosso problema em questão, definiremos

$$h(t_i, t) = b^{t-t_i},$$

sendo $0 \leq b \leq 1$ e $t \geq t_i$. Esta escolha foi feita pela natureza da função exponencial em questão, com um decaimento a cada instante de tempo, mas sem nunca chegar a zero. No caso onde $b = 0$, todos os indivíduos terão sua avaliação nula e portanto a probabilidade de serem selecionados tanto para cruzamento quanto para descarte será dada por uma distribuição uniforme, tornando a busca totalmente aleatória. No outro extremo, quando $b = 1$, temos $f(x_i, t) = f_0(x_i)$, com comportamento análogo ao anterior onde $h(t_i, t) = c$. Para valores intermediários, indivíduos mais velhos terão sua avaliação diminuída. Este comportamento é interessante porque apenas indivíduos que tiveram um bom desempenho e tenham sido avaliados recentemente terão uma boa avaliação global tendo, portanto, maiores chances de reprodução. Por outro lado indivíduos que tiveram uma avaliação boa em uma situação anterior ainda terão um valor razoavelmente alto, tendo, analogamente, chances razoáveis de reprodução. Este característica é desejável por sabermos que uma solução que era boa em um determinado momento não passa a ser uma péssima solução em poucas gerações, podendo ter, portanto, características interessantes para a função atual. Por outro lado, indivíduos ruins não passam a ser indivíduos muitos bons de um instante para outro, sendo interessante que suas avaliações continuem baixas tendo eles sido gerados recentemente ou não.

O algoritmo 4.1 detalha os passos do método proposto por este trabalho. A única diferença deste algoritmo para o algoritmo 3.2 é a adição da linha 3, que atualiza as avaliações. Para que disponhamos sempre dos valores de avaliação no instante de tempo corrente, basta que a cada iteração o valor de avaliação de todos os indivíduos seja multiplicado pela constante b. Apesar de explicitarmos este passo nesse algoritmo, ele pode ser implementado de uma maneira diferente, mais eficiente, como veremos no capítulo 5.

Algoritmo 4.1 Algoritmo Genético Steady State Adaptativo

```
1: gerar população inicial
2: enquanto não satisfizer condição de parada faça
3:     atualiza avaliação dos indivíduos
4:     seleciona par de pais de acordo com sua aptidão
5:     gera um filho a partir de cruzamento
6:     efetua possíveis mutações
7:     descarta um indivíduo da população
8:     adiciona novo individuo à nova população
9:     avalia novo indivíduo
10: fim de enquanto
```

Uma diferença significativa na ação deste algoritmo é a velocidade de resposta a uma mudança na função de avaliação (decorrente, por exemplo, de uma mudança no ambiente ou, no contexto de jogos, de uma mudança de comportamento do jogador). Na situação tradicional, uma avaliação superestimada de determinados indivíduos (que em um momento anterior eram considerados bem adaptados ao ambiente e portanto receberam valores altos de avaliação) pode fazer com que esses indivíduos se reproduzam demasiadamente, guiando a busca para regiões do espaço onde não há boas soluções. Com a alteração proposta, este tipo de indivíduo passa a ter sua importância reduzida com o passar do tempo.

Uma outra mudança que melhora consideravelmente o desempenho do Algoritmo Steady State tradicional é o uso da estratégia de descarte do indivíduo mais velho, como apresentado em (VAVAK; FOGARTY, 1996). Dessa forma, gradativamente os indivíduos com avaliações errôneas são eliminados da população e a busca guia-se pelos indivíduos mais recentes e suas avaliações. No capítulo 6 apresentaremos uma comparação do desempenho do Algoritmo Steady State com a versão Adaptativa com a utilização das estratégias de descarte do indivíduo mais velho e do indivíduo com a pior avaliação. Uma vantagem clara do Algoritmo Adaptativo em relação ao tradicional com a estratégia de descarte é a velocidade de reação do algoritmo às mudanças. Enquanto o algoritmo com a estratégia de descarte dos mais antigos de-

pende da eliminação dos indivíduos da população para se adaptar ao novo contexto, o algortimo adaptativo consegue fazer isso gradativamente, ao reduzir a avaliação dos indivíduos mais antigos. Outro aspecto interessante é que, com a versão adaptativa proposta, a idade do indivíduo é levada em consideração também na estratégia de descarte do indivíduo menos apto. Assim os dois critérios, idade e adaptação, passam a ser levados em consideração no momento do descarte. Vale ainda ressaltar que a modificação proposta pode ser utilizada com qualquer uma das estratégias de descarte apresentadas na seção 3.2.

4.1.3 Outras Aplicações

Como dito anteriormente, a função $h(t_i, t)$ é muito importante para um bom funcionamento do algoritmo. Alguns exemplos foram citados acima, como casos onde a função poderia tornar o algoritmo uma busca aleatória, uma maneira de acelerar a convergência com a proposta anterior e também uma forma de simplesmente torná-la neutra, fazendo com que o Algoritmo Genético de comporte de forma padrão. Para ilustrar outros usos desta função, apresentaremos três exemplos adicionais. O primeiro deles inspirado no ciclo de reprodução na natureza. O segundo tenta evitar uma convergêcia genética precoce e ainda um terceiro que transforma um Algoritmo Genético Steady State em um Geracional, mostrando que, assim como o Steady State pode ser considerado um caso particular do Geracional (utilizando elitismo de $n - 1$ indivíduos), a recíproca pode ser considerada verdadeira.

4.1.3.1 Reprodução Verossímil

Uma característica que pode causar certa estranheza nas versões Steady State do algoritmo é a reprodução de um indivíduo recém-gerado. Fazendo o paralelo com a

natureza, seria como se um indivíduo jovem demais estivesse gerando descendentes. Sabe-se que na natureza, a idade da primeira reprodução dos mamíferos, por exemplo, tende a diminuir impulsionada por uma baixa expectativa de vida (HARVEY; ZAMMUTO, 1985). Ainda assim, existe uma idade mínima referente à maturidade física dos indivíduos. Para tornar o algoritmo genético mais parecido com a natureza, podemos tornar a reprodução de indivíduos jovens menos provável. Da mesma forma, após determinada idade as chances de reprodução diminuem (ainda que possam depender do sexo do indivíduo) e, para uma simulação verossímil devemos diminuir também suas chances de reprodução. Para isso, uma possível função a ser utilizada é uma gaussiana dada por

$$h(t_i, t) = e^{-\frac{[(t-t_i)-c]^2}{2\sigma^2}},$$

onde c é o centro da função, ou seja, a idade onde a chance de reprodução encontra seu máximo e σ é a variância, que define a velocidade de decaimento da função ao se afastar do centro. Um exemplo de gráfico desta função pode ser visto na figura 4.1.

Em muitas situações não existe apenas uma idade onde as chances de reprodução são maiores e sim uma faixa de idades $[c_i, c_f]$ onde todos os indivíduos tem altas chances de reprodução. Além disso, é bastante provável que a função não seja simétrica. Para dar uma maior flexibilidade podemos usar uma função baseada na combinação de duas Gaussianas com parâmetros c_1, σ_1 e c_2, σ_2 respectivamente, dada por

$$h(t_i, t) = \begin{cases} e^{-\frac{[(t-t_i)-c_i]^2}{2\sigma_i^2}} & \text{se} \quad (t-t_i) < c_i \\ 1 & \text{se} \quad c_i \leq (t-t_i) \leq c_f \\ e^{-\frac{[(t-t_i)-c_f]^2}{2\sigma_f^2}} & \text{se} \quad (t-t_i) > c_f \end{cases}$$

e que, assim como no caso anterior, é contínua e derivável e, consequentemente, suave. Um exemplo de gráfico para essa função pode ser visto na figura 4.2.

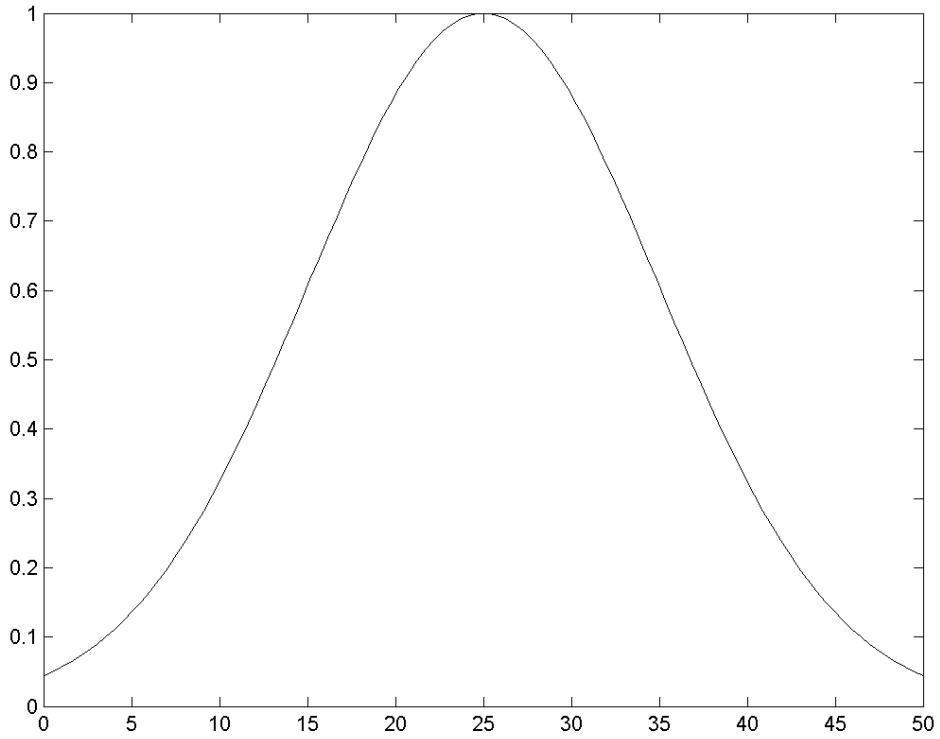

Figura 4.1: Exemplo de Função Gaussiana com $c = 25$ e $\sigma = 10$

4.1.3.2 Reprodução Tardia

Outra aplicação para a função $h(t_i, t)$ é seu uso para atrasar a convergência genética. Ao possibilitar que um indivíduo recém-gerado se reproduza, caso este indivíduo seja muito bom, ele pode acabar se reproduzindo excessivamente, forçando uma convergência precipitada. Como dito anteriormente, este fenômeno normalmente é indesejado pelo alto risco de conduzir a busca a um mínimo local indesejado prematuramente. Uma maneira de se evitar este problema é evitando que indivíduos recém-gerados se reproduzam. Para isso, basta diminuir sua avaliação e, por consequência, suas chances de reprodução. Para isso, uma possível função é a função

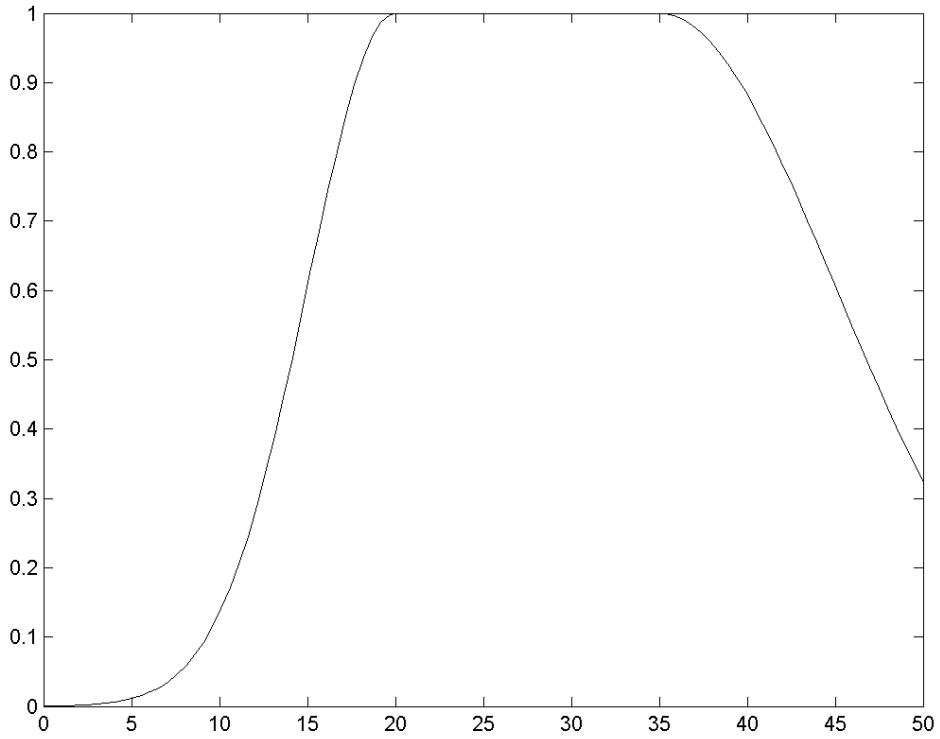

Figura 4.2: Exemplo de Combinação de Gaussianas com $c_1 = 20$, $\sigma_1 = 5$, $c_2 = 35$ e $\sigma_2 = 10$ representando as chances de reprodução dos indivíduos em função das idades

sigmóide, definida, em nosso contexto, como

$$h(t_i, t) = \frac{1}{1 + e^{-\lambda[(t-t_i)-c]}},$$

onde c é centro da função, onde ela assume o valor $h(t_i, t) = 0,5$ e λ é uma constante responsável pela inclinação da curva. A figura 4.3 nos dá um exemplo de gráfico para esta função.

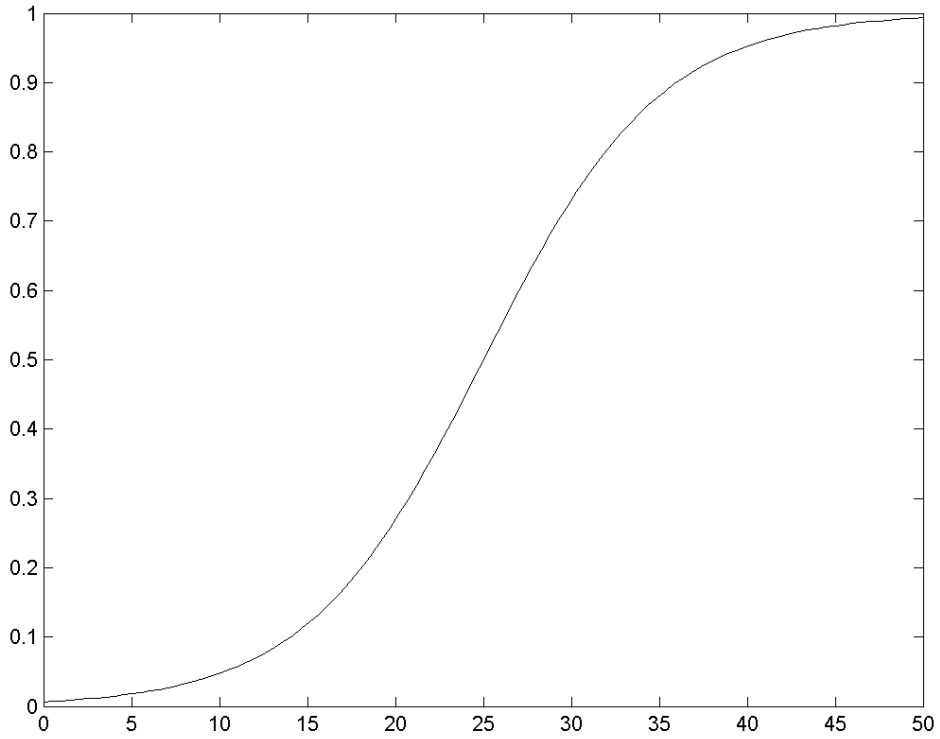

Figura 4.3: Exemplo de função sigmóide com centro $c = 25$ e inclinação $\lambda = 0.2$

4.1.3.3 Transformando Steady State em Geracional

O último exemplo de uso da função $h(t_i, t)$ é para uma uso um tanto inusitado. A idéia é que, a partir de um Algoritmo Genético Steady State, possamos simular o comportamento de um Algoritmo Geracional. Sendo n o tamanho da população do algoritmo original, simularemos o comportamento de um algoritmo geracional com uma população de $n' = n/2$ indivíduos. Para isso utilizaremos a função

$$h(t_i, t) = \begin{cases} 1 & \text{se} \quad n' \leq n' \left\lceil \dfrac{t}{n'} \right\rceil - t_i < n \\ \\ 0 & \text{caso contrário} \end{cases}$$

que serve para decidir se um indivíduo tem chance de reprodução ou não. Além desta função, devemos também utilizar a estratégia de descarte do indivíduo mais velho, para que possamos simular corretamente o Algoritmo Genético Geracional.

Com esta função, o algoritmo funcionará da seguinte forma. Após a geração da população inicial de n indivíduos, os primeiros $n' = n/2$ começarão a ser eliminados para dar lugar a novos indivíduos gerados. Através da função $h(t_i, t)$, a população será dividida em dois subconjuntos, um deles correspondendo a $h(t_i, t) = 0$ contendo os indivíduos mais velhos (que serão descartados gradativamente) e os mais novos (que substituem os mais velhos), e outro subconjunto contendo os indivíduos com $h(t_i, t) = 1$, que são aqueles que se reproduzirão. Após n' iterações, este primeiro subconjunto terá sido totalmente renovado, com indivíduos novos e, por conta da função "teto" na função $h(t_i, t)$, a situação se alternará, com o segundo subconjunto sendo utilizado para os novos indivíduos e o primeiro subconjunto sendo o conjunto dos pais que darão origem a novos filhos. Assim, a cada n' iterações a situação se alterna. A tabela 4.1 apresenta a relação entre os parâmetros do algoritmo sendo executado de fato e do comportamento simulado.

Tabela 4.1: Comparação entre os parâmetros do algoritmo real e do simulado.

	Incremental	Geracional
Tamanho da População	n	$n' = n/2$
Indivíduos Úteis Gerados	m	$m - n'$
Total de Gerações	-	$m/(n' - 1)$

Uma última observação pertinente é que os n' primeiros indivíduos gerados pelo algoritmo não gerarão descendentes. Sendo assim, estes n' indivíduos são gerados desnecessariamente. Entretanto, a magnitude de n' quando comparada à de m é muito pequena, sendo este esforço desprezível, principalmente pelo fato de ser constante e independente da quantidade de gerações desejadas. Além disso, esta geração de indivíduos desnecessários é facilmente contornada na implementação.

5 IMPLEMENTAÇÃO EFICIENTE

Em um contexto onde um bom desempenho é crucial, um capítulo dedicado à implementação se mostra adequado. Não discutiremos aqui detalhes específicos de plataforma ou a implementação realizada para os testes do capítulo 6, mas questões gerais, que podem comprometer o desempenho assintoticamente.

5.1 Atualização das Avaliações

O primeiro ponto que surge com a proposta da seção 4.1.2 é a necessidade de atualizar a avaliação de todos os n indivíduos da população a cada iteração. Entretanto, como há outros passos de complexidade $O(n)$ referentes à seleção e ao descarte de indivíduos, poder-se-ia argumentar que uma melhora no desempenho neste passo não estaria diminuindo a complexidade assintótica. Porém, mais adiante mostraremos que mesmo as operações de seleção e descarte podem ser feitas de maneira mais eficiente. Além disso, como estamos lidando com evolução em tempo real em jogos, contexto de recursos preciosos e escassos, não podemos disperdiçar tempo de processador, sendo importante, portanto, qualquer ganho de desempenho.

Para que não precisemos efetuar operações com todas avaliações, podemos "atualizá-las" apenas no instante que elas forem utilizadas. Para isso, ao invés de multiplicar a avaliação de cada indivíduo pela constante b a cada iteração, podemos efetuar o cálculo no momento de sua utilização. Para evitar cálculos desnecessários, considerando que o valor de b é constante, podemos ainda utilizar uma estrutura auxiliar que armazena os valores das potências de base b, evitando a realização deste cálculo no decorrer do jogo. Esta estrutura pode consistir de um vetor simples, onde a sua i-ésima posição teria o valor b^i.

5.2 Melhorando a Eficiência do Algoritmo

Quando utilizado o método de seleção por torneio, a seleção pode ser feita eficientemente pois não é necessário percorrer a lista dos indivíduos. O problema surge então quando utilizamos o método da roleta.

Para determinar o indivíduo a ser selecionado, a estratégia mais frequente é varrer a lista de indivíduos e acumular as probabilidades até que sua soma ultrapasse o valor obtido na amostra da variável aleatória. Porém, no pior caso, todos os indivíduos serão visitados, nos dando uma complexidade de $O(n)$. Mesmo considerando o caso médio, a complexidade fica em $O(n)$, por serem percorridas, em média, $(n+1)/2$ posições da lista.

Surge, então, a necessidade de uma estrutura mais eficiente, onde possamos selecionar os indivíduos de maneira menos custosa. A primeira idéia que nos ocorre é a utilização de uma estrutura como uma árvore balanceada ou um heap, semelhantes aos descritos em (SZWARCFITER; MARKENZON, 1994). Porém, a cada iteração a probabilidade de seleção dos indivíduos é alterada, além de um indivíduo ser substituido. Portanto, para utilizar uma dessas estruturas devemos fazer alterações

em sua estrutura e comportamento de forma que se adequem às nossas necessidades.

Por ser de implementação mais simples, utilizaremos o heap, embora o procedimento também fosse de possível implementação em uma árvore balanceada, com mesma complexidade.

Um heap é uma estrutura utilizada para a implementação de listas de prioridades. Em processos de um computador, por exemplo, podemos precisar saber qual é o processo com maior prioridade em uma determinada lista de processos. Em situações como essa, são comuns dois tipos de operação. A primeira delas é a inserção, que consiste basicamente em adicionar um novo elemento à lista. A segunda é a remoção ou seleção do elemento com maior prioridade.

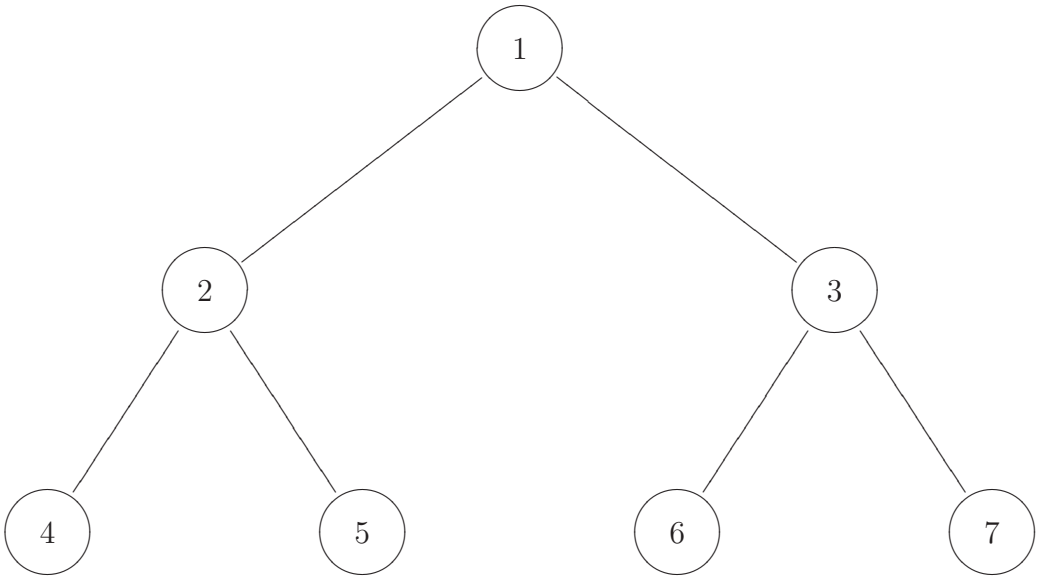

Figura 5.1: Representação gráfica de um heap em forma de árvore com os vértices numerados de acordo com sua posição no vetor.

Esta estrutura é normalmente visualizada como uma árvore binária completa com a propriedade de que os filhos de um nó sempre têm prioridade inferior à sua própria. A implementação desta estrutura é normalmente feita utilizando-se um vetor onde na primeira posição encontra-se a raiz da árvore, em seguida os nós do segundo nível da árvore, seguidos pelos nós do terceiro nível e assim sucessivamente. A figura 5.1 relaciona a posição de um elemento do vetor com sua posição na árvore.

Na operação de remoção de um elemento do heap, a sua raiz é removida e em seu lugar é colocado o elemento de maior índice (o último elemento da sua representação vetorial). Neste momento, faz-se necessário o ajuste das posições dos elementos na estrutura pois o indivíduo que passou a ocupar a raiz não é necessariamente o de maior prioridade (e de fato não o será caso a árvore tenha três ou mais níveis). Para este ajuste é utilizado o algoritmo 5.1.

Algoritmo 5.1 Descendo um elemento no heap

```
Desce(i)
  j ⇐ 2i
  se j ≤ n então
    se j < n então
      se v[j + 1].p > v[j].p então
        j ⇐ j + 1
      fim de se
    fim de se
    se v[i].p > v[j].p então
      temp = v[i]
      v[i] = v[j]
      v[j] = temp
      Desce(j)
    fim de se
  fim de se
```

O algoritmo basicamente compara a prioridade do elemento com cada um de seus filhos caso existam. Caso algum deles tenha a prioridade maior que a do elemento em questão, estes trocam de lugar e o algoritmo é chamado recursivamente para

a nova posição que o elemento ocupa. O vetor v é um vetor de tamanho n que armazena as informações dos elementos e $v[i].p$ é a prioridade do i-ésimo elemento do vetor.

Para a inserção de um novo indivíduo o procedimento é de certa forma semelhante, porém de natureza oposta. Um novo indivíduo é inserido na primeira posição após a última posição ocupada. Neste caso também se faz necessário um procedimento que verifique as prioridades, fazendo com o elemento suba na árvore até alcançar a posição adequada, como descrito no algoritmo 5.2.

Algoritmo 5.2 Subindo um elemento no heap

```
Sobe(i)
  j ⇐ ⌊i/2⌋
  se j ≥ 1 então
    se v[i].p > v[j].p então
      temp = v[i]
      v[i] = v[j]
      v[j] = temp
      Sobe(i)
    fim de se
  fim de se
```

Observe que, embora estejamos falando de prioridades, esta estrutura serve para qualquer problema onde a operação de remoção de interesse seja sempre do indivíduo com maior ou menor valor em alguma característica. Para diferenciá-los, costuma-se utilizar as denominações "Heap de máximo" ou "Heap de mínimo", respectivamente.

A operação de remoção está diretamente relacionada à escolha do indivíduo a ser descartado. Porém, para isso é necessário adequar o campo "prioridade" ao tipo de estratégia de descarte utilizado. Para o descarte do indivíduo mais velho, a prioridade deve estar relacionada à idade do indivíduo. Já para o descarte do menos apto, a própria avaliação poderia estar relacionada à prioridade. Entretanto, como no algoritmo proposto as avaliações não são estáticas, sendo modificadas com o

passar do tempo, torna-se necessário acrescentar algumas informações à estrutura, porém sem alterar seu funcionamento básico. Definiremos, então, cada elemento tendo os campos descritos na tabela 5.1.

Tabela 5.1: Campos presentes em cada elemento e seu significado

cromossomo	Cromossomo do indivíduo
t	Tempo no qual o indivíduo foi avaliado pela última vez
a	Avaliação recebida no tempo t
t_{esq}	Tempo no qual a subárvore esquerda foi atualizada pela última vez
a_{esq}	Avaliação da subárvore esquerda no tempo t_{esq}
p	Critério de prioridade do heap. Possivelmente uma função de a e t

Agora que definimos quais as informações serão necessárias, podemos detalhar o funcionamento da estrutura modificada para as operações de interesse: a seleção e o descarte.

5.2.1 Seleção

Embora a natureza de um heap não ofereça operações de seleção de indivíduos que não sejam o de maior prioridade, com o uso de das informações adicionais presentes na tabela 5.1 é possível efetuar estas operações de maneira eficiente, sem que seja necessário visitar cada indivíduo do heap. Para que isto seja possível, é necessária uma informação adicional, mantida globalmente e não em um nó, referente ao total as avaliações dos indivíduos presentes na árvore. Esta informação é importante para o cálculo da probabilidade que, como visto no capítulo 3, é calculado como

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}.$$

No nosso caso particular, considerando a versão adaptativa do algoritmo, obtemos

$$p_i = \frac{f(x_i, t)}{\sum_{j=1}^{n} f(x_j, t)}.$$

Para evitar o recálculo do somatório, utilizaremos uma variável *soma*, que armazenará seu valor. A cada iteração, devemos atualizar seu valor de maneira simples. Primeiramente, seu valor é multiplicado pela constante b, uma vez que cada indivíduo tem sua avaliação alterada pelo mesmo fator. Após esta operação, devemos subtrair o valor da avaliação do indivíduo descartado, facilmente obtido na raiz da árvore, e adicionar o valor da avaliação do indivíduo que o substituiu.

Para a seleção de um indivíduo para a reprodução, inicialmente é sorteado um valor aleatório p, de acordo com uma distribuição uniforme no intervalo $[0, 1]$. A seguir, é chamada a função Seleção$(1, p)$, descrita no algoritmo 5.3 que retornará o indivíduo desejado. Os parâmetros da função são o nó a ser visitado e uma variável associada à probabilidade.

Algoritmo 5.3 Seleção de um indivíduo no heap

```
Seleção(i, p)
  se p < (v[i].a_esq × h(v[i].t_esq, t))/soma então
    retorna Seleção(2i, p) {Subárvore Esquerda}
  senão se p < (v[i].a_esq × h(v[i].t_esq, t) + v[i].a × h(v[i].t, t))/soma então
    retorna i {O indivíduo desejado é o próprio indivíduo i}
  senão
    retorna Seleção(2i+1, p−(v[i].a_esq × h(v[i].t_esq, t)+v[i].a × h(v[i].t, t))/soma)
    {Subárvore Direita}
  fim de se
```

O algoritmo 5.3 inicialmente olha a raiz da árvore. Como a variável $v[i].a_{esq}$ armazena o total das avaliações na subárvore esquerda do nó i no instante de tempo $v[i].t_{esq}$, sabemos que a avaliação no instante atual pode ser dada por $v[i].a_{esq} \times$

$h(v[i].t_{esq}, t)$. Sabemos ainda que a probabilidade de o individuo procurado estar nesta subárvore pode ser dada por $v[i].a_{esq} \times h(v[i].t_{esq}, t)/soma$. Assim, o primeiro "se" do algoritmo decide se o indivíduo desejado se encontra na subárvore esquerda. O segundo "se" por sua vez, efetua cálculo semelhante, porém incluindo também o indivíduo na posição i. E, finalmente, caso o indivíduo não esteja na subárvore esquerda nem seja o próprio nó que estamos avaliando, a única opção que resta é a subárvore direita.

Podemos observar facilmente que a altura da árvore completa representada no heap é sempre $O(\log n)$. Observamos também que as operações efetuadas nas comparações do algoritmo 5.3 são todas de tempo constante. Como no pior caso o algoritmo irá iniciar na raiz e percorrer a árvore até uma das folhas, podemos afirmar que sua complexidade de pior caso é $O(\log n)$.

5.2.2 Descarte

Como dito anteriormente, em um heap o elemento de maior (ou menor) prioridade sempre se encontra na raiz. Dessa forma, a escolha do indivíduo a ser descartado ocorre de maneira trivial. Além disso sabemos que, ao remover um indivíduo, devemos substituí-lo pelo indivíduo de maior índice da árvore. Porém, como no problema em questão uma operação de descarte é sempre seguida por uma operação de inserção, podemos inserir o novo indivíduo gerado diretamente na raiz da árvore. Após esta operação, ajustes semelhantes aos descritos nos algoritmo 5.1 se fazem necessários. Estes ajustes são descritos a seguir.

5.2.3 Inserção

Ao efetuar a operação de descida no heap para ajuste de prioridades, faz-se necessária uma atenção especial em relação ao conteúdo das variáveist_{esq} e a_{esq}. Estes campos dependem da posição relativa dos nós na árvore e, portanto, devem ser atualizados conforme as alterações de posição dos nós forem sendo realizadas. Uma versão revisada do algoritmo 5.1, atualizada para satisfazer nossos propósitos e levando estes aspectos em consideração é apresentada no algoritmo 5.4.

O algoritmo realiza basicamente duas operações. A primeira delas consiste na atualização das avaliações para o instante de tempo atual. Embora esta operação não seja fundamental (pois podemos sempre realizar o cálculo no momento de sua utilização) ela simplifica os cálculos posteriores e sua interpretação. A outra operação consiste em atualizar as informações sobre as subárvores. Esta atualização é relativamente simples pois em cada passo ocorre no máximo uma vez e apenas caso a troca ocorra entre um elemento e seu filho esquerdo. Com isso, conseguimos efetuar a remoção (descarte) de um indivíduo em tempo constante, ou seja,$O(1)$ e a inserção do novo indivíduo gerado em tempo $O(\log n)$, correspondente à altura da árvore.

A tabela 5.2 apresenta uma comparação entre as complexidades de tempo da versão tradicional do algoritmo e da versão proposta acima. A variável n se refere ao tamanho da população e a variável m se refere ao total de indivíduos gerados.

Observamos que o uso de uma estrutura de dados adequada pode melhorar consideravelmente o desempenho assintótico. Entretanto, considerando-se as constantes multiplicativas omitidas na notação O e também a dificuldade de implementação, a uso dessa estrutura não é indicado para populações muito pequenas. Por outro lado, para populações de tamanho médio ou grande, a diferença começa a ser notada. O limiar depende de detalhes de implementação e de ambiente. Para implementações

Algoritmo 5.4 Descendo um elemento no heap levando em consideração t_{esq} e a_{esq}

```
Desce2(i)
  j ⇐ 2i
  se j ≤ n então
    se j < n então
      se v[j + 1].p > v[j].p então
        j ⇐ j + 1
      fim de se
    fim de se
    se v[i].p > v[j].p então
      temp = v[i] {Troca-se os elementos de posição como anteriormente}
      v[i] = v[j]
      v[j] = temp
      v[i].a_esq = v[i].a_esq × h(v[i].t_esq, t) {Atualiza-se as avaliações}
      v[i].a = v[i].a × h(v[i].t, t)
      v[i].t_esq = t
      v[i].t = t
      v[j].a_esq = v[j].a_esq × h(v[j].t_esq, t)
      v[j].a = v[j].a × h(v[i].t, t)
      v[j].t_esq = t
      v[j].t = t
      se j = 2i então {j é filho esquedo de i}
        temp = v[j].a_esq − v[i].a + v[j].a
        v[j].a_esq = v[i].a_esq
        v[i].a_esq = temp
      senão {j é filho direito de i}
        temp = v[j].a_esq
        v[j].a_esq = v[i].a_esq
        v[i].a_esq = temp
      fim de se
      Desce(j) {Efetua-se novamente o processo de modo recursivo}
    fim de se
  fim de se
```

Tabela 5.2: Operações, sua complexidade em Algoritmos Genéticos tradicionais e com a estrutura de dados proposta

Seleção	$O(n)$	$O(\log n)$
Descarte	$O(n)$	$O(1)$
Inserção	$O(1)$	$O(\log n)$
Uma iteração	$O(n)$	$O(\log n)$
Tempo total	$O(mn)$	$O(m \log n)$

genéricas, entretanto, recomenda-se o uso da estrutura, por apresentar desempenho superior na maioria das situações e ligeiramente inferior em aplicações pequenas.

6 EXPERIMENTOS E RESULTADOS

Para aplicações de interesse deste trabalho em ambientes não-estacionários, onde a evolução deve ocorrer online e a função de avaliação pode sofrer mudanças, Algoritmos Genéticos Steady State, ou Incrementais, se mostram superiores a Algoritmos Genéticos Geracionais (VAVAK; FOGARTY, 1996). Por isso, nos testes foram consideradas apenas versões Incrementais (Steady State) do algoritmo, comparando o desempenho entre a versão tradicional e a versão alterada pela nossa proposta.

Uma discussão sobre os resultados encontrados será apresentada no próximo capítulo, bem como possibilidades de formas de prosseguimento e melhoria deste trabalho.

6.1 Funções Utilizadas

Para os testes, foram utilizadas duas funções distintas, com comportamentos bem diferentes. A primeira delas foi escolhida por sua natureza previsível e seu comportamento suave, com mínimos e máximos locais. Com isso esperamos ter uma maior facilidade para interpretação dos resultados. A segunda delas apresenta van-

tagens diretas na interpretação dos resultados por estes estarem diretamente ligados à sua codificação. Em ambos os casos foi utilizada a codificação binária com uso de 32 bits em cada cromossomo. Para a avaliação do algoritmo proposto foram efetuadas mudanças em tempo real nas funções, para caracterizar o ambiente como não-estacionário. Além disso, também foi simulado o comportamento de evolução do jogador, resultando em avaliações superestimadas, como descrito na seção 4.1.1.

6.1.1 Função f_1

A primeira função utilizada é definida por

$$f_1(x) = x \operatorname{sen}(10\pi x) + 1$$

restrita ao domínio $[-1, 2]$, como utilizada em (MICHALEWICZ; FOGEL, 2000) e exposta na figura 6.1.

Entre as vantagens de seu uso em nosso trabalho podemos citar o conhecimento do valor máximo ($\approx 2,85$) e sua irregularidade com existência de muitos máximos locais, que pode ser uma armadilha para os métodos de busca. Além disso, o fato de o máximo global estar distante do centro do domínio evita que o cruzamento de soluções nas extremidades conduzam a uma boa solução "por acaso".

O cromossomo de um indivíduo é representado por um inteiro de 32 bits, gerando, então, 2^{32} indivíduos possíveis. Cada um deles é representado por um inteiro no intervalo $[0, 2^{32} - 1]$ e, portanto, se faz necessário o uso de uma função que faça a escala e translação deste domínio para o desejado, que é$[-1, 2]$. Para isso utilizamos a função de conversão

$$x(i) = -1 + \frac{3i}{2^{32} - 1},$$

onde i é o inteiro referente à representação do indivíduo e x_i é seu valor real no

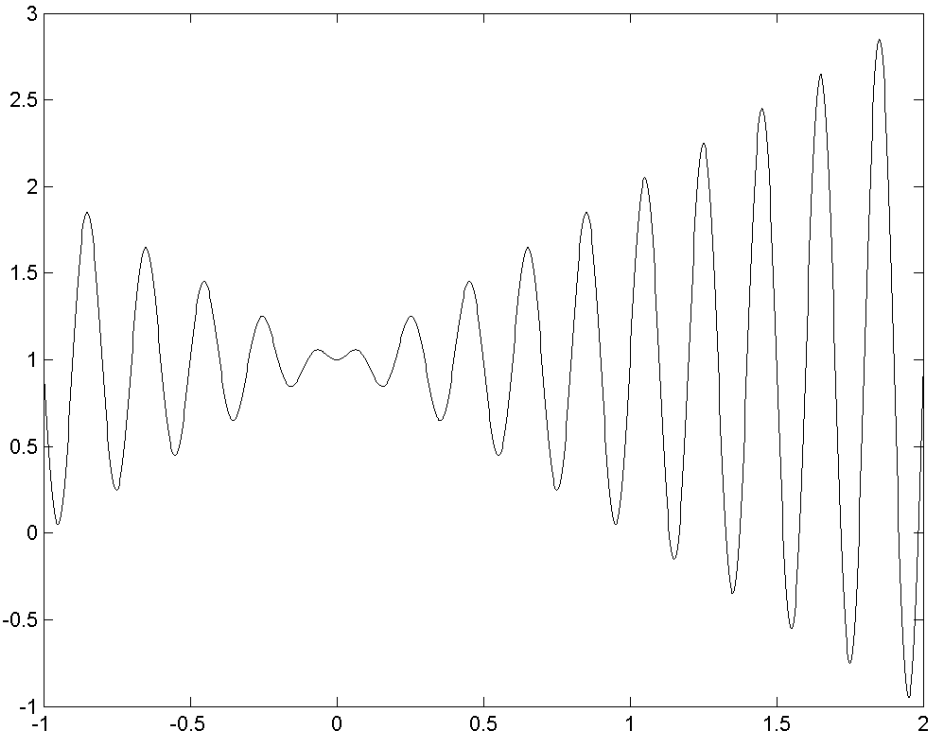

Figura 6.1: Gráfico da função $f_1(x) = x\,\mathrm{sen}(10\pi x) + 1$ no intervalo $[-1, 2]$.

domínio de $f_1(x)$.

A simulação do aprendizado, de acordo com o modelo descrito na seção 4.1.1, é inserida na função $f_1(x)$ como um quociente, resultando na seguinte equação

$$g_1(x,t) = \frac{f_1(x)}{r(t)} = \frac{x\,\mathrm{sen}(10\pi x) + 1}{1 - (1 - r_i)e^{-\lambda t}}.$$

Supondo que o conhecimento inicial do jogador é de 20% do máximo possível, temos $r_i = 0,2$. Para evitar que o aprendizado ocorra muito rapidamente (e consequentemente seja imperceptível), o valor de λ deve ser pequeno. Para os experimentos, utilizamos valores $0,005 \leq \lambda \leq 0,05$. O gráfico da função $g_1(x,t)$ com estes parâ-

metros pode ser visto na figura 6.2.

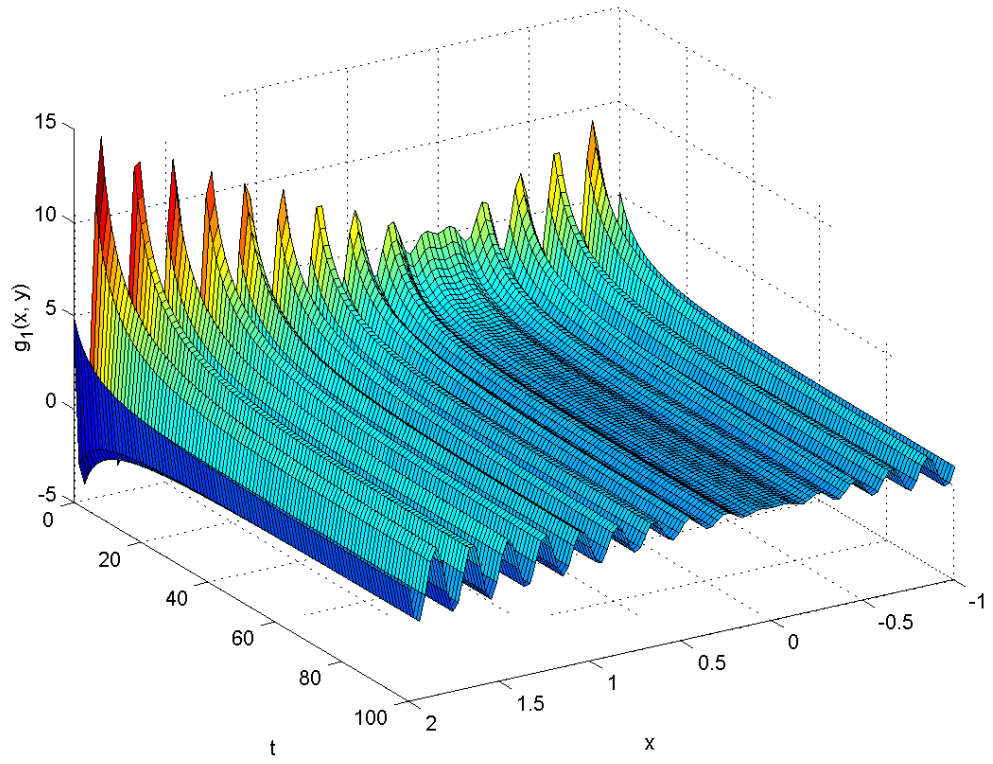

Figura 6.2: Função $g_1(x,t)$, altera a função original $f_1(x)$ com a inserção da componente do tempo.

Nos testes que envolviam a simulação de mudanças no ambiente em tempo real, a modificação era feita alterando-se a fase da função. Para isso, é utilizada mais uma variável, responsável por essa mudança. Inicialmente temos a fase $\phi = 0$ e, a cada Δt_m unidades de tempo esta variável era alterada por um incremento aleatório, dado por uma distribuição de probabilidade uniforme dentro do intervalo $[-c; c]$, onde $0 < c \leq 0,5$. Quando utilizada esta modificação, a nova função $f_1(x)$ passa então a ser dada por

$$f_1(x) = x\,\mathrm{sen}(10(\pi x + \phi)) + 1.$$

6.1.2 Função f_2

A função $f_2(x)$ de um indivíduo x é a quantidade de bits em seu cromossomo iguais aos bits de um padrão p pré-determinado arbitrariamente. Como o cromossomo tem um total de 32 bits, essa função pode ser escrita como

$$f_2(x) = 32 - dist(p, x)$$

onde a função $dist(p, x)$ nos dá a Distância de Hamming entre os indivíduos p e x. Esta função é a mesma utilizada em (VAVAK; FOGARTY, 1996), um trabalho com objetivos semelhantes aos nossos. Por conta de sua irregularidade, a visualização desta função para quantidades de bits altas torna-se complicada. Por isso, um exemplo com o uso de 6 bits é exibido na figura 6.3.

De forma análoga ao caso da função $g_1(x, t)$, definimos a função $g_2(x, t)$ como

$$g_1(x, t) = \frac{f_1(x)}{r(t)} = \frac{32 - dist(p, x)}{1 - (1 - r_i)e^{-\lambda t}}$$

e os valores utilizados para r_i e λ foram os mesmos da função $g_1(x, t)$.

Já para a simulação de mudanças no ambiente em tempo real, foi utilizada também a mesma estratégia utilizada em (VAVAK; FOGARTY, 1996). Esta estratégia consiste na alteração de uma quantidade de bits do padrão. Com a alteração de, digamos,k bits, a alteração da avaliação de um indivíduo poderá variar também em, no máximo, k unidades para mais ou para menos. Com isso pode-se fazer alterações controladas e avaliar ser comportamento sem mudanças drásticas. As alterações foram feitas em posições aleatórias do cromossomo a cada Δt_m unidades de tempo.

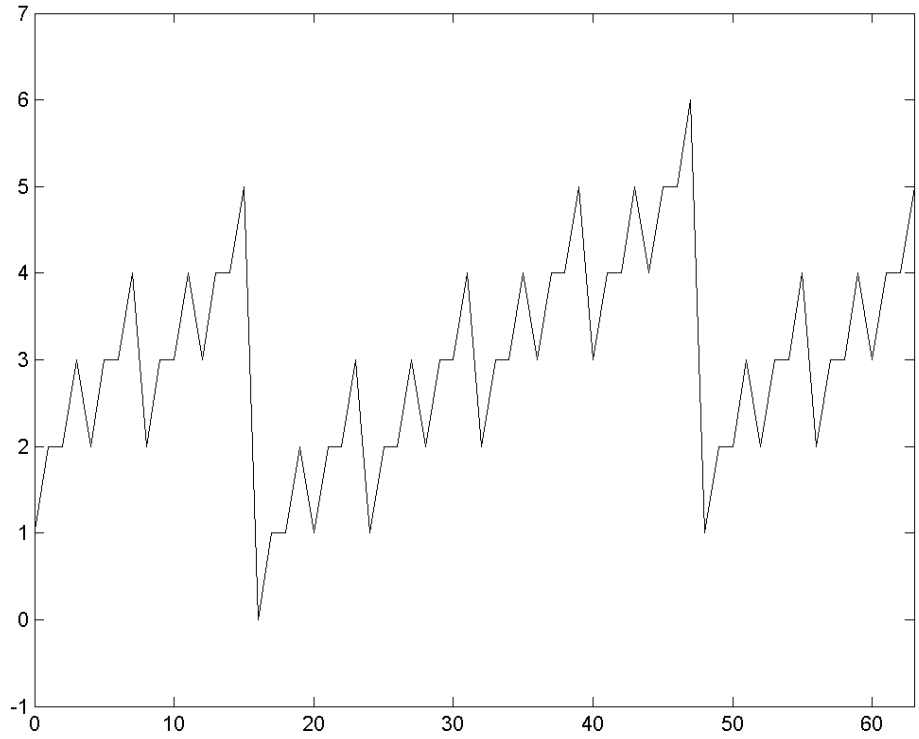

Figura 6.3: Exemplo de função $f_2(x)$. Situação com 6 bits, domínio no intervalo $[0, 2^6 - 1]$. Padrão sorteado $p = (101111)_2 = 47$.

6.2 O Parâmetro b

Para o ajuste do parâmetro b, a base da função exponencial dada por

$$h(t_i, t) = b^{t-t_i},$$

foram testados diversos valores entre 0 e 1, e também a equação

$$b = \sqrt[n]{0,5},$$

onde n é o tamanho da população utilizada. Nesta, após n iterações, que é o tempo médio de vida de um indivíduo, a função $f(x_i, t)$ terá a metade de seu valor inicial

$f_0(x_i)$, obtido quando o indivíduo x_i foi gerado, ou seja,

$$f(x_i, n) = \frac{f_0(x_i)}{2}.$$

Além de ter obtido bons resultados, esta equação mostra-se mais robusta, por não precisar ser ajustada manualmente ao se alterar o tamanho da população. Assim esta constante é definida diretamente a partir dos parâmetros tradicionalmente utilizados em Algoritmos Genéticos, o que facilita o uso do método, diminuindo a quantidade de testes necessários até a obtenção de um conjunto de parâmetros eficiente.

Os resultados obtidos experimentalmente com valores definidos manualmente confirmam a escolha desta forma de cálculo como adequada, com bom desempenho quando utilizados valores de b próximos ao sugerido. Sendo assim, esta forma de cálculo foi utilizada na maioria dos experimentos, exceto quando especificado o contrário.

Vale ressaltar que nem sempre esta é a melhor forma de cálculo de b e, quando necessário, um ajuste fino da variável deve ser considerado. Apesar disso, esta forma de cálculo pode ser considerada satisfatória para casos onde não seja necessário um refinamento maior.

O uso desta forma de cálculo pode ainda ser justificado por outro argumento. Suponha que seja escolhido um valor baixo para b, como $b = 0,5$. Para este valor, estamos fazendo com que na iteração seguinte à que um indivíduo foi gerado, sua avaliação cai pela metade. E na iteração seguinte, tem sua avaliação reduzida pela metade novamente. Ou seja, os indivíduos acabam tendo sua avaliação diminuída muito rapidamente. Assim, a busca acaba ignorando indivíduos mais velhos, reduzindo drasticamente a diversidade genética e perdendo sua capacidade de exploração do espaço. A imagem 6.4 mostra o desempenho médio dos algoritmos para$b = 0,5$. Assim, o uso de valores de b baixos reduz a qualidade do algoritmo genético. A fórmula de cálculo proposta, acaba sempre resultando em valores próximos a 1, fa-

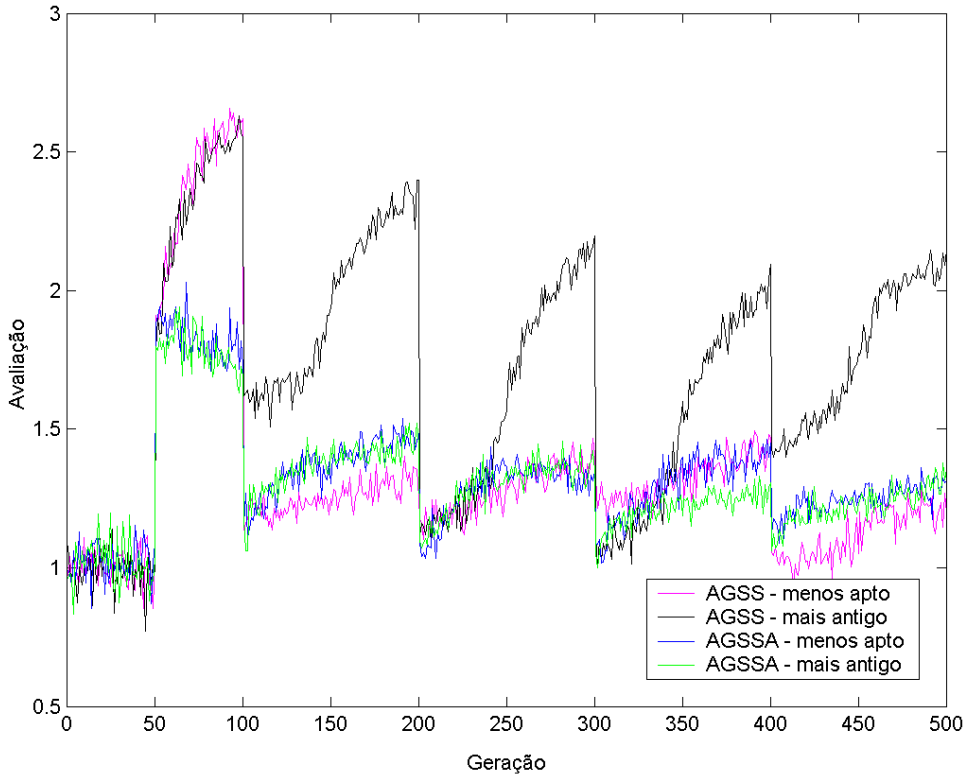

Figura 6.4: Execução dos algoritmos com $b = 0,5$.

zendo com que indivíduos antigos sejam penalizados mas sem reduzir a capacidade de exploração do espaço de busca.

6.3 Testes Realizados

Todos os teste foram realizados no Matlab[1], pela facilidade de implementação proveniente de funções matemáticas e gráficas pré-existentes, e também pela praticidade na execução dos testes. Todas as versões dos algoritmos utilizados foram imple-

[1] http://www.mathworks.com/

mentadas a partir do zero e houve uma preocupação com a semelhança entre os algoritmos, de forma que as únicas diferenças fossem inerentes às diferenças entre os métodos, evitando, assim, a existência de fatores externos que pudessem interferir na comparação dos resultados.

Nas seções seguintes abordaremos a comparação entre o AGSSA e o Algoritmo Genético Steady State Tradicional (AGSS) sob algumas perpectivas diferentes, considerando sempre a execução de 2 versões de cada um dos algoritmos: o AGSSA com descarte do indivíduo menos adaptado e do mais antigo e, de forma análoga, o AGSS com descarte do menos apto e do mais antigo.

Para a exibição dos resultados, utilizaremos gráficos comparativos com o desempenho de cada uma destas variantes dos algoritmos nas cores azul, verde, cor-de-rosa e preto, respectivamente. Para a obteção destes gráficos, cada versão foi executada diversas vezes e, para cada geração, a média dos indivíduos gerados nas diversas execuções foi calculada. Esta metodologia foi utilizada para evitar que eventuais distorções pontuais ocorridas devido às componentes randômicas dos algoritmos genéticos dificultassem a interpretação dos resultados.

Os valores das funções exibidos nos gráficos são referentes à qualidade (aptidão) do indivíduo naquele instante. Mesmo no caso dos testes envolvendo simulação do aprendizado, onde os algoritmos recebem avaliações "distorcidas", o valor exibido é a avaliação real, uma vez que o objetivo que estamos buscando maximizar é aquele referente à qualidade do agente. Deve-se observar ainda que, nas primeiras iterações dos algoritmos, seu desempenho é idêntico para todas as versões e totalmente aleatório, pois ainda estão sendo gerados os indivíduos iniciais da população.

Em todos os testes realizados, os resultados das diferentes versões dos algoritmos apresentaram desvios padrão semelhantes. Portanto, omitiremos esta informação

quando analisarmos e compararmos estes algoritmos.

Os tamanhos de população considerados nos testes foram de 30 e 50 indivíduos. A probabilidade de mutação utilizada em todos os casos foi de 1%. Com esta probabilidade, a chance de ocorrer uma mutação em um indivíduo é de cerca de 28%. O método de seleção utilizado foi o método do torneio, para que os algoritmos fossem mais robustos às mudanças de funções e também à alteração proposta. A quantidade de indivíduos selecionados para o torneio era proporcional ao tamanho da população. Nos resultados apresentados, essa quantidade era de 6 ou 8 indivíduos. Estes valores foram escolhidos de forma que a seleção não fosse muito aleatória (quando a quantidade de indivíduos é muito pequena) nem forçasse uma convergência precoce (com muitos indivíduos sendo selecionados para o torneio, indivíduos menos adaptados acabando tendo uma chance muito pequena de se reproduzir).

6.3.1 Alterações no Ambiente

Nos testes de reação dos algoritmos em relação às mudanças no ambiente, as duas versões do AGSSA tiveram resultados bastante próximos como pode ser observados nas figuras 6.5 e 6.6.

Já as versões do AGSS apresentaram desempenhos bem diferentes. A versão que eliminava o menos apto obteve um desempenho inicial semelhante aos demais. Porém, com o passar do tempo, à medida que a função de avaliação se altera, a impossibilidade do algoritmo de reavaliar os indivíduos faz com que existam indivíduos que obtiveram boas avaliações em instantes anteriores mas com péssimas avaliações com a função atual. Por outro lado, a versão com descarte do indivíduo mais antigo foi capaz de se adaptar, uma vez que o tempo de vida de um indivíduo é limitado. Assim, a população se altera de acordo com as mudanças do ambiente, ainda que

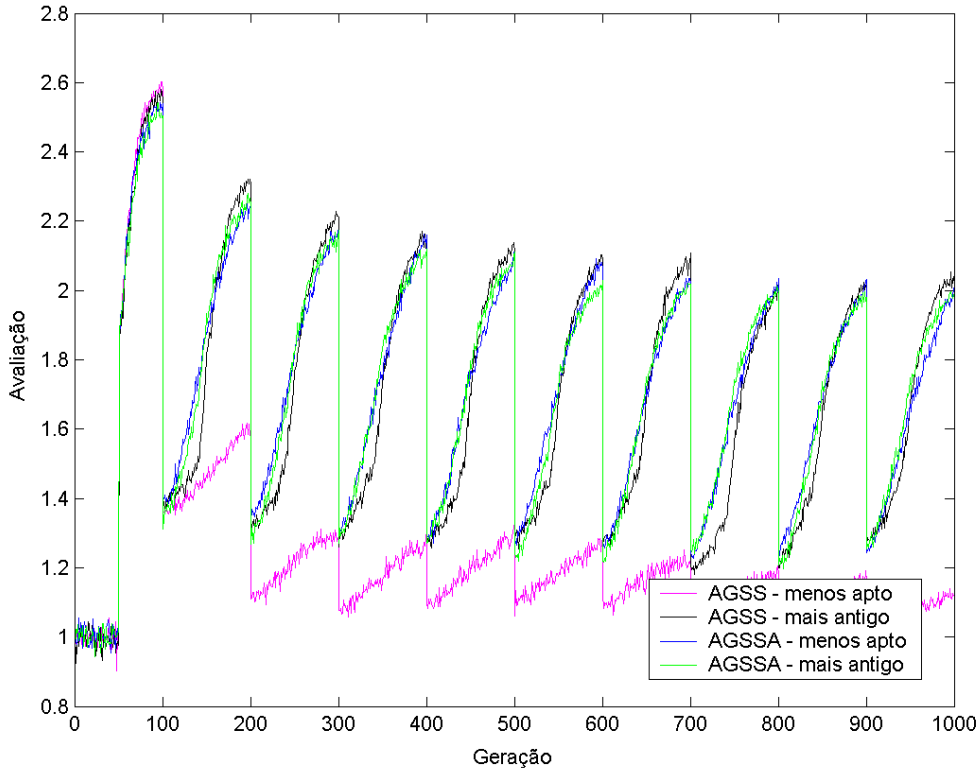

Figura 6.5: Desempenho médio dos algoritmos para a função f_1, alterando a fase de um valor aleatório no intervalo $[-0,25; 0,25]$ a cada 100 iterações, em uma população com 50 indivíduos.

seja necessário um tempo para isso ocorrer. Essa deficiência caracteriza uma das situações em que o AGSSA tem grande potencial de apresentar melhores resultados que o AGSS. Além disso, como o tempo de vida de um indivíduo depende do tamanho da população, este parâmetro do algoritmo passa a desempenhar um papel ainda mais importante, como veremos mais adiante, na seção 6.3.4.

As duas versões do AGSSA obtiverem desempenho semelhante à versão do AGSS de eliminação do mais antigo. Entretanto, quando o parâmetro b utilizado possuia um valor adequado, as versões do AGSS apresentavam desempenho superior, como

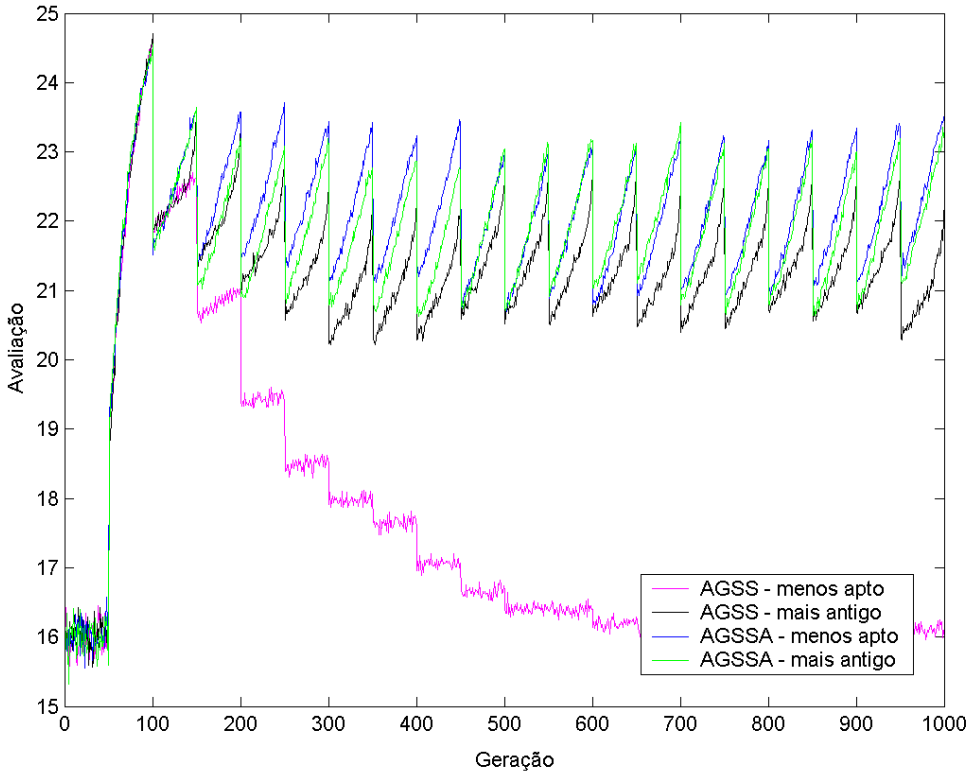

Figura 6.6: Desempenho médio dos algoritmos para a função f_2, com $b = 0,995$ alterando 5 bits do padrão objetivo a cada 50 iterações, em uma população com 30 indivíduos.

pode ser observado nas figuras 6.6 e 6.7.

Uma característica interessante que podemos observar na figura 6.5 é que, como as alterações no ambiente ocorrem com uma frequência baixa, a cada 100 iterações, nas primeiras iterações após uma mudança o AGSSA apresenta um melhor desempenho, enquanto a versão com descarte do mais antigo do AGSS fica presa ainda ao cenário anterior. Com o passar do tempo, cerca de 50 iterações depois, o algoritmo acaba descartando os indivíduos que haviam sido avaliados no cenário anterior e evolui rapidamente alcançando novamente o desempenho do AGSSA. Podemos observar

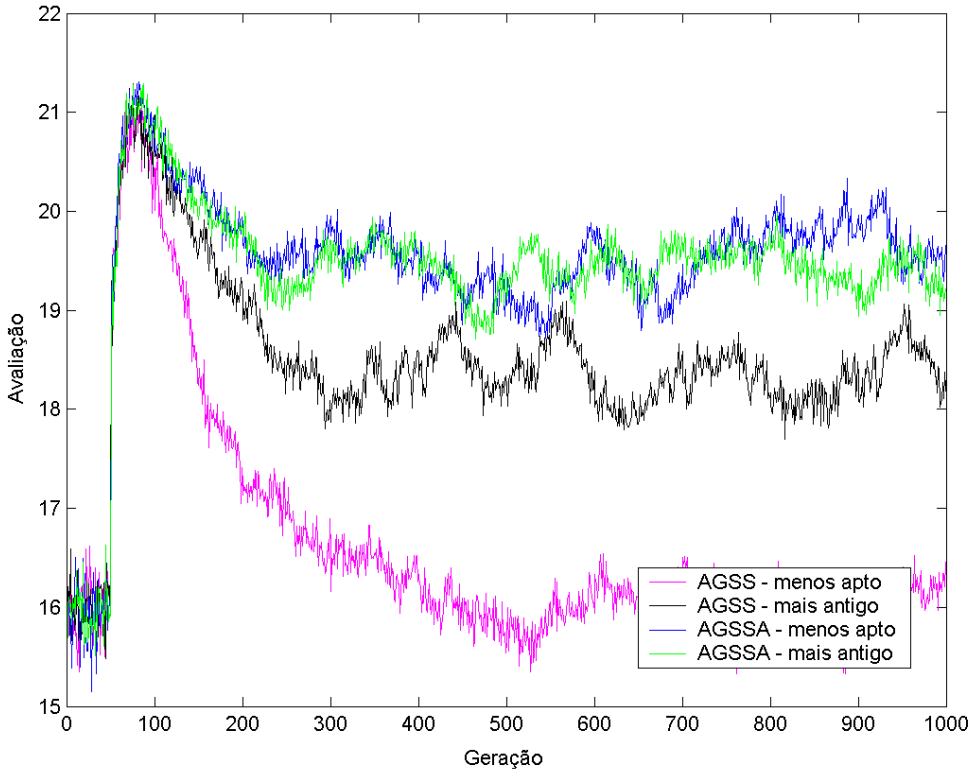

Figura 6.7: Desempenho médio dos algoritmos para a função f_2, alterando 1 bit do padrão objetivo a cada 4 iterações, em uma população com 50 indivíduos.

que, em situações de estabilidade do ambiente, o uso da funçãoh atribuindo um peso pela idade de cada indivíduo acaba agindo como uma forma de ruído na avaliação. Isso explica o crescimento do desempenho do AGSS 50 iterações após a mudança no ambiente.

6.3.2 Simulação de Aprendizado de um Jogador

Para a simulação do aprendizado de um jogador humano, foi utilizada a modelagem abordada no capítulo 4. Os únicos parâmetros a serem definidos são r_i e λ. Em

todos os testes foi utilizado o valor $r_i = 0,2$ para o conhecimento inicial do jogador. Para a taxa de aprendizado, foram utilizados diversos valores diferentes, explicitados em cada um dos exemplos.

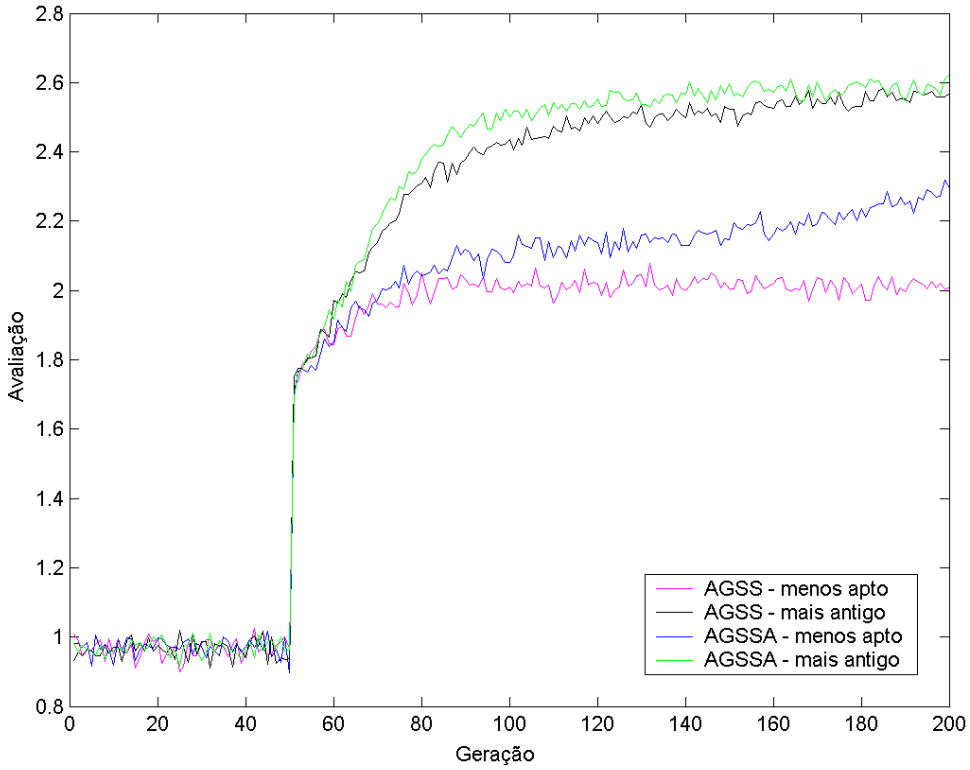

Figura 6.8: Desempenho médio de cada um dos algoritmos com parâmetro$b = \sqrt[50]{0,8}$ e população de 50 indivíduos, para a função f_1 com simulação de aprendizado com $\lambda = \frac{1}{150}$.

Novamente nos testes o AGSS apresentou resultado ruim para o caso onde era eliminado sempre o pior indivíduo. Além disso, mesmo o AGSSA não apresentou resultados muito satisfatórios para o método de descarte do menos apto. Ainda assim, o desempenho foi superior ao do AGSS.

A superioridade do AGSSA sobre o AGSS se manteve nas versões com o descarte

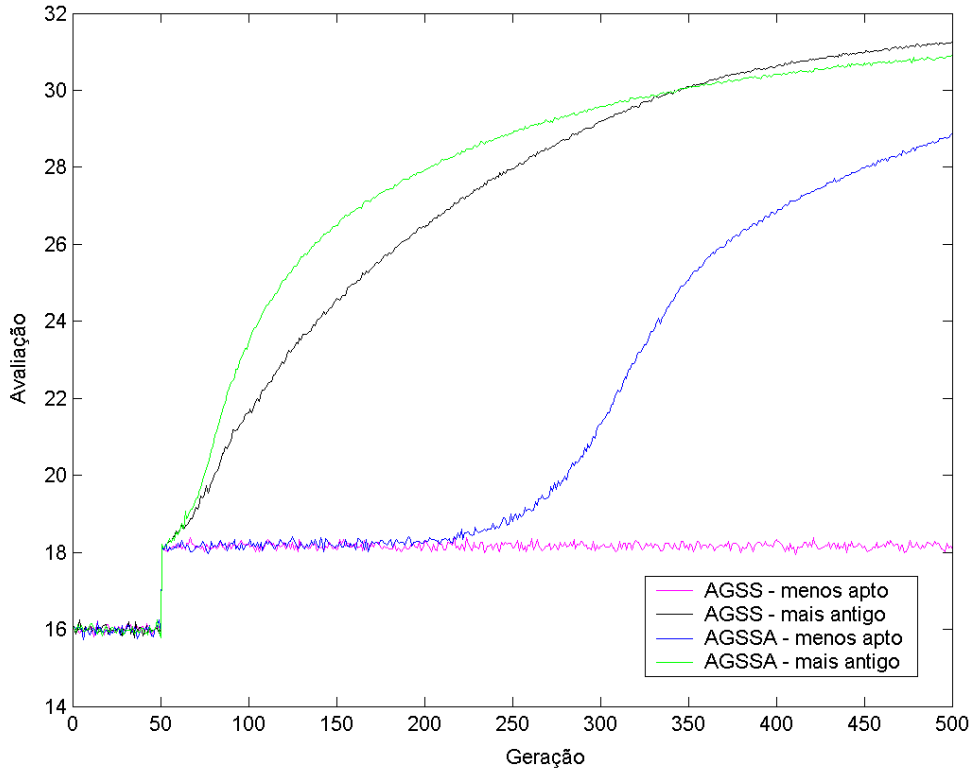

Figura 6.9: Desempenho médio de cada um dos algoritmos com parâmetro$b = \sqrt[50]{0,8}$ e população de 50 indivíduos, para a função f_2 com simulação de aprendizado com $\lambda = \frac{1}{100}$.

do indivíduo mais antigo, embora menos discrepante, como pode ser observado nas figuras 6.8 e 6.9. Apesar disso, após uma certa quantidade de iterações o desempenho de ambos acaba se tornando equivalente, ou o AGSS até mesmo supera o AGSSA, quando o aprendizado do jogador humano praticamente deixa de ocorrer.

6.3.3 Sensibilidade ao Parâmetro b

Como era de se esperar, os algoritmos se mostraram bastante sensíveis ao parâmetro b. Em particular, quando $b = 1$, o AGSSA e o AGSS se comportam da mesma forma. De maneira geral, os valores bem próximos de 1, como $b = \sqrt[50]{0,5} \approx 0,986$ no caso de uma população com 50 indivíduos, mostraram os melhores resultados. Além disso, foi observado que os diferentes tipos de descarte reagiam de maneira diferente às variações de b.

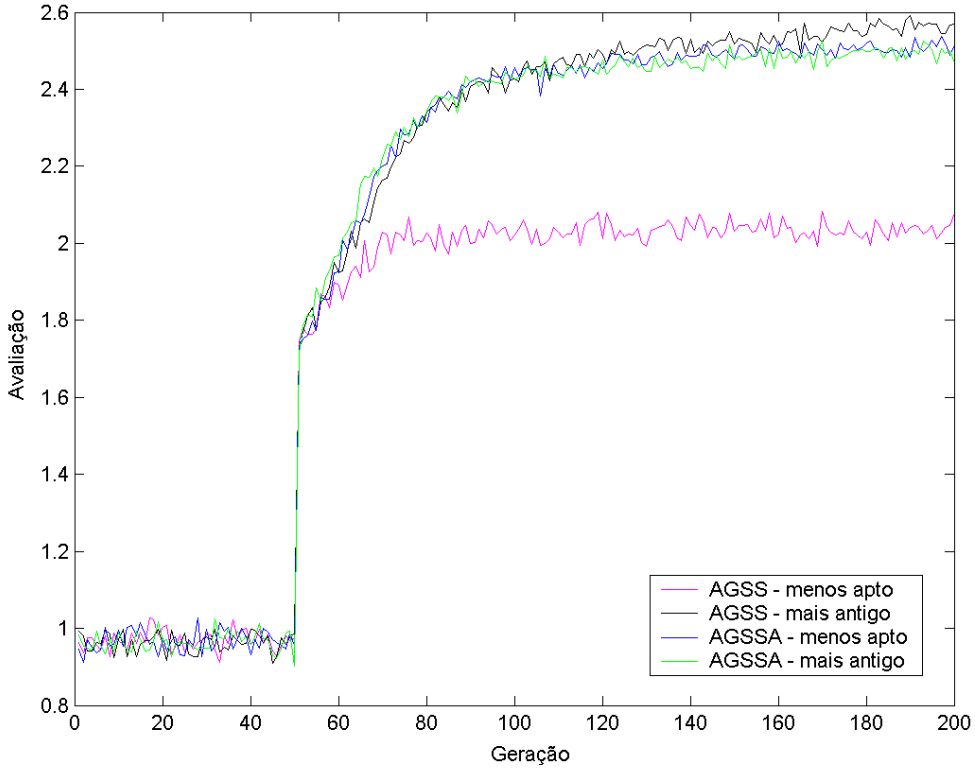

Figura 6.10: Desempenho médio de cada um dos algoritmos com parâmetro $b = \sqrt[50]{0,2}$ e população de 50 indivíduos, para a função f_1 com simulação de aprendizado com $\lambda = \frac{1}{150}$.

Como observado na figura 6.8, na qual utilizamos $b = \sqrt[50]{0,8}$, o AGGSA com descarte

do mais antigo apresentou resultados superiores a todos os demais. Por outro lado, quando $b = \sqrt[50]{0,2}$, as duas versões do AGSSA obtiveram desempenho similar ao AGSS com descarte do indivíduo mais antigo, como pode ser observado na figura 6.10. Por outro lado, com $b = \sqrt[50]{0,5}$ obtemos resultados semelhantes ao primeiro caso para o AGGSA com descarte do mais antigo, porém resultados muito melhores para aquele com descarte do menos apto, como exposto na figura 6.11.

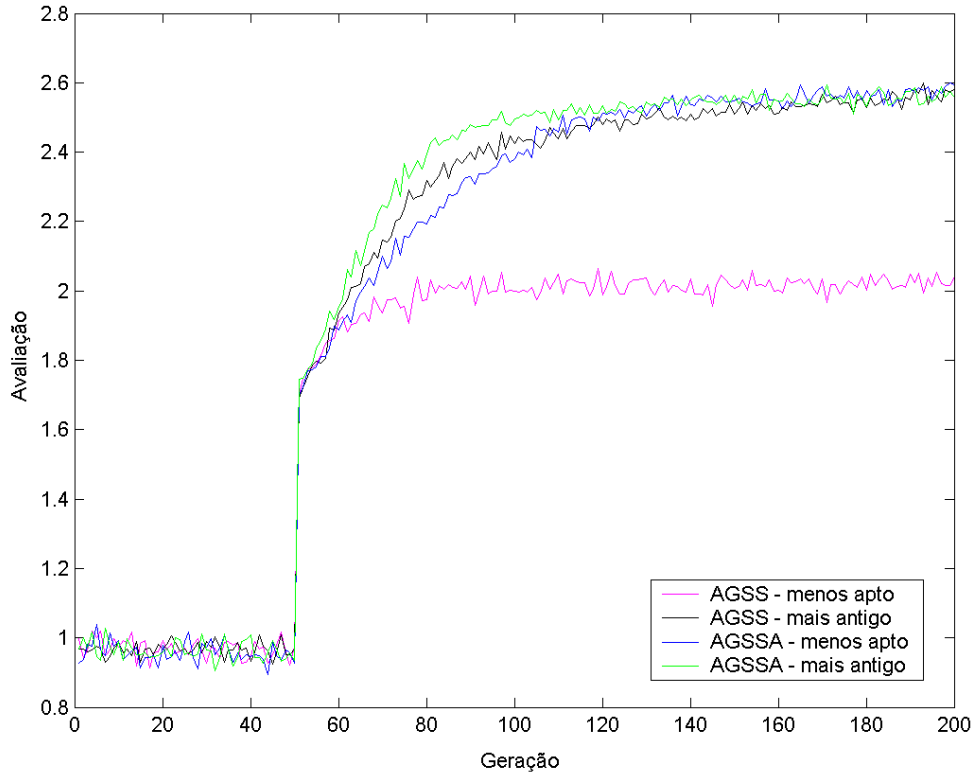

Figura 6.11: Desempenho médio de cada um dos algoritmos com parâmetro $b = \sqrt[50]{0,5}$ e população de 50 indivíduos, para a função f_1 com simulação de aprendizado com $\lambda = \frac{1}{150}$.

Quando utilizamos o AGSSA com o descarte do indivíduo mais antigo, o impacto de b é observado no momento da reprodução: indivíduos mais novos apresentam maiores chances de reprodução. Já no caso do AGSSA com descarte do indivíduo

menos apto, o valor de b passa a ter um papel importante também no momento da seleção do indivíduo a ser descartado, ou seja, a decisão de descarte passa a depender dos dois parâmetros combinados, o tempo de vida do indivíduo e sua aptidão. Um estudo mais profundo do impacto de b no desempenho dos algoritmos é umas possibilidades de trabalhos futuros, que abordaremos no próximo capítulo.

6.3.4 Sensibilidade ao Tamanho da População

Como poderíamos suspeitar, o tamanho da população tem uma influência notável no desempenho dos algoritmos. Quando tratamos de ambientes dinâmicos, a relação entre a frequência com que as mudanças ocorrem e o tamanho da população passa a ser de suma importância. No caso dos algoritmos tradicionais, o AGSS com descarte do menos apto acaba ficando preso a paradigmas anteriores, mantendo na sua população indivíduos com bons níveis de aptidão em cenários anteriores, porém com desempenho ruim no estado atual. Com a alteração da estratégia de descarte para o descarte do indivíduo mais antigo, este impacto acaba sendo menor, uma vez que indivíduos antigos, que possivelmente apresentam avaliações incompatíveis com o cenário atual, acabam sendo desprezados com o passar do tempo. Entretando, caso as mudanças aconteçam com muita frequência, pode ocorrer uma defasagem devido ao tamanho da população.

O AGGSA por sua vez, acaba levando a defasagem em consideração por natureza. Em qualquer instante, um indivíduo mais antigo receberá uma penalidade por sua idade, uma vez que sua avaliação já não é mais tão confiável quanto no momento em que este foi gerado. Este comportamento acaba tornando o algoritmo mais robusto a populações com tamanhos muito grandes. Obviamente, o valor de$b$ tem um papel importante aqui, acarretando em uma busca muito localizada quando o seu valor é baixo (uma vez que indivíduos mais velhos passam a ser quase que desprezados)

ou uma busca totalmente desinformada da idade dos indivíduos no caso em que seu valor é demasiadamente próximo de 1.

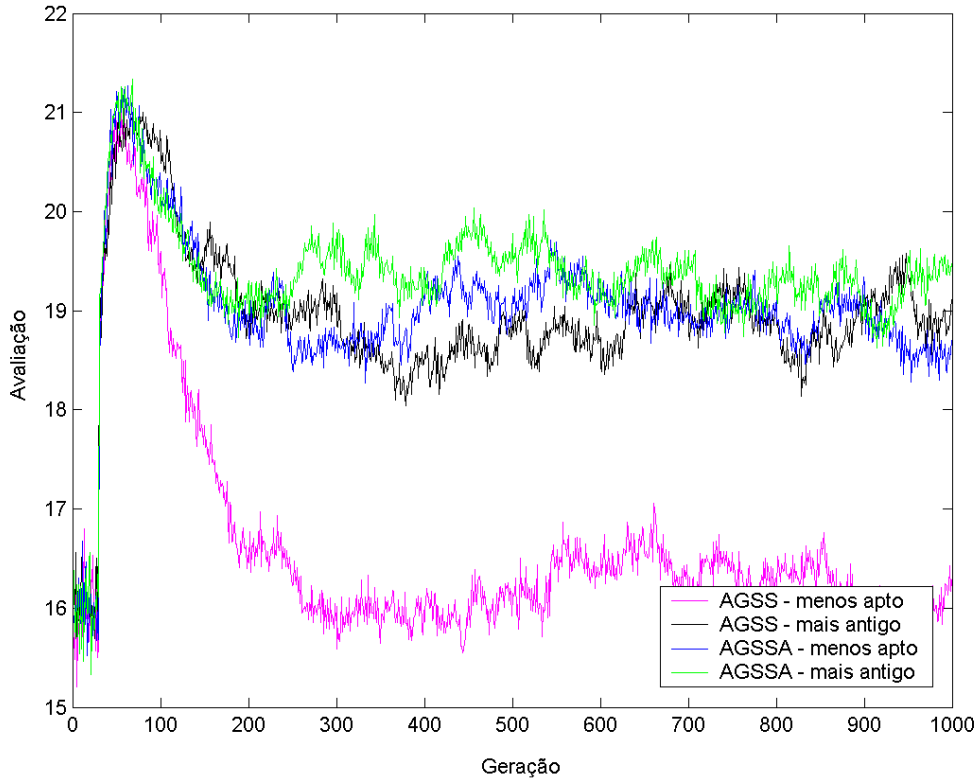

Figura 6.12: Desempenho médio dos algoritmos para a função f_2, alterando 1 bit do padrão objetivo a cada 4 iterações, em uma população com 30 indivíduos.

A sensibilidade dos algoritmos à variação da população pode ser visualizada com mais facilidade através de um exemplo. Retornando à figura 6.6, onde a população possuía 50 indivíduos, podemos observar que o desempenho do AGSS é inferior ao seu desempenho quando com uma população de 30 indivíduos (figura 6.12). De maneira geral, o AGSSA é menos sensível à variação no número de indivíduos da população.

7 CONCLUSÕES E TRABALHOS FUTUROS

Este capítulo apresenta um resumo das contribuições e conclusões deste trabalho e também faz uma breve discussão sobre possíveis caminhos a serem trilhados a partir deste trabalho e outras questões relacionadas que mereçam atenção.

Uma primeira conclusão que pode ser tirada deste trabalho é uma melhor compreensão das deficiências de ambos os métodos de descarte. No caso da eliminação do mais antigo há uma demora na reação quando ocorrem alterações no ambiente, uma vez que indivíduos bem adaptados ao ambiente anterior continuarão a se reproduzir com alta probabilidade. Já o caso da eliminação do menos adaptado apresenta, além desta desvantagem, dificuldades para o descarte de indivíduos antigos, mesmo após a mudança de ambiente, por estes indivíduos podem ter recebido avaliações muito altas naquele instante.

A modificação aos Algoritmos Genéticos sugerida por este trabalho no capítulo 4 apresenta uma alternativa a ser utilizada em problemas onde o ambiente se altera dinamicamente. Os testes realizados com o Algoritmo Genético Steady State Adaptativo (AGSSA) no capítulo 6, mostraram que é possível melhorar o desempenho dos algoritmos tradicionais. Além disso, o AGSSA apresentou uma maior robustez à

variação do tamanho da população, ao contrário do AGSS que tem seu desempenho diminuído quando a população aumenta. Analogamente, as mesmas características se mantêm quando há uma maior variação na função objetivo, representnado mudanças no ambiente (ou alterações no comportamento do jogador humano, no caso de jogos).

Ainda assim, seu uso não deve ser feito de modo impensado: em situações como a de jogos, onde a habilidade do jogador pode parar de evoluir após um determinado número de iterações, o uso do AGSSA torna-se desnecessário depois de um certo tempo. Por outro lado, caso haja a possibilidade de alteração de estratégia do jogador para explorar possíveis pontos fracos, torna-se necessária a manutenção do uso do algoritmo.

Como visto no capítulo anterior, em determinadas situações o algoritmo obteve desempenho superior e, em outras, desempenho inferior aos métodos convencionais. Entretanto, desde que escolhidos os parâmetros adequados de acordo com o problema, podemos afirmar que o AGSSA sempre obtém bom desempenho pois, no pior caso, podemos ajustar seus parâmetros para que o algoritmo atue de forma idêntica a um Algoritmo Genético Incremental. Uma desvantagem clara do uso da modificação proposta no AGSSA é a diminuição do desempenho do algoritmo, uma vez que são necessárias mais variáveis e maior processamento destas informações. Entretanto esta queda de desempenho não compromete o uso da técnica, pois as operações são simples e em pequena quantidade.

Além das situações de interesse abordadas, com evolução de um jogador humano no instante inicial e mudanças de estratégia no decorrer do jogo (ou, em outro contexto, mudanças no ambiente dos indivíduos), há uma outra situação onde o algoritmo apresenta vantagens: em situações onde a avaliação de um indivíduo não pode ser obtida de forma precisa, mas apenas estimada. Neste tipo de situação, é

comum a ocorrência de erros de avaliação. Com o uso do AGSSA, este tipo de erro acaba sendo minimizado, pois o indivíduo passa a ter sua avaliação diminuída com o passar do tempo.

Apesar dos resultados promissores, estudos futuros com novas funções e com aplicações reais precisam ser feitos para que se possa compreender melhor as consequências das alterações propostas.

7.1 Dificuldades

Apesar da motivações deste trabalho ter surgido do estudo de técnicas de inteligência aplicadas a jogos, a dificuldade de se testar os algoritmos neste tipo de ambiente acabou impedindo a realização de testes em ambientes reais. Entre as dificuldades levantadas, podemos destacar a necessidade de realização de grande quantidade de testes com indivíduos diferentes. Como a proposta do trabalho se baseia no aprendizado do jogador, um único indivíduo não pode realizar mais de um teste, pois em uma segunda ocasião ele já estaria mais acostumado com as possibilidades do jogo e com os controles. Assim, para se comparar diferentes algoritmos, tornar-se-ia necessário testá-los com jogadores diferentes. E para que qualidades individuais destes testadores não interferissem no resultado final, seria importante a realização de uma grande quantidade de testes.

Além desta dependência de uma grande quantidade de testadores, havia ainda outros empecilhos como a necessidade de implementação dos próprios jogos em si e, em particular, jogos onde a abordagem faça sentido, ou seja, são necessários dispositivos (sejam eles agentes, obstáculos, etc) que possam ser ajustados de forma a aumentar o nível de dificuldade do jogo. Além disso ainda seria necessário que os jogos possibilitassem a evolução do jogador com o passar do tempo e também onde

fosse possível efetuar mudanças na estratégia para que estes dois cenários fossem testados satisfatoriamente.

Uma outra dificuldade encontrada no decorrer deste trabalho foi sua abrangência. Inicialmente foram estudadas diversas abordagens e pretendia-se aplicar diversas delas. Entretanto, com o passar do tempo preferiu-se restringir o escopo e concentrar o esforço em contribuições mais pontuais.

7.2 Trabalhos Futuros

Como dito anteriormente, novos estudos e simulações devem ser feitos utilizando-se o algoritmo proposto, de forma que possamos dominar o conhecimento sobre as implicações de seu uso. Além disso, o algoritmo não obteve resultados semelhantes para as diferentes funções, tendo resultados de eficácia diferente em cenários com funções diferentes. O estudo das razões pelas quais este fenômeno ocorreu, bem como uma possível caracterização dos tipos de problemas aos quais o uso do AGSSA é recomendado ou não são possíveis continuações deste trabalho.

Outro aspecto interessante a ser abordado é o parâmetro b. Como com o passar do tempo o método se torna desnecessário (pois o jogador "para de evoluir"). Estratégias dinâmicas podem ser adotadas de forma a detectar a necessidade de uso de uma adaptação, ou seja, detectar que o ambiente está se alterando, e também ajustar o valor de b de acordo com essas necessidades. Em jogos onde o jogador não pode alterar sua estratégia e melhora sua perícia com o passar do tempo, uma possibilidade é fazer o valor b tender a 1 com o decorrer da iteração com o jogador.

No capítulo 4, mais especificamente na seção 4.1.3, foram abordadas outras possibilidades de usos da função $h(t_i, t)$. O estudo do impacto de funções de diferentes

naturezas nos algoritmos genéticos é uma possibilidade de trabalho interessante que pode revelar possibilidades de aplicação até mesmo em problemas de outra natureza.

Como abordado neste texto, uma característica central que difere a aplicação de métodos de otimização em ambientes dinâmicos genéricos de sua aplicação em jogos é a necessidade de geração de indivíduos com desempenho satisfatório na maioria das iterações (SPRONCK; SPRINKHUIZEN-KUYPER; POSTMA, 2003). Para resolver esta restrição, em particular, propomos duas medidas que podem ser tomadas mas que não foram abordadas detalhadamente neste trabalho por questões de escopo: a geração da população inicial poderia se dar de maneira controlada, através da seleção, possivelmente aleatória, de indivíduos gerados previamente, garantindo assim uma qualidade mínima aos agentes das primeiras iterações com o jogador; a avaliação de um agente através de parâmetros pré-definidos antes de permitir o confronto com o jogador, avaliando se este agente apresenta um grau de qualidade mínimo antes de apresentá-lo ao jogdor. As maneiras de se realizar estas duas melhorias são diversas, mas como dito, fogem do escopo deste trabalho.

Uma consequência da existência de uma função que atribui um peso maior aos indivíduos mais novos é que o algoritmo tende a acelerar a convergência genética (uma vez que indivíduos gerados recentemente tendem a se reproduzir mais facilmente). Como trabalho futuro pode-se estudar este fenômeno e procurar meios de conciliar os objetivos de resposta rápida às mudanças, geração de poucos indivíduos ruins e manutenção da variabilidade genética.

REFERÊNCIAS

ALVIM, L.; OLIVEIRA CRUZ, A. de. A Fuzzy State Machine applied to an emotion model for electronic game characters. **Fuzzy Systems, 2008. FUZZ-IEEE 2008. (IEEE World Congress on Computational Intelligence). IEEE International Conference on**, [S.l.], p.1956–1963, June 2008.

ANDERSON, E. F. Playing smart - artificial intelligence in computer games. In: CON03 CONFERENCE ON GAME DEVELOPMENT, 2003. **Proceedings...** [S.l.: s.n.], 2003.

ANDRADE, G.; SANTANA, H.; FURTADO, A.; LEITÃO, A.; RAMALHO, R. Online Adaptation of Computer Games Agents: a reinforcement learning approach. In: BRAZILIAN WORKSHOP ON COMPUTER GAMES, SBGAMES, 2004, Curitiba, Brasil. **Proceedings...** [S.l.: s.n.], 2004.

BRANKE, J. **Evolutionary Approaches to Dynamic Optimization Problems - Updated Survey**. 2001.

CHAMPANDARD, A. J. **AI Game Development**: synthetic creatures with learning and reactive behaviors. 1st.ed. Berkeley: New Riders, 2003.

CROCOMO, M. K. **Um algoritmo evolutivo para aprendizado on-line em jogos eletrônicos**. 2008. Dissertação de Mestrado — Ciências de Computação e Matemática Computacional, USP, São Paulo, SP, Brasil.

DEMASI, P. **Estratégias Adaptativas e Evolutivas em Tempo Real para Jogos Eletrônicos**. 2003. Dissertação de Mestrado — Programa de Pós-Graduação em Informática, Instituto de Matemática, UFRJ, Rio de Janeiro, RJ, Brasil.

DEMASI, P.; OLIVEIRA CRUZ, A. J. de. Online Coevolution for Action Games. **International Journal of Intelligent Games and Simulation**, [S.l.], v.2, n.2, p.80–88, Oct. 2003.

Entertainment Software Association. Disponível em: <`http://www.theesa.com/facts/top_10_facts.php`>. Acesso em Abril 5, 2008.

Games Making Interesting Use of Artificial Intelligence Techniques. Disponível em: <`http://www.gameai.com/games.html`>. Acesso em Abril 5, 2008.

GEYER-SCHULZ, A. Holland classifier systems. In: INTERNATIONAL CONFERENCE ON APPLIED PROGRAMMING LANGUAGES, 1995, San Antonio, Texas, United States. **Proceedings...** ACM, 1995. p.43–55.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization, and Machine Learning.** 1ª.ed. EUA: Addison-Wesley, 1989.

GRIECO, B. **Criação de Oponentes Virtuais com Características Humanas em Jogos Eletrônicos através de Técnicas de Inteligência Computacional** 2007. Dissertação de Mestrado — Programa de Pós-Graduação em Informática, Instituto de Matemática, UFRJ, Rio de Janeiro, RJ, Brasil.

HART, P.; NILSON, N.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE Transactions. Systems Science and Cybernetics**, [S.l.], v.4, n.2, p.100–107, Feb 1968.

HARVEY, P. H.; ZAMMUTO, R. M. Patterns of mortality and age at first reproduction in natural populations of mammals. **Nature**, [S.l.], v.315, p.319–320, May 1985.

HAYKIN, S. **Neural Networks**: a comprehensive foundation (2nd edition). [S.l.]: Prentice Hall, 1998.

HOLLAND, J. H. **Adaptation in natural and artificial systems**: an introductory analysis with applications in biology, control, and artificial intelligence. 1ª.ed. EUA: University of Michigan Press, 1975.

IERUSALIMSCHY, R.; FIGUEIREDO, L. H. de; FILHO, W. C. Lua - An Extensible Extension Language. **Software: Practice and Experience**, [S.l.], v.26, n.6, p.635–652, 1996.

JONES, J.; SOULE, T. Comparing genetic robustness in generational vs. steady state evolutionary algorithms. In: GECCO '06: PROCEEDINGS OF THE 8TH ANNUAL CONFERENCE ON GENETIC AND EVOLUTIONARY COMPUTATION, 2006, New York, NY, USA. **Anais...** ACM, 2006. p.143–150.

LINDEN, R. **Algoritmos Genéticos**. 1ª.ed. Rio de Janeiro: Brasport, 2006.

MELANIE, M. **An Introduction to Genetic Algorithms**. 1ª.ed. EUA: MIT Press, 1998.

MICHALEWICZ, Z.; FOGEL, D. B. **How to solve it**: modern heuristics. New York, NY, USA: Springer-Verlag New York, Inc., 2000.

PONSEN, M. **Improving Adaptive Game AI with Evolutionary Learning** 2004. Dissertação de Mestrado — Faculty of Media and Knowledge Engineering, Delft University of Technology Instituto de Informática, Delft, The Netherlands.

REYNOLDS, C. W. Flocks, herds and schools: a distributed behavioral model. In: SIGGRAPH '87: PROCEEDINGS OF THE 14TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, 1987, New York, NY, USA. **Anais...** ACM, 1987. p.25–34.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence**: a modern approach. [S.l.]: Prentice Hall, 1995.

SILVA, V. S. R.; THOME, A. C. G. Um Comitê de Redes Neurais para o Reconhecimento de Letras Manuscritas. In: ENIA SBRC, 2007, Rio de Janeiro. **Anais...** [S.l.: s.n.], 2007.

SPRONCK, P.; SPRINKHUIZEN-KUYPER, I.; POSTMA, E. Online Adaptation of Game Opponent AI in Simulation and in Practice. In: INTERNATIONAL CONFERENCE ON INTELLIGENT GAMES AND SIMULATION, GAME-ON, 2003, Londres, Reino Unido. **Proceedings...** [S.l.: s.n.], 2003.

SWEETSER, P. **Current AI in Games**: a review. School of ITEE, University of Queensland.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos**. 2nd.ed. [S.l.]: Editora LTC, 1994.

THOMPSON, C. J.; HAHN, S.; OSKIN, M. Using modern graphics architectures for general-purpose computing: a framework and analysis. In: MICRO 35: PROCEEDINGS OF THE 35TH ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 2002, Los Alamitos, CA, USA. **Anais...** IEEE Computer Society Press, 2002. p.306–317.

VAVAK, F.; FOGARTY, T. Comparison of steady state and generational genetic algorithms for use in nonstationary environments. **Evolutionary Computation, 1996., Proceedings of IEEE International Conference on**, [S.l.], p.192–195, May 1996.

WANDERLEY, M. F. B.; SILVA, J. C. P. da; BORGES, C. C. H.; VASCONCELOS, A. T. R. Application of Genetic Algorithms to the Genetic Regulation Problem. In: ADVANCES IN BIOINFORMATICS AND COMPUTATIONAL BI-

OLOGY: THIRD BRAZILIAN SYMPOSIUM ON BIOINFORMATICS, 2008.**Proceedings...** Springer, 2008. p.140–151.

YANNAKAKIS, G.; HALLAM, J. Evolving Opponents for Interesting Interactive Computer Games. In: INT. CONF. ON THE SIMULATION OF ADAPTIVE BEHAVIOR (SAB04), 8., 2004, Toronto, Canada.**Proceedings...** [S.l.: s.n.], 2004. p.499–508.

YANNAKAKIS, G. N. **AI in Computer Games**: generating interesting interactive opponents by the use of evolutionary computation. 2005. Tese de Doutorado — Institute of Perception, Action and Behaviour, School of Informatics, University of Edinburgh, Edinburgh.

ZADEH, L. A. Fuzzy Sets. **Information and Control**, [S.l.], v.8, n.3, p.338–353, 1965.