2 A Construção de Sistemas com Agentes Inteligentes

Este capítulo apresenta as principais ferramentas de Engenharia de Software e algoritmos utilizados nessa tese. A primeira seção apresenta uma linguagem de modelagem orientada a agentes chamada ANote. Essa ferramenta é indispensável para qualquer projeto de software de sistema multi-agente de larga escala. As próximas duas seções apresentam ferramentas importantes que permitem reuso de código e *design*. Essas ferramentas diminuem o tempo de desenvolvimento e reduzem a complexidade de implementar sistemas multi-agente. A última seção deste capítulo apresenta um resumo de vários algoritmos utilizados nessa tese.

2.1. ANote – Uma linguagem de Modelagem Simples e Poderosa

ANote (Choren, 2002; Choren et al., 2004a; Choren et al., 2004b) é uma linguagem de modelagem que oferece um padrão para descrever conceitos relacionados ao processo de modelagem orientado a agentes. A linguagem possui um meta-modelo conceitual que define, em um nível mais alto, conceitos de interação, de ambiente, e de sociedade tais como objetivos, protocolos de interação, recursos do ambiente e organizações. Além disso, a linguagem também define conceitos básicos para cada agente individualmente tais como ações, comunicação e planos.

Esses conceitos definem uma variedade de aspectos ou visões, que podem se complementar ou sobrepor na especificação do sistema. A linguagem ANote define sete visões baseadas em seu meta-modelo conceitual: objetivo, agente, cenário, planejamento, interação, ontologia e organização. Cada visão gera um artefato ou diagrama, e representa uma especificação parcial do sistema. Assim, o *designer* do sistema pode se concentrar em um conjunto pequeno de propriedades a cada momento, e considerar apenas as propriedades importantes para cada contexto.

A visão de objetivo do ANote fornece uma identificação inicial de uma árvore de objetivos para definir as funcionalidades do sistema. Nesta visão,

objetivos complexos podem ser decompostos funcionalmente em outros objetivos e fluxos, e isso fornece uma descrição das funcionalidades em termos de uma árvore hierárquica de objetivos. A figura 7 apresenta uma ilustração da visão de objetivo.

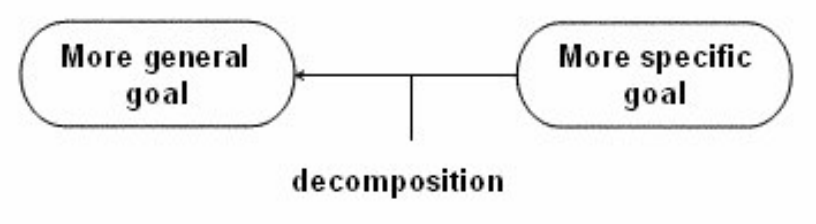

Figura 7: A visão de objetivo do ANote.

A visão do agente especifica as classes dos agentes na aplicação e seus relacionamentos. Nesta visão não há detalhes sobre o comportamento dos agentes, pois especifica apenas a estrutura do sistema. As classes de agentes especificam os papéis que serão executados para alcançar os objetivos definidos na visão de objetivo. Essas classes podem constituir também uma ou mais organizações. Quando dois agentes precisam interagir no sistema, uma associação é criada para representar um relacionamento estrutural. A figura 8 apresenta uma ilustração da visão do agente.

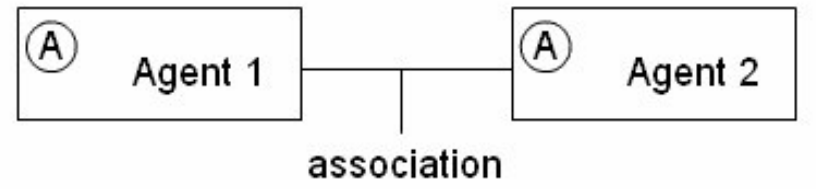

Figura 8: A visão do agente do ANote.

A visão do cenário especifica o comportamento de um cenário do agente. Esse cenário é especificado textualmente, e apresentam planos de como o agente deve atingir os seus objetivos. Esses cenários são úteis por dois motivos: (i) ilustrar como objetivos podem ser atingidos ou não; e (ii) mostrar as circunstâncias em que o agente pode se adaptar, aprender, ou mostrar um comportamento autônomo. Os cenários são gerados por um grupo de ações normais e emergentes, que surgem de comportamentos adaptativos ou em

contextos excepcionais. Essas são partes de um cenário que precisam ser especificados: agente principal, pré-requisitos, plano principal e plano variante. A figura 9 apresenta uma ilustração da visão do cenário.

Cenário I	
Agente Principal	Agente I
Pré-Condições	Evento do ambiente externo
Plano Principal das Ações	Enquanto VERDADE Ação 1 Ação 2 Envia Mensagem para Agente II Fim Enquanto
Interação	Agente II
Variante do Plano	

Figura 9: A visão do cenário do ANote.

A visão de planejamento apresenta as ações de planos descritas no cenário. Essa visão utiliza uma representação de estados com ações e transições para apresentar as ações internas e sua seqüência. As visões de cenários podem ser mapeadas diretamente para estas visões de planejamento, e vice versa. Além disso, uma notação de adaptação do agente é apresentada com transições adaptativas para ações variantes ou comportamentos emergentes. Essas transições adaptativas permitem ao *designer* ilustrar quando e em que circunstâncias um agente deve modificar o seu comportamento. A figura 10 apresenta os componentes da visão de planejamento.

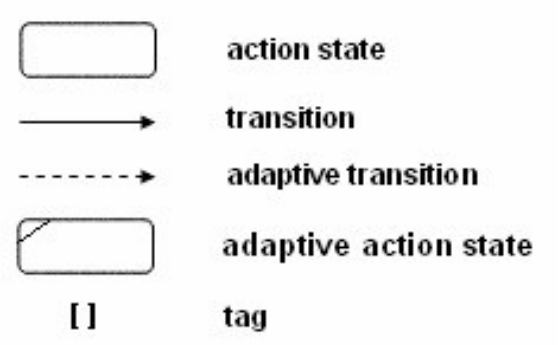

Figura 10: A visão de planejamento do ANote.

A visão de interação é utilizada para representar um conjunto de mensagens trocadas pelos agentes durante a execução de ações de planos. As interações são representadas nesta visão como diagramas de conversação que descrevem o discurso entre os agentes, por exemplo, o protocolo de mensagens e os estados da interação. A figura 11 apresenta os componentes da visão de interação.

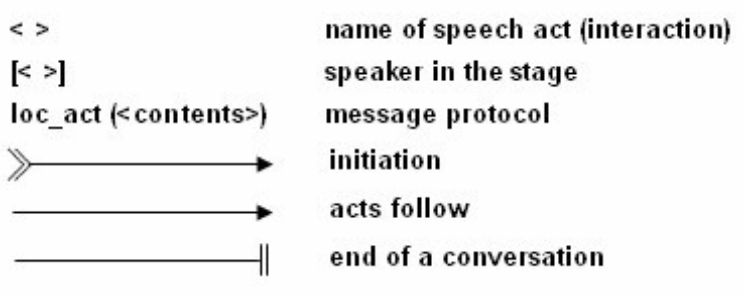

Figura 11: A visão de interação do ANote.

Um sistema multi-agente não é composto apenas por agentes, pois possui também componentes que compõem o ambiente onde os agentes atuam. A visão de ontologia é responsável por especificar os recursos do ambiente e a base de conhecimento do agente. Em ANote, esses componentes que não representam agentes são modelados como objetos. Conseqüentemente, um diagrama de classes UML, podendo ser detalhado com OCL, é utilizado para representar o ambiente do sistema.

A visão de organização modela a sociedade de agentes como uma unidade que oferece serviços. Esses serviços possuem um conjunto de objetivos, e podem ser acessados por uma interface através de protocolos de mensagens. As organizações podem participar de uma relação de dependência para mostrar como elas são organizadas no modelo cliente-servidor. Por exemplo, ilustrar como os agentes de uma organização requisitam os serviços de agentes pertencentes a outra organização.

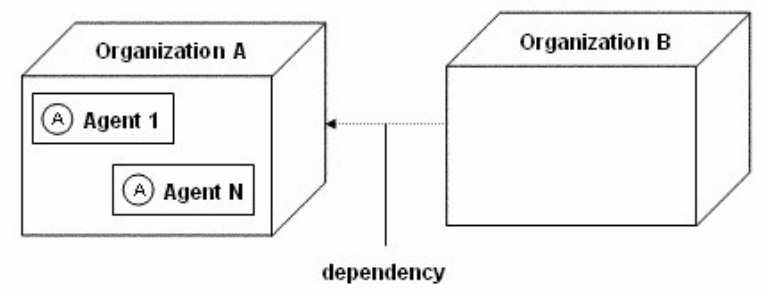

Figura 12: A visão de organização do ANote.

2.2. Frameworks Orientados a Objetos

Frameworks orientados a objetos (Fayad et al., 1999) é uma tecnologia de software que permite a construção rápida e fácil de um software. A demanda por este tipo de tecnologia é cada vez mais constante. Por exemplo, uma ferramenta de *e-commerce* necessita de uma tecnologia que permita mudanças constantes no software para que possa se adequar às constantes alterações nas regras de comércio entre as empresas. A tecnologia dos *frameworks* orientados a objetos permite essas mudanças rápidas de desenvolvimento para que o software possa se adequar às alterações necessárias.

Frameworks orientados a objetos geram aplicações instanciadas com muita rapidez, acompanhando assim a constante necessidade de gerar serviços e produtos inovadores. As suas principais características são:

(i) Reutilização - A possibilidade de reutilizar código nas várias aplicações instanciadas pelo *frameworks* orientados a objetos é que permite diminuir o tempo de desenvolvimento do software. Utilizando uma modelagem orientada a objetos reutilizável (Fayad et al., 1999), *frameworks* permitem o re-uso das classes, subsistemas ou até o sistema para a geração de novas aplicações instanciadas.

(ii) Flexibilidade - *Frameworks* orientados a objetos são flexíveis suficientes para construir variações e extensões no software. Isso permite desenvolver uma gama de aplicações que cobrem um domínio bem amplo.

Frameworks orientados a objetos possuem um núcleo de software, com trechos de código já escritos, e vários pontos de flexibilização (Fayad et al., 1999), que necessitam de desenvolvimentos futuros. Nesses pontos de flexibilização são

implementadas as variações e extensões necessárias para a criação da aplicação final.

Para o desenvolvimento do núcleo é preciso que se faça, antes de qualquer coisa, uma análise de requisitos das várias aplicações pertencentes ao domínio. Nessa análise de requisitos são identificados os pontos em comum das aplicações e as variações. No desenvolvimento dos *frameworks* orientados a objetos são implementados os vários pontos em comum, e indicados os locais onde implementar os vários pontos de flexibilização.

2.3. Padrões de Projeto (Design Patterns)

"Um padrão descreve um problema que ocorre inúmeras vezes em determinado contexto, e descreve ainda a solução para esse problema, de modo que essa solução possa ser utilizada sistematicamente em distintas situações" (Alexander et al, 1977). Conseqüentemente, os *design patterns* são muito importantes para o processo de desenvolvimento de software, especialmente nas áreas de manutenção e reuso de código.

Os *design patterns* são definidos como soluções para problemas recorrentes. Christopher Alexander apresentou em seu trabalho (Alexander et al, 1977; Alexander, 1979) os primeiros *design patterns*. Ele escreveu sobre a sua experiência em engenharia civil para resolver problemas de projetos que apareciam de forma recorrente em construções e cidades. Esse processo de documentação de soluções para problemas recorrentes começou na área de desenvolvimento de software a cerca de 15 anos atrás. Os princípios de Alexander estão presentes na criação das primeiras documentações de *design patterns*.

Os documentos funcionavam como um guia para desenvolver software, e seu público alvo era programadores novatos que queriam adquirir conhecimento com os engenheiros de software mais experientes. Um dos trabalhos mais práticos nessa área resultou no livro do *gang of four* (Gamma et al., 1995) em 1995 por Eric Gamma, Richard Helm, Ralph Johnson e John Vlissides. O livro descreve 23 padrões que surgiram a partir da experiência dos autores no desenvolvimento de software.

Os principais objetivos dos padrões de projetos são (Gamma et al., 1995):

(i) Capturar a experiência e o conhecimento de um especialista em projeto de software;

(ii) Especificar abstrações com um nível acima de classes, objetos ou componentes;

(iii) Definir um vocabulário comum para que os problemas e soluções possam ser discutidos;

(iv) Definir uma arquitetura de software com fácil documentação e manutenção;

(v) Determinar propriedade que auxiliem projetos de arquitetura;

(vi) Auxiliar o desenvolvimento de arquiteturas complexas.

A descrição dos *design patterns* utiliza alguns formatos comuns, tais como *Alexandrian* (Alexander, 1979), *GoF* (Gamma et al., 1995), e *POSA* (Buschmann et al., 1996). O padrão apresentado no capítulo 4 utiliza um formato que combina elementos do *GoF* e *POSA*. Os elementos mais importantes são:

(i) Nome e Intenção – definem de forma sucinta a essência do padrão; A escolha do nome é extremamente importante, pois é incorporado ao vocabulário dos engenheiros de software;

(ii) Contexto – descreve as situações onde o padrão deve ser aplicado, e fornece evidências de sua generalidade;

(iii) Problema – descreve as dificuldades de *design* normalmente encontradas por engenheiros de software;

(iv) Forças - descrevem de forma sucinta os principais objetivos;

(v) Solução – descreve como resolver o problema de *design* já definido. Essa solução define elementos ou participantes que criam um *design* em termos de abstrações de um paradigma de programação. Essa solução deve atingir todos os objetivos de *design* definidos, e resolver os problemas para todas as aplicações do contexto apresentado;

2.4. Algoritmos para Agentes Inteligentes

Esta seção apresenta de forma sucinta alguns algoritmos de Inteligência Artificial, *Machine Learning*, Predição em Séries Temporais, Otimização Combinatória, e Análise de Agrupamento utilizadas nesta tese. Esses algoritmos são úteis para implementar agentes inteligentes que resolvem problemas complexos e aprendem a se adaptar a novas experiências.

A seção 2.4.1 apresenta algumas definições de Inteligência Artificial, e descreve o método de busca Minimax com a técnica de poda Alpha-Beta. A seção 2.4.2 apresenta uma subárea da Inteligência Artificial chamada *Machine Learning*, e apresenta duas técnicas desta área: Redes Neurais e Algoritmos Genéticos. A seção 2.4.3 apresenta algumas técnicas de Predição em Séries Temporais: a Média Móvel e a Exponencial Suavizada. A seção 2.4.4 apresenta a Otimização Combinatória, e a técnica de formulação de Programação Inteira. A seção 2.4.5 apresenta a Análise de Agrupamento e um algoritmo chamado *Isodata*.

2.4.1.Inteligência Artificial

Um agente racional é uma entidade que faz a coisa correta (Russell et al., 1995). Porém, o que é fazer a coisa correta? Os autores afirmam que é equivalente a definir o "como" e "quando" esse agente atinge o sucesso. Conseqüentemente, o termo medida de performance é utilizado para definir esse grau de sucesso e responder o: (i) "como" – um critério para determinar o quanto o agente é bem sucedido, e o (ii) "quando" – o momento certo para fazer essa avaliação.

Winston (Winston, 1992) define Inteligência Artificial (IA) como "o estudo das computações que permitem uma entidade sentir, pensar e agir". A área possui dois objetivos principais para o autor: (i) visão de engenharia – a IA tem o objetivo de resolver problemas reais utilizando um ferramental de idéias sobre representação do conhecimento, a utilização do conhecimento, e integração de sistemas; e (ii) visão científica – a IA é utilizada para determinar quais ideais sobre representação do conhecimento, a utilização do conhecimento, e integração de sistemas, melhor explicam os vários tipos de inteligência.

2.4.1.1.Busca Minimax

O Minimax (Russell et al., 1995) é uma técnica de busca para determinar a estratégia ótima em um cenário de jogo com dois jogadores. O objetivo dessa estratégia ótima é decidir a melhor jogada para um dado estado do jogo. Há dois jogadores no Minimax: o MAX e o MIN. Uma busca em profundidade é feita a partir de uma árvore onde a raíz é a posição corrente do jogo. As folhas dessa árvore são avaliadas pela ótica do jogador MAX, e os valores dos nós internos são atribuídos de baixo para cima com essas avaliações. As folhas do nível minimizar são preenchidas com o menor valor de todos os seus nós filhos, e o nível de maximizar são preenchidos com o maior valor de todos os nós filhos. Essa técnica é utilizada combinada com uma técnica de poda chamada Alpha-Beta (Russell et al., 1995).

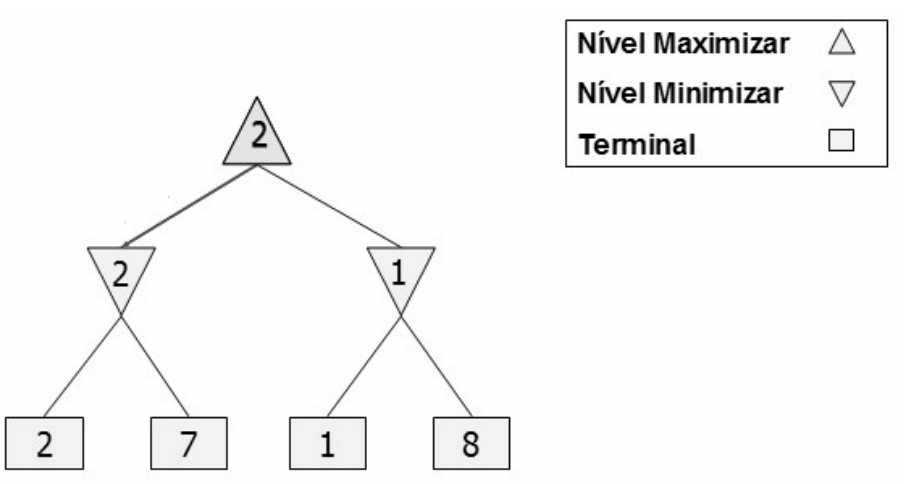

Figura 13: A árvore Minimax.

2.4.2.Machine Learning

Machine Learning é uma subárea da Inteligência Artificial que está relacionada com os programas que aprendem com experiência (Russell et al., 1995). As técnicas de *Machine Learning* são cruciais para fornecer estratégias conhecidas aos agentes em ambiente imprevisíveis e heterogêneos como a Internet.

A área de *Machine Learning* estuda as questões de como construir programas de computadores que melhoram automaticamente com a experiência

(Mitchell, 1997). Essas técnicas podem ser classificadas em relação ao processo de aprendizado que visa melhorar automaticamente a sua performance:

(i) Aprendizado Supervisionado – O algoritmo de aprendizado é treinado com um conjunto de entradas conhecidas e saídas corretas correspondentes. A técnica de aprendizado adapta o conhecimento para que o erro entre a saída atual e a saída correta seja o menor possível. Esse tipo de aprendizado se assemelha ao aprendizado de um aluno com um professor amigo que fornece exemplos e orientações.

(ii) Aprendizado Não Supervisionado – O algoritmo de aprendizado é treinado apenas com as entradas conhecidas, mas sem as saídas corretas correspondentes. O algoritmo procura por padrões interessantes, regularidades ou agrupamentos nessas entradas conhecidas. Esse tipo de aprendizado se assemelha ao aprendizado de um aluno que procura identificar regularidades e generalizações a partir apenas de observações empíricas.

(iii) Aprendizado Semi-Supervisionado - O algoritmo de aprendizado é treinado com um conjunto de dados rotulados (entradas e saídas corretas conhecidas) e outra parte não rotulada (apenas entradas conhecidas). O conjunto não rotulado é utilizado para incrementar a performance de modelos de classificação tradicionais, baseados no processo de aprendizado supervisionado que utilizam apenas os dados rotulados.

(iv) Aprendizado por Reforço - O algoritmo de aprendizado é treinado sem nenhum conjunto de treinamento. O algoritmo deve adquirir o conhecimento através de ações feitas em um ambiente. Esse ambiente simplesmente fornece recompensas e penalidades, ou reforços, por ações boas ou ruins executadas pelo algoritmo. Conseqüentemente, o algoritmo deve adaptar o seu conhecimento para sugerir apenas ações boas. Esse tipo de aprendizado se assemelha ao aprendizado de um aluno com um crítico, que fornece apenas mensagens do tipo “muito bom” ou “péssima alternativa”.

2.4.2.1. Redes Neurais

As Redes Neurais (Mitchell, 1997) são sistemas inspirados nos neurônios biológicos e o poder de processamento paralelo do cérebro com vários neurônios. Uma rede neural artificial é composta por várias unidades de processamento chamadas neurônios artificiais.

Assim como o sistema nervoso é composto por bilhões de células nervosas, a rede neural artificial também é formada por unidades que simulam o funcionamento de um neurônio. Estes módulos devem receber e retransmitir informações tal como um neurônio biológico.

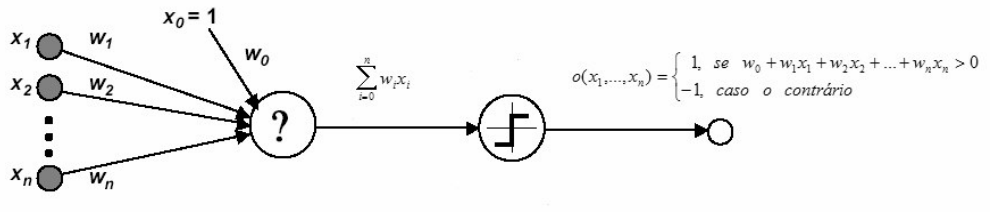

Figura 14: O Neurônio Artificial.

Um tipo de rede neural muito popular é baseado na unidade chamada *perceptron* (Rosenblatt, 1959). O *perceptron* (Mitchell, 1997) recebe um vetor de entrada com números reais, calcula a combinação linear dessas entradas, e gera uma saída igual a 1 se o valor é maior do que um mínimo estabelecido. Caso contrário, o valor é -1. Precisamente, dadas as entradas x_1 até x_n , a saída $o(x_1, ..., x_n)$ é calculada da seguinte forma:

$$o(x_1,...,x_n) = \begin{cases} 1, & se \quad w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n > 0 \\ -1, & caso \quad o \quad contrário \end{cases}$$

Onde cada w_i é uma constante de valor real, ou peso, que determina a contribuição da entrada x_i na saída do *perceptron*. Note que a quantidade $-w_0$ é um valor mínimo que a combinação $w_1x_1 + ... + w_nx_n$ precisar atingir para que o *perceptron* gere a saída 1.

O aprendizado no *perceptron* é um processo de escolha de valores para os pesos $w_0,...,w_n$. Esse processo ocorre com a adaptação dos pesos a partir de um

conjunto de entradas conhecidas e saídas corretas correspondentes. Uma maneira de aprender todos esses pesos é começar com valores aleatórios, e depois iterativamente aplicar o *perceptron* para cada exemplo do conjunto de treinamento, modificando os seus pesos toda vez que a saída gerada diferir da saída esperada. A Regra do Perceptron (Rosenblatt, 1959) é um algoritmo para atualizar os pesos do *perceptron* a cada passo no processo de aprendizado:

$$w_i = w_i + \eta \, . \, (t - o).x_i$$

Onde w_i é i-ésimo peso do *perceptron*, x_i é a i-ésima entrada do *perceptron*, t é a saída esperada para o exemplo de treinamento, o é a saída gerada pelo *perceptron*, e η é uma constante positiva chamada taxa de aprendizado. Essa taxa tem o objetivo de regular o grau de mudança dos pesos a cada passo, e normalmente é um valor pequeno como, por exemplo, 0.1.

2.4.2.2. Algoritmos Genéticos

Os algoritmos genéticos (Winston, 1992) utilizam um procedimento de busca inspirado na evolução natural. Os passos utilizados dentro dos algoritmos usam rotinas análogas, de uma certa forma, ao cruzamento de indivíduos, cruzamento de cromossomos, mutação de genes, e seleção natural.

Um ponto importante a ressaltar na seleção natural é a baixa eficiência apresentada ao utilizar mecanismos simples de seleção. Para obter resultados melhores, o mecanismo precisa levar em consideração a diversidade entre os indivíduos e a capacidade individual.

Nos algoritmos genéticos, a pouca diversidade apresenta um resultado interessante: uma busca apenas de um ótimo local. Porém, o objetivo da busca não é esse, mas sim encontrar um ótimo global. Para encontrar esse ótimo global, a diversidade se torna um fator fundamental. Assim como na evolução natural, os algoritmos genéticos sacrificam parte da sua população em ótimos locais para que outros indivíduos consigam atingir o ótimo global.

Um cromossomo nesses algoritmos representa um ponto no espaço de busca, e o objetivo é criar novos cromossomos a partir dos existentes que estejam mais próximos do ótimo global. É importante notar que cada cromossomo:

(i) possui uma lista de elementos chamados de genes;

(ii) representa um ponto no espaço de busca com um valor associado que representa o quão forte (*fitness*) esse indivíduo é;

(iii) um novo cromossomo pode ser criado utilizando o conceito de cruzamento de outros dois cromossomos;

(iv) um novo cromossomo também pode ser criado utilizando o conceito de mutação, onde esse novo cromossomo possui um gen modificado.

O algoritmo abaixo imita a evolução natural (Winston, 1992).

(i) crie uma população inicial de um cromossomo;

(ii) faça a mutação de um ou mais genes utilizando um ou mais cromossomos existentes em sua população, produzindo assim um novo cromossomo a cada nova mutação;

(iii) faça o cruzamento de um ou mais cromossomos;

(iv) adicione os cromossomos novos gerados a partir da mutação e do cruzamento à população atual;

(v) crie uma nova geração mantendo os melhores cromossomos de sua população, junto com alguns outros cromossomos escolhidos aleatoriamente de sua população atual. Nessa escolha aleatória, aumente a probabilidade de escolha para os cromossomos que tiverem um "*fitness*" maior;

(vi) volte ao passo dois até que um cromossomo de sua nova geração tenha atingido o objetivo (maior "*fitness*").

2.4.3. Predição em Séries Temporais

A predição em séries temporais, ou técnicas de *Forecasting* (Bowerman et al, 1993; Chopra et al., 2004), são muito utilizadas para calcular valores futuros de demanda ou preços baseados em dados históricos. Por exemplo, os métodos de *forecasting* de demanda são a base do planejamento estratégico em uma cadeia de suprimento. O objetivo das técnicas de *forecasting*, na pratica, é apoiar as decisões, tais como (i) quantos produtos devem ser produzidos esse mês? (ii) qual

o tamanho do lote de compra de uma determinada matéria prima? (iii) quando efetuar esse pedido de compra de matéria prima?

Uma das técnicas de *forecasting* em séries temporais mais simples se chama Média Móvel, e pode ser definida pela seguinte fórmula:

$$s_i = \frac{1}{n} \sum_{j=i}^{i+n-1} a_j.$$

onde:

$\{a_i\}_{i=1}^{N}$ é a série histórica (*N* termos)

$\{s_i\}_{i=1}^{N-n+1}$ é a seqüência de *n* preços previstos (janela de tamanho *n*)

Uma outra técnica de *forecasting* em séries temporais se chama Exponencial Suavizada, e pode ser definida pela seguinte fórmula:

$$s_i = \alpha.a_{i-1} + (1-\alpha).s_{i-1}$$

onde:

$\{a_i\}_{i=1}^{N}$ é a série histórica (*N* termos)

$\{s_i\}_{i=1}^{n}$ é a seqüência de *n* preços previstos

α é a constante de suavização

Observe que pela sua definição recursiva, o s_i pode re-escrito como:

$$s_i = \alpha.a_{i-1} + \alpha.(1-\alpha).a_{i-2} + \alpha.(1-\alpha)^2.a_{i-3} + ...$$

Essa fórmula mostra o efeito do desconto no valor da informação introduzida por α, que é usualmente escolhido no intervalo $0 < \alpha < 1$.

2.4.4. Otimização Combinatória

Um *Solver*, ou otimizador, (Solver.com, 2005; Wolsey, 1998) é uma ferramenta de software que ajuda a encontrar a melhor maneira de alocar recursos.

Esse recursos podem ser matéria prima, homem – hora, ou qualquer coisa com um estoque limitado. A solução ótima pode ser essa alocação que gere o maior lucro, o menor custo, ou encontre a melhor qualidade possível. Para utilizar um *solver*, um modelo precisa ser criado para especificar:

(i) Os recursos a serem utilizados, ou variáveis de decisão;

(ii) Os limites de recursos que podem ser utilizados, ou restrições;

(iii) A medida de otimização, ou o objetivo.

A formulação de programação inteira é uma linguagem matemática para criar esses modelos de otimização.

Seja:

$\max\{\ cx : Ax \leq b, x \geq 0\ \}$ um programa linear.

Onde A é uma matriz de m por n elementos, c é um vetor linha de dimensão n, b é um vetor coluna de dimensão m, e x é um vetor coluna de dimensão n com variáveis de decisão. Na programação inteira as componentes de x são todas variáveis inteiras. Conseqüentemente:

(i) A Função Objetivo é definida por:
$\max cx$

(ii) Variáveis de decisão são definidas por:
$x \geq 0$, e x possui apenas valores inteiros.

(iii) Restrições são definidas por:
$Ax \leq b$

A tradução de um problema para esta formulação nem sempre é uma tarefa simples. Definir variáveis e restrições para esse modelo de otimização quase sempre requer um trabalho de engenharia de algoritmos, pois o problema de decisão da programação inteira é NP-completo (Wolsey, 1998).

2.4.5. Análise de Agrupamento

O principal objetivo dos algoritmos de Agrupamento (Duda et al., 1973; Lebart et al., 1997) é agrupar dados que possuem padrões semelhantes. Esses grupos também são conhecidos por *clusters* ou partições. O problema de

agrupamento pode ser considerado também como um método de particionamento de um espaço. O ideal é que esse particionamento organize os dados de tal maneira que se possa tomar decisões corretas. Caso isso não seja possível, o objetivo é obter uma probabilidade de erro bem pequena.

Os algoritmos de agrupamento se tornam muito interessantes para a área de Inteligência de Mercado que está preocupada também em encontrar grupos de consumidores com padrões de consumo ou hábitos semelhantes. Esses padrões encontrados permitem criar produtos e serviços personalizados para esses grupos, aumentando assim o foco da empresa e a satisfação dos clientes.

2.4.5.1. Algoritmo Isodata

O *Isodata* (Duda et al., 1973; Lebart et al., 1997) é um algoritmo de agrupamento muito conhecido pela sua simplicidade e rapidez de processamento. A entrada desse algoritmo é o conjunto *I* de *n* indivíduos a serem particionados, onde cada indivíduo possui *p* atributos. Esse conjunto representa os dados a serem agrupados. Suponha que o espaço R^p possui uma distância *d* entre os *n* pontos-indivíduo (por exemplo, a distância euclidiana). Além disso, um limite máximo de *q* classes é estabelecido para o resultado do algoritmo.

O algoritmo (Lebart et al., 1997) ilustra as etapas que determinam as classes:

Etapa 0: Determine *q* centros provisórios das classes (por exemplo, de maneira pseudo-aleatória). Os centros são:

$$\{C_1^0,...,C_k^0,...,C_q^0\}$$

Calcule uma primeira partição P_0 do conjunto *I* em *q* classes:

$$\{I_1^0,...,I_k^0,...,I_q^0\}$$

Um individuo *i* pertence a uma classe I_k^0 se ele é mais próximo a C_k^0 do que a qualquer outro centro.

Etapa 1: Determine os *q* novos centros das classes:

$$\{C_1^1,...,C_k^1,...,C_q^1\}$$

Calculando os centros de gravidade a partir das partições:

$$\{I_1^0,...,I_k^0,...,I_q^0\}$$

Esses novos centros permitem o cálculo de uma nova partição P_1 de I, utilizando a mesma regra empregada para determinar P_0. A nova partição P_1 é formada pelas classes:

$$\{I_1^1,...,I_k^1,...,I_q^1\}$$

Etapa *m*: Determine os q novos centros das classes:

$$\{C_1^m,...,C_k^m,...,C_q^m\}$$

Calculando os centros de gravidades a partir das partições:

$$\{I_1^{m-1},...,I_k^{m-1},...,I_q^{m-1}\}$$

Esses novos centros permitem o cálculo de uma nova partição P_m de I, utilizando a mesma regra empregada para determinar P_0. A nova partição P_m é formada pelas classes:

$$\{I_1^m,...,I_k^m,...,I_q^m\}$$

Aos poucos o processo converge e o algoritmo pode terminar o seu processamento quando duas iterações sucessivas apresentarem as mesmas partições. Uma maneira para determinar se as partições são idênticas é utilizando a medida da variância intra-classes. Muitas vezes, o algoritmo é implementado para não esperar a sua convergência completa. Mas para isso, uma margem de erro é definida para que seja possível determinar o ponto de parada. Essa margem de erro permite calcular quanto que uma partição pode diferir da partição sucessiva.

O resultado das partições é representado pelos centros ou centróides. Um centróide pode ser considerado um indivíduo que melhor representa as características de sua partição.