UNIVERSIDADE FEDERAL DE GOIÁS – UFG

CAMPUS CATALÃO – CaC

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO – DCC

Bacharelado em Ciência da Computação

Projeto Final de Curso

Abordagem Imunológica para Elaboração Automática de Quadros Horários

Autor: Thiago de Paulo Faleiros

Orientador: Prof. Dr. Alexsandro Santos Soares

Catalão - 2007

Thiago de Paulo Faleiros

Abordagem Imunológica para Elaboração Automática de Quadros Horários

Monografia apresentada ao Curso de
Bacharelado em Ciência da Computação da
Universidade Federal de Goiás - Campus Catalão
como requisito parcial para obtenção do título de
Bacharel em Ciência da Computação

Área de Concentração: Inteligência Artificial
Orientador: Prof. Dr. Alexsandro Santos Soares

Catalão - 2007

Faleiros, Thiago de Paulo

Abordagem Imunológica para Elaboração Automática de Quadros Horários/ Prof. Dr. Alexsandro Santos Soares - Catalão - 2007

Número de páginas: 71

Projeto Final de Curso (Bacharelado) Universidade Federal de Goiás, Campus Catalão, Curso de Bacharelado em Ciência da Computação, 2007.

Palavras-Chave: 1. Sistema Imunológico Artificial. 2. Elaboração de Quadro Horário. 3. Algoritmos Evolutivos

Thiago de Paulo Faleiros

Abordagem Imunológica para Elaboração Automática de Quadros Horários

Monografia apresentada e aprovada em _____ de _____
Pela Banca Examinadora constituída pelos professores.

_____

Prof. Dr. Alexsandro Santos Soares – Presidente da Banca

_____

Avaliador A

_____

Avaliador B

Aos meus pais, pelo incentivo e amparo.

AGRADECIMENTOS

A Deus, pois a ele agradeço a força de vontade para realizar este trabalho. Uma força com que tanto necessito para superar desafios, e acredito que novos virão! Entretanto, pela fé e por mais força de vontade eles serão superados.

Também tenho o justo dever de agradecer meus pais, João Batista de Paulo e Vilma Faleiros de Paulo, pelo apoio e incentivo em meus estudos. Sem esse apoio, esse trabalho, e qualquer outro realizado na graduação, seria praticamente impossível de ser concluído com dedicação total.

Aos professores do Departamento de Ciência da Computação da Universidade Federal de Goiás - Campus Catalão (UFG-CAC) por fornecerem uma boa instrução.

A todos meus amigos e alunos de Ciência da Computação da UFG-CAC, que proporcionam constantes trocas de experiências vividas durante nossa atual fase de descobrimento da ciência.

Aos amigos Rodney Elias Marçal e Leonardo Garcia Marquez. Um por me auxiliar na revisão deste trabalho, e o outro por criar essa excelente formatação de documento em LaTeX.

Ao meu orientador, Prof. Dr. Alexsando Santos Soares, pelo auxílio e orientação. Desvendando-me o ”caminho das pedras”na pesquisa científica.

”if an adaptive pool of antibodies can produce ’intelligent’ behavior, can we harness the power of this computation to tackle the problem of preference matching, recommendation and intrusion detection?”. (Dispankar Dasgupta and Uwe Aickelin)

RESUMO

Faleiros, T. Abordagem Imunológica para Elaboração Automática de Quadros Horários. Curso de Ciência da Computação, Campus Catalão, UFG, Catalão, Brasil, 2007, 71p.

Este trabalho apresenta o uso de uma técnica meta-heurística, inspirada no sistema imunológico dos animais vertebrados, para implementar e validar um algoritmo de otimização de quadro de horários. São apresentados os conceitos biológicos envolvidos nos processos imunológicos, onde a linha de pesquisa do chamado Sistema Imunológico Artificial é baseado. Além disso, é descrito o problema de elaboração automática de quadros horários de cursos universitários, de maneira que, o modelo construído com base no paradigma orientado a objeto seja aplicado a instâncias de testes experimentais. Uma implementação foi construída utilizando a linguagem de programação Java para a obtenção dos resultados, e a partir disso, comparações com outros trabalhos foram realizadas.

Palavras-Chaves: Sistema Imunológico Artificial, Elaboração de Quadro Horário, Algoritmos Evolutivos

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A forma tradicional de soluções tecnológicas existentes, aquelas definidas por conjuntos de regras bem definidas, e, de certa forma simplista, onde o problema global é simplificado para que possa ser estudado e previsto, não mostra sucesso para a resolução de certos tipos de problemas [de Castro Silva, 2001]. Esses problemas, em geral, exigem grande esforço computacional, utilizando tais formas de soluções, devido, principalmente, à complexidade do problema, necessitam de técnicas que busquem ir além do tradicional paradigma.

Entretanto, de onde buscar essas formas ”não tradicionais”e como obter a idéia para resolver tais problemas? Talvez, essa seja a pergunta que vários pesquisadores se questionaram antes de observar que, na própria natureza, alguns processos semelhantes a problemas computacionalmente difíceis já eram resolvidos.

Hoje, algumas técnicas inspiradas na natureza são bastante conhecidas e bem sucedidas para resolução de problemas com complexidade elevada [de Castro Silva, 2001]. Tem-se o exemplo do cérebro humano (redes neurais), como um sistema de processamento bastante citado em livros da área de inteligência artificial [Russell e Norvig, 1995], sobretudo pela fascinação e a necessidade de compreender todo o processo de inteligência humana a fim de se buscar, com isso, um sistema computacional com semelhantes características. Entender como o cérebro com dimensões físicas reduzidas (quando comparado a computadores de grande porte), poderia perceber, compreender e manipular um mundo extremamente amplo e diversificado, traria base para estudos no campo computacional.

Da mesma forma, tem-se também, na natureza, o sistema imunológico dos vertebrados, que está proporcionando uma fonte de inspiração para processos computacionais. À maneira como os conjuntos gerais de princípios imunológicos trabalham levam a desenvolvimento de novas ferramentas [Somayaji et al., 1997].

O sistema imunológico artificial está se relacionando, principalmente, com os sistemas de segurança computacionais. Assim, a analogia entre o sistema de defesa do organismo e o sistema de segurança computacional é facilmente criada quando leva em consideração os seguintes princípios: a necessidade de manter a sobrevivência (ou funcionamento) do sistema, a capaci-

dade de detecção de intrusos e a de se recuperar após danos. Tais princípios nem sempre encontrados em sistemas de segurança computacionais. Por essa razão, existem vários trabalhos que buscam no sistema imunológico uma arquitetura para construção de sistema de segurança [de Paula, 2004].

Outros princípios encontrados no sistema imunológico, como autonomia, adaptabilidade, cobertura dinâmica, detecção de anomalia, dão a capacidade de determinar a quantidade de detectores para combater uma invasão, aprender a combater novos antígenos (corpos estranhos), renovar e melhorar o repertório de anticorpos, formar uma arquitetura distribuída sem nenhum mecanismo central. Com todas essas características, o sistema imunológico é uma rica fonte de inspiração [Somayaji et al., 1997]. Além disso, o processo de otimização na geração de anticorpos, a regulação, o mecanismo de adaptação e vários processos que governam a complexa arquitetura formada pelo sistema imunológico o tornam um fantástico centro de processamento [de Castro Silva, 2001].

Neste trabalho, portanto, é utilizado um modelo imunológico, o algoritmo de seleção clonal, inspirado no processo de regulação de anticorpos, com o propósito de aplicar tais processos no problema de geração automática de quadro horário, e implementar tal modelo afim de validar sua aplicabilidade no problema estudado.

O problema de geração automática de quadros horários é demasiado complexo [Cooper e Kingston, 1995, Schaerf, 1995], além de que as várias restrições exigidas, a quantidade de recursos que necessita ser agendada, dentro da tabela de horários, demonstra a dificuldade de automatizar tal processo.

Vários trabalhos relatam a complexidade da formulação de quadros horários de cursos universitários, sendo tal problema pertecente a classe de complexidade NP-Completo [Cooper e Kingston, 1995]. Também, várias formas de resolvê-lo foram propostas, recentemente, técnicas dentro da área de inteligência artificial, estão sendo aplicadas para melhorar o processo de geração das soluções.

Neste estudo, é buscada uma solução totalmente automática para o problema, sendo dependente do processo computacional a determinação da melhor solução. Não serão consideradas técnicas interativas que envolvam intervenções humanas para o direcionamento de uma melhor solução.

O presente trabalho está organizado, da seguinte forma. O capítulo 2 apresenta uma revisão dos métodos de sistemas imunológicos artificiais, introduzindo seus conceitos principais e descrevendo os algoritmos de seleção clonal e seleção negativa, além de oferecer todo o embasamento biológico necessário para um entendimento da técnica usada.

O capítulo 3 especifica o problema estudado, a determinar os tipos principais de quadros horários universitários, que são trabalhados nas pesquisas sobre esse problema. Especifica, ainda, o conjunto de restrições, a complexidade do problema, a explicar como são aplicadas técnicas de algoritmo genéticos e *simulated annealing*.

No capítulo 4, é apresentada a arquitetura proposta para a solução do problema objeto de interesse. Nele, serão mostradas, de modo detalhado, além da modelagem computacional do problema, com base do paradigma da programação orientada a objetos, o algoritmo e seus operadores.

O capítulo 5 trata dos testes executados, tendo em vista a análise do problema, o estudo do quadro de horários de curso universitário gerado, onde os gráficos de evolução e os valores de afinidade são apresentados como forma de relatar os resultados obtidos da implementação.

Por fim, no capítulo 6, são feitas as conclusões e sugestões para trabalhos futuros.

Capítulo 2

Sistema Imunológico Artificial

Sobreviver em um ambiente repleto de organismos agressores, como vírus, bactérias, fungos e parasitas, espalhados pelo ar ou água, e também, constantemente, procurando por organismos hospedeiros, é o grande desafio que o sistema de defesa dos animais vertebrados vem tratando há gerações.

Inserido nesse ambiente hostil, como é possível o ser humano sobreviver por mais de 70 anos, sendo que, durante toda a sua vida, está sujeito a ataques (vírus, bactérias, parasitas, entre outras ameaças)? Qual o procedimento de defesa do organismo humano para garantir tal proteção? A completa resposta dessas questões é o desejo de cientistas da área imunológica, pois além de dar o entendimento de como o organismo humano reage ao ataque no organismo, possibilitará a descoberta de novas curas [de Castro Silva, 2001].

Portanto, todo esse mecanismo natural dos organismos vivos é resultado de um grande período de adaptação e evolução. Por milhares de anos, a natureza vem tratando problemas que, em uma analogia no ambiente computacional, é extremamente complexo, e por tal motivo, os cientistas da computação estão buscando, na natureza, a inspiração para projetar dispositivos básicos que executem tarefas específicas.

2.1 Sistema Imunológico Natural

Inicialmente, para o entendimento dos mecanismos do sistema imunológico artificial, é necessário um entendimento sobre alguns aspectos biológicos envolvidos no processo imunológico.

Pode-se dividir o sistema imunológico natural em duas partes importantes do sistema geral: o sistema imunológico inato e o sistema imunológico adaptativo [Hofmeyr, 2000].

O sistema imunológico inato é a primeira linha de defesa contra partículas estranhas, sendo caracterizadas por células fagocitárias, responsáveis pela ingestão de partículas estranhas ao organismo, e por outros tipos de defesas, como barreiras físicas (pele) e químicas. O termo ”inato”se refere à parte do sistema imunológico adquirida desde o nascimento, não apresentando

nenhum mecanismo de adaptação além de constituir uma rápida e inicial linha de defesa que protege o organismo [de Castro Silva, 2001].

Em um segundo nível, está presente o sistema imunológico adaptativo, capaz de reconhecer microorganismos como vírus, bactéris, fungos, protozoários, helmintos e alguns tipos de vermes e em seguida iniciar uma resposta para tal infecção [Hofmeyr, 2000]. Outra característica importante, existente no sistema imunológico adaptativo, é a chamada memória imunológica. Ela é capaz de gravar informações de tal agente patogênico, em uma primeira detecção, a fim de acelerar uma resposta a alguma infecção que possa ocorrer em qualquer momento futuro. Todos as descrições deste trabalho, como os mecanismos de detecção de antígenos, seleção clonal, seleção negativa, são processos inclusos no complexo sistema imunológico adaptativo.

A complexidade desse sistema de proteção é notória devido à vasta quantidade de células e moléculas, todas trabalhando de forma harmônica, objetivando respostas a substâncias estranhas ao organismo, os chamados antígenos [Somayaji et al., 1997].

Todo esse processo complexo pode ser descrito, de um modo simplificado (veja figura 2.1), quando algum patógeno (agente infeccioso) é ingerido por uma molécula apresentadora de antígeno especializado (APCs). Após tal processo de ingestão, o patógeno é processado e apresentado à superfície da molécula APC. Em seguida, as células T, as quais possuem moléculas receptoras, em sua superfície, capazes de reconhecer antígenos processados pela APC, vão para o estado de ativação, onde se dividem e secretam sinais químicos (linfocina) que sinalizam a outros componentes do sistema imunológico, como as células B, algum antígeno encontrado (veja esse processo na figura 2.1).

As células B, semelhantes às células T, também possuem moléculas receptoras de especificidade única em sua superfície, entretanto, as células B são capazes de reconhecer antígenos livres no organismo, sem a necessidade da ingestão e digestão das células apresentadoras de antígenos. Células B possuem anticorpos ligados à membrana e que são secretados quando ativadas. Cada célula B possui apenas um único tipo de anticorpo, que através da complementaridade é capaz de se ligar a uma porção do antígeno chamada epítopo. Cada anticorpo é capaz de reconhecer um epítopo, sendo que, se um antígeno contém vários epítopos, logo, uma molécula de anticorpo é capaz de reconhecer mais de um antígeno[de Castro Silva, 2001].

Note que todo o processo é realizado com a cooperação entre o conjunto de células formadoras do sistema imunológico, sendo cada uma responsável por uma função relativamente simples, e, no conjunto, realiza um trabalho extremamente complexo.

Para um entendimento mais apurado do processo (principalmente para ajudar em modelos computacionais), algumas questões devem ser esclarecidas, inicialmente, a respeito de como é realizado o processo de detecção, como é o processo de criação de detectores e o controle dos recursos finitos do organismo para combater uma quantidade infinita de antígenos.

Nas próximas seções, essas questões serão discutidas, pois suas respostas poderão servir como base na construção de modelos computacionais para otimizar processos, recursos e,

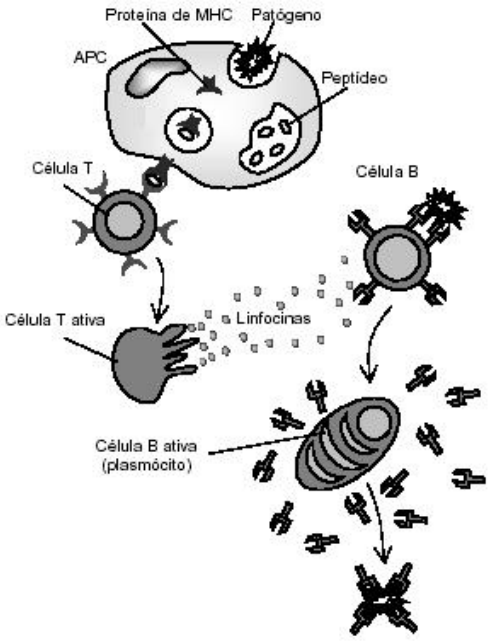

Figura 2.1: Esquema simplificado de ativação e reconhecimento do Sistema Imunológico (figura retirada de [de Castro Silva, 2001])

também, possibilitar a construção de modelos de segurança computacional [Somayaji et al., 1997].

2.1.1 Maturação, Afinidade e Teoria da Seleção Clonal

Os linfócitos são as principais células de defesa dos animais vertebrados, existentes, exclusivamente, nos mamíferos, proporcionando um nível de proteção maior do que aquele existente no sistema inato, especialmente, em uma segunda invasão de um específico antígeno [de Castro Silva, 2001]. Isso se deve à memória imunológica, ou seja, ao reconhecimento do mesmo antígeno, caso ele entre novamente em contato com o organismo (essa imunidade adquirida é bastante notória através da vacinação e da exposição inicial à doença).

Cada linfócito virgem que penetra na corrente circulatória é portador de receptores de antígenos com uma única especificidade. A especificidade deste receptor, contudo, é determinada graças ao processo de rearranjo gênico, realizada durante o desenvolvimento do linfócito na medula óssea e no timo [de Castro Silva, 2001]. O processo de rearranjo gênico, realizado na medula óssea e no órgão linfático chamado timo, é importante, principalmente, para o entendimento de algumas técnicas estudadas no sistema imunológico artificial, para entender como ocorre a criação dos detectores e a variação de afinidade.

O processo é realizado a partir de quatro cadeias polipeptídicas que formam o anticorpo [de Castro Silva, 2001]. O fato é que, no genoma de cada indivíduo, existem múltiplos segmentos gênicos com seqüências relativamente distintas, que codificam uma parte do receptor do anticorpo, ou seja, existem bibliotecas de fragmentos gênicos que são agrupados de forma

aleatória para formar uma molécula completa de anticorpo.

Essas cadeias polipeptídicas (fragmentos gênicos) do anticorpo formam a região de ligação ao antígeno, de modo que, cada um deles, possui um receptor de antígeno. O anticorpo é uma glicoproteína composta por quatro cadeias polipeptídicas: duas cadeiras leves (l) idênticas e duas cadeias pesadas (h) também idênticas, como ilustrado na Figura 2.2. Análises realizadas por pesquisadores constataram que as cadeias polipeptídicas das moléculas de imunoglobulina são compostas por uma região aminoterminal altamente variável (região variável) e uma região carboxiterminal (região constante) com poucos tipos distintos [Tonegawa, 1983, Janeway et al., 2000, Perelson e Weisbuch, 1997].

A região variável, ou região-V, é responsável pelo reconhecimento antigênico, em que subregiões são usualmente chamadas de regiões determinadas por complementariedade. A região constante, ou região-C, é responsável por uma variedade de funções efetoras, como fixação do complemento e ligação a outros receptores celulares do sistema imune. Além dessa união de fragmentos gênicos, para a formação do anticorpo (molécula de imunoglobulina ativa nos linfócitos), são acrescentadas ainda altas taxas de mutações aumentando a diversidade genética. Portanto, para a formação da diversidade populacional dos anticorpos, existem dois mecanismos, um de rearranjo gênico e outro de mutação [de Castro Silva, 2001].

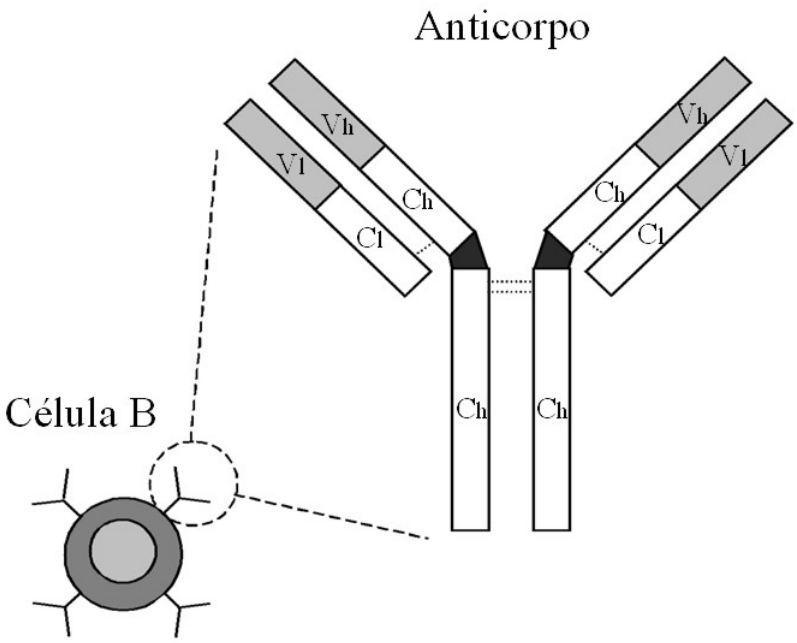

Figura 2.2: Célula B com destaque para a molécula de anticorpo em sua superfície [de Castro Silva, 2001].

Quanto ao reconhecimento antigênico (anticorpo reconhecendo um antígeno), a combinação é efetuada por complementaridade entre uma região de ligação do receptor e uma porção do antígeno chamada epítopo. Vê-se que existe uma afinidade entre o anticorpo e antígeno, sendo

que, quanto maior a afinidade, melhor é o anticorpo para o reconhecimento[Hofmeyr, 2000]. Assim, todo o processo descrito acima, de rearranjo gênico e mutação, é relevante para uma maior variabilidade da população de anticorpos.

Existe também o processo de maturação de afinidade, em que os anticorpos com maior afinidade são selecionados, de maneira semelhante à seleção natural (apenas os melhores indivíduos sobrevivem) [Ao, 2005], sendo que, na imunologia, esse processo é chamado de teoria da seleção clonal [de Castro e Zuben, 2000].

Cada célula de anticorpo apresenta uma forma de combinação com o antígeno, entretanto, não é garantido, que todas as células irão trabalhar como células efetoras na detecção de um específico antígeno, devido à baixa afinidade ou às regiões de ligação não serem complementares [Hofmeyr, 2000]. A fim de gerar anticorpos na quantidade correta para a proteção contra a infecção, aquelas células ativadas (que já tenha detectado um antígeno e possuem uma boa afinidade para detecção) irão se proliferar por clonagem em todo o organismo. Esse processo, chamado de teoria da seleção clonal (veja a figura 2.3) [de Castro e Zuben, 2000], supostamente explica o mecanismo de memória imunológica [de Castro Silva, 2001], em que um animal, quando é exposto a uma doença em um primeiro momento, é submetido a anticorpos que combatem a doença. Somente os anticorpos que responderem ao estímulo antigênico restarão no organismo e, em um segundo momento, em uma infecção futura, aquelas células, chamadas de células de memória, que já estão presentes no organismo, devido ao primeiro estímulo, são reativadas, sendo, dessa vez, capazes de secretar anticorpos para executar uma resposta mais rápida e mais efetiva.

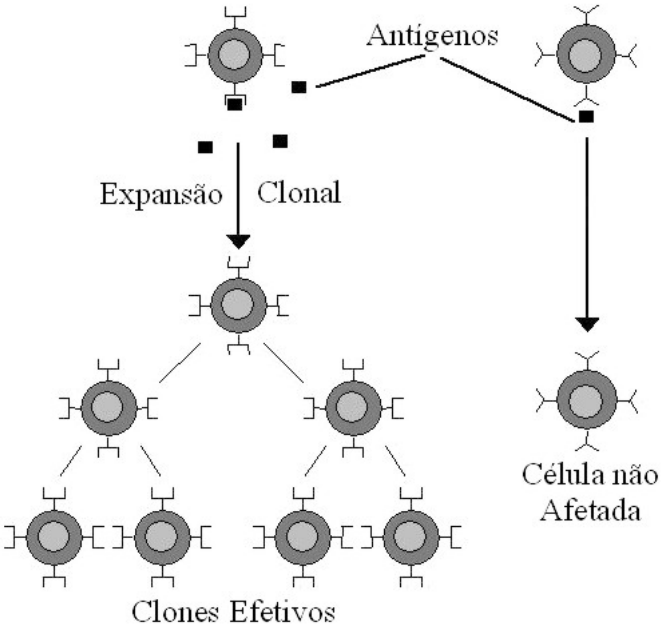

Figura 2.3: Processo de Seleção Clonal

O princípio da seleção clonal está associado, principalmente, a uma resposta imune adaptativa a um estímulo antigênico, a estabelecer que apenas aquelas células capazes de reconhecer um determinado estímulo antigênico irão ser selecionadas para o processo de clonagem e mutação.

O estímulo antigênico pode ser compreendido mais precisamente com a seguinte seqüência de fatos [de Castro e Zuben, 2000]: inicialmente, quando uma célula B, um linfócito responsável pela produção de anticorpos, é exposta a um antígeno, e também por um segundo sinal co-estimulatório efetivado pela célula T, é estimulado a proliferação (dividir) e transformação em células B maturas, capazes de secretar anticorpos em altas taxas, essas células são os plasmócitos. Além de poder se transformar em plasmócitos, as células B também se diferenciam em células B de memória. As células B de memória percorrem todo o organismo através do sangue durante um longo período, e não produzem anticorpos, mas, quando expostas em uma re-infecção do antígeno, é iniciado o processo de diferenciação em plasmócito, capazes de produzir anticorpos pre-selecionados com alta afinidade [de Castro Silva, 2001].

Visto que, no processo de criação dos anticorpos, como descrito anteriormente, o mecanismo de rearranjo gênico e mutação aumentam a variabilidade dos anticorpos e a seleção clonal seleciona aqueles com melhor afinidade, resta, ainda, compreender como o sistema imunológico, possuindo recursos finitos, consegue criar anticorpos que cubram um espaço de antígenos que é extremamente superior. É estimado que o sistema imune tenha aproximadamente 10^6 proteínas, e que existem 10^{16} proteínas diferentes ou padrões a serem reconhecidos [Hofmeyr, 2000].

Para superar esse problema, a seleção clonal atua diversificando e aumentando dinamicamente a diversidade de detectores. Constantemente os detectores, cada um cobrindo uma parte do espaço de antígenos, são renovados pelo processo de seleção clonal, e apenas aqueles que reconhecem um antígeno são permanecidos, clonados e mutados, aumentando a diversidade e o espaço de coberptura da população de anticorpos [Hofmeyr, 2000].

2.1.2 Seleção Negativa das células T

Para que o sistema imunológico funcione corretamente, é necessário que ele seja capaz de reconhecer o que são células estranhas ao organismo, ou organismos invasores, e células próprias do organismo. Para tal diferenciação, uma teoria que explica essa capacidade é a seleção negativa no timo [Stewart e Coutinho, 2004, Forrest et al., 1994].

O timo é um órgão localizado na porção superior do tórax onde ocorre o desenvolvimento da célula T, sendo protegido por uma barreira sanguínea, composta por uma grande quantidade de células apresentadoras de antígenos (APCs), que faz com que essas APCs apresentem complexos MHC-próprio ao repertório de células T que estão sendo formadas [de Castro Silva, 2001]. O Complexo de histocompatibilidade principal (MHC $-$ *Major Histocompatibility Complex*),

são proteínas presentes nas células, que apresentam os antígenos fragmentados na superfície da APC, para a detecção das células T.

De um modo mais simples, pode-se dizer que o timo possui uma barreira contra células não-próprias do organismo. Fazendo com que, em seu interior, as células de anticorpos em formação combinem apenas com outras células próprias. Nesse caso, que aqueles anticorpos (anticorpos formados dentro do timo) que detectarem células próprias morrerão (processo chamado de deleção clonal), restando apenas aqueles que não combinarem com nenhuma mólécula própria, consequentemente, célula que reconhece apenas substâncias não próprias.

Logo, espalhará pelo organismo, apenas aqueles anticorpos que passarem por esse processo de ”validação”. Isso é chamado de seleção negativa [de Castro Silva, 2001], essa teoria explica a distinção de próprio e não próprio realizada pelo sistema imunológico [Forrest et al., 1994].

2.2 Sistema Imunológico Artificial

A área de pesquisa relacionada à computação inspirada na natureza, chamada de computação natural, é composta de várias técnicas, algumas bastante conhecidas, como algoritmos genéticos, redes neurais, técnicas que se mostraram eficazes na resolução de algumas espécies de problemas. Uma nova linha de pesquisa, que também busca na natureza a inspiração para resolução de problemas do mundo real, é o sistema imunológico artificial. Nesta linha de idéias extraídas do processo exercido pelo sistema imunológico biológico, a analogia entre os componentes biológico (anticorpos, células, moléculas, órgãos) e os problemas a serem tratados são criados e, em seguida, o processo exercido pelo sistema imunológico é aplicado ao problema [Dasgupta, 1998]. Toda uma metodologia exigida pelo sistema imunológico artificial, tendo como base o processo natural, é aplicada sobre o problema. Os processos exercidos, relevantes ao aspecto em que o problema se identifica, são modelados de modo que a analogia seja criada, e o problema possa ser resolvido, levando em consideração, as vantagens oferecidas pelos princípios biológicos.

Nos úlltimos quinze anos, essa técnica imunológica vem se consolidando [Dasgupta, 2006], alguns trabalhos foram realizados e várias definições são encontradas para o sistema imunológico artificial.

Definição 1: ”Os Sistemas imunológicos Artificiais são metodologias de manipulação de dados, classificação, representação e raciocínio que seguem um paradigma biológico plausível: o sistema imunológico artificial”[Starlab, WWW].

Definição 2: ”Um sistema imunológico artificial é um sistema computacional baseado em metáforas do sistema imunológico natural”[Timmis, 2000]

Definição 3: ”Os sistemas imunológicos artificiais são compostos por metodologias inteligentes, inspirados no sistema imunológico biológico, para a solução de problemas de mundo

real"[Dasgupta, 1998].

Um trabalho bastante notório na área, sendo encontrado como referência em toda nota bibliográfica relacionada ao assunto, é a tese de doutorado de Leandro Nunes de Castro [de Castro Silva, 2001]. Nesta tese citada ao longo deste estudo, um novo modelo, mais formal e genérico é apresentado, chamado de *Engenharia Imunológica*.

Definição: "A engenharia imunológica é um processo de meta-síntese, o qual vai definir a ferramenta de solução de um determinado problema baseado nas características do próprio problema, e depois aplicá-las na obtenção da solução. Ao invés de buscar a reconstrução parcial ou total do sistema imunológico tão fielmente quanto possível, a engenharia imunológica deve procurar desenvolver e implementar modelos pragmáticos inspirados no sistema imunológico que preservem algumas de suas propriedades essenciais e que se mostrem passíveis de implementação computacional e eficazes no desenvolvimento de ferramentas de engenharia" [de Castro Silva, 2001].

A partir desta definição, observa-se a não obrigatoriedade da construção de um sistema fiel aos aspectos biológicos. A engenharia imunológica pode restringir à diversidade de células e moléculas dos sistemas imunológicos, mas, ainda pode utilizar a nomenclatura e conceitos já existentes na área de sistema imunológico artificial [de Castro Silva, 2001].

Com a tentativa de modelar genericamente os componentes do sistema imunológico para o desenvolvimento das ferramentas de engenharia imunológica, são mostrados modelos de representação da interação entre as células e maneiras de medir essa interação (medida de similaridades), sendo também apresentados algoritmos para simular os mecanismos e princípios imunológicos biológicos. É importante salientar que: "O enfoque da engenharia imunológica está voltado para um único tipo celular, as células B, e mecanismos de reconhecimento" [de Castro Silva, 2001]. Basicamente, as células B são as responsáveis pela produção de anticorpos e neste trabalho, a distinção de célula B, célula T não é especificada, todas são chamadas de anticorpos.

Para implementar o básico do sistema imunológico artificial, quatro decisões devem ser feitas [Dasgupta, 1998]: forma de representação, medida de similaridade, seleção e mutação. Uma vez escolhida a forma de representação e fixada uma adequada medida de similaridade, o algoritmo irá executar a seleção e a mutação, ambas baseadas na medida de similaridade, isso tudo realizado até o critério de parada ser conhecido.

2.2.1 Representação Celular e Suas Afinidades

Antes de explicar como é realizada a representação de um antígeno e um anticorpo, um conceito que deve ser entendido é como ocorre a interação entre eles. A combinação não ocorre apenas com antígenos e anticorpos exatamente complementares, existe um limiar, um valor mínimo em que a afinidade de um anticorpo reconhece um antígeno. Com isso, um anticorpo

pode reconhecer vários antígenos que possuem um valor maior que seu limiar de afinidade [Somayaji et al., 1997]. Esse princípio inspira vários pesquisadores na área de segurança, principalmente, devido à capacidade de diminuir a quantidade de recursos necessários para formar detectores de uma invasão [de Paula, 2004].

Um fator importante para a representação de um antígeno (Ag) e um anticorpo (Ab), é definir algum relacionamento de afinidade. Um antígeno será matematicamente representado como $Ag = Ag_1, Ag_2, Ag_3, Ag_4, ..., Ag_L$ e anticorpo $Ab = Ab_1, Ab_2, ..., Ab_L$, onde L representa as dimensões do espaço de inclusão dos anticorpos. Considere o espaço incluso por N moléculas de anticorpos, todos contendo um valor de complementaridade com o antígeno. A afinidade entre Ag e Ab serão relacionada pela distância entre eles. A distância D pode ser calculada pela fórmula de Euclides (2.1) ou Manhattan (2.2) [de Castro Silva, 2001].

$$D = \sqrt{\sum_{i=1}^{L}(Ab_i - Ag_i)^2} \tag{2.1}$$

$$D = \sum_{i=1}^{L}|Ab_i - Ag_i| \tag{2.2}$$

Se esta distância D for maior ou igual a um limiar de afinidade, então ocorreu ligação entre as moléculas. Para uma representação simbólica do espaço de cobertura dos anticorpos é utilizada a distância de Hamming, na qual Ag e Ab são representados por seqüências de atributos [de Castro Silva, 2001]. Distância de Hamming é o número de atributos que diferem na mesma posição em duas seqüências de igual tamanho.

Para uma representação computacional, assuma que os atributos são as cadeias binárias representando as moléculas. Dentro do espaço de inclusão dos anticorpos existe ainda várias maneiras de determinar a afinidade. São apresentadas as seguintes:

- Número de bits complementares (figura 2.4): é realizada uma operação de ou-exclusivo entre duas *strings* de bits, e em seguida é somada a quantidade de 1's da *string* resultante.
- Quantidade de r bits complementares consecutivos (figura 2.5): contam-se a maior seqüência de bits idênticos nas duas *strings*.
- Algoritmo de Hunt (figura 2.6): soma-se os bits semelhantes nas duas *strings* e que estejam na mesma posição, mais 2^{l_i}, onde l_i é o comprimento l de cada parte i da região de ligação que possui mais do que 2 bits complementares.

Dentre as formas de calcular a afinidade, a partir de *string de bits*, como citadas acima, a de r bits complementares foi utilizada no trabalho desenvolvido pela pesquisadora da Universidade do Novo México, Stephanie Forrest, para a criação do algoritmo de seleção negativa [Forrest et al., 1994]. Os outros métodos não serão abordados neste trabalho.

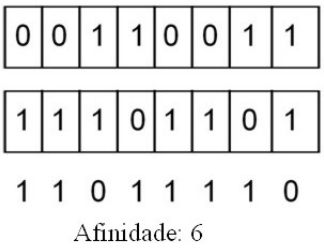

Figura 2.4: Número de bits complementares.

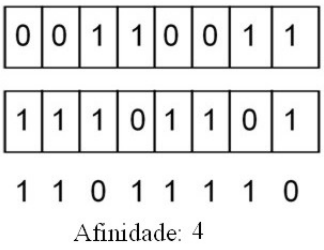

Figura 2.5: r bits complementares consecutivos.

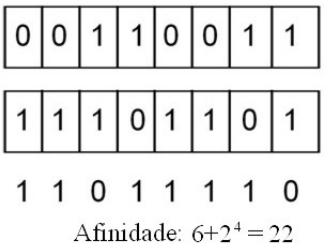

Figura 2.6: Algoritmo de Hunt.

2.2.2 Algoritmo de Seleção Negativa

Um dos primeiros trabalhos que relacionaram a teoria retirada do processo exercido pelo sistema imunológico para a elaboração de um modelo computacional foi o algoritmo de seleção negativa. No trabalho de Stephanie Forrest e vários outros pesquisadores da Universidade do Novo México [Forrest et al., 1994], o algoritmo de seleção negativa foi aplicado ao problema de detecção de vírus computacionais.

O principal motivo de uma busca por um processo alternativo na detecção de vírus foi devido à forma de discriminação entre dados com alguma intrusão por vírus, realizada atualmente, através de monitores de atividades, scanners de assinatura de vírus em arquivos. Esses métodos se mostravam ineficazes, principalmente, devido às novas formas de vírus computacionais que estavam surgindo nos anos 90, como vírus polimórficos, encriptografados [Ford, 2004]. Portanto, o algoritmo de seleção negativa foi uma tentativa de transportar toda a eficácia do sistema

imunológico natural para a detecção de vírus em um sistema computacional.

O algoritmo descrito no artigo [Forrest et al., 1994] é considerado um detector de alteração ou um método de autentificação de arquivo, em que poderão notificar qualquer alteração realizada no dado a ser protegido. O algoritmo, chamado de seleção negativa, possui duas fases.

Na primeira fase, chamada de fase de censoriamento (veja a figura 2.7), são gerados aleatoriamente *strings*. Essas *strings*, definidas sob algum alfabeto, são comparadas com o conjunto próprio (dados a serem protegidos), a *string* que não "combinar"com o conjunto próprio, obedecendo a alguma regra especificada de combinação, será adicionada ao conjunto de detectores e as *strings* que combinarem serão descartadas.

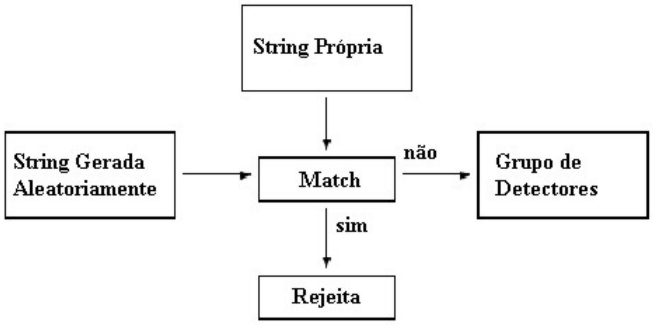

Figura 2.7: Fase de censoriamento do algoritmo de seleção negativa

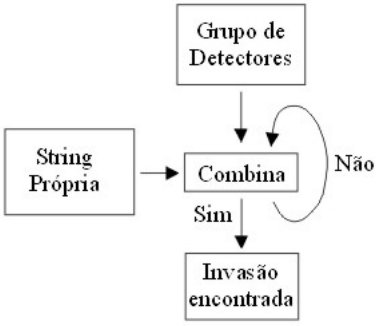

Figura 2.8: Fase de monitoramento do algoritmo de seleção negativa

A segunda fase do algoritmo, chamada de fase de monitoramento (figura 2.8), aplica a regra de "combinação"entre o conjunto próprio e os detectores gerados na fase de censoriamento. Assim, se houver uma combinação, então foi identificado alteração no conjunto próprio. Os dados a serem monitorados (dados a serem protegidos) podem ser uma *string* própria definida sobre o alfabeto $0, 1$ (qualquer dado computacional possui tal representação).

Uma *string* pode também ser definida sobre um alfabeto de *mnemônicos* (instruções de linguagem *assembly*), *string* de dados, etc. E uma vez definido o conjunto de dados a serem protegidos, bem como, quando a fase de censoriamento já tiver gerado o conjunto de detectores que não combinam com o conjunto próprio, a fase de monitoramento irá, continuamente, verificar os dados a serem protegidos a fim de detectar qualquer alteração não autorizada.

Quanto ao modo em que será realizada a combinação de *strings* para a verificação de afinidade, foi utilizado o método de *r-bits consecutivos*. O método se baseia na contagem da maior seqüência de bits distintos nas duas strings. Por exemplo, nas *strings* de bits 10101010 e 00101101, realizando a regra dos *r-bits consecutivos*, tem-se entre elas, uma afinidade de 3, que é a quantidade de bits a aparecerem, consecutivamente diferente, nas duas cadeias. Após a definição da afinidade, será definido um limiar entre as duas cadeias, dado que, se a seqüência

de r-bits consecutivos for superior a tal limiar, então entende-se que, houve uma ”combinação”; caso contrário, não havendo ”combinação”, a *string* detectora será selecionada para fazer parte do conjunto próprio.

Este método é considerado probabilístico, vez que existe uma probabilidade para duas strings, uma gerada aleatoriamente, combinar em pelo menos *r-bits consecutivos*. No trabalho de [Forrest et al., 1994], foi realizada uma estimativa de o quanto o método é eficaz para realizar a detecção, e também para regulação do sistema. A regulação diz respeito à maneira como o sistema deve estabelecer o tamanho da *string* de detectores, o número de detectores, o número de *strings* próprias e o quanto de sensibilidade a alteração no sistema permitirá. Essas informações sobre a regulação são importantes, pois, assintoticamente, o tempo de gerar a *string* aleatória e comparar duas *strings* para verificar a afinidade são constantes, mas na fase de sensoriamento, há um custo computacional alto para encontrar os detectores que não ”combinam”. O aumento da probabilidade de detecção resulta no aumento do custo computacional (devido ao aumento de *strings* geradas aleatoriamente e o número de detectores exigidos).

Logo, constata-se que o algoritmo se mostrou viável para detecção nos resultados encontrados em [Forrest et al., 1994], haja vista que todos os experimentos comprovaram as análises teóricas, embora ainda reste a dificuldade computacional de se gerar o repertório de detectores a partir de um conjunto aleatório [Ayara et al., 2002]. Se a geração dos detectores não for aleatória, o sistema corre o risco de gerar detectores com algum padrão, e esse padrão, por sua vez, pode ser explorado por agentes maliciosos.

2.2.3 Algoritmo de Seleção Clonal

Muitos dos trabalhos desenvolvidos que utilizam Sistema imunológico Artificial já foram executados usando algoritmos genéticos e técnicas de computação evolutiva [Dasgupta, 1998]. Existe uma semelhança entre esses campos [Dasgupta et al., 2003], mas a principal distinção é como ocorre o desenvolvimento da população. Em algoritmos genéticos a população está envolvida por processos de crossover e mutação, entretanto, em Sistema imunológico Artificial, a reprodução é asexuada e, nela, cada filho produzido por uma célula é a sua própria cópia (processo chamado de clonagem).

Em ambos os sistemas são usados processos de mutação para alterar os indivíduos criados (a prole) e introduzir a variação genética, de modo que o modelo computacional de seleção clonal pega emprestado teorias imunológicas, entretanto, como esse modelo está dentro da engenharia imunológica, não há exatamente uma simulação rígida do comportamento do sistema imunológico biológico [de Castro Silva, 2001].

O princípio biológico extraído da teoria da seleção clonal para a modelagem computacional é a seleção e proliferação de células com alta afinidade (células estimuladas por antígenos), produzindo um rápido crescimento na população com afinidade alta, morte das células menos esti-

muladas (deleção clonal), maturação de afinidade, re-seleção dos clones com maior afinidades, geração e manutenção de diversidade através de processo de mutação [Somayaji et al., 1997].

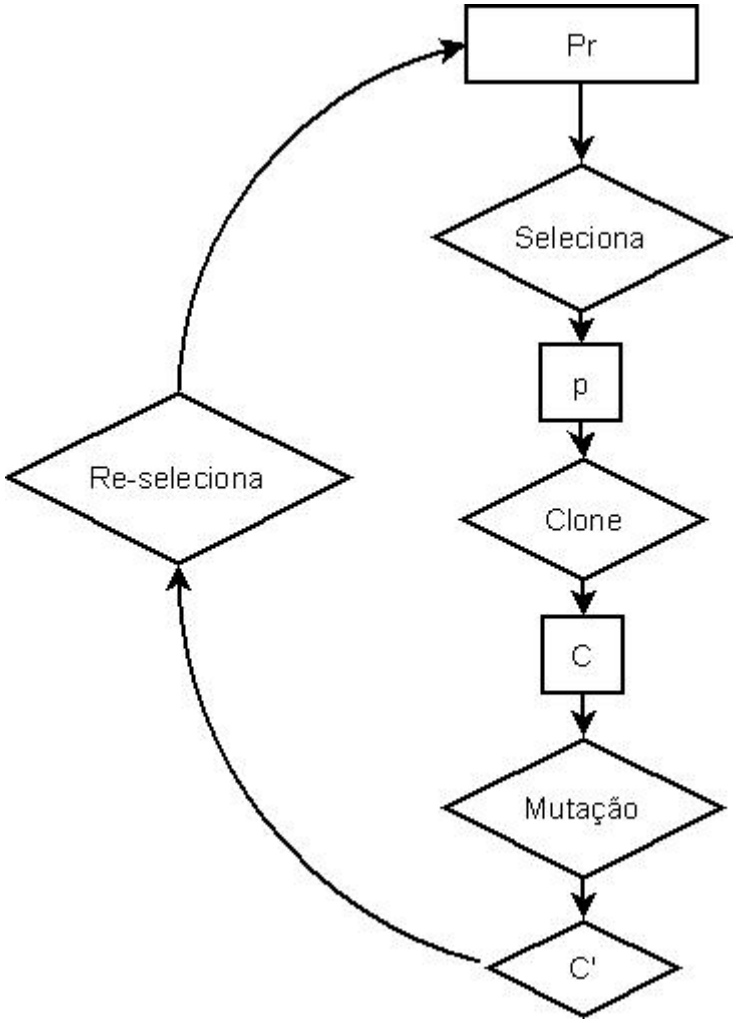

Figura 2.9: Diagrama de bloco do algoritmo de Seleção Clonal

O algoritmo funciona da seguinte maneira (veja a figura 2.9):

1) Gerar aleatoriamente um grupo Pr de anticorpos (soluções);

2) Selecionar o melhor anticorpo p da população Pr, baseado na medida de afinidade;

3) Reproduzir (clonar) o melhor anticorpo da população. A quantidade de clones é relativa a afinidade. Formando a população de clones C;

4) Submeter a população de clones a um esquema de mutação proporcional a afinidade do anticorpo. Uma população madura de anticorpos é gerada (C');

5) Re-selecionar os anticorpos criado de C';

6) Substituir anticorpos com baixa afinidade por novos.

O passo inicial do algoritmo é a criação da população em que, cada anticorpo é formado aleatoriamente, garantindo uma variabilidade entre os indivíduos. Esse processo se baseia na recombinação gênica de cada anticorpo, formado por regiões variáveis, ou região-V (responsável pelo reconhecimento antigênico).

A seleção clonal é análoga a seleção natural das espécies [Ao, 2005] a prever que, apenas os melhores anticorpos, aqueles que são capazes de reconhecer um antígeno, serão selecionados e se diferenciarão em plasmócitos. No algoritmo em si, os anticorpos selecionados serão copiados (processo de clonagem) e sofrerão mutações, com o propósito de improvisar uma melhora na afinidade.

Dessa forma, com a seleção clonal, tem-se várias soluções representadas por anticorpos e vários pontos no espaço de soluções sendo cobertos. Além disso, melhoras emergem em cada solução, o que significam os passos em direção a algum ótimo local, realizados pelos processo de mutação e seleção.

Capítulo 3

Alocação de Quadros de Horários

O problema de alocação de quadro horário consiste em uma seqüência de lições entre professores e estudantes em um determinado período do tempo (tipicamente uma semana), satisfazendo um grupo de restrições de vários tipos [Schaerf, 1995].

A palavra *Timetabling* é o termo da língua inglesa para o problema de geração automática de quadros horários e, neste trabalho, os termos ”geração de quadro de horários”e *”timetabling”* serão usados com o mesmo significado.

O quadro horário se refere à tabela de eventos arranjada de acordo com o período de tempo. Os eventos são, geralmente, encontros de pessoas em uma determinada locação, indicando a localização e quando irão acontecer esses encontros, lembrando que, cada período de tempo de um quadro horário é chamado *slot*. Outro fator base para a formação do quadro horário é o dever de possuir os eventos requisitados, tentando, sobremaneira, satisfazer o desejo de todos os que estão envolvidos no agendamento de tempo, e que tais indivíduos estejam presentes em, no máximo, um evento.

A maneira mais tradicional de resolver este problema é a manual. Quando se fala em alocação manual de eventos em um quadro horário, é lembrado o grande esforço exigido, principalmente, em instituições onde a quantidade de aulas e alunos envolvidos é grande e as restrições são variadas e em considerável quantidade. Além disso, a solução manual corre o risco de não suprir algumas restrições, como por exemplo, um aluno poderá deixar de fazer a matéria por incompatibilidade de horário entre suas várias disciplinas [Junior e da Rocha, 2006].

Por esse motivo, várias aplicações e técnicas, com a tentativa de prover uma forma automática e eficaz de resolver a alocação de quadros horários, vêm sendo aplicadas
[Junior e da Rocha, 2006, S.N.Silva et al., 2005, Gueret et al., 1995, Cambazard et al., 2005]
[Gaspero e Schaerf, 2002]. Além disso, outro fator atrativo para o estudo deste problema é a classe em que ele se encaixa no grupo de problemas, sendo considerado difícil, do ponto de vista da complexidade computacional [Cooper e Kingston, 1995], além de ser pesquisado por cientistas da computação, a fim de elaborarem técnicas eficazes de resolução de problemas ”intratáveis”.

3.1 Variações

O quadro horário, de uma instituição para outra, pode variar em sua composição e na forma de suas restrições [Schaerf, 1995]. Dependendo da instituição, a maneira em que as restrições são exigidas e como os eventos devem ser alocados podem variar, tornando esse problema bastante específico a uma instituição em particular e, consequentemente, o problema não apresenta um modelo geral que possa ser aplicado em todas as instituições.

Entretanto, na literatura, o problema de elaboração de quadros horários para universidades é normalmente classificado em duas classes principais [Schaerf, 1995]:

- *Quadro Horário de Cursos Universitário*: A alocação semanal para todas as lições de um grupo de cursos de universidade, minimizando a sobreposição de lições do curso com alunos em comum.

- *Quadro Horário para Alocação de Exames*: A alocação de exames de um grupo de cursos universitários, evitando a sobreposição de exames com alunos em comum, e espalhando os exames para os estudantes o melhor possível.

Essa classificação não é estrita, pois, dependendo do problema, peculiaridades de classes distintas podem ser encontradas em necessidades de uma determinada instituição. Neste trabalho, será tratado especificadamente o problema de Quadro Horário de Cursos Universitário. Os motivos para essa escolha concernem, a semelhança existente entre as especificações deste problema e o real problema encontrado no Departamento de Ciência da Computação de Catalão da Universidade Federal de Goiás.

3.2 Especificação do Problema

A especificação do problema é retirada do artigo que descreve as regras da Segunda Competição Internacional de *Timetabling* (ITC - 2007), aberta em 1 de agosto de 2007 [Gaspero et al., 2007]. No artigo, são apresentadas as regras gerais do *ITC - 2007*, a formulação do problema, a descrição das instâncias de entrada, sendo tais formulações consideradas um padrão, dentro dessa área de pesquisa, em que vários pesquisadores usam essas especificações para construção de seus trabalhos [Gaspero e Schaerf, 2002].

O problema descrito no artigo é originalmente de especificações aplicadas à Universidade de Udine (Itália), tendo também características encontradas em várias outras universidades. Ainda assim, é citado que foram realizadas leves simplificações ao problema real para manter um certo nível de generalização.

O problema consiste nas seguintes entidades:

- Dias, Slots: É fornecido um número de aulas que devem ser ministradas em uma semana (tipicamente 5 ou 6 dias). Tendo em cada dia a divisão de tempo em um número fixo de *slots*, que é sempre igual para todos os dias definidos no quadro horário.

- Cursos e Professores: Cada curso consiste de um número fixo de aulas a serem agendadas em períodos distintos, com um determinado número de estudantes matriculados e lecionadas por um professor. Para cada curso existe um número mínimo de dias em que a aula deve ser espalhada no quadro horário, de modo que haverá períodos em que o curso não poderá ser agendado.

- Salas: Cada sala tem sua capacidade, expressada em termos de números de cadeiras livres e todas as salas são igualmente apropriadas para todos os cursos, desde que a capacidade seja o suficiente para acomodar todos os alunos matriculados.

- Turmas: Uma turma é um grupo de cursos, de maneira que, qualquer par de cursos, nesse grupo, terá estudantes em comum (estudantes de um mesmo período que cursam as mesmas matérias). Logo, haverá a existência de conflito, caso tais cursos de uma mesma turma forem alocados a um mesmo *slot*.

3.2.1 Restrições

Devido às variações encontradas para o problema, procurou utilizar o padrão de especificação do quadro horário usado em pesquisas, tanto como base de dados quanto para formulação (incluindo as restrições). Portanto, o conjunto de restrições não necessariamente é o mesmo encontrado no curso de Ciência da Computação da Universidade Federal de Goiás Campus de Catalão, mas o problema estudado se assemelha bastante, visto que uma adaptação pode ser realizada com o acréscimo de algumas restrições específicas do departamento.

O problema estudado consiste em encontrar qualquer agendamento de eventos que satisfaça todas as restrições incondicionais, mas em alguns casos, exige-se que seja aperfeiçoado, também, um conjunto de restrições que não são obrigatórias para sanar o problema, pois, de certo modo, uma solução que satisfaça todas as restrições pode ser impossível de ser encontrada, o que intencionou, a partir disso, erigir grupos de restrições com grau de obrigatoriedade [Schaerf, 1995].

Esses grupos de restrições são divididos em dois: restrições invioláveis e restrições violáveis. As restrições invioláveis são todas as restrições exigida pelo problema e que se satisfeitas, torna uma determinada solução apropriada (restrições que obrigatoriamente não podem ser restringidas). O outro grupo, chamado de restrições violáveis, na qual uma solução sem a obediência dessas restrições ainda torna a solução satisfatível, porém, menos atraente do que outras que as satisfazem [Schaerf, 1995].

Restrições invioláveis

- Todos os cursos devem ser atribuídos ao quadro horário. Para algum curso C_i, que é um conjunto de aulas minstradas durante a semana ($C_i = \{c_1, ..., c_n\}$, n é o número de aulas dadas), tem-se: $\forall c \in C_i$, c, devem ser atribuídos ao conjunto W de *slots* do quadro horário semanal.

- Dois cursos não podem ser atribuídos a mesma sala. $\forall r_i \in R$, sendo R o conjunto de recursos, neste problema o recurso utilizado é uma sala, todos os encontro $E' \subset E$ (encontro é o agrupamento de recursos e pessoas em um determinado período de tempo)que foram atribuídos ao *slot* w não podem ser marcados na mesma sala.

- Aulas da mesma turma, ou dadas pelo mesmo professor não podem ser atribuídas ao mesmo *slot*. Considerando os cursos de uma mesma turma T, tal que $C_i, C_j \in T_k$, estes e que sejam atribuídos a um encontro E_i, E_j com os mesmos recursos (professores ou salas, por exemplo), não poderão ser agendados em um mesmo *slot* w.

- Se o professor não está disponível naquele período de tempo, em um determinado *slot* w, então, todos os encontros $E' \in E$ relacionados a este professor não podem ser alocados no slot w.

Restrições violáveis

- Para cada aula, o número de estudantes que estão matriculados no curso deve ser menor ou igual a capacidade da sala em que a aula será ministrada. $\forall c \in C_i$, considere o n o número de estudantes matriculados no curso C_i, e m a capacidade de uma determinada sala R. Logo, tem-se no quadro horário $n \leq m$, a fim de satisfazer essa restrição.

- As aulas de cada curso devem ser espalhadas dentro de um dado número mínimo de dias definido pela entrada do problema. Dado um curso C qualquer, considere q o mínimo número de dias que tal curso deve ser ministrado. Dado também, um conjunto W, conjunto de todos os *slots* do quadro horário, tal que $W = \{w_{1,1}, ..., w_{1,k}, w_{2,1}, ..., w_{2,k}, ..., w_{d,k}\}$, onde d é o número de dias e k e o número de períodos do dia, tal que d varia entre o dia 0 (segunda-feira) até o dia máximo definido no quadro horário naquela semana e k variando entre o primeiro período do dia (primeiro *slot* do dia) até o último período (último *slot* do dia). Para satisfazer tal restrição, é necessário que as aulas do curso C, que foram atribuídas a um sub-conjunto de *slots* W' com d' dias, satisfaça a condição $d' \geq q$.

- Aulas pertecentes a uma turma deve ser adjacentes a outras aulas (em *slots* consecutivos). Para uma certa turma é contada uma violação, toda vez que existir uma aula não adjacente a qualquer outra aula da mesma turma, dentro do mesmo dia. Dado um conjunto T_i de

cursos, tal que $T = \{T_1, ..., T_v\}$ e v o número de turmas, para tal restrição ser satisfeita, $\forall C_i \in C'$, C' o conjunto de cursos atribuído ao *slot* $w_{d,k}$ (d um dia qualquer) e $\forall C_j \in C''$, C'' o conjunto de cursos atribuído ao *slot* $w_{d,k+1}$, deve-se ter $C' \in T$ e $C'' \in T$.

3.3 Complexidade do Problema

O objetivo do problema é satisfazer todas as restrições invioláveis, determinando se existe uma solução no espaço de busca, e que tal solução corresponda os requisitos exigidos pelo conjunto de restrições invioláveis. É esperado também que os resultados sejam os melhores possíveis, trazendo uma solução em que o número de restrições violáveis seja mínimo, constituindo o ótimo, uma solução que satisfaça, inclusive, todas as restrições violáveis [Schaerf, 1995].

A solução de um problema de busca é aquela que supri todas as restrições, ou seja, qualquer solução factível é aceita como uma solução para o problema. Deste modo, a elaboração automática de quadro horário é considerado um problema de busca, em que é necessário cumprir todas as restrições invioláveis. E também possui características que o classifica como um problema de otimização, quando observado a capacidade de melhorar a solução, diminuindo a quantidade de restrições violáveis [Schaerf, 1995].

Para mostrar que o problema de alocação de quadros horários não pode ser resolvido por um algorítmo rápido, o trabalho de [Neufeld e Tartar, 1974] provou que o problema de elaboração automática de quadros horários pertence à classe de problemas difíceis, os chamados problemas NP-Completos. Para demonstrar isso, foi proposta uma redução para o problema de coloração de grafos que pertence à classe NP-Completo. Nessa redução, o grafo correspondente à tabela de quadros horários foi definida de maneira que cada lição é associada com um vértice, e existe uma aresta entre cada par de vértice da lição que não pode estar no mesmo *slot*. Assim, aulas que compartilham o mesmo professor ou que sejam da mesma turma são agrupadas por uma aresta.

O resultado da coloração do grafo pode ser facilmente transformada em um tabela do quadro horário, apenas atribuindo cada período do quadro horário (*slot*) a cada cor, e conseqüentemente, agendando as lições concernentes a um vértice para um *slot* correspondendo uma cor [Burke et al., 2004].

3.3.1 Uma formulação a partir de Grafo

Um problema de geração de quadros horários é um problema com quatro parâmetros [Burke et al., 2004]: W, um grupo finito de períodos (slots); R, um grupo finito de recursos; E, um grupo finito de encontros; e S, um grupo finito de restrições. O problema é atribuir um período e recursos, no encontro determinado pelo quadro horário, de modo que as restrições sejam satisfeitas o melhor possível.

- *Slot*: Um *slot* w é um elemento do grupo de slots W de uma instância do problema de geração automática de quadro horário.

- Recursos: Os recursos estão relacionados com os agendamentos de professores, salas, itens de equipamentos especiais, estudantes (ou grupo de estudantes), e assim por diante. Um recurso r é um elemento do grupo de recursos R de uma instância do problema de *timetabling*.

- Encontros: Um conjunto de encontro E é uma coleção de *slots* e recursos atribuídos à grade horária, especifica em que período do dia (em qual *slot*) o recurso será aplicado.

- Restrições: O conjunto de restrições $S = \{S_1, ..., S_2\}$ em que serão consideradas as restrições invioláveis, determina uma função binária $h : L \rightarrow \{0, 1\}$. Onde L é um grupo de soluções do problema, e l é uma solução pertecente a L.

$$h(l) = \begin{cases} 0 & \text{se l não satisfaz a condição} \\ 1 & \text{caso contrário} \end{cases}$$

Para a elaboração de quadros horários de cursos universitários, especificadamente, considere as seguintes definições. Um conjunto $C = \{C_1, ..., C_n\}$ denotam as coleções de cursos oferecidos durante a semana W, onde W é o grupo de *slots*. Assume-se que cada curso C_i consiste de n aulas na semana, ou seja $C_i = \{C_{i,1}, ..., C_{i,n}\}$. Considere $T = \{T_1, ..., T_m\}$ denotar a coleção de turmas, sendo cada turma uma coleção de cursos, ou seja, $T_i = \{C_1, ..., C_k\}$. As turmas são definidas pelo semestre em que os alunos estão matriculados. Um exemplo de turmas são um conjunto de disciplinas exigidas na grade curricular a alunos de um mesmo ano (ou semestre, considerando o rítmo semestral).

Na figura 3.1 é dado um exemplo de grafo de quadro horário para curso. O modelo básico consiste de nodos, representando aulas, e arestas unindo esses nodos. No modelo de grafo utilizado por [Burke et al., 2004], as arestas possuem pesos, de acordo com o conflito que elas representam. O peso, ou penalidade, de cada aresta $a_{i,j}$, refere-se ao conflito existente entre aulas de cursos C_i e C_j em número de estudantes que estão matriculadas em ambos os cursos. Com isso, é criado um grafo ponderado definido como seguinte: para cada Curso C_i, existem nodos $C_{i,r}$, representando as aulas. Para cada par $C_{i,r}$ e $C_{i,s}$ de cada par r, s, $r \neq s$. Além disso, uma aresta com peso ∞ é criada entre nodos $C_{i,r}$ $C_{i,s}$ para cada par possível r, s representando uma proibição correspondendo a aulas de um mesmo curso.

A atribuição de *slots* para cada nodo (aula), definida no grafo da figura 3.1, são através de números inteiros representando as cores.

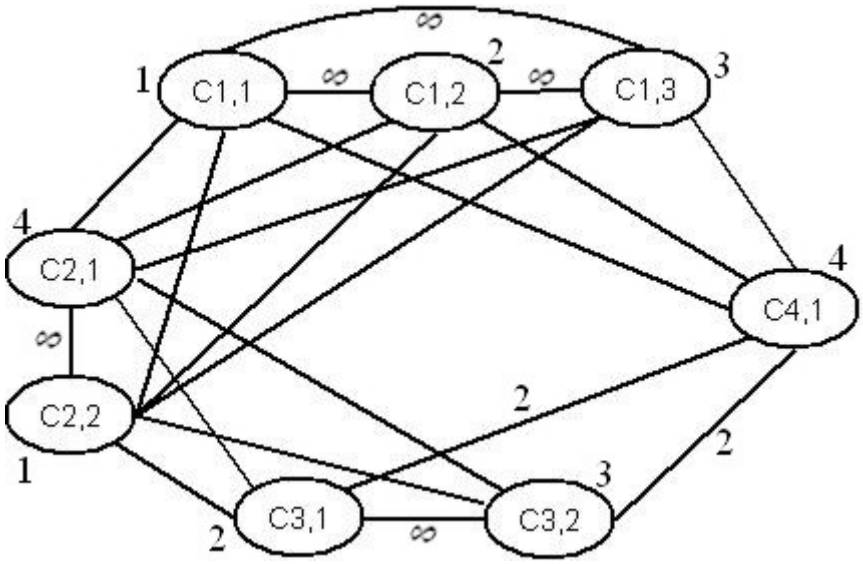

Figura 3.1: Grafo de um quadro horário.

3.4 Métodos de Solução

Diversas técnicas de soluções do problema de *timetabling* têm sido proposta, indo de técnicas heurísticas, que buscam características de uma resolução manual (alocar aula a aula até preencher o quadro horário), até métodos pertecentes ao campo de inteligência artificial.

Os métodos advindos do campo de inteligência artificial são os recentemente aplicados em pesquisas, sendo encontrados diversos trabalhos a utilizarem diferentes técnicas. Podem-se destacar técnicas como, *simulated annealing*, busca tabu, algoritmos genéticos e programação por restrição. Na próxima seção, serão citadas as técnicas de *simutaled annealing* e algortimos genéticos, devido à semelhança com o algoritmo estudado neste trabalho.

3.4.1 *Simulated Annealing*

A técnica de *simulated annealing* usa conceitos adivindos da termodinâmica, técnica que, também conhecida como têmpera ou recozimento simulado, foi proposta originalmente por [Kirkpatrick et al., 1983]. O processo térmico de resfriamento de um material a uma alta temperatura é realizado lentamente até atingir o ponto de solidificação. Dependendo da velocidade em que haja o resfriamento, o material pode conter níveis diferentes de imperfeições. No modelo computacional proposto, o processo poderia ser usado para buscar soluções factíveis, com o objetivo de encontrar uma solução ótima, semelhante à técnica de busca local, entretanto, permitindo movimentos que aumentem o valor da função objetivo (piorando a solução), com a esperança de fugir de um ótimo local.

O algoritmo inicia com uma solução inicial s. No laço de iteração, são gerados aleatoriamente soluções s' a partir da solução corrente s. Uma fução f é utilizada para avaliar as soluções a partir do cálculo de restrições s' e s e calcular a variação Δ entre tais soluções

($\Delta = f(s') - f(s)$), se $\Delta < 0$, s' passa a ser a nova solução corrente. Caso contrário, ele é aceito com uma probabilidade $e^{(-\Delta/T)}$, onde T é uma parâmetro do método chamado temperatura. Todo esse processo é repetido até que T seja tão pequeno que nenhuma nova solução possa ser aceita. Neste caso, o sistema está sendo considerado estável [e Silva et al., 2005].

3.4.2 Algoritmos genéticos

Algoritmos genéticos são uma família de modelos computacionais inspirada na teoria de seleção natural das espécies [Ao, 2005]. Tais algoritmos incorporam uma estrutura semelhante a de um cromossomo e aplicam operações de seleção e mutação a essas estruturas [Junior e da Rocha, 2006], de maneira a preservar informações da solução do problema. Os cromossomos, para o problema de *timetabling*, são, assim, representados por vetores. Sendo, cada posição destes, formada por sala e *slot* [Fernandes et al., 1999].

O grupo de soluções do problema de *timetabling* (representados pelos cromossomos) é chamado de população. Inicialmente, esta população é criada por processo aleatório, em que cada evento (cada parte da lição) é atribuído aos *slots* e às salas [Fernandes et al., 1999].

Após a inicialização da população, cada indivíduo (cada solução do conjunto de soluções), é passado por uma função de avaliação, chamada de *fitness*. A avaliação serve para medir uma devida solução e, normalmente, para o problema de *timetabling* é contada a quantidade de restrições violadas; quanto menor a quantidade de restrições violadas, melhor a solução.

Dada a avaliação de cada indivíduo, o próximo passo é selecioná-los a partir do método chamado *roulette-wheel*. Nesse método, a probabilidade para que um indivíduo seja selecionado é proporcional ao valor calculado pela função de *fitness*. Com os indivíduos selecionados é realizada a operação chamada *crossover*. *Crossover* seleciona genes dos cromossomos (indivíduos) selecionados, agora chamado de pais, e cruzam esses cromossomos com o propósito de criar novos decendentes.

Com o objetivo de aumentar a variabilidade da população, e, consequentemente, erigir um espaço de busca maior, para o conjunto de soluções da população, são realizados processos de mutação. A mutação é efetuada alterando-se o valor de um gene de um indivíduo sorteado, aleatoriamente, com uma determinada probabilidade, denominada probabilidade de mutação, ou seja, vários indivíduos da nova população podem ter um de seus genes alterado aleatoriamente.

Capítulo 4

Sistema Imunológico Artificial na Resolução do Problema de Geração de Quadro Horário

Afim de aplicar o modelo imunológico para resolver o problema de alocação automática de quadros horários, foi escolhido o algoritmo de seleção clonal. O motivo da escolha não deu-se devido à exclusividade deste modelo imunológico para o determinado problema, mesmo porque a definição de cada modelo a um específico tipo de problema é ainda um assunto em aberto, dentro da área de sistema imunológico artificial [Dasgupta, 2006]. Mesmo assim, entre os modelos mais estudados, o algoritmo de seleção clonal é tipicamente aplicado a problemas de otimização.

Quanto aos outros modelos, existe ampla aplicação destes em problemas de segurança computacional, detecção de falha, segurança em rede, arquitetura de agentes móveis. Isso se deve aos aspectos encontrados no sistema imunológico, que é a capacidade de distinguir o próprio do não próprio, detecção de antígeno e o comportamento dinâmico e agregado.

Antes de descrever todo o processo realizado pela seleção clonal, um fator importante é conhecer como é a entrada do problema e, principalmente, como se dá a representação no modelo computacional.

4.1 Entrada do Problema

A entrada do problema são instâncias definidas em um simples arquivo contendo os cursos, professores, salas, dias, períodos, turmas e as restrições. Abaixo, tem-se um exemplo de como são definidos tais instâncias [Gaspero et al., 2007]:

```
Name: Instância 1
Courses: 4
```

```
Rooms: 2
Days: 5
Periods_per_day: 4
Curricula: 2
Constraints: 8

COURSES:
Algoritmo Thiago 3 3 30
Programacao1 João 3 2 42
Redes Vilma 5 4 40
Administracao Wesley 5 4 18

ROOMS:
A 32
B 50

CURRICULA:
Cur1 3 Algoritmo Administracao Redes
Cur2 2 Programacao1 Redes

UNAVAILABILITY_CONSTRAINTS:
Algoritmo 2 0
Algoritmo 2 1
Algoritmo 3 2
Algoritmo 3 3
Redes 4 0
Redes 4 1
Redes 4 2
Redes 4 3

END.
```

Os dados em cada seção devem ser interpretados da seguinte maneira:

```
Courses:
<id. do Curso> <Professor> <n. de aulas> <mínimo de
diasaulas> <n. de estudantes>

Rooms:
```

```
<id. de Sala> <Capacidade>

Curricula:
<id. de Turma> <n. de Cursos> <id. de Cursos> ..... <id. de Cursos>

Unavailability_Constraints:
<id. do Curso> <Dia> <periodo do dia>
```

Quanto as restrições (*Unavailability_Constraints*), por exemplo ”Algoritmo 3 2”, significa que o curso Algoritmo não pode ser agendado no terceiro (2) período de uma quinta feira (3), lembrando que os dias e o período começam de 0.

Essa entrada é processada e as informações são utilizadas junto a alocação das instâncias das classes modeladas, dessa forma, o espaço em memória alocada pelo sistema inicialmente, depende do tamanho desse arquivo de entrada (quantidade de cursos, restrições, turmas, salas). Após a instânciação inicial a partir dos arquivos, o algoritmo começa a sua execução, manipulando essas classes já alocadas e realizando o processamento dessas informações, reutilizando as classes já alocadas.

4.2 Modelagem do problema

4.2.1 Curso

A figura 4.1 representa o diagrama da classe *Curso*, os dados são instânciados a partir do arquivo de entrada. O atributo *cursoId* é uma *string* que identifica o curso, o atributo *professor* é uma *string* que armazena o nome do professor que ministrará o curso, *nAulas* refere-se à quantidade de aulas que tal curso deverá cumprir na semana. O atributo *minAulasDias* refere-se a quantidade mínima de aulas do mesmo curso, permitida no mesmo dia, e por último, o atributo do tipo inteiro *nEstudantes*, que mantém a quantidade de estudantes matriculados no respectivo curso.

4.2.2 Sala

No diagrama da classe *Sala* (figura 4.2) tem-se o atributo *salaId*, que é uma *string* que identifica a sala, e o atributo *capacidade* que, logicamente, determina a capacidade máxima de alunos na respectiva sala.

4.2.3 Turma

No diagrama da classe *Turma* (figura 4.3), tem-se uma simples *string* identificadora da turma, *turmaId*, e mais dois atributos, um referindo à quantidade de cursos *nCursos* e o outro,

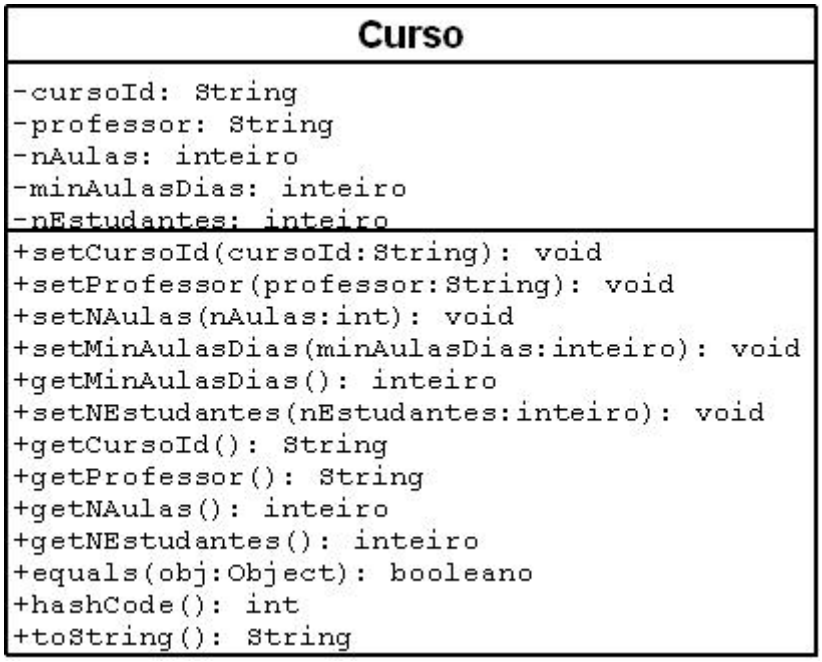

Figura 4.1: Diagrama da Classe Curso

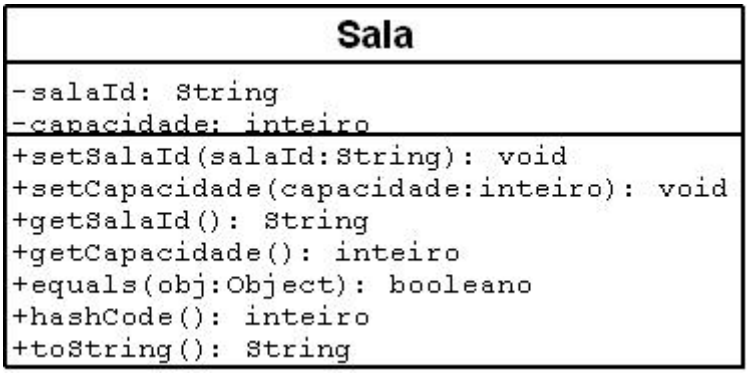

Figura 4.2: Diagrama da Classe Sala

vetCursos, um vetor de *string*, contendo todos os cursos que formam esta turma.

4.2.4 Restrições dos Cursos

É modelada uma classe (figura 4.4) que identifica um curso, através do atributo *cursoId* e também refere-se a uma específica restrição, tendo os atributos *dia* e *período*, correspondendo respectivamente aos dias e períodos em que tal curso não poderá ser agendado no quadro horário.

4.2.5 Quadro Horário

Cada quadro horário gerado é definido como um anticorpo da população, sendo que, cada quadro horário foi modelado como uma matrix M_{pxd} (veja a tabela 4.1), tal que p representa o

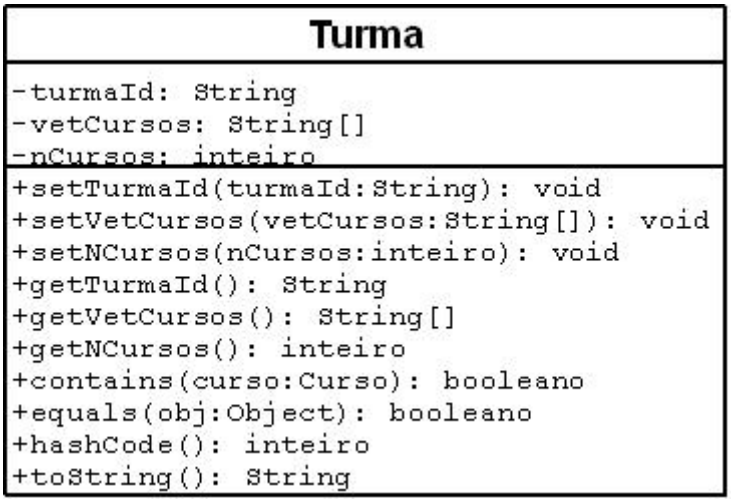

Figura 4.3: Diagrama da Classe Turma

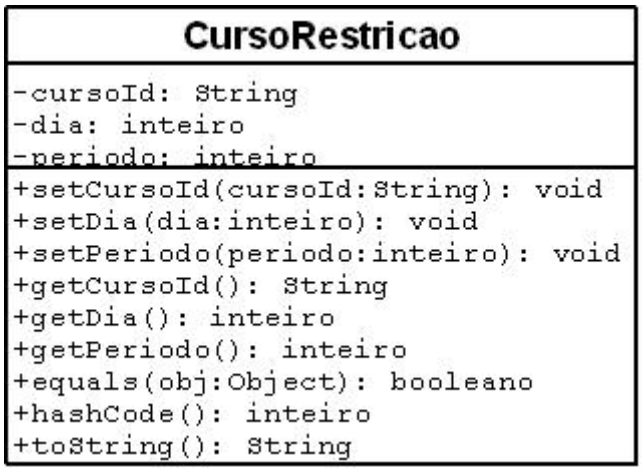

Figura 4.4: Diagrama da Classe CursoRestricao

número de períodos e d o número de dias da semana, ou seja, os períodos p são a quantidade de *slots* de um determinado dia da semana e os dias d são aqueles em que as aulas acontecerão, normalmente, de segunda a sexta-feira.

Em cada *slot* são alocados vários eventos, de forma a ser definida uma estrutura de dados para agrupar vários eventos em cada *slot*, utilizando uma lista para representar cada *slot* da tabela, tendo assim, cada quadro horário como uma matriz de listas.

Os eventos, especificadamente para o problema de quadro horários de curso universitário, é o agrupamento dos cursos e salas, definindo uma aula a ser dada, considerando que cada curso seja aplicado em mais de uma aula na semana, para que a quantidade de eventos que o quadro horário deva agendar seja maior que a quantidade de cursos oferecida pela instituição. Na figura 4.5, o diagrama da classe *Aula*, cujos atributos *curso* e *sala* definem respectivamente o curso e a sala, formando o evento a ser agendado.

Com isso, foi definida uma interface, chamada de anticorpo (ver figura 4.6), que possui os

Matriz de *Slots*					
períodos	segunda	terça	quarta	quinta	sexta
1	slot 1	slot 6	slot 11	slot 16	slot 21
2	slot 2	slot 7	slot 12	slot 17	slot 22
3	slot 3	slot 8	slot 13	slot 18	slot 23
4	slot 4	slot 9	slot 14	slot 19	slot 24
5	slot 5	slot 10	slot 15	slot 20	slot 25

Tabela 4.1: Matriz de *Slots*.

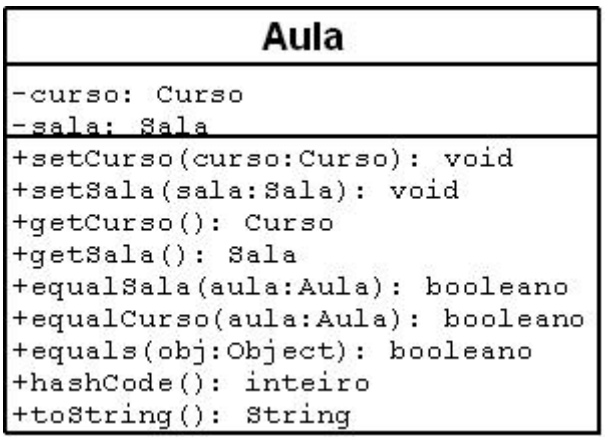

Figura 4.5: Diagrama da Classe Aula

operadores genéricos necessários para o processamento do algoritmo de seleção clonal. Em uma implementação para o problema de curso universitário, a interface anticorpo foi estendida e especificada.

Figura 4.6: Diagrama da Interface Anticorpo

No algoritmo, um anticorpo é uma solução do problema, que, além de implementar os operadores a definirem um anticorpo no algoritmo, necessita também que a classe a implementar *Anticorpo* possua uma especialização, segundo o problema a ser resolvido.

Para o problema de geração automática de cursos universitários, foi especializada a interface *Anticorpo* para uma classe chamada *Timetabling*. Cada instância desta classe define uma

solução, em que são agrupados todos os cursos e salas envolvidos na especificação do problema.

Na classe *Timetabling*, o atributo *table* contendo a matriz de lista de aulas, onde são atribuídos todos os cursos com suas respectivas salas e obedecendo o conjunto de restrições do problema.

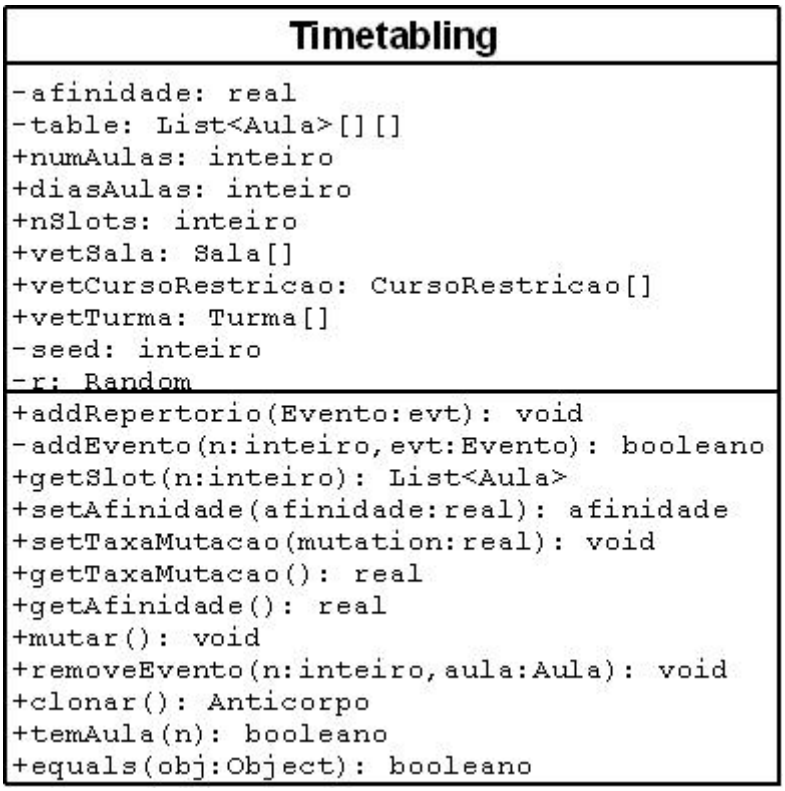

Figura 4.7: Diagrama da Classe Timetabling

4.2.6 Modelo Completo

Na figura 4.9, esboçada adiante, o diagrama do sistema completo, com todas as classes modeladas. A classe abstrata *CSA* (*Clonal Selection Algorithms*, veja a figura 4.8) contém a implementação do algoritmo de seleção clonal e alguns métodos abstratos. É nessa classe que todos os processos de inicialização, seleção, mutação, entre outros operadores do algoritmo, serão realizados. Alguns métodos, como aqueles responsáveis por inicializar e pelo cálculo de *fitness*, são diferenciados a cada tipo de problema (como quadros horários para curso, exames, entre outras variantes). Portanto, é justificável deixá-los como métodos abstratos, já que o método a definir o critério de parada e o da roleta são abstratos para permitir uma flexibidade na implementação de uma especificação, ao passo que o método plotar serve apenas como um auxílio na visualização gráfica do crescimento da população.

A classe concreta *CSACourse* é uma especificação da classe abstrata *CSA*, em que os detalhes do problema de geração automática de quadro horário para curso universitário são implementados. Note pela figura 4.9 que a relação entre a classe *Timetabling* é de um (1) para muitas

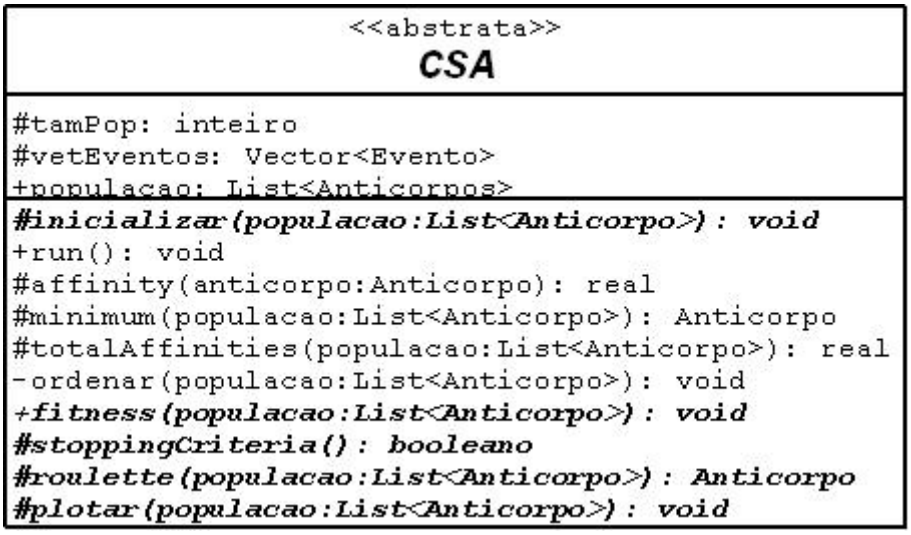

Figura 4.8: Diagrama da Classe Abstrata CSA

instâncias da classe *Timetabling*, determinando o conjunto população do algoritmo de seleção clonal.

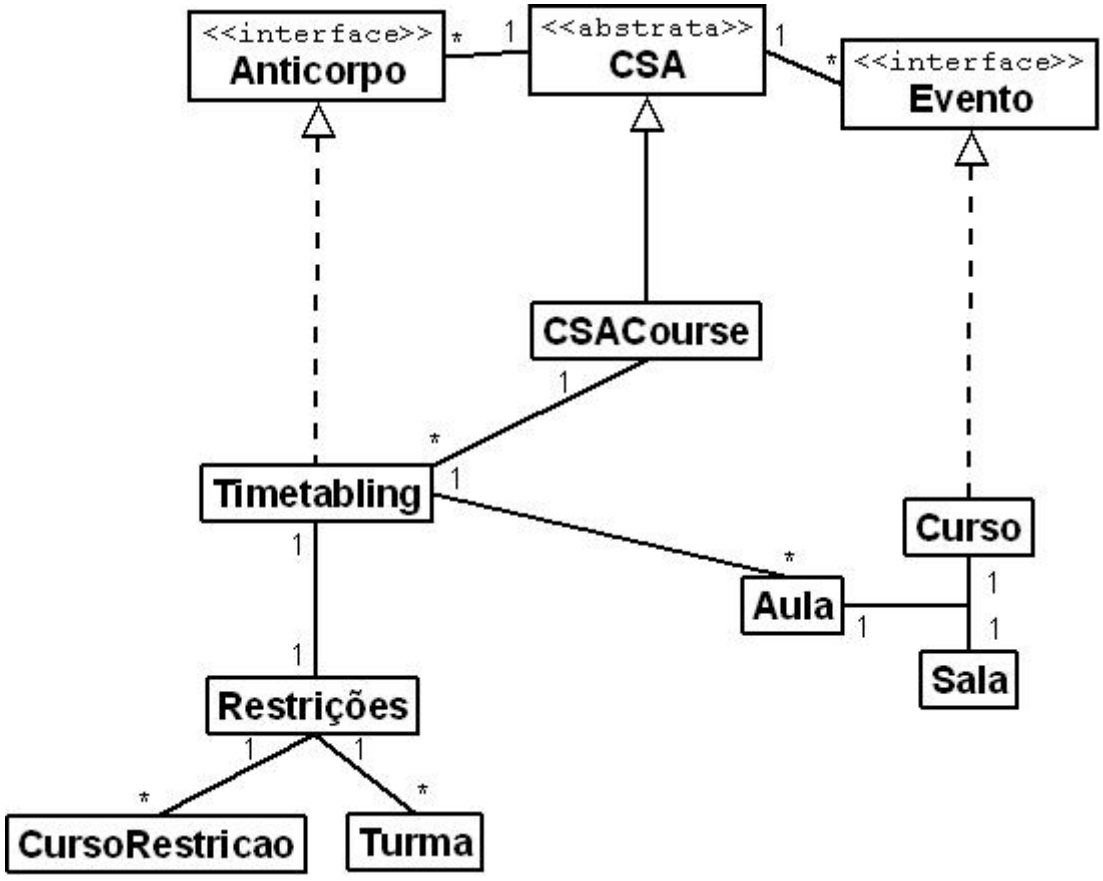

Figura 4.9: Diagrama da Classe do Sistema de Geração Automática de Quadros Horários de Cursos Universitários

Na figura 4.10 são referenciadas o pacote contendo as classes mais genéricas do diagrama

completo (figura 4.9). A partir da especificação dessas classes, tanto o problema de geração de quadros horários para cursos universitários quanto para exames poderão ser implementados.

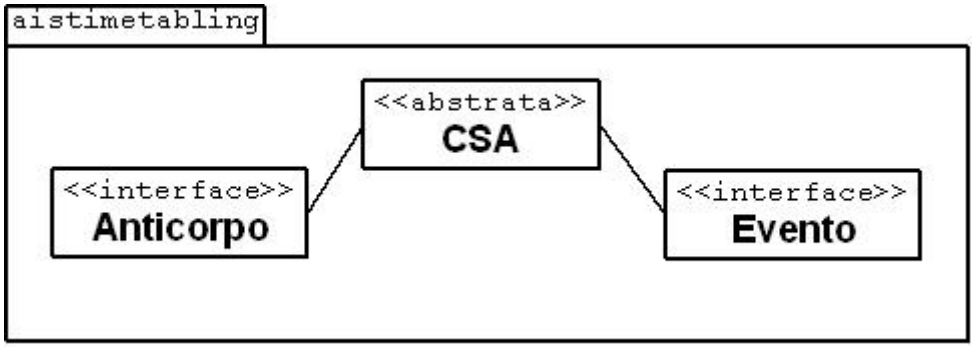

Figura 4.10: Pacote *aistimetabling* com as principais classes genéricas.

4.3 Algoritmo de Seleção Clonal

Quando um antígeno é reconhecido por todo o processo de detecção realizado pelo sistema imunológico, aquele anticorpo que melhor reconhece tal antígeno, ou seja, sofre um estímulo antigênico, é selecionado e proliferado por um processo de clonagem e mutação. Esse mecanismo de reconhecimento e proliferação é basicamente o princípio de seleção clonal (veja a seção 2.1.1).

Logo, no algoritmo de seleção clonal, para o problema de elaboração de quadros horários (figura 4.11), para cursos universitários [Malim et al., 2006], estão presentes operadores básicos, como seleção, clonagem e mutação. Nos próximos subtópicos são discutidas as características do algoritmo, como a inicialização da população de anticorpos, avaliação da afinidade entre cada anticorpo, o processo de seleção, a clonagem e o mecanismo de mutação.

Abaixo é descrito o algoritmo de seleção clonal, proposto por [Malim et al., 2006]:

A partir do algoritmo proposto por [Malim et al., 2006] foi implementado uma versão na linguagem de programação Java. Veja no apêndice **A.1** na classe Classe Abstrata *CSA* o método *run*, que implementa o algoritmo da figura 4.11.

4.4 Inicialização

A população inicial de anticorpos é gerada aleatoriamente. Cada anticorpo (*timetabling*) inicial tem suas aulas agendadas de forma que apenas as restrições invioláveis são respeitadas. Como não há a preocupação de atender as restrições violáveis, a afinidade de cada anticorpo da população inicial é baixa, ou seja, as soluções iniciais não são, geralmente, consideradas boas (apesar de satisfazer as restrições invioláveis).

```
1  1. Inicialização:
2  inicialize a população de anticorpos (quadro horário satisfatível)
3  para cada anticorpo (quadro horário)
4     aleatoriamente selecione um evento (curso/sala) um por um
5        atribua evento e sala aleatoriamente para um slot
6        (obedecendo as restrições invioláveis)
7     se nenhum anticorpo idêntico (quadro horário duplicados)
8        adiciona anticorpo (quadro horário) para a população
9     se não elimine anticorpo
10
11 2. Loop População:
12 para cada geração de anticorpos (quadros horários satisfazível)
13    para cada anticorpo faça
14       determine a afinidade do anticorpo via função de afinidade
15       (afinidade = 1/fitness)
16       calcule probabilidade de seleção (taxa de clonagem)
17       (probabilidade de seleção = afinidade / afinidade total)
18    aleatoriamente selecione um anticorpo (quadro horário)
19    (usando método da roleta baseado na probabilidade de seleção)
20    Clonagem: Clone o anticorpo selecionado
21    (número de clones = tamanho da população x probabilidade de seleção)
22       Mutação: para cada geração de clones faça
23       (taxa de mutação = 1 / probabilidade de seleção)
24       se um valor aleatório  <= taxa de mutação, mutação = falso;
25          enquanto mutação = false, selecione um evento aleatoriamente
26             reatribua evento e sala para um slot aleatoriamente
27             (obedecendo as restrições invioláveis)
28             se todas as restrições são satisfeitas e não há quadro horário duplicado
29                determine a afinidade do novo clone
30                se a afinidade (novo clone) >= afinidade (clone original)
31                   mutação = sucesso;
32       se a afinidade (novo clone) > afinidade mínima da população
33          remove anticorpo com menor afinidade
34          adicione novo clone a população
35 repeta (2. Loop População) até critério de parada for encontrado.
```

Figura 4.11: Algoritmo de seleção Negativa para Resolução de Quadros Horários

Além de ser gerada aleatoriamente, cada anticorpo da população inicial deve ser distinto de cada outro anticorpo, ou seja, por toda a matriz de lista de aulas de um anticorpo (*timetabling*), não deve existir matrizes com a idêntica disponibilidade de aulas. Com essa característica, é garantido uma variabilidade na população inicial.

Cada instância da classe *Curso* e *Sala* é agrupada na classe *Aula*. Dessa forma, os cursos e salas são selecionados aleatoriamente para formar a aula. A atribuição no quadro horário, no processo de inicialização, é, também, realizada de forma aleatória. A atribuição é realizada apenas se naquele *slot* a aula puder ser ministrada, tendo em vista as restrições invioláveis.

4.5 Avaliando a Afinidade e Seleção

Ainda que o anticorpo já tenha sido definido como a solução, por outro lado, tem-se para o antígeno a solução objetivo, a solução desejada. Dessa forma, é intento de cada anticorpo detectar um antígeno, ou que, em outras palavras, cada solução gerada deva tender à solução objetivo, ou a melhor solução possível.

Então, quando se considera o anticorpo como uma solução gerada, deve-se também definir um grau de afinidade. A afinidade de um anticorpo determina a capacidade de encontrar um

antígeno, ou seja, a distância de uma solução ótima (distância entre antígeno e anticorpo), isto é, quanto menor a distância, melhor será a afinidade.

A avaliação de um anticorpo x, com o objetivo de determinar a sua afinidade, é realizada a partir de uma função $f(x)$, tal que $f(x) = 1/fitness(x)$. A função de *fitness* é calculada a partir da quantiade de restrições violáveis desrespeitadas.

Mais precisamente, a função de *fitness* é o número de estudantes sem lugar para sentar em uma aula (onde a capacidade da sala é menor que a quantidade de matrículas do curso alocado), mais o número de cursos que foi atribuido a uma quantidade menor de dias definida pelo número mínimo de dias multiplicaldo por 5, e mais o número de aulas não adjacente a qualquer outra aula da mesma turma em um determinado dia (note que a segunda restrição possui peso 5).

Para calcular o *fitness* deve-se fazer as seguintes considerações envolvendo as restrições violáveis:

- $\forall c \in C_i$, considere o n o número de estudantes matriculados no curso C_i, e m a capacidade de uma determinada sala R. Logo, se $(n - m) > 0$ atribua essa diferença a uma variável r_1;

- Dado um curso C qualquer, considere q o mínimo número de dias que tal curso deve ser ministrado. Dado também, um conjunto W com todos os *slots* do quadro horário, tal que $W = \{w_{1,1}, ..., w_{1,k}, w_{2,1}, ..., w_{2,k}, ..., w_{d,k}\}$, onde d é o número de dias e k e o número de períodos do dia. As aulas do curso C, que foram atribuídas a um sub-conjunto de *slots* W' com d' dias, atribui-se a r_2 a diferença entre $|q - d'|$ se $(q - d') > 0$;

- Dado um conjunto T_i de cursos, tal que $T = \{T_1, ..., T_v\}$ e v o número de turmas: $\forall C_i \in C'$, C' o conjunto de cursos atribuído ao *slot* $w_{d,k}$ (d um dia qualquer e k um período do dia qualquer) e $\forall C_j \in C''$, C'' o conjunto de cursos atribuído ao *slot* $w_{d,k+1}$, deve-se atribuir a r_3 a quantidade de cursos $C_i \in C'$ tal que $C_i \ni T$ e $C'' \in T$.

A partir dos itens acima, a função de *fitness* f é calculada da seguinte maneira (equação 4.1):

$$f = r_1 + (r_2 \times 5) + r_3 \tag{4.1}$$

Quando a afinidade de cada anticorpo é calculada, é retirado da população o anticorpo com maior afinidade. A seleção é realizada através do método da roleta. No método da roleta, cada anticorpo da população é representado proporcionalmente a sua afinidade. Dessa forma, para cada anticorpo com alta afinidade, é dada uma porção maior da roleta, enquanto aos indivíduos de afinidade mais baixa é dada uma porção relativamente menor. A roleta é girada e, a cada giro, um anticorpo é apontado pela seta e selecionado. Todo esse processo é semelhante a um sorteio,

sendo que cada anticorpo escolhido terá a participação no processo de clonagem e mutação. A figura 4.12 ilustra esse método.

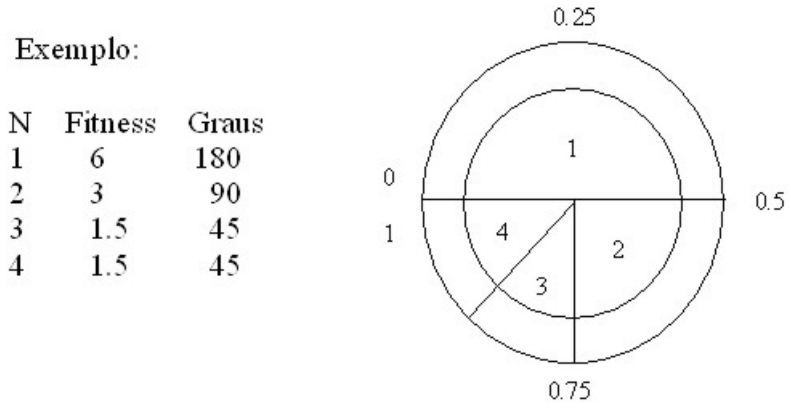

Figura 4.12: Representação de uma roleta.

4.6 Clonagem, Mutação e Critério de Parada

Quando um anticorpo é selecionado, em seguida, é realizado o processo de clonagem (cópia do anticorpo). O número de clones é o tamanho da população (p) vezes a probabilidade de seleção (probalidade de seleção é a afinidade do anticorpo dividido pela soma das afinidades de todos os anticorpos da população). Portanto, a função g, que determina o número de clones de um anticorpo x_i é a seguinte (equação 4.2):

$$g(x_i) = p \times \frac{f(x_i)}{\sum_{k=1}^{p} x_k} \tag{4.2}$$

O próximo passo realizado pelo algoritmo de seleção clonal é a mutação do clone. Diferentemente de algoritmos genéticos, a taxa de mutação t de um clone é definida pelo próprio algoritmo, o cálculo é o seguinte: $t = 1 - (afinidade/afinidadetotal)$. Todos os clones serão mutados desde que a taxa de mutação seja alta .

A mutação sobre um quadro horário é uma operação que muda as características do quadro de horários inicial. A mutação consiste na re-atribuição de uma aula do curso em um determinado período (slot) para um outro período escolhido, mudando a sala em que a aula será dada, sendo a escolha, tanto do período como da nova sala, feita aleatoriamente. É importante salientar que qualquer alteração no quadro horário não deve desobedecer o conjunto de restrições invioláveis.

Na seleção clonal, quando um anticorpo passa pelo processo de clonagem, a afinidade desse novo clone não pode ser inferior a afinidade do anticorpo original. Assim, a clonagem será

considerada bem sucessida, quando houver um incremento na afinidade, e, com isso, apenas os clones que apresentarem uma melhora na afinidade serão selecionados.

Após os clones serem mutados e selecionados, é escolhido, da população de anticorpos maturos, os indivíduos com as menores afinidades, e esses são substituídos por aqueles clones mutados e selecionados que apresentarem um valor de afinidade alta. Dessa forma, a população de anticorpos apresenta um aumento da afinidade, em um processo realizado ao final de cada interação.

As interações, que vão do processos de avaliação da afinidade e seleção dos indivíduos até a clonagem e mutação, são realizadas até um certo critério de parada. Na implementação deste trabalho, o critério de parada foi um inteiro definindo a quantidade de vezes que o algoritmo deve interar.

Portanto, as interações, contento todos os operadores de seleção clonal, foram aplicadas no problema, a partir do que é necessário para o processo de seleção clonal. E a modelagem foi construída, utilizando o paradigma orientado a objetos, para agregar o processamento imunológico a uma fácil modelagem computacional que atenda os requisitos de um quadro horário.

Enfim, os critérios necessários para o processo de seleção clonal, que devem ser definidos em um modelo imunológico (forma de representação, medida de similaridade, seleção e mutação), estão todos presentes na modelagem apresentada. Fazendo deste trabalho consistente para a área de engenharia imunológica.

Capítulo 5

Resultados e Análises

O problema de quadro horário se particulariza em cada instituição considerada. No entanto, é importante salientar que existem características que são comuns a diversas instituições, de modo que se possa definir as diversas modalidades desse problema, como apresentado no capítulo 3. Dessa forma, para efeito de teste, foram utilizadas três instâncias do problema especificado.

O propósito deste trabalho foi implementar e validar, através de base de dados experimentais, um modelo imunológico para automatização de quadro horário, e comparar com trabalhos correlatos. Desta forma, procurou-se avaliar a eficácia da solução, assim como sua eficiência computacional. Acrescente-se a isto, que, a partir deste trabalho, será possível aplicar todo o conjunto de classes e interfaces desenvolvidas, em um sistema específico para o departamento de Ciência da Computação da Universidade Federal de Goiás - Campus Catalão.

Todas as instâncias de testes podem ser encontradas no site `www.diegm.uniud.it/satt/projects/EduTT/CourseTT/`. Também, junta a elas, é disponibilizado um programa codificado na linguagem C, que valida as soluções dessas instâncias. No programa, chamado *validator.cc*, são passados como parâmetros de entrada, a instância de teste e a suposta solução. Como saída, são contadas as restrições invioláveis e violáveis desobedecidas.

Neste trabalho, as soluções geradas durante o processo de inicialização, bem como as soluções resultantes das interações do algoritmo de seleção clonal (soluções finais), foram armazenadas em arquivos para serem avaliadas por uma compilação do programa (*validator.cc*).

Como resultado, foi confirmado que todas as soluções geradas pela implementação, tanto na inicialização quanto nas soluções finais, não desobedeceram nenhuma restrição inviolável, e que o valor da afinidade (quantidade de restrições violáveis) calculado, corresponde corretamente a aquela definida pelo programa validador. Portanto, todos os valores de afinidade demonstrados nos testes estão corretos.

O algoritmo foi codificado em linguagem de programação *Java* (jdk 1.6) e testado em um PC rodando o sistema operacional GNU/Linux Ubuntu, equipado com um processador Intel Pentium 4 Dual Core e 512 MB de memória RAM.

Foram realizadas em média cinco execuções do programa para três variações diferente de quantidade de indivíduos e número de interações, a fim de se obter o melhor valor e a média da população de anticorpos (quadro horário). Abaixo seguem as tabelas de resultados e os gráficos de crescimento da afinidade para a melhor solução encontrada (os gráficos das outras soluções são encontrados no CD-ROM, junto com arquivos de entrada, de saída, *logs* e o programa validador). No trabalho de [Gaspero e Schaerf, 2002], nos quais os valores de *fitness* são mostrados, o tempo de execução da melhor solução não é citado. Entretanto, os testes foram realizados com um tempo limite de 600 segundos (10 minutos).

Outras considerações importantes sobre os dados das instâncias de testes, são em relação a algumas definições, como por exemplo a de períodos. Os períodos são cada hora definida em um dia, sendo que cada curso será ministrado em um determinado período do dia, ou um *slot* dentro do quadro horário. Os *slots* são os períodos dentro de uma semana. Em um quadro horário com 5 dias e 5 períodos terá uma quantidade de 25 *slots*. As restrições são as proibições impostas para cada curso de ser ministrada em um determinado período do dia. Essas informações são úteis para um melhor entendimento das descrições referentes as instâncias de testes.

5.1 Teste 1

Na primeira instância, tem-se as seguintes descrições na tabela 5.1 [Malim et al., 2006]:

Cursos	Aulas (A)	Salas (S)	Dias (D)	Períodos (P)	*slots* (W)	Turmas	Restricoes
46	207	12	5	4	20	26	20

Tabela 5.1: Descrição da instância 1.

O número total de aulas por *slot* é de 10,35 (A/W). Como são 12 salas oferecidas, tem-se uma ocupação total de salas por *slot* de 86,25% ($A/(S \times W)$). Visto que o algoritmo opera aleatoriamente, essa percentagem indica com que facilidade uma sala sorteada poderá ser atribuída a algum *slot*.

Lembrando que a avaliação de uma solução é feita a partir da contagem da quantidade de restrições violáveis desobedecidas, como mostrado no capítulo 4, pode-se citar o melhor valor de avaliações realizadas em outros trabalhos científicos:

- [Gaspero e Schaerf, 2002], melhor resultado encontrado: 200.
- [Malim et al., 2006], melhor resultado encontrado: 284. Tempo gasto: 26 minutos

	melhor	melhor fit.	média	média fit.	tempo(min.)
0	0,003584	279	0,003574	279	40,51
1	0,003484	287	0,003391	294	110,29
2	0,003623	**276**	0,003514	284	34,30

Tabela 5.2: Tabela de resultados para instância 1: 500 anticorpos 1000 gerações.

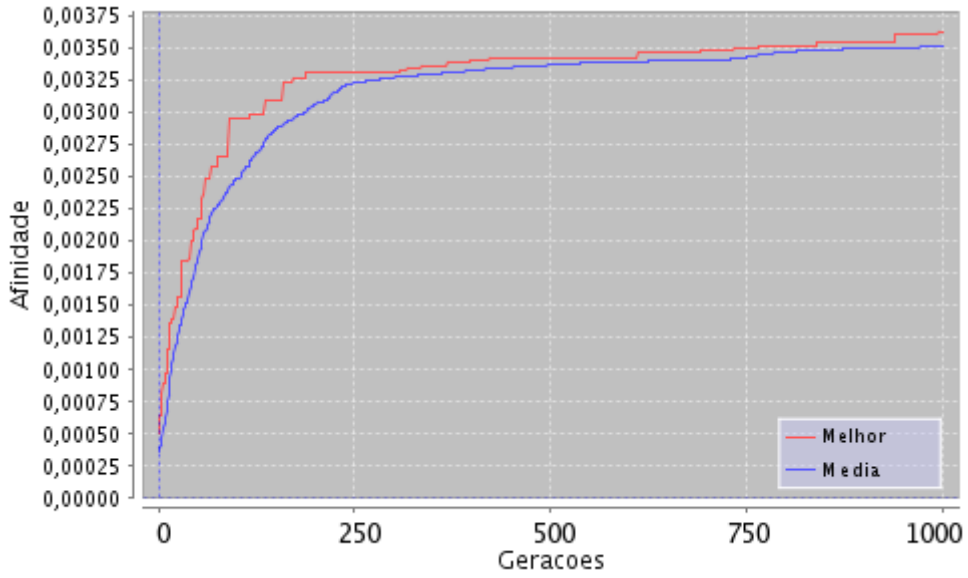

Figura 5.1: Teste 1: gráfico de execução do algoritmo para a solução de melhor *fitness* (500 anticorpos, 1000 gerações e *fitness* de 276).

5.1.1 Resultados e Análises

População de 500 anticorpos e 1000 gerações (tabela 5.2)

População de 300 anticorpos e 500 gerações (tabela 5.3)

	melhor	melhor fit.	média	média fit.	tempo(min.)
0	0,003205	312	0,003130	319	16,77
1	0,003774	**265**	0,003757	266	17,02
2	0,003509	285	0,003460	289	188,22
3	0,003311	302	0,003236	308	17,62
4	0,003322	301	0,003260	306	18,86

Tabela 5.3: Tabela de resultados para instância 1: 300 anticorpos 500 gerações.

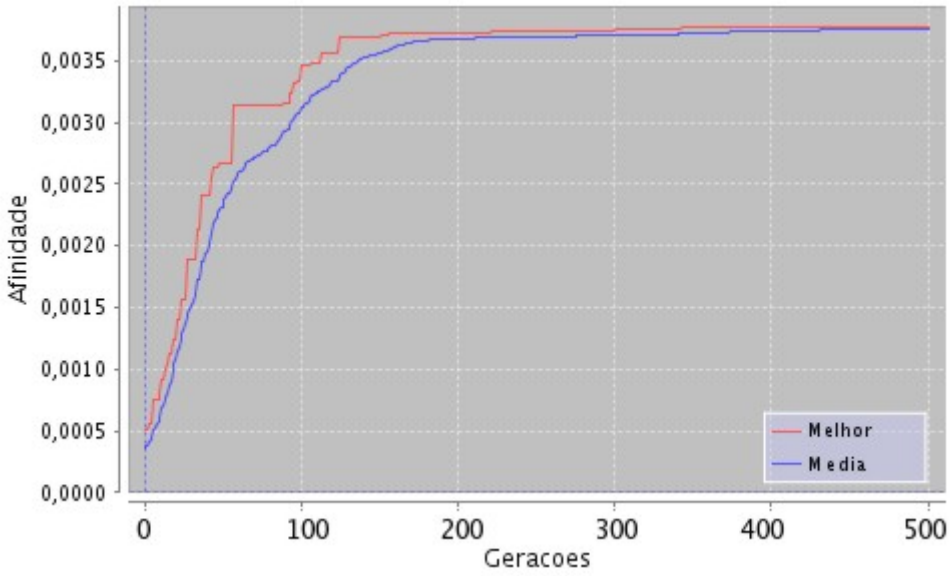

Figura 5.2: Teste 1: gráfico de execução do algoritmo para a solução de melhor *fitness* (300 anticorpos, 500 gerações e *fitness* de 265).

	melhor	melhor fit.	média	média fit.	tempo(min.)
0	0,003040	329	0,003022	330	3,52
1	0,003195	**313**	0,003171	315	3,25
2	0,002874	348	0,002837	352	3,33
3	0,002841	352	0,002758	362	3,36
4	0,003175	315	0,003122	320	3,78

Tabela 5.4: Tabela de resultados para instância 1: 100 anticorpos 100 gerações.

População de 100 anticorpos e 100 gerações (tabela 5.4)

Lembrando que a afinidade é o inverso do *fitness* ($afinidade = 1/fitness$). Em uma análise a partir dos gráficos, nota-se que as soluções iniciais geradas possuem um valor de *fitness* superior a 2000. Esse valor alto se deve à maneira aleatória com que essas soluções são geradas na inicialização.

Quando o programa começa a executar todos os operadores de seleção clonal, definidos como interações, pode ser observado no gráfico (figura 5.2) um rápido crescimento da afinidade, diminuindo assim as restrições violáveis (*fitness*). No gráfico da figura 5.1, nota-se o rápido crescimento da afinidade em até, aproximadamente, 300 gerações, e, após isso, toda a população começa a estagnar, não apresentando melhorias consideráveis.

Após várias interações, a população de soluções (anticorpos) começa a diminuir a variabilidade, conforme apresentado no gráfico (figura 5.1) pela distância entre os valores de melhor e média e, consequentemente, aproxima-se de um ótimo local, onde não são sustentáveis melhorias nas soluções.

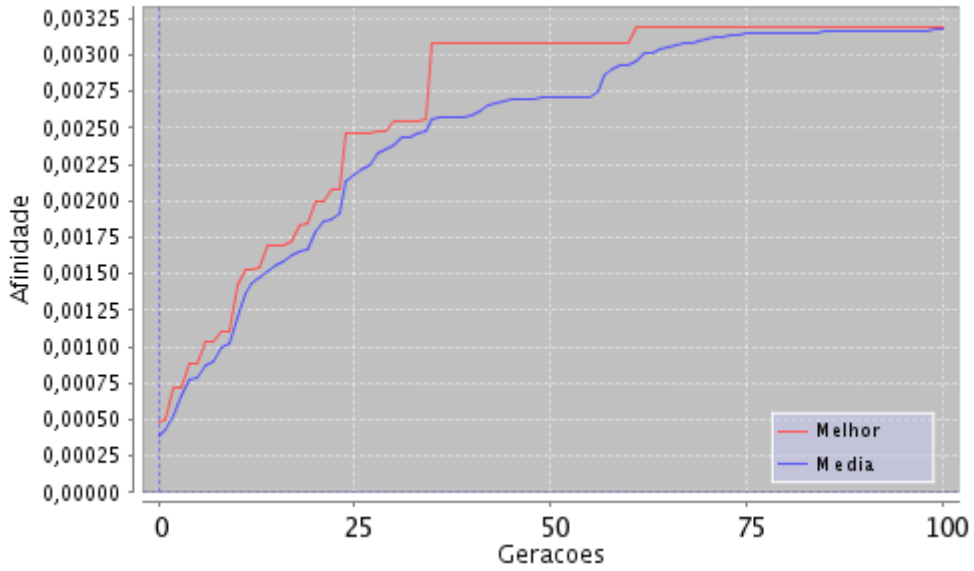

Figura 5.3: Teste 1: gráfico de execução do algoritmo para a solução de melhor *fitness* (100 anticorpos, 100 gerações e *fitness* de 313).

Quando a população de anticorpos é pequena, rapidamente a população começa a estagnar (gráficos das figuras 5.2 e 5.3) convergindo a um ótimo local. Entretanto, nada impede que aleatoriamente as soluções geradas converjam a um ótimo global, independente do tamanho da população. Como visto nos testes realizados com populações de 500, 300 e 100 anticorpos e 1000, 500 e 100 gerações, a melhor solução encontrada (valor de *fitness* igual a 265) foi aquela com apenas 300 indivíduos em 500 gerações (gráfico da figura 5.2).

5.2 Teste 2

Na segunda instância, tem-se as seguintes descrições (tabela 5.5 [Malim et al., 2006]):

Cursos	Aulas (A)	Salas (S)	Dias (D)	Períodos (P)	*slots* (W)	Turmas	Restricoes
52	223	12	5	4	20	30	148

Tabela 5.5: Descrição da instância 2.

A percentagem de ocupação de salas por *slots* desta instância é de 92,92%. Somando-se ainda a quantidade de restrições impostas (148), observa-se que essa instância é mais difícil que a primeira. Em contradição, os resultados encontrados são melhores (*fitness* menores). Observe os itens abaixo com os valores encontrados em outros trabalhos:

- [Gaspero e Schaerf, 2002], melhor resultado encontrado: 13.
- [Malim et al., 2006], melhor resultado encontrado: 21. Tempo gasto: 5,78 minutos.

5.2.1 Resultados e Análises

População de 500 anticorpos e 1000 gerações (tabela 5.6)

	melhor	melhor fit.	média	média fit.	tempo(min.)
0	0,041660	24	0,034370	29	36,96
1	0,050000	20	0,042954	23	36,41
2	0,062500	**16**	0,052923	18	38,25
3	0,055556	18	0,044946	22	37,17
4	0,052632	19	0,044726	22	86,84

Tabela 5.6: Tabela de resultados para instância 2: 500 anticorpos 1000 gerações.

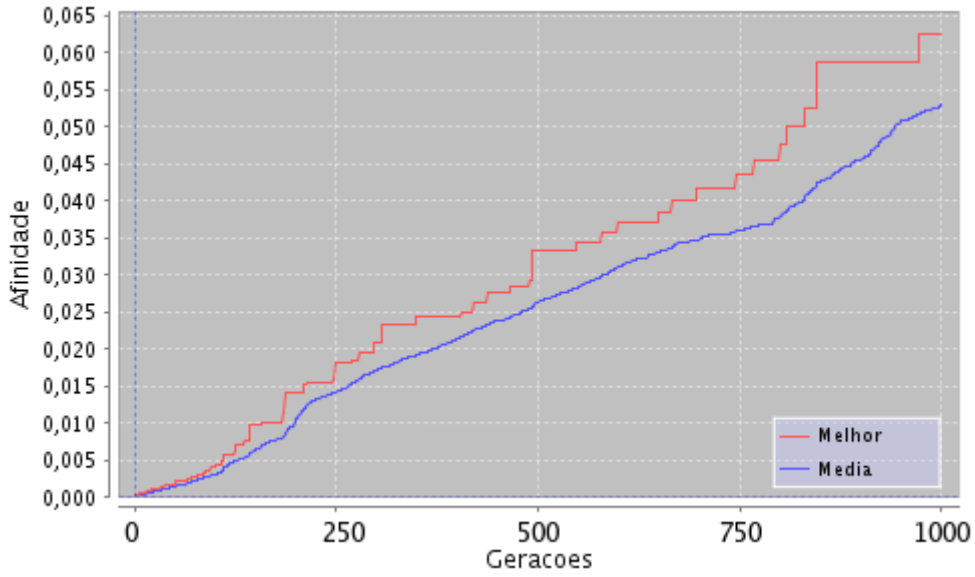

Figura 5.4: Teste 2: gráfico de execução do algoritmo para a solução de melhor *fitness* (500 anticorpos, 1000 gerações e *fitness* de 16).

População de 300 anticorpos e 500 gerações (tabela 5.7)

População de 100 anticorpos e 100 gerações (tabela 5.8)

Em comparação com a primeira instância, a segunda possui mais restrições e a percentagem de ocupação é maior. Devido a esses fatos, não foram apresentados nos gráficos (gráficos das figuras 5.4, 5.5 e 5.6) um crescimento brusco da afinidade. É provável que um número maior de gerações possa resultar em um valor melhor do que a solução atual, isto é, fitness igual a 16.

Nesta instância, mais exigente devido as restrições, nota-se a importância de prover um grande número de anticorpos. No gráfico da figura 5.6, a média da população convergiu rapidamente para a melhor solução do momento. Tal efeito se deve principalmente a falta de

	melhor	melhor fit.	média	média fit.	tempo(min.)
0	0,028571	35	0,026425	37	20,86
1	0,035714	28	0,029759	33	19,02
2	0,032258	31	0,028170	35	19,85
3	0,032258	31	0,029004	34	18,14
4	0,035714	**28**	0,031261	31	18,86

Tabela 5.7: Tabela de resultados para instância 2: 300 anticorpos e 500 gerações.

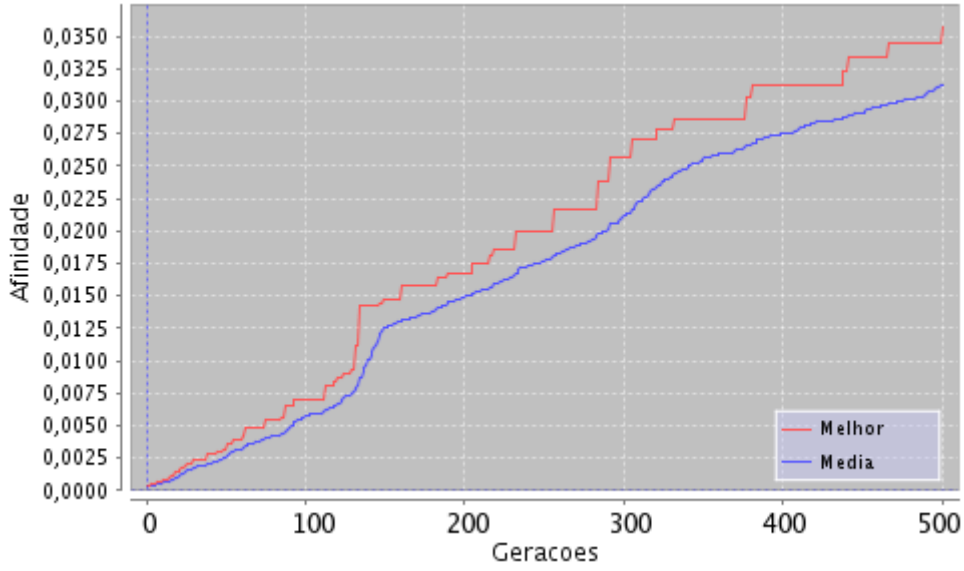

Figura 5.5: Teste 2: gráfico de execução do algoritmo para a solução de melhor *fitness* (300 anticorpos, 500 gerações e *fitness* de 28).

	melhor	melhor fit.	média	média fit.	tempo(min.)
0	0,014706	**68**	0,012601	79	3,15
1	0,013333	75	0,012237	81	3,17
2	0,012821	78	0,012108	82	3,27
3	0,014493	69	0,011514	86	3,48
4	0,011905	84	0,010650	93	2,68

Tabela 5.8: Tabela de resultados para instância 2: 100 anticorpos e 100 gerações.

diversidade. A diversidade se refere a uma maior cobertura no espaço de soluções, dando assim a possibilidade de fugir de ótimos locais.

A melhor solução encontrada (*fitness* igual a 16 conforme gráfico da figura 5.4), teve tempo de execução superior àquela encontrada no trabalho de [Malim et al., 2006]. A justificativa mais plausível para esse aumento deve-se à falta de informação no trabalho de [Malim et al., 2006] sobre a quantidade de anticorpos da população utilizada em seus experimentos.

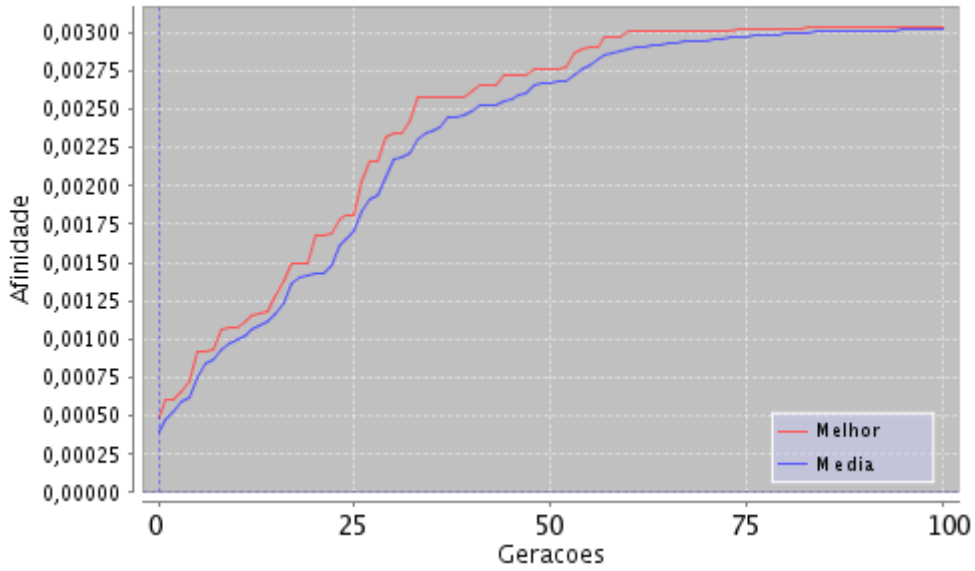

Figura 5.6: Teste 2: gráfico de execução do algoritmo para a solução de melhor *fitness* (100 anticorpos, 100 gerações e *fitness* de 68).

5.3 Teste 3

Na terceira instância, tem-se as seguintes descrições (tabela 5.9) [Malim et al., 2006]:

Cursos	Aulas (A)	Salas (S)	Dias (D)	Períodos (P)	*slots* (W)	Turmas	Restricoes
56	252	13	5	4	20	55	229

Tabela 5.9: Descrição da instância 3.

Nesta instância, a percentagem de ocupação de salas por *slots* é de 96,92%. São 229 restrições e 55 turmas, onde a quantidade de turmas também restringe, pois obriga que cursos da mesma turma não sejam agendados no mesmo *slot*. Logo, esta instância é considerada difícil, principalmente no processo aleatório de mutação definido pelo algoritmo.

Valores encontrado em outros trabalhos:

- [Gaspero e Schaerf, 2002], melhor resultado encontrado: 55.
- [Malim et al., 2006], melhor resultado encontrado: 69. Tempo gasto: 26.95 minutos.

5.3.1 Resultados e Análises

Haja vista as dificuldades da instância (alta percentagem de ocupação de sala e várias restrições), a implementação realizada neste trabalho não mostrou sucesso na solução. Alguns resultados foram colhidos, porém, devido ao tempo com que cada mutação gastava, a coleta de dados foi inviabilizada.

No processo de mutação, cada sala e curso são selecionados aleatoriamente. Em seguida é sorteado um *slot*, no qual será atribuído a sala e o curso. Como na instância 3 são várias restrições, é baixa a probabilidade de encontrar um *slot* válido para atribuição. Dessa forma, quando não é encontrado um *slot* válido, o algoritmo tenta novamente sortear outro *slot*, o qual também possui pouca probabilidade. Assim, o programa fica inerte no processo de mutação, gastando muito tempo entre cada interação.

Foram gastos quase dois dias para executar aproximadamente 450 interações. A execução do programa deu início às 21 horas e 16 minutos de sábado e foi encerrada, sem terminar todas as 1000 gerações, segunda-feira às 21 horas. O gráfico da figura 5.8 representa a execução deste programa.

Os *logs* de resultados contendo o tempo gasto, a semente do gerador de número aleatório, a média e o melhor valor de afinidade, são escritos apenas no final da execução do programa. Como no teste 3 foram gastos vários dias e, possivelmente, iam-se mais, ficou inviável manter o programa em execução. Apenas o gráfico (veja a figura 5.8) pôde ser salvo para análise.

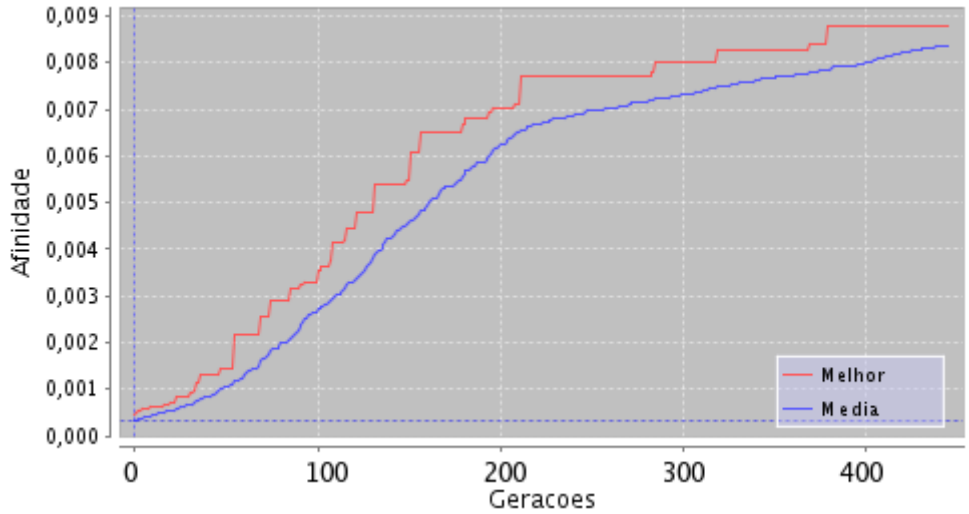

Figura 5.7: Teste 3: gráfico de execução da instância 3

O gráfico 5.8 indica que a afinidade está próxima de 0,009. Calculando o valor de *fitness* ($fitness = 1/afinidade$), obtém-se 111, o qual é um bom valor considerando a quantidade de interações e resultados de outros trabalhos, porém, prejudicado pelo tempo gasto para execução do programa.

Com a expectativa de melhorar o tempo de execução do programa, foram adicionadas no algoritmo, especialmente para a instância 3, duas heurísticas. Na primeira, foram agrupados os *slots* que não pertençam ao grupo de restrições (*Unavailability_Constraints* - veja a seção 4.1) de cada curso, fazendo que apenas sejam escolhidos, aleatoriamente, valores dentro desse grupo. Na segunda heurística foi incrementado um contador para limitar o processo de mutação até um valor determinado. Observe abaixo o gráfico da figura 5.8 com os resultados encontrados

mostrados na tabela 5.10.

	melhor	melhor fit.	média	média fit.	tempo(min.)
0	0,001798	556	0,001636	611	26
1	0,002220	**451**	0,002216	462	28
2	0,001879	532	0,001758	569	29

Tabela 5.10: Tabela de resultados para instância 3: 500 anticorpos e 1000 gerações.

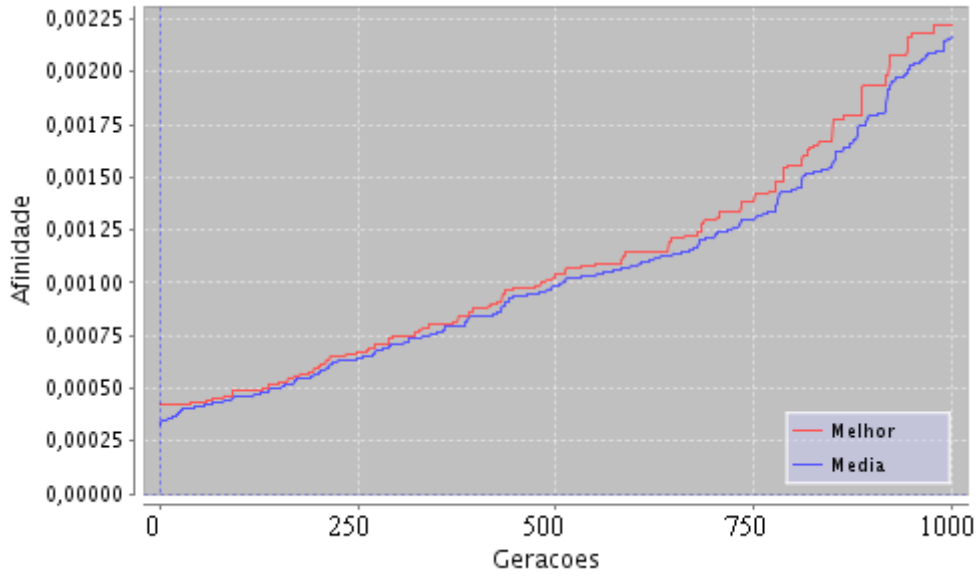

Figura 5.8: Teste 3: gráfico de execução da instância 3 utilizando heurísticas.

A partir do gráfico (figura 5.8), observe que a distância entre melhor e média é pequena, indicando pouca variabilidade da população de anticorpos. Essa pouca diferença entre média e melhor é justificada pela limitação do processo de mutação causado pela heurística.

O melhor anticorpo é aquele mais próximo de um ótimo, dessa forma, a mutação gasta mais tempo no processo de melhora. Dessa forma, utilizando a heurística, seus clones serão interrompidos quando a tentativa alcançar o limiar definido, Consequentemente, os piores terão maior chance de aumentar a afinidade (por serem mais fáceis de mutar). Portanto, a média da população (como é visto pelo gráfico) será elevada, porém, não oferecendo um grande aumento no melhor valor.

No trabalho de [Malim et al., 2006], onde o algoritmo de seleção clonal para automação de quadro horário foi publicado, desconsiderou-se o teste com uma quarta instância (encontrada no mesmo site das outras três), instância esta, com 100% de ocupação, poderá fazer o processo de mutação impossível. Dessa forma, a dificuldade de resolver a instância 3, com quase 100% de ocupação (96,92%), pode ser justificado.

Capítulo 6

Conclusões Finais e Trabalhos Futuros

6.1 Conclusões Finais

Em todos os testes, o valor de *fitness* encontrado se aproxima daquele obtido em outros trabalhos científicos [Malim et al., 2006, Gaspero e Schaerf, 2002]. Em contrapartida, o tempo de execução, principalmente para a terceira instância, não correspondeu à expectativa, demonstrando disparidade entre os resultados.

Verifica-se, portanto, a necessidade de um estudo mais específico envolvendo os parâmetros de configuração, tais como tamanho da população e número de indivíduos, para objetivar melhorias no processo.

Além disso, boas heurísticas poderiam ser aplicadas na implementação para melhorar o processo de mutação. O problema ocorrido na mutação, como foi constatado, deve-se à sua própria característica. Como visto, a mutação ocorrerá enquanto os clones gerados não forem melhores, ou iguais, ao anticorpo selecionado. Agora, suponha que este anticorpo é o ótimo local: como não é possível melhorar uma solução ótima, o programa fica inerte no processo de mutação. Outro problema encontrado na mutação e que inviabilizou o tempo de processamento da instância 3 de teste foi a dificuldade de atribuir os eventos em um quadro horário repleto de restrições e com alta percentagem de ocupação.

Nas instâncias 1 e 2, por serem mais fáceis de encontrar um *slot* válido, devido a pouca quantidade de restrições e a baixa ocupação (comparado com a instância 3), o processo de mutação rapidamente finalizava, encontrando um bom anticorpo (quadro horário satisfatível), de maior ou igual afinidade. Assim, na quantidade máxima de gerações definidas, mil gerações ocorreram com progresso. Porém, é provável que, se o número aumentasse (mais de mil gerações), em algum momento o programa ficaria inerte no processo de mutação.

De todo, pode-se concluir que o algoritmo de seleção clonal tendem a ser viáveis para o problema de quadros horários, desde que, a quantidade de restrições não sejam exorbitantes. Mesmo por que, como citado no trabalho de [Malim et al., 2006], "o algoritmo não foi projetado

para tratar quadros horários com 100% de ocupação".

6.2 Trabalhos Futuros

Algumas propostas de trabalhos futuros são apresentadas a seguir. São indicações ou de continuidade do relato aqui exposto ou novas implementações, não abordadas no desenvolvimento desse trabalho de pesquisa.

- Melhorias no processo de mutação, devido a dificuldade com que esse operador pussui para problemas com várias restrições.
- Desenvolver um projeto de software, incluindo os requisitos necessários para um sistema de geração automática de quadros horários, utilizando as classes que neste trabalho foram desenvolvidas.
- Incluir novas restrições, tanto violáveis quanto invioláveis, a fim de, especificar um sistema para resolver o problema de geração automática de quadros horários do Departamento de Ciência da Computação da Universidade Federal de Goiás.

Referências

Ao, P. (2005). Laws in darwinian evolutionary theory. *Physics of Life Reviews*, 2:117.

Ayara, M., Timmis, J., de Lemos, R. de Castro, L., e Duncan, R. (2002). Negative selection: How to generate detectors. In Timmis, J. e Bentley, P., editors, *1st International Conference on Artificial Immune Systems*, pages 89–98, University of Kent at Canterbury. Unversity of Kent at Canterbury Printing Unit.

Burke, E., de Werra, D., e Kingston, J. (2004). Applications to timetabling. In Gross, J. L. e Yellen, J., editors, *Handbook of Graph Theory*, pages 445–474. CRC Press London.

Cambazard, H., Demazeau, F., Jussien, N., e David, P. (2005). Interactively solving school timetabling problems using extensions of constraint programming. ISBN 3-540-30705-2.

Cooper, T. B. e Kingston, J. H. (1995). The complexity of timetable construction problems. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling (ICPTAT '95)*, pages 511–522.

Dasgupta, D. (1998). *Artficial Immune Systems and Their Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Dasgupta, D. (2006). Advances in artificial immune systems. IEEE Computational Intelligence Magazine.

Dasgupta, D., Ji, Z., e Gonzlez, F. (2003). Artificial immune system.

de Castro, L. N. e Zuben, F. J. V. (2000). The clonal selection algorithm with engineering applications. In *Artificial Immune Systems*, pages 36–39, Las Vegas, Nevada, USA.

de Castro Silva, L. N. (2001). *Engenharia Imunológica: Desenvolvendo Aplicação e Ferramentas Computacionais inspiradas em Sistemas Imunológicos Artificiais*. PhD thesis, Universidade Estadual de Campinas.

de Paula, F. S. (2004). *Um arquitetura de segurança computacional inspirada no sistema imunológico*. PhD thesis, Universidade Estadual de Campinas - UNICAMP.

e Silva, A. S. N., Sampaio, R. M., e Alvarenga, G. B. (2005). Uma aplicação de simulated annealing para o problema de alocaçãoo de salas.

Fernandes, C., Caldeira, J. P., Melicio, F., e Rosa, A. (1999). High school weekly timetabling by evolutionary algorithms. In *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*, pages 344–350, New York, NY, USA. ACM.

Ford, R. (2004). The wrong stuff? *IEEE Security and Privacy*, 2(3):86–89.

Forrest, S., Perelson, A. S., Allen, L., e Cherukuri, R. (1994). Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 202–212, Oakland, CA. IEEE Computer Society Press.

Gaspero, L. D., McCollum, B., e Schaerf, A. (2007). The second international timetabling competition (itc-2007): Curriculum-based course timetabling.

Gaspero, L. D. e Schaerf, A. (2002). Multi-neighbourhood local search with application to course timetabling. In *PATAT*, pages 262–275.

Gueret, C., Jussien, N., Boizumault, P., e Prins, C. (1995). Building university timetables using constraint logic programming. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling (ICPTAT '95)*, pages 393–408.

Hofmeyr, S. A. (2000). An interpretative introduction to the immune system. *Design Principles for the Immune System and other Distributed Autonomous Systems, Oxford University Press*.

Janeway, C. A., Walport, M., Travers, P., e Capra, J. D. (2000). *Imunobiologia: o Sistema Imunológico na Saúde e na Doença*. Artes Médicas, 4ª Ed.

Junior, A. F. L. J. e da Rocha, C. A. J. (2006). Aghora: Algoritmos genéticos para geração de horários de aula. Universidade da Amazônia (UNAMA); Belem; PA.

Kirkpatrick, S., Gelatt, C. D., e Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680.

Malim, M. R., Khader, A. T., e Mustafa, A. (2006). Artificial immune algorithms for university timetabling. *E. K. Burke, H. Rudová (Eds): PATAT 2006*, pages 234–245.

Neufeld, G. A. e Tartar, J. (1974). Graph coloring conditions for the existence of solutions to the timetable problem. *Commun. ACM*, 17(8):450–453.

Perelson, A. S. e Weisbuch, G. (1997). Immunology for physicists. *Reviews of Modern Physics*, 69(4):1219+.

Russell, S. J. e Norvig, P. (1995). *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Schaerf, A. (1995). A survey of automated timetabling. In *115*, page 33. Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X.

S.N.Silva, A., Sampaio, R. M., e Alvarenga, G. B. (2005). Uma aplicação de simulated annealing para o problema de alocação de salas. *INFOCOMP Journal of Computer Science*, 4(3):59–66.

Somayaji, A., Hofmeyr, S., e Forrest, S. (1997). Principles of a computer immune system. In *Meeting on New Security Paradigms, 23-26 Sept. 1997, Langdale, UK*, pages 75–82. New York, NY, USA : ACM, 1998.

Starlab (WWW). Url: `http://www.starlab.org/genes/ais/`; último acesso: 16 de junho de 2007.

Stewart, J. e Coutinho, A. (2004). The affirmation of self: a new perspective on the immune system. *Artif. Life*, 10(3):261–276.

Timmis, J. (2000). *Artificial Immune Systems: A novel data analysis technique inspired by the immune network theory*. PhD thesis, Department of Computer Science, University of Wales, Aberystwyth. Ceredigion. Wales.

Tonegawa, S. (1983). Somatic generation of antibody diversity. *Nature*, 302(5909):575–81.

Apêndice A

Código Fonte

A.1 Classe Abstrata `CSA`

```
1  /*
2   * Classe CSA (Clonal Selection Algorithm)
3   *
4   * Essa classe abstrada implementa
5   * os principais operadores de seleção clonal
6   * juntamente com a especificação dos
7   * métodos auxiliares abstratos.
8   */
9
10 package aistimetabling;
11
12 import aistimetabling.tabela.Restricoes;
13 import java.util.Collections;
14 import java.util.Comparator;
15 import java.util.LinkedList;
16 import java.util.List;
17 import java.util.Vector;
18
19 public abstract class CSA implements Runnable{
20
21     protected int tamPop;       //tamanho da população
22     protected Vector<Evento> vetEventos;    //vetor de Eventos
23     public static List<Anticorpo> pop;      //Lista de anticorpos da População
24
25     /** Cria uma nova instância de CSA */
26     public CSA(int tamPop, Vector<Evento> eventos) {
27         this.tamPop = tamPop;
28         this.vetEventos = eventos;
29     }
30
31     /*
32      * retorna a Lista de anticorpos da população
33      */
34     public List<Anticorpo> getPop(){
35         return pop;
36     }
37
38     /*
39      * executa todas as operações de seleção clonal
40      */
41     public void run(){
42         pop = new LinkedList();     //instancia uma lista ligada
43         int nClones = 0;        // número de clones
44         boolean mutation;       // flag que indica a mutação
45
46         //1.inicializa uma população de anticorpos
47         inicializar(pop);
48
49         //para cada geração de anticorpos (prováveis timetabling)
```

```
50          for (Anticorpo anticorpo: pop){
51              //determina a afinidade do anticorpo (affinity = 1/fitness)
52              anticorpo.setAfinidade(affinity(anticorpo));
53          }
54
55          //faça para cada geração de anticorpos
56          do{
57              //ordena a população de anticorpos a partir da afinidade
58              //os indivíduos com menor afinidade são os primeiros
59              ordenar(pop);
60              //randomicamente seleciona um anticorpo (timetabling)
61              Anticorpo anticorpo = roulette(pop);
62              //selection probability = affinity/total affinities
63              double probability = anticorpo.getAfinidade()/totalAffinities(pop);
64              //nClones = population size x cumulative selections probability
65              nClones = ((int)(pop.size() * probability))+100;
66              //vetor de anticorpos
67              Anticorpo[] vetClones = new Anticorpo[nClones];
68              for(int i = 0; i<nClones; i++)
69                  //clona o anticorpo e atribui na posição i do vetor de clones
70                  vetClones[i] = anticorpo.clonar();
71              //para cada geração de clones
72              for(int i =0; i<nClones; i++){
73                  //calcula a taxa de mutação do clone = 1 - selection probability
74                  vetClones[i].setTaxaMutacao(1-probability);
75                  //guarda a afinidade original, antes da mutação
76                  double originalCloneAffinity = vetClones[i].getAfinidade();
77                  mutation = false;
78                  //se um valor aleatório é menor que a taxa de mutação
79                  if (getNAleatorio() <= vetClones[i].getTaxaMutacao()){
80                      while (mutation == false){
81                          //satisfazendoas as restrições invioláveis
82                          //são realizadas as mutações no anticorpo
83                          Anticorpo t = vetClones[i].clonar();
84                          vetClones[i].mutar();
85                          //se não existe clones semelhantes
86                          //clones semelhantes no problema de timetabling
87                          //são quadros horários com a mesma disposição de aulas
88                          if (!Restricoes.duplicado(vetClones[i],vetClones)){
89                              //determina a finidade do novo clone
90                              vetClones[i].setAfinidade(affinity(vetClones[i]));
91                              /* se a afinidade do novo clone é maior que a do
92                                 anticorpo original (anticorpo 'pai')
93                              */
94                              if (vetClones[i].getAfinidade() >= originalCloneAffinity){
95                                  //flag sinalização final do processo de mutação
96                                  mutation = true;
97                                  if (i+1 < nClones)
98                                  vetClones[i+1] = vetClones[i];
99                              }
100                             else
101                                 vetClones[i] = t;
102                         }
103                     } // fim do while
104                 } //fim do if
105                 // procura o anticorpo com a menor afinidade (o pior anticorpo)
106                 Anticorpo X = this.minimum(pop);
107                 //se a afinidade do clone na posição i é maior que a o pior anticorpo
108                 if (vetClones[i].getAfinidade() > X.getAfinidade()){
109                     //se a população não contém algum anticorpo semelhante ao clone gerado
110                     if (!pop.contains(vetClones[i])){
111                         //remove o pior da população, anticorpo X
112                         pop.remove(X);
113                         //adiciona o novo clone
114                         pop.add(vetClones[i]);
115                     }
116                 }
117             }//fim do for: Mutaion
118             //plota o gráfico de crescimento da afinidade
119             plotar(pop);
120             //realiza esse processo até o critério de parada ser obedecido
121         }while(stoppingCriteria());
122     }
123
```

```
124     /*
125      * Cálculo da Afinidade de um anticorpo
126      *
127      */
128     protected double affinity(Anticorpo anticorpo){
129         return 1/fitness(anticorpo);
130     }
131 
132     /*
133      * Método que retorna o anticorpo com a menor afinidade da população
134      *
135      */
136     protected Anticorpo minimum(List<Anticorpo> pop){
137         Anticorpo menorAnticorpo = Collections.min(pop,new Comparator<Anticorpo>(){
138             public int compare(Anticorpo o1, Anticorpo o2) {
139                 if (o1.getAfinidade() > o2.getAfinidade()) return 1;
140                 else if (o1.getAfinidade() < o2.getAfinidade()) return -1;
141                 return 0;
142             }
143         });
144         return menorAnticorpo;
145     }
146 
147     /*
148      * Soma a afinidade de todos os anticorpos da população
149      *
150      */
151     protected double totalAffinities(List<Anticorpo> pop) {
152         double total = 0;
153         for(Anticorpo anticorpo: pop)
154             total+=anticorpo.getAfinidade();
155         return total;
156     }
157 
158     /*
159      * Ordena a população em ordem crescente
160      *  pelo valor da afinidade
161      */
162     private void ordenar(List<Anticorpo> pop){
163         Collections.sort(pop,new java.util.Comparator(){
164             public int compare(Object o1, Object o2){
165                 Anticorpo anticorpo1 = (Anticorpo)o1,
166                         anticorpo2 = (Anticorpo)o2;
167                 if (anticorpo1.getAfinidade() > anticorpo2.getAfinidade())
168                     return 1;
169                 else if (anticorpo1.getAfinidade() < anticorpo2.getAfinidade())
170                     return -1;
171                 return 0;
172             }
173         });
174     }
175 
176     /*
177      * Método abstrado, utilizado para iniciar cada anticorpo
178      * da população
179      */
180     protected abstract void inicializar(List<Anticorpo> pop);
181 
182     /*
183      * Método abstrado que realiza o cálculo do fitness de um quadro horário
184      */
185     public abstract double fitness(Anticorpo anticorpo);
186 
187     /*
188      * Método abstrado que determina o critério de parada
189      */
190     protected abstract boolean stoppingCriteria();
191 
192     /*
193      * Método abstrado que implementa o métoda da roleta
194      */
195     protected abstract Anticorpo roulette(List<Anticorpo> pop);
196 
197     /*
```

```
198         * Método abstrado utilizado para uma implementar o mecanismo de
199         * visualização do gráfico de
200         * crescimento da afinidade durante as interações
201         */
202        protected abstract void plotar(List<Anticorpo> pop);
203
204        /*
205         * Método que retorna um número randômico gerado
206         */
207        protected double getNAleatorio(){
208            return Math.random();
209        }
210    }
```

A.2 Interface `Anticorpo`

```
1   /*
2    * Interface especificando os métodos necessário para um
3    * anticorpo
4    */
5
6   package aistimetabling;
7
8   public interface Anticorpo {
9
10      /*adiciona um repertorio ao anticorpo
11       * No problema de Timetabling, um repertório
12       * é cada evento que será
13       * adicionado a tabela de quadros horários
14       */
15      public void addRepertorio(Evento evt);
16
17      /*
18       * seta a afinidade do anticorpo
19       * A afinidade é calculada a partir do fitness, sendo que
20       * afinidade = 1/fitness
21       */
22      public void setAfinidade(double afinidade);
23
24      /*
25       * retorna a afinidade do anticorpo
26       */
27      public double getAfinidade();
28
29      /*
30       * seta a Taxa de Mutação do anticorpo
31       */
32      public void setTaxaMutacao(double mutation);
33
34      /*
35       * retorna a taxa de Mutação do anticorpo.
36       */
37      public double getTaxaMutacao();
38
39      /*
40       * Realiza o processo de Mutação do anticorpo
41       *
42       */
43      public void mutar();
44
45      /*
46       * retorna um clone o anticorpo
47       */
48      public Anticorpo clonar();
49  }
```

A.3 Interface `Evento`

```
1  /*
2   * Interface marcadora que especifica
3   * um Evento
4   */
5
6  package aistimetabling;
7
8  public interface Evento {
9
10
11 }
```

A.4 Classe `Curso`

```
1  /*
2   * Classe que especifica um Curso,
3   * onde os atributos são carregados a partir
4   * do arquivo de especificação do problema
5   *
6   * Para o problema de quadros horários de
7   * curso universitário, um curso é um evento
8   * que deve ser agendado no quadro horário.
9   * Por esse motivo, ele implementa a interface
10  * evento.
11  */
12
13 package aistimetabling.course;
14
15 import aistimetabling.Evento;
16
17 public class Curso implements Evento{
18
19     private String cursoId,
20             professor;
21     private int nAulas, minAulasDias, nEstudantes;
22     private int aulasAtribuidas;
23
24     public Curso(String cursoId) {
25         this.cursoId = cursoId;
26     }
27
28     public void setCursoId(String cursoId){
29         this.cursoId = cursoId;
30     }
31
32     public void setProfessor(String professor){
33         this.professor = professor;
34     }
35
36     public void setNAulas(int nAulas){
37         this.nAulas = nAulas;
38     }
39
40     public void setMinAulasDias(int minAulasDias){
41         this.minAulasDias = minAulasDias;
42         this.aulasAtribuidas = minAulasDias;
43     }
44
45     public int getMinAulasDias(){
46         return minAulasDias;
47     }
48
49     public void setNEstudantes(int nEstudantes){
50         this.nEstudantes = nEstudantes;
51     }
52
53     public String getCursoId(){
54         return this.cursoId;
55     }
```

```
56
57      public String getProfessor(){
58          return this.professor;
59      }
60
61      public int getNAulas(){
62          return this.nAulas;
63      }
64
65      public int getNEstudantes(){
66          return this.nEstudantes;
67      }
68
69      /* verifica se um objeto passado como parâmetro
70       * é igual a própria instância dessa classe.
71       */
72      public boolean equals(Object obj){
73          return obj instanceof Curso?
74              ((Curso)obj).getCursoId().equals(this.cursoId) : false;
75      }
76
77      // retorna o código Hash
78      public int hashCode(){
79          return cursoId.hashCode();
80      }
81
82      public String toString(){
83          return cursoId+"\t"+professor+"\t"+nAulas+
84                 "\t"+minAulasDias+"\t"+nEstudantes;
85      }
86  }
```

A.5 Classe `Sala`

```
1   /*
2    * Classe que especifica uma sala,
3    * onde os atributos são carregados a partir
4    * do arquivo de especificação do problema
5    */
6
7   package aistimetabling.course;
8
9   public class Sala {
10
11      //String identificadora da Sala
12      private String salaId;
13      private int capacidade;
14
15      public Sala(String salaId) {
16          this.salaId = salaId;
17      }
18
19      public void setSalaId(String salaId){
20          this.salaId = salaId;
21      }
22
23      public void setCapacidade(int capacidade){
24          this.capacidade = capacidade;
25      }
26
27      public String getSalaId(){
28          return this.salaId;
29      }
30
31      public int getCapacidade(){
32          return this.capacidade;
33      }
34
35      //verifica se os objetos são iguais
36      public boolean equals(Object obj){
37          return obj instanceof Sala?
```

```
38              ((Sala)obj).getSalaId().equals(salaId) : false;
39      }
40
41      //Retorna o código Hash da classe
42      public int hashCode(){
43          return salaId.hashCode();
44      }
45
46      public String toString(){
47          return this.salaId;
48      }
49
50  }
```

A.6 Classe `Turma`

```
1   /*
2    * Classe que especifica uma turma,
3    * onde os atributos são carregados a partir
4    * do arquivo de especificação do problema
5    */
6
7   package aistimetabling.course;
8
9   public class Turma {
10
11      private String turmaId;
12      private String[] vetCursos;
13      private int nTurma;
14
15      public Turma(String turmaId, int nTurma) {
16          vetCursos = new String[nTurma];
17          this.nTurma = nTurma;
18      }
19
20      public void setTurmaId(String turmaId){
21          this.turmaId = turmaId;
22      }
23
24      public void setVetCursos(String[] vetCursos){
25          this.vetCursos = vetCursos;
26      }
27
28      public void setNTurma(int nTurma){
29          this.nTurma = nTurma;
30      }
31
32      public String getTurmaId(){
33          return this.turmaId;
34      }
35
36      public String[] getVetCursos(){
37          return this.vetCursos;
38      }
39
40      /*
41       * Método que verifica se para essa turma
42       * contém o curso passado como parâmetro
43       */
44      public boolean contains(Curso curso){
45          for(String strCurso: vetCursos)
46              if (strCurso.equals(curso.getCursoId()))
47                  return true;
48          return false;
49      }
50
51      public int getNTurma(){
52          return this.nTurma;
53      }
54
55      public boolean equals(Object obj){
```

```
56          return obj instanceof Turma ?
57             ((Turma)obj).getTurmaId().equals(turmaId) : false;
58      }
59
60      public int hashCode(){
61          return turmaId.hashCode();
62      }
63  }
```

A.7 Classe `CursoRestricao`

```
1   /*
2    * Classe que determina as
3    * restrições carrega do arquivo
4    * de especificação do Problema
5    */
6
7   package aistimetabling.course;
8
9   public class CursoRestricao {
10
11      //string identificadora do curso
12      private String cursoId;
13      private int dia, periodo;
14
15      public CursoRestricao() {
16      }
17
18      public void setCursoId(String cursoId){
19          this.cursoId = cursoId;
20      }
21
22      public void setDia(int dia){
23          this.dia = dia;
24      }
25
26      public void setPeriodo(int periodo){
27          this.periodo = periodo;
28      }
29
30      public String getCursoId(){
31          return this.cursoId;
32      }
33
34      public int getDia(){
35          return this.dia;
36      }
37
38      public int getPeriodo(){
39          return this.periodo;
40      }
41
42      /* verifica se um objeto obj
43       * é igual a esta própria instância
44       */
45      public boolean equals(Object obj){
46          if (obj instanceof CursoRestricao)
47              return false;
48          CursoRestricao aux = (CursoRestricao)obj;
49          return aux.getCursoId().equals(cursoId) &&
50                 aux.getDia() == dia &&
51                 aux.getPeriodo() == periodo;
52      }
53
54      /*
55       * Código de Hash da Classe
56       */
57      public int hashCode(){
58          return cursoId.hashCode()+dia+periodo;
59      }
60  }
```

A.8 Classe `Aula`

```
1  /*
2   *
3   * Essa classe apenas agrega um curso
4   * e uma sala definindo uma sala
5   *
6   */
7
8  package aistimetabling.tabela;
9
10 import aistimetabling.Evento;
11 import aistimetabling.course.Curso;
12 import aistimetabling.course.Sala;
13
14 /**
15  *
16  * @author Thiago
17  */
18 public class Aula {
19
20     private Curso curso;
21     private Sala sala;
22
23     /** Creates a new instance of Aula */
24     public Aula(Curso curso, Sala sala) {
25         this.curso = curso;
26         this.sala = sala;
27     }
28
29     public Aula(){}
30
31     public Curso getCurso(){
32         return this.curso;
33     }
34
35     public Sala getSala(){
36         return this.sala;
37     }
38
39     /*
40      * Método que verifica se uma sala
41      * passada como parâmetro
42      * é igual a aquela agregada nesta classe
43      */
44     public boolean equalSala(Aula aula){
45         return aula.getSala().equals(this.sala);
46     }
47
48     /*
49      * Método que verifica se um curso
50      * passada como parâmetro
51      * é igual a aquela agregada nesta classe
52      */
53     public boolean equalsCurso(Aula aula){
54         return aula.getCurso().equals(this.curso);
55     }
56
57     /* Verifica se um objeto obj,
58      * é igual a o definido por essa instância.
59      * Aulas iguais possuem o mesmo curso e
60      * a mesma sala
61      */
62     public boolean equals(Object obj){
63         if (!(obj instanceof Aula))
64             return false;
65         Aula aula = (Aula)obj;
66         return aula.getCurso().equals(this.curso)
67             && aula.getSala().equals(this.sala);
68     }
69
70     //retorna o código Hash dessa classe
71     public int hashCode(){
72         return curso.hashCode()+sala.hashCode();
```

```
73      }
74
75      public String toString(){
76          return "["+curso.getCursoId()+"]";
77      }
78
79  }
```

A.9 Classe `CSACourse`

```
1   /*
2    *
3    *  Classe que extende classe abstrata CSA
4    *  Nessa classe, é especificada para o problema
5    *  de cursos universitários.
6    *
7    * Os métodos que calculas as restrições são também
8    * implementados
9    */
10
11  package aistimetabling.tabela;
12
13  import aistimetabling.*;
14  import aistimetabling.course.*;
15  import java.util.*;
16  import org.jfree.ui.RefineryUtilities;
17  import grafico.XYGrafico;
18
19  public class CSACourse extends CSA{
20
21      private int nInteracoes = 100000;
22      public int interaCorrente = 0;
23      public CursoRestricao[] vetCursoRestricao = null;
24      private Sala[] vetSala;
25      private Turma[] vetTurma;
26      private int dias, periodos;
27
28      //Objeto gerador de Número aleatórios
29      private java.util.Random r =
30              new java.util.Random((System.currentTimeMillis()));
31
32      /** Creates a new instance of CSACourse */
33      public CSACourse(int tamPop, Vector<Evento> vetEvento,
34              Sala[] vetSala, CursoRestricao[] vetCursoRestricao,
35              Turma[] vetTurma, int dias, int periodos, int nInteracoes) {
36          super(tamPop, vetEvento);
37          this.vetCursoRestricao = vetCursoRestricao;
38          this.dias = dias;
39          this.periodos = periodos;
40          this.vetSala = vetSala;
41          this.vetTurma = vetTurma;
42          this.nInteracoes = nInteracoes;
43      }
44
45      public void setRandomGenerator(java.util.Random r){
46          this.r = r;
47      }
48
49      public void inicializar(List<Anticorpo> pop) {
50          while (pop.size() < tamPop){
51              //cria anticorpo
52              Anticorpo anticorpo = new Timetabling(periodos,dias,
53                      vetSala, vetCursoRestricao, vetTurma, this.r);
54              for(Evento evt: vetEventos){    //para cada evento
55                  Curso curso = (Curso)evt;
56                  for (int i=0; i<curso.getNAulas(); i++)
57                      //aleatoriamente atribui um evento
58                      //ao timetabling satisfazendo as restrições
59                      anticorpo.addRepertorio(evt);
60              }
61              //se o anticorpo não é duplicado
```

```
62              if (!pop.contains(anticorpo))
63                  //adiciona o anticorpo a população
64                  pop.add(anticorpo);
65          }
66      }
67
68      /*
69       * Método que retorna o cálculo do fitness.
70       * Todas as restrições violáveis específica do
71       * problema são calculadas
72       */
73      public double fitness(Anticorpo anticorpo) {
74          double fit = nEstudanteSemLugar(anticorpo)+
75                       (nCursosMenor(anticorpo)*5)+
76                       (custoTurmaCompactness(anticorpo));
77          return fit;
78      }
79
80      /* Restrição Violável:
81       * nº de estudante sem lugar pra sentar
82       */
83      public int nEstudanteSemLugar(Anticorpo anticorpo){
84          int count = 0;
85          Timetabling timetabling = (Timetabling)anticorpo;
86          for (int i=0; i<timetabling.nSlots; i++){
87              List<Aula> slot = timetabling.getSlot(i);
88              for(Aula aula: slot){
89                  int diferenca = aula.getSala().getCapacidade() -
90                       aula.getCurso().getNEstudantes();
91                  if (diferenca < 0)
92                      count += Math.abs(diferenca);
93              }
94          }
95          return count;
96      }
97
98      /* Restrição Violável:
99       * nº de dias que foi atribuído a
100      * uma quantidade de slots menor
101      * do que o número mínimo de dias
102      */
103     public int nCursosMenor(Anticorpo anticorpo){
104         Timetabling timetabling = (Timetabling)anticorpo;
105         int count = 0;
106         for (Evento evt: vetEventos){
107             Curso curso = (Curso)evt;
108             if (diasTrabalhados(curso,timetabling) <
109                   curso.getMinAulasDias())
110                 count += curso.getMinAulasDias() -
111                   diasTrabalhados(curso,timetabling);
112         }
113         return count;
114     }
115
116     /*
117      * Soma os dias em que um curso foi ministrado
118      */
119     public int diasTrabalhados(Curso curso,
120                                Timetabling timetabling){
121         int count = 0;
122         proximo_dia:
123             for (int i=0; i<timetabling.diasAulas; i++){
124                 for (int j = 0; j<timetabling.numAulas; j++){
125                     for(Aula aula: timetabling.table[i][j]){
126                         if (aula.getCurso().equals(curso)){
127                             count ++;
128                             continue proximo_dia;
129                         }
130                     }
131                 }
132             }
133             return count;
134     }
135
```

```
136     /* Restrição Violável:
137      * Conta o número de cursos que não são adjacentes
138      * a qualquer outro da mesma turma
139      */
140     public int custoTurmaCompactness(Anticorpo anticorpo){
141         Timetabling tabela = (Timetabling)anticorpo;
142         int count = 0, ppd = tabela.numAulas;
143         for (Turma turma: vetTurma){
144             for(int p=0; p<tabela.nSlots; p++){
145                 if (turmaPeriodoAula(turma,p,tabela) > 0){
146                     if ((p % ppd == 0) &&
147                            turmaPeriodoAula(turma,p+1,tabela) == 0)
148                         count += turmaPeriodoAula(turma,p,tabela);
149                     else if ((p % ppd == ppd-1) &&
150                            turmaPeriodoAula(turma,p-1,tabela) == 0)
151                         count += turmaPeriodoAula(turma,p,tabela);
152                     else if (turmaPeriodoAula(turma,p+1,tabela) == 0 &&
153                         turmaPeriodoAula(turma,p-1,tabela) == 0)
154                         count += turmaPeriodoAula(turma,p,tabela);
155                 }
156             }
157         }
158         return count;
159     }
160 
161     /*
162      * Método que conta, a partir no quadro horário,
163      * os cursos que  são de uma mesma turma
164      */
165     public int turmaPeriodoAula(Turma turma, int periodo,
166                     Timetabling timetabling){
167         if (periodo < 0 || periodo >= timetabling.nSlots)
168             return 0;
169         int count = 0;
170         //para cada aula do slot
171         for(Aula aula: timetabling.getSlot(periodo)){
172             Curso curso = aula.getCurso();
173             //se o curso pertence a Turma (Curriculum)
174             if (turma.contains(curso))
175                 count++;
176         }
177         return count;
178     }
179 
180     //Determina critério de parada
181     protected boolean stoppingCriteria() {
182         this.interaCorrente++;
183         return ( interaCorrente-1 < nInteracoes);
184     }
185 
186     // Implementa o método da Roleta
187     protected Anticorpo roulette(List<Anticorpo> pop) {
188         double afinAcumulada = 0;
189         for (int i=1; i<pop.size(); i++){
190             afinAcumulada += pop.get(i).getAfinidade();
191         }
192         double nAleatorio = r.nextDouble()*afinAcumulada;
193         double aux = 0;
194         int i=0;
195         for(i=0; ( i<pop.size() && (aux < nAleatorio)); ++i){
196             aux += pop.get(i).getAfinidade();
197         }
198         i--;
199         return pop.get(i);
200     }
201 
202     /* Esse método pode ser implementado, para auxiliar
203      * em uma vizualização das interações ocorrida durante
204      * a execução do programa
205      */
206     protected void plotar(List<Anticorpo> pop){
207 
208     }
209 
```

```
210 }
```

A.10 Classe `Restricoes`

```
1  /*
2   * Esta classe possui métodos
3   * estáticos que implementam
4   * todas as as definições das restrições
5   * Invioláveis.
6   *
7   */
8
9  package aistimetabling.tabela;
10
11 import aistimetabling.Anticorpo;
12 import aistimetabling.course.*;
13 import java.util.List;
14
15 public abstract class Restricoes {
16
17     /*
18      * Este método verifica se as restrições
19      * impostas pela especificação de entrada
20      * (Unaviability_Constraints), são obedecidas
21      *
22      */
23     public static boolean conflitoTurma(Aula aula,
24             CursoRestricao vetCurRest[],
25             int dia, int periodo){
26         for (CursoRestricao cursoRestricao: vetCurRest)
27             if(cursoRestricao.getCursoId().equals(
28                     aula.getCurso().getCursoId())){
29                if (cursoRestricao.getDia() == dia &&
30                     cursoRestricao.getPeriodo() == periodo)
31                    return true;
32             }
33         return false;
34     }
35
36     /*
37      * Verifica se em um slot do quadro horário,
38      * o professor já esta oculpado.
39      */
40     public static boolean conflitoProfessor(Aula aula,
41              List<Aula> slot){
42         for(Aula auxAula: slot)
43             if (aula.getCurso().getProfessor().equals(
44                    auxAula.getCurso().getProfessor()))
45                 return true;
46         return false;
47     }
48
49     /*
50      * Verifica se em um slot do quadro horário,
51      * a aula passada como parâmetro, não é igual
52      * a alguma outra da mesma turma naquele slot.
53      */
54     public static boolean mesmaTurma(Aula aula,
55                List<Aula> slot, Turma[] vetTurma){
56         for(Aula aulaSlot: slot){
57             for(Turma turma: vetTurma){
58                 if (turma.contains(aulaSlot.getCurso()) &&
59                      turma.contains(aula.getCurso())){
60                     return true;
61                 }
62             }
63         }
64         return false;
65     }
66
67     /*
```

```
68       * Verifica se em um slot do quadro horário,
69       * o curso já está agendado.
70       */
71      public static boolean conflitoCurso(Aula aula,
72                List<Aula> slot){
73          for(Aula auxAula: slot)
74              if (aula.equalsCurso(auxAula))
75                  return true;
76          return false;
77      }
78
79      /*
80       * Verifica se em um slot do quadro horário,
81       * a sala já esta oculpada.
82       */
83      public static boolean conflitoSala(Aula aula,
84                        List<Aula> slot){
85          for(Aula auxAula: slot)
86              if (aula.equalSala(auxAula))
87                  return true;
88          return false;
89      }
90
91      /*
92       * Verifica se o timetabling (1º argumento)
93       * é igual a algum timetabling da população
94       */
95      public static boolean duplicado(Anticorpo timetabling,
96                     Anticorpo[] populacao){
97          for(Anticorpo anticorpo: populacao){
98              //se o conteúdo for igual,
99              //mas se forem objetos diferentes
100             if (anticorpo.equals(timetabling) &&
101                           timetabling !=anticorpo){
102                 return true;
103             }
104         }
105         return false;
106     }
107
108 }
```

A.11 Classe `Lotacao`

```
1   /*
2    * Classe auxiliar:
3    * Essa classe utiliza uma matriz de valores boleanos
4    * para marcar todos os slots visitados da tabela
5    * agregada na classe Timetablint.
6    *
7    */
8
9   package aistimetabling.tabela;
10
11  public class Lotacao {
12
13
14      private boolean[][] tab; //matriz auxiliar de marcação
15      private int ndias, naulas;
16
17      /** Cria uma nova Instância de Lotação */
18      public Lotacao(int naulas, int ndias) {
19          this.ndias = ndias;
20          this.naulas = naulas;
21          tab = new boolean[ndias][naulas];
22      }
23
24      //cria uma nova tabela
25      public void zeraTable(){
26          tab = null;
27          tab = new boolean[ndias][naulas];
```

```
28      }
29
30      /*
31       * Insere em um slot ( parametro "nslot")
32       * verdadeira, indicando que tal slot já
33       * foi visitado
34       */
35      public void inserir(int nSlot){
36          setar(nSlot, true);
37      }
38
39      /*
40       * Remove a marcação de visita do slot
41       */
42      public void remover(int nSlot){
43          setar(nSlot, false);
44      }
45
46      /*
47       * verifica se a tabela já foi totalmente visitada
48       * a partir dos valores de marcação
49       * booleano
50       */
51      public boolean lotado(){
52          for (int i=0; i<ndias; i++)
53              for (int j=0; j<naulas; j++)
54                  //se ja visitou este slot
55                  //continua a execucao
56                  if (tab[i][j]) continue;
57                  else return false;
58          return true;
59      }
60
61      /*
62       * Marca qual o estado (ocupado ou não)
63       * de um slot (nSlot).
64       */
65      private void setar(int nSlot, boolean ocupado){
66          int coluna = nSlot/naulas;
67          int linha = nSlot%naulas;
68          tab[coluna][linha] = ocupado;
69      }
70
71  }
```

A.12 Classe `Timetabling`

```
1   /*
2    * Classe Timetabling
3    *
4    * Essa classe implementa
5    * a especificação dos métodos
6    * definidos na interface Anticorpo
7    */
8
9   package aistimetabling.tabela;
10
11  import aistimetabling.Anticorpo;
12  import aistimetabling.Evento;
13  import aistimetabling.course.*
14  import java.util.*
15
16  public class Timetabling implements Anticorpo{
17
18      //atributo determinando a afinidade do anticorpo
19      private double afinidade,
20              mutation; //taxa de mutação do anticorpo
21
22      //matrix que armazena todas as aulas agendadas
23      public List<Aula> table[][];
24
```

```
25      //numAulas = períodos
26      //diasAulas = dias
27      public int numAulas, diasAulas,
28           nSlots;  // nº de slotes da tabela
29
30      //atributos carregados a partir do arquivo de
31      // especificação
32      public Sala[] vetSala;
33      public CursoRestricao[] vetCursoRestricao;
34      public Turma[] vetTurma;
35
36      //objeto gerador de números aleatório
37      private java.util.Random r;
38
39      // Cria uma nova instância da classe Timetabling
40      public Timetabling(int numAulas, int diasAulas,
41              Sala[] vetSala, CursoRestricao[] vetCursoRestricao,
42              Turma[] vetTurma, java.util.Random r) {
43          this.vetSala = vetSala;
44          this.numAulas = numAulas;
45          this.diasAulas = diasAulas;
46          this.nSlots = numAulas*diasAulas;
47          this.vetCursoRestricao = vetCursoRestricao;
48          this.vetTurma = vetTurma;
49          this.r = r;
50          //copia uma tabela em outra instância
51          table = new List[diasAulas][numAulas];
52          for (int i=0; i<diasAulas; i++)
53              for (int j=0; j<numAulas; j++)
54                  table[i][j] = new LinkedList();
55      }
56
57      // Cria uma nova instância da classe Timetabling
58      public Timetabling(int numAulas, int diasAulas, Sala[] vetSala,
59                List<Aula> tablePai[][], CursoRestricao[] vetCursoRestricao,
60                Turma[] vetTurma, java.util.Random r){
61          this.vetSala = vetSala;
62          this.numAulas = numAulas;
63          this.diasAulas = diasAulas;
64          this.nSlots = numAulas*diasAulas;
65          this.vetCursoRestricao = vetCursoRestricao;
66          this.vetTurma = vetTurma;
67          this.r = r;
68          //copia uma tabela em outra instância
69          table = new List[diasAulas][numAulas];
70          for (int i=0; i<diasAulas; i++)
71              for (int j=0; j<numAulas; j++)
72                  this.table[i][j] = new LinkedList(tablePai[i][j]);
73      }
74
75      /*
76       * Insere o gerador de números aleatórios
77       */
78      public void setRandomGenerator(java.util.Random r){
79          this.r = r;
80      }
81
82      /*
83       * Adiciona um Evento evt, ao repertório.
84       * Em outras palavras, adiciona um Curso (Evento)
85       * na tabela de quadros horários (obedecendo as restrições)
86       */
87      public void addRepertorio(Evento evt) {
88          Lotacao tabAux = new Lotacao(numAulas,diasAulas);
89          int nAleatorio;
90          //faça enquanto não conseguir adicionar um evento
91          do {
92              nAleatorio = r.nextInt(nSlots);
93              tabAux.inserir(nAleatorio);
94              if (tabAux.lotado()){  //se todos os slots foram
95                                     // consultados e não
96                                    //conseguiu inserir corretamente
97                  //troca os cursos de slot
98                  mutar();
```

```
99                  //zera a tabela auxiliar que indica lotação
100                 tabAux.zeraTable();
101             }
102         } while(!addEvent(nAleatorio, evt));
103     }
104
105     /*
106      * Adiciona Evento em um slot n;
107      * Antes, verifica se nenhuma restrição inviolável é desobedecida
108      * Retorna verdade se a inserção foi efetuada
109      */
110     public boolean addEvent(int n, Evento evt){
111         int dia = n/numAulas;
112         int periodo = n%numAulas;
113         Aula aula = new Aula((Curso)evt,vetSala[r.nextInt(vetSala.length)]);
114         List<Aula> slot = getSlot(n);
115         //verifica restrições invioláveis
116         if (!Restricoes.mesmaTurma(aula,slot,this.vetTurma) &&
117                 !Restricoes.conflitoSala(aula,slot) &&
118                 !Restricoes.conflitoProfessor(aula,slot) &&
119                 !Restricoes.conflitoCurso(aula,slot) &&
120                 !Restricoes.conflitoTurma(aula,vetCursoRestricao,
121                       dia,periodo)){
122             slot.add(aula);  //adiciona a aula na tabela
123             return true;
124         }
125         return false;
126     }
127
128     /*
129      * Retora um slot n da tabela.
130      */
131     public List<Aula> getSlot(int n){
132         int coluna = n/numAulas;
133         int linha = n%numAulas;
134         try{
135             return table[coluna][linha];
136         }catch(ArrayIndexOutOfBoundsException e){
137             e.printStackTrace();
138         }
139         return null;
140     }
141
142     public void setAfinidade(double afinidade) {
143         this.afinidade = afinidade;
144     }
145
146     public void setTaxaMutacao(double mutation) {
147         this.mutation = mutation;
148     }
149
150     public double getTaxaMutacao() {
151         return this.mutation;
152     }
153
154     public double getAfinidade() {
155         return this.afinidade;
156     }
157
158     /*
159      * Método que faz as trocas de cursos:
160      * * Remove um Evento escolhido da aleatoriamente da tabela
161      * * Adiciona um evento em um slot escolhido aleatoriamente
162      */
163     public void mutar() {
164         //remove Evento Aleatoriamente
165         int slotAleatorio = 0;
166         //faça enquando o slot aleatório possui pelo menos 1
167         // exame
168         do {
169             slotAleatorio = r.nextInt(nSlots);
170         } while(!temExame(slotAleatorio));
171         //retorna slot
172         List<Aula> slot = getSlot(slotAleatorio);
```

```
173             int aulaAleatorio = r.nextInt(slot.size());
174             //retorna uma aula do slot (aleatoriamente)
175             Aula aula = slot.get(aulaAleatorio);
176             // remove a aula do quadro horário
177             removeEvent(slotAleatorio,aula);
178 
179             //adiciona Evento Aleatoriamente
180             addRepertorio(aula.getCurso());
181         }
182 
183         //Remove um Evento (Curso) de um slot especificado nSlot
184         private void removeEvent(int nSlot, Aula aula){
185             int linha = nSlot/numAulas;
186             int coluna = nSlot%numAulas;
187             table[linha][coluna].remove(aula);
188         }
189 
190         /*
191          * Cria uma nova instância (clone)
192          * da classe timetabling (anticorpo)
193          */
194         public Anticorpo clonar(){
195             Timetabling clone = new Timetabling(numAulas,diasAulas,vetSala,
196                  table, this.vetCursoRestricao, this.vetTurma,this.r);
197             clone.setAfinidade(this.getAfinidade());
198             clone.setTaxaMutacao(this.getTaxaMutacao());
199             return clone;
200         }
201 
202         //verdadeiro se no slot tem pelo menos 1 exame
203         public boolean temExame(int nSlot){
204             List<Aula> slot = getSlot(nSlot);
205             if (slot.size() == 0)
206                 return false;
207             return true;
208         }
209 
210         /*
211          * Verifica se duas tabelas (timetabling) são iguais
212          * São iguais, se em cada slot, foram agendadas os mesmos cursos
213          */
214         public boolean equals(Object obj){
215             if (!(obj instanceof Timetabling))
216                 return false;
217             Timetabling timetabling = (Timetabling)obj;
218             for(int i=0; i< timetabling.diasAulas; i++){
219                 for(int j=0; j<timetabling.numAulas; j++){
220                     if (!(table[i][j].equals(timetabling.table[i][j]))){
221                         return false;
222                     }
223                 }
224             }
225             return true;
226         }
227 
228     }
```