Nataniel Pereira Borges Junior

Framework para execução distribuída de algoritmos genéticos utilizando Hadoop

Florianopolis

2014

Nataniel Pereira Borges Junior

Framework para execução distribuída de algoritmos genéticos utilizando Hadoop

Trabalho apresentado ao Curso de Sistemas de Informação da Universidade Federal de Santa Catarina como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Universidade Federal de Santa Catarina – UFSC

Centro Tecnológico – CTC

Departamento de Informática e Estatística – INE

Orientador: Prof. Ricardo Pereira e Silva, Dr.

Florianopolis

2014

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Borges Junior, Nataniel Pereira
Framework para execução distribuída de algoritmos genéticos utilizando Hadoop / Nataniel Pereira Borges Junior ; orientador, Ricardo Pereira e Silva - Florianópolis, SC, 2014.
108 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico. Graduação em Sistema de Informação.

Inclui referências

1. Sistema de Informação. 2. Algoritmos genéticos. 3. Frameworks. 4. MapReduce. 5. Hadoop. I. Silva, Ricardo Pereira e. II. Universidade Federal de Santa Catarina. Graduação em Sistema de Informação. III. Título.

Nataniel Pereira Borges Junior

Framework para execução distribuída de algoritmos genéticos utilizando Hadoop

Trabalho apresentado ao Curso de Sistemas de Informação da Universidade Federal de Santa Catarina como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Florianopolis, 2 de dezembro de 2014:

Prof. Ricardo Pereira e Silva, Dr.
Orientador

Prof. Luciana de Oliveira Rech, Dra.
Universidade Federal de Santa Catarina

Prof. Patrícia Vilain, Dra.
Universidade Federal de Santa Catarina

Florianopolis
2014

RESUMO

Algoritmos genéticos são mecanismos de otimização genéricos que possuem arquitetura e fluxo de controle estáticos. Eles são utilizados em áreas como *machine learning* e *design optimization*. *Frameworks* orientados a objetos são uma abordagem consolidada para reuso de arquitetura e fluxo de controle de aplicações, sendo seu principal objetivo o aumento da velocidade e qualidade do processo de desenvolvimento de um software. Atualmente ambientes multicomputadores, como *clusters* e *grids*, encontram-se disponíveis a baixo custo. Este trabalho apresenta um *framework* orientado a objetos que possibilita o desenvolvimento de algoritmos genéticos com população variável em ambientes multicomputadores através do modelo de programação MapReduce, avaliando o reuso proporcionado.

Palavras-chaves: algoritmo genético. framework. hadoop. map reduce.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACO	Otimização baseada em colônia de formigas (*Ant Colony Optimization*)
APGA	Algoritmo genético com população de tamanho adaptável proposto por Bäck et al. (2000) (*Genetic Algorithm with Adaptive Population Size*)
DGA	Algoritmo genético distribuído (*Distributed Genetic Algorithm*)
GA	Algoritmo genético (*Genetic Algorithm*)
GAvaPS	Algoritmo genético com população variável proposto por Arabas et al. (1994) (*Genetic Algorithm with Variable Population Size)*)
GUI	Interface gráfica do utilizador (*Graphical user interface*)
OO	Orientadas a objeto (*Object-oriented*)
PLGA	Algoritmo genético sem parametros proposto por Harik e Lobo (1999) (*Parameter-less Genetic Algorithm*)
PSO	Otimização por enxame de partículas (*Particle Swarm Optimization*)
SGA	Algoritmos genéticos simples (*Simple Genetic Algorithm*)
SGA-SCM	Algoritmos genéticos simples com operações realizadas na seguinte ordem: seleção, reprodução e mutação (*Simple Genetic Algorithm - Selection, Crossover e Mutation*)
SO	Sistema operacional
UML	Linguagem de modelagem unificada (*Unified Modeling Language*)

1 INTRODUÇÃO

Recentemente a Ciência da Computação tem se beneficiado da inteligência computacional onde unem-se normalmente as áreas tecnologia, biologia e medicina. Dentre as interações entre essas áreas foi formulada uma família de modelos computacionais inspirados pelo processo evolutivo dos seres biológicos para serem utilizados como técnica de busca. Essa família de modelos, e consequentemente a técnica de busca resultante de sua aplicação, são conhecidos como algoritmos genéticos (*genetic algorithms* - GAs) (WHITLEY, 1994, p. 65).

Diversos métodos de otimização foram desenvolvidos nas áreas de matemática e pesquisa operacional, entretanto esses métodos são especializados, ou seja, desenvolvidos para a solução de uma família específica de problemas. De maneira sintética[1] GAs são conhecidos como mecanismos de busca genéricos que não utilizam técnicas baseadas em gradiente (KEARNEY et al., 1987) tornando possível a solução de, por exemplo, funções não diferenciáveis[2]. Entretanto GAs são métodos não determinísticos e pouco eficientes, ou seja, caso exista um algoritmo especializado para a solução de um problema certamente este será mais eficiente e eficaz do que um algoritmo genético (WHITLEY, 1994, p. 68).

Segundo Chuang e Wu (2000, p. 269) GAs diferem dos mecanismos de otimização tradicionais em diversos aspectos. GAs utilizam mecanismos probabilísticos na escolha de uma solução em vez de mecanismos determinísticos. A busca é realizada sobre uma população de pontos no espaço de busca em vez de um único ponto inicial como em técnicas tradicionais de gradiente. Além disso é necessário pouco conhecimento sobre a função objetiva do problema, informações como continuidade e diferenciabilidade, por exemplo, são desnecessárias.

O conceito que inspirou a criação de GAs foi o princípio da evolução das espécies de Darwin onde dada uma população de indivíduos, pressões externas, como a disponibilidade limitada de alimentos e parceiros para reprodução, implicam na seleção natural (sobrevivência do mais aptos) fazendo com que os indivíduos mais aptos se reproduzam e propagem seus genes para as novas gerações fazendo com que indivíduos cada vez melhores sejam criados (EIBEN; SCHOENAUER, 2002, p. 1).

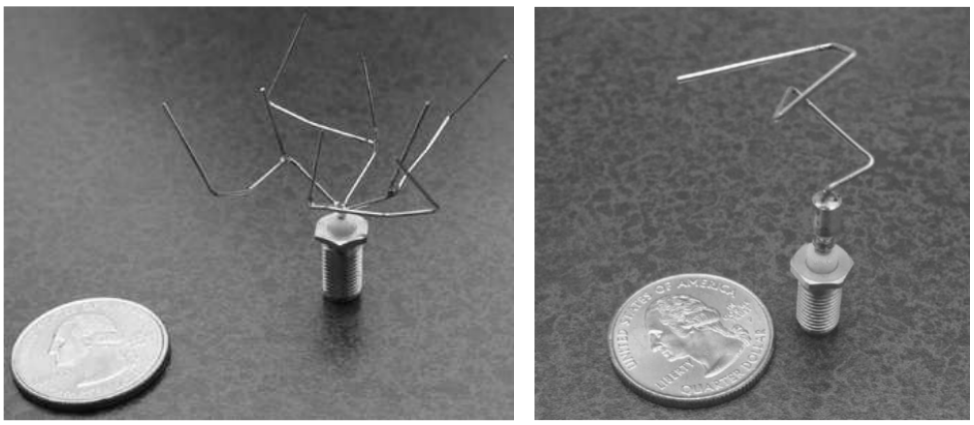
Atualmente GAs são utilizados na solução de problemas em diversas áreas. Nas engenharias foram utilizados na automatização de processos de design automatizado de equipamentos como apresentado em (SANTARELLI et al., 2006) e (BOLCHINI et al., 2010). Além de sua utilização no projeto de equipamentos, GAs foram aplicados na solução

[1] Definições mais específicas são apresentadas na seção 2.1.
[2] Funções onde os limites laterais de um ponto são divergentes.

de problemas diversos como os referentes a transmissão de energia elétrica como transmissão econômica (BAKIRTZIS et al., 1994), (SHEBLE; BRITTIG, 1995), planejamento de expansão de rede (MIRANDA et al., 1994), fluxo de carga (YIN; GERMAY, 1991), (YIN, 1993), seleção de capacitores ideias (BOONE; CHIANG, 1993), (SUNDHARARAJAN; PAHWA, 1994), diagnóstico de falhas (WEN; HAN, 1995), problemas classificados como *NP-hard*[3].

Uma das aplicações mais famosas da GA é a apresentada na Figura 1, utilizadas com sucesso no projeto *Space Technology 5* (ST5) da NASA (HORNBY et al., 2006, p. 4-5) onde o design da antena foi projetado através da utilização de um GA cujo objetivo era maximizar a recepção do sinal. Segundo os autores, antes de utilizarem um GA para o projeto da antena, empresas especializadas foram contratadas, entretanto os projetos apresentados não apresentaram os resultados esperados. O GA deste trabalho realizou um processo de otimização baseado em quatro operações: avançar, para adicionar um segmento reto a uma parte da antena, rotação-x, rotação-y e rotação-z, para rotacionar a parte da antena em uma dimensão fazendo com que o próximo segmento reto fique inclinado ao anterior, combinando-os repetidamente de forma a maximizar uma função desenvolvida pelos autores que considera, com diferentes pesos, requisitos como *gain pattern*, relações de onda, peso e dimensões.

Figura 1 – Protótipos de antenas projetadas com GAs para o projeto *Space Technology 5* (ST5) da NASA.



(a) Antena ST5-3-10 (b) Antena ST5-33-142-7

Fonte: (HORNBY et al., 2006, p. 5)

Outras áreas além das engenharias beneficiaram-se de algoritmos de GAs, na biologia e na química GAs são utilizados na identificação de parâmetros visando a otimização de reações como apresentado em (ANGELOVA et al., 2011) em um processo de fermen-

[3] NP-Hard, também conhecido como NP-Difícil ou NP-Complexo, significa *Non-deterministic polynomial time*, uma classificação definida na teoria da computação para agrupar problemas complexos.

tação de leveduras. Ao problema de inversão sísmica[4] estudado na geologia foi proposta uma solução baseada exclusivamente em GAs em (MONTESINOS et al., 2005). Estudos em outras áreas como administração, música e telecomunicações foram realizados respectivamente em (ROSÁRIO, 2011), (PAPADOPOULOS; WIGGINS, 1998) e (MEUNIER et al., 2000).

Embora GAs possam ser aplicados a diversas áreas eles são algoritmos que possuem estrutura e fluxo de execução fixos[5] o que os torna propícios para a aplicação de uma abordagem voltada ao reuso como *frameworks* orientados a objeto visando o aumento da produtividade dos desenvolvedores (MARKIEWICZ; LUCENA, 2001, p. 1). Além disso, embora GAs sejam ineficientes eles são altamente passíveis de paralelização[6], podendo ser executados em ambientes paralelos (como GPUs) ou distribuídos (como *clusters* e *grids* computacionais) através de abordagens como o modelo de programação *MapReduce*, tornando sua performance escalável.

1.1 OBJETIVOS

1.1.1 Objetivo geral

O presente trabalho tem como objetivo geral o projeto e implementação de um *framework* orientado a objetos para a execução distribuída de GAs com população variável.

1.1.2 Objetivos específicos

A partir do objetivo geral estabelecido foram definidos os seguintes objetivos específicos para o presente trabalho:

- Projetar um *framework* para execução distribuída de algoritmos genéticos com população variável e MapReduce, utilizando UML2;
- Implementar o *framework* projetado utilizando a linguagem Python;
- Implementar um *framework* utilizando o modelo MapReduce;
- Implementar um *framework* de domínio utilizando um *framework* de suporte;
- Avaliar o *framework* por meio do desenvolvimento de três aplicações.

[4] O problema da inversão sísmica consiste na determinação da estrutura dos dados de subsolo a partir da prospecção geológica, tendo como objetivo primário obter uma seção geológica ou um modelo 3D (LINDEN, 2006).

[5] Essas características são apresentadas na seção 2.1.

[6] Seu paralelismo é abordado na seção 2.1.

1.2 JUSTIFICATIVA

Atualmente, diante do aumento dos recursos computacionais disponíveis, é possível tratar diversos problemas com uma quantidade maior de detalhes, resultando por sua vez em abstrações mais fiéis das situações existentes no mundo real. Entretanto, essas melhores abstrações ampliam a quantidade de elementos existentes no espaço de soluções candidatas de um problema, tornando sua otimização um processo que consome cada vez mais recursos computacionais.

Devido à abordagem utilizada na solução de um problema, algoritmos genéticos são altamente paralelizáveis podendo fazer uso de ambientes multi-computados, como *clusters* e *grids* computacionais, que hoje encontram-se disponíveis a baixo custo. Entretanto modelos de programação distribuída como *MapReduce* não são amplamente dominados pelos desenvolvedores, resultando no aumento do custo de desenvolvimento e na utilização de soluções ineficientes.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado da seguinte forma: na seção 2.1 são apresentados algoritmos genéticos e são abordados seus principais conceitos, modelos formais, utilização de população variável, além de apresentadas as principais operações de seleção, mutação e *crossover*.

A seção 2.2 realiza a uma revisão bibliográfica sobre *frameworks* orientados a objetos, onde é realizada uma comparação entre abordagens voltadas ao reuso de *software* e onde são mostrados os conceitos e classificações de *framework* orientados a objetos além de sua relação com padrões de projeto.

O modelo de programação distribuída *MapReduce* é apresentado na seção 2.3, são abordados seus principais conceitos além de sua implementação no *framework* Hadoop. O Capítulo 3 apresenta o framework desenvolvido e o Capítulo 4 apresenta as aplicações de avaliação implementadas. Por último, no Capítulo 5 são apresentadas as conclusões deste trabalho.

2 REVISÃO BIBLIOGRÁFICA

2.1 ALGORITMOS GENÉTICOS

Essa seção apresenta a fundamentação teórica para o funcionamento de GAs. Na subseção 2.1.1 são abordados os conceitos básicos sobre algoritmos genéticos, suas principais operações e elementos. Os modelos formais que embasam o funcionamento de GAs são apresentados na subseção 2.1.2, enquanto os principais mecanismos de seleção, *crossover* (reprodução) e mutação são apresentados, em sua ordem padrão denotada por SGA-SCM[1], respectivamente nas subseções 2.1.3, 2.1.4 e 2.1.5. Por último são apresentadas variações do algoritmo para o funcionamento com populações variáveis (subseção 2.1.6).

2.1.1 Conceitos básicos

GAs são "técnicas de busca estocásticas[2] baseadas em mecanismos de seleção natural e genética" (GUO et al., 2010, p. 2990), visão que é compartilhada por diversos autores como Beasley et al. (1993a, p. 2), Shi et al. (2005, p. 255), Escuela et al. (2007, p. 437) e Munawar et al. (2008, p. 897).

Segundo Fogel (1994, p. 3) os princípios da teoria da evolução de Darwin são mecanismos robustos de busca e otimização onde cada bioma evoluído demonstra um comportamento complexo e otimizado. Os problemas que as espécies solucionaram durante o seu processo evolutivo envolvem fatores como caos, sorte e temporalidade que são, por sua vez, características de problemas que se mostraram intratáveis através dos mecanismos de otimização tradicionais.

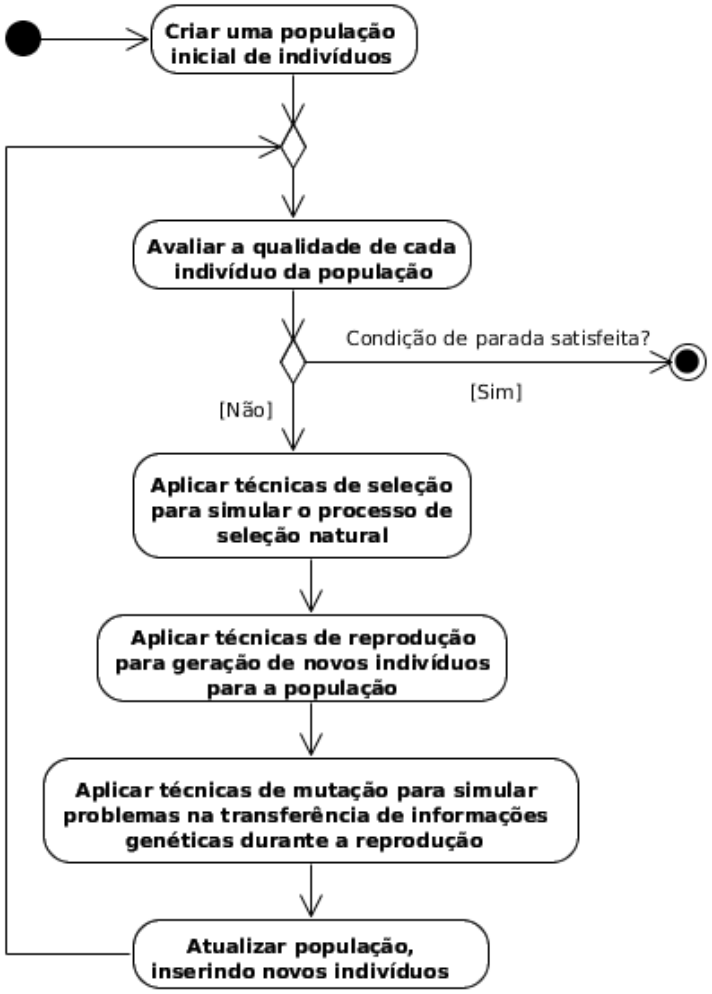
De maneira abstrata Beasley et al. (1993a, p. 58) definem a evolução das espécies na natureza como um processo de competição, onde os indivíduos disputam recursos como alimentos, água e parceiros. Os indivíduos mais adaptados sobreviverão e se reproduzirão, gerando uma quantidade maior de descendentes, enquanto indivíduos menos aptos tendem a gerar quantidades menores de descendentes, Fogel (1994, p. 3) ressalta que sem essa competição, se todos os indivíduos conseguissem se reproduzir, haveria um crescimento exponencial da população. Esse processo de seleção favorece a propagação dos genes dos pais mais aptos para seus filhos e é repetido a cada nova geração. A propagação e combinação de diferentes genes "bons" possui boa probabilidade de ocasionar o surgimento

[1] O padrão de nomenclatura adotado é o proposto em Roeva et al. (2012, p. 178), decorrente da ordem de execução das operações de seleção, *crossover* e mutação, os autores citam algoritmos alternativos que realizam as operações em outras ordens como SGA-MSC (mutação, seleção e *crossover*) e SGA-CMS (*crossover*, mutação e seleção).

[2] Não determinísticos, sujeito a eventos aleatórios.

de indivíduos mais aptos que qualquer um dos seus ancestrais (FORREST, 1993, p. 2), embora ocorram casos onde são gerados indivíduos com pouca ou nenhuma aptidão que serão extintos em algumas gerações. Além disso durante o processo de transferência de informações genéticas entre as gerações (*crossover*) podem ocorrer falhas, essas falhas são chamadas de mutações e são uma importante ferramenta no processo de evolução. Todo esse processo é traduzido em operações cujo sequenciamento é descrito na Figura 2. Nas etapas desse algoritmo diferentes técnicas de seleção natural, reprodução (*crossover*) e mutação podem ser utilizadas, essas técnicas são apresentadas nas próximas seções.

Figura 2 – Fluxo de execução de algoritmos genéticos.



GAs operam sobre um conjunto de indivíduos chamados de cromossomos que representam soluções-candidatas a um problema. Fogel (1994, p. 3) trata indivíduos como a dualidade entre seu genótipo e fenótipo, onde o genótipo é o local de armazenamento dos genes recebidos dos antepassados[3], e fenótipo é a representação de seu código genético em um ambiente, sendo possível que estruturas genéticas completamente diferentes produzam resultados semelhantes.

[3] Normalmente o armazenamento dessas informações ocorre através de strings binárias.

Ainda segundo o autor, o efeito que uma alteração no código genético de um indivíduo terá em seu fenótipo é, normalmente, imprevisível devido a existência de *pleiotropy*, onde a alteração de um único gene afeta diversos traços do fenótipo, e *polygeny*, onde a alteração de um único traço do fenótipo depende da alteração de diversos genes. Alguns autores, como Beasley et al. (1993a), Whitley (1994) e Verel (2013), resumem essas características como *epistasis*. Beasley et al. (1993a) ressalta inclusive que em funções multimodais sempre existe alguma interação entre os genes.

Além do fenótipo e do genótipo os cromossomos possuem um *fitness*, o *fitness* é uma nota de avaliação do cromossomo baseada nos valores de seu fenótipo no problema a ser solucionado. A função de avaliação por sua vez é um dos itens que varia conforme o problema a ser solucionado pelo GA[4]. Normalmente o *fitness* é representado como um valor real positivo e é proporcional a qualidade do cromossomo, ou seja, quanto melhor for uma solução candidata maior o valor de seu *fitness*, entretanto em problemas de minimização é possível que um valor menor represente um indivíduo melhor. Em um problema de maximização de lucro por exemplo o valor de *fitness* poderia ser o lucro obtido (quanto maior o lucro, melhor o *fitness)* enquanto em uma comparação de curvas o valor do *fitness* poderia ser a distância de Hausdorff (HUTTENLOCHER et al., 1993, p. 850) entre uma curva base e uma curva candidata (nesse caso quanto menor a distância melhor o fitness).

Considerando que deseje-se resolver um problema de regressão linear (localizar a equação de reta que melhor se ajusta a um conjunto de pontos), que a Figura 3 represente uma solução candidata nesse problema e que o genótipo apresentado na mesma esteja codificado na forma binária. Neste caso os genes 1 e 2 seriam traduzidos, respectivamente, como os valores 2653 e 25345 e o genótipo do cromossomo seria representado por uma reta definida pela equação $Y = 2653X + 25345$. Neste problema o valor do *fitness* do cromossomo pode ser considerado o somatório dos erros[5], onde um valor menor de *fitness* representaria uma melhor solução candidata.

Figura 3 – Cromossomo composto por genes binários.

	Gene 01																Gene 02															
Genótipo	0	0	0	0	1	0	1	0	0	1	0	1	1	1	0	1	0	1	1	0	0	0	1	1	0	0	0	0	0	0	0	1
Genótipo (dec)	2653																25345															
Fenótipo	Y = 2653X + 25345																															

População é o conjunto de cromossomos manipulados por um GA durante uma ite-

[4] A função de avaliação calcula o *fitness* de cada cromossomo da população de forma independente, sendo esse o principal ponto de paralelização do algoritmo em ambientes como *clusters* e *grids* computacionais.

[5] Distâncias cartesianas entre o valor predito e o valor de cada ponto.

ração. É normalmente constante[6] e inicialmente gerada de maneira aleatória[7](WHITLEY, 1994, p. 65). O procedimento tradicional de inicialização da população de cromossomos de um GA ocorre criando aleatoriamente cromossomos, verificando se os mesmos já existem[8] na população (visando garantir a variabilidade da população inicial) e inserindo-os caso não estejam, procedimento o qual é descrito na Figura 4. Utilizando o exemplo da regressão linear apresentado anteriormente e considerando o tamanho da população inicial como 14 indivíduos, a Figura 5 seria a representação de uma população inicial válida para esse problema.

Figura 4 – Procedimento de geração da população inicial de um GA.

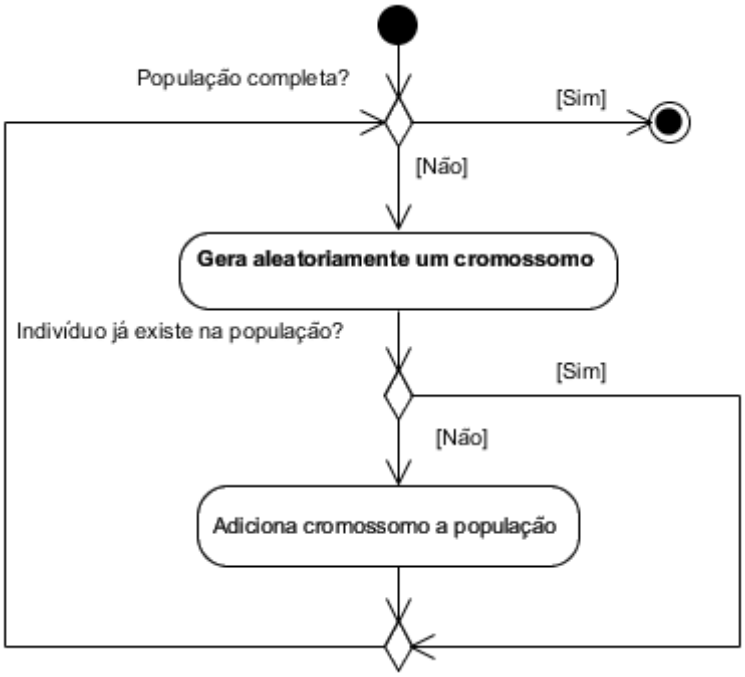


Figura 5 – População inicial para o problema de regressão linear.

	Genótipo																														Fenótipo
Cromossomo 1	0	0	0	0	1	0	1	0	0	1	0	1	1	1	0	1	0	1	1	0	0	0	1	1	0	0	0	0	0	0	1	Y = 2653X + 25345
Cromossomo 2	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	Y = 2560X + 24576
Cromossomo 3	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Y = 10240X + 49152
Cromossomo 4	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	Y = 40960X + 1537
Cromossomo 5	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	1	0	0	0	Y = 10240X + 1
Cromossomo 6	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	Y = 640X + 24
Cromossomo 7	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	Y = 10240X + 1
Cromossomo 8	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	Y = 320X + 96
Cromossomo 9	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	Y = 20480X + 1536
Cromossomo 10	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	Y = 80X + 1537
Cromossomo 11	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	Y = 640X + 24
Cromossomo 12	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	Y = 40X + 1
Cromossomo 13	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	Y = 640X + 25
Cromossomo 14	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	Y = 32X + 3

[6] Na subseção 2.1.6 são apresentados, dentre outros, o modelo proposto por Arabas et al. (1994) onde é introduzido o conceito de idade e o trabalho com populações variáveis e uma adaptação realizada por Bäck et al. (2000)

[7] Existem estudos, como o de Diaz-Gomez e Hougen (2007, p. 43), que demonstram que a qualidade da população inicial afeta a qualidade do resultado da execução do algoritmo.

[8] Cromossomos são considerados iguais quando seu material genético é o mesmo.

Uma das decisões críticas no processo de criação de um GA é o tamanho da população inicial. Grefenstette (1986 apud PETIT; SWIGGER, 1983) e Lobo e Lima (2006, p. 1241) afirmam que GAs podem convergir para soluções com pouca qualidade caso a população inicial seja pequena, entretanto caso a população inicial seja grande o algoritmo tornar-se-á inaceitavelmente lento. Ainda segundo os autores, estudos apontam que o tamanho da população ideal varia conforme o problema a ser resolvido, não havendo um valor padrão.

2.1.2 Definições e modelos formais

Segundo Forrest (1993, p. 4) GAs são normalmente aplicados a situações complexas, sujeitas a alterações durante o tempo ou com ruídos, características que tornam virtualmente impossível predizer sua performance. Entretanto existem teorias que embasam seu funcionamento, sendo as principais a teoria dos esquemas (*schemata theory*) que é apresentada na subseção 2.1.2.1, e a teoria dos blocos de construção (*building blocks*) que é apresentada na subseção 2.1.2.2, descritas respectivamente em (HOLLAND, 1975) e (GOLDBERG; HOLLAND, 1988).

A análise de GAs inicia-se pelo conceito de um espaço de busca, podendo um GA ser considerado um procedimento de busca de uma *string* binária de comprimento l com *fitness* alto em um conjunto de *strings* binárias de comprimento l (FORREST, 1993, p. 4). Na literatura, como em Guo et al. (2010, p. 4) e Verel (2013, p. 4), esse espaço de busca é frequentemente descrito como *fitness landscape* tratando indivíduos com maior *fitness* como elevações no terreno e um GA como um mecanismo de localização de picos (para comparação com técnicas de gradiente), esta visão está representada na Figura 6 onde tem-se o valor de uma variável Y determinado pelo valor das variáveis X e Z. No caso do problema de regressão linear apresentado anteriormente pode-se considerar a variável Y como o somatório dos erros, a variável X como o valor a e a variável Z como b (definindo uma equação de reta como $Y = aX + b$).

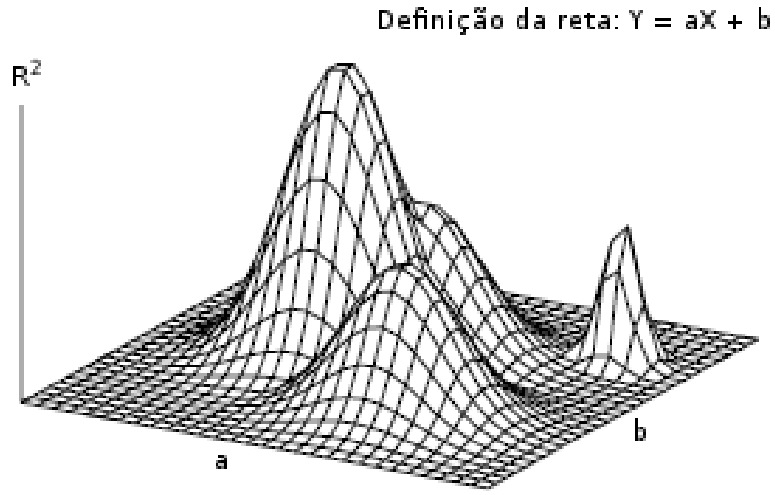
2.1.2.1 Teoria dos esquemas e teorema fundamental dos GAs

A teoria dos esquemas introduz o conceito de esquema para explicar formalmente porque GAs funcionam. Srinivas e Patnaik (1994b, p. 20) definem um esquema como modelo de similaridade definido sobre um alfabeto $\{0, 1, *^{9}\}$ com o mesmo comprimento l das representações binárias do terreno, onde cada esquema descreve um subconjunto de *strings* com valores idênticos em uma posição específica[10]. Ainda segundo os autores, o

[9] Alguns autores como Booker et al. (1989, p. 7) utilizam o símbolo # ou a expressão "ignorado" (*don't care*) para essa representação.

[10] Alguns autores como Forrest (1993, p. 5) e Whitley (1994, p. 4) chamam esses subconjuntos do espaço de busca de hiperplanos.

Figura 6 – Representação gráfica de um *fitness landscape*



Fonte: Adaptado de (VEREL, 2013, p. 8).

caractere * é definido como uma posição de valor livre, aceitando tanto o valor 0 quanto 1.

Cada *string* representada por um esquema é chamada de instância do esquema sendo que um único cromossomo está contido em até 2^l esquemas, característica que é conhecida como paralelismo implícito. Este paralelismo possibilita a avaliação de múltiplos esquemas simultaneamente encontrando-se ilustrado na Figura 7 onde um cromossomo de comprimento 3 ("101") está contido em 7 esquemas diferentes, ou seja a avaliação de seu *fitness* implicaria, implicitamente, na avaliação do *fitness* de 7 esquemas. Ainda conforme essa teoria, um esquema composto por $\{*\}^l$ define todas as *strings* binárias de tamanho l, as posições fixas do esquema representam a sua ordem e o comprimento de um esquema é definido pela distância entre as posições fixas em seus extremos, a critério de exemplo um esquema definido como *1*00* possuiria um comprimento de 3.

Whitley (1994) propõe que a visualização dos hiperplanos gerados pelos esquemas seja feita como um hipercubo[11], semelhante ao apresentado na Figura 8 onde, no exemplo apresentado, cada extremidade refere-se a uma *string* binária completa, cada superfície refere-se a um esquema de ordem 1 e cada aresta representa um esquema de ordem 2. Nesta imagem, ao avaliar-se o cromossomo "101" da Figura 7 são avaliados os esquemas de ordem 1: *0* (verde), **1 (azul) e 1** (vermelho) e os esquemas de ordem 2: $\overrightarrow{001\ 101}$, $\overrightarrow{100\ 101}$ e $\overrightarrow{111\ 101}$ (arestas laranja).

Sob a ótica dos esquemas pode-se interpretar, segundo Forrest (1993, p. 4), o mecanismo de busca de um GAs como um processo implícito de amostragem de esquemas onde o retorno (*feedback*) da função de *fitness*, combinado com métodos de seleção, re-

[11] Hipercubo é um análogo N-dimensional a um cubo cuja dimensão é 3

Figura 7 – Representação dos esquemas contidos em um cromossomo.

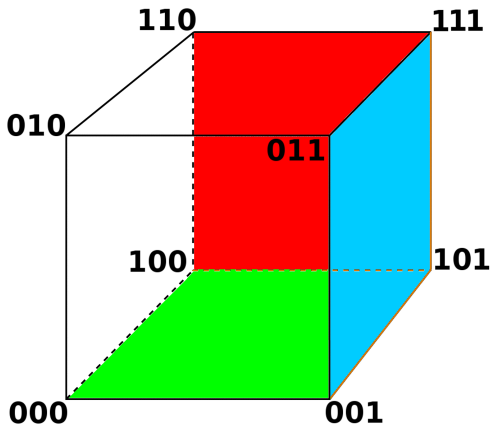
Genótipo do cromossomo

Cromossomo A = 101

Esquemas ao qual o cromossomo pertence

Esquema 0	=	***
Esquema 1	=	1**
Esquema 2	=	*0*
Esquema 3	=	**1
Esquema $1 \cap 2$	=	10*
Esquema $1 \cap 3$	=	1*1
Esquema $2 \cap 3$	=	*01

Figura 8 – Representação dos hiperplanos gerados pelos esquemas como cubo.



combinação (*crossover*) e distorções no processo de amostragem (mutação) direcionam, ao longo do tempo, o algoritmo para longe dos esquemas que possuem um baixo *fitness* médio, focando nos que possuem um *fitness* médio mais elevado. Srinivas e Patnaik (1994b, p. 20) entretanto analisam que o funcionamento de um GA ocorre devido à competição existente entre os esquemas dos cromossomos para se tornar o esquema predominante da população.

2.1.2.2 Teoria dos blocos

A teoria dos blocos de construção (*building blocks*) caracteriza a importância de uma boa codificação do problema para o código do cromossomo[12], ela, segundo Srinivas e Patnaik (1994b, p. 21), afirma que *strings* são compostas por blocos e que as operações genéticas realizadas por um GA como mutação, *crossover* e seleção geram, promovem e justapõem esses blocos buscando a criação de uma *string* otimizada. Beyer e Schwefel

[12] Uma boa codificação, segundo Goldberg e Holland (1988, p. 95) é uma codificação onde genes relacionado fiquem proximos (na composição do genótipo), além disso recomenda-se a utilização de codificações como Gray Code (MEHTA et al., 1996, p. 178-179) onde números consecutivos são diferenciados por somente um caracter.

(2002), de maneira sintética, afirmam que essa teoria baseia-se na premissa de que a combinação e concatenação de blocos bons[13] de pais diferentes com *fitness* elevado produzirá um resultado com *fitness* elevado, reproduzindo a ideia intuitiva de que a combinação das características boas dos pais resultará em um filho também bom[14].

Srinivas e Patnaik (1994b, p. 21) relacionam as principais operações realizadas por um GA com a sua influência nos blocos que compõe a *string*. Segundo os autores o *crossover* tende a manter as informações das *strings* à serem combinadas o que faz com que a criação de novos blocos se torne difícil quando as mesmas são semelhantes. Ao contrário das operações de *crossover* as operações de mutação não conservam blocos, sendo utilizadas para a introdução de novos blocos nos cromossomos, enquanto as operações de seleção servem de guia para a construção de blocos com maior *fitness* a cada geração. Baseando-se nos conceitos dessa teoria, Beasley et al. (1993a, p. 7) afirma que GAs devem ser projetados visando aproximar genes relacionados uma vez que a justaposição de blocos é uma operação importante para o funcionamento do mesmo.

2.1.3 Operadores de seleção

Beyer e Schwefel (2002, p. 11) afirmam que GAs necessitam de um objetivo que guie a busca para regiões mais promissoras do *fitness landscape*, da mesma forma que o processo de seleção natural guia a evolução das espécies, esse guia é definido através de operadores de seleção. Miller e Goldberg (1995, p. 195) e posteriormente Sivaraj e Ravichandran (2011, p. 3793) definem que os operadores de seleção devem dar preferência aos melhores indivíduos (com *fitness* mais elevado) possibilitando que seus genes sejam passados para as gerações futuras, proibindo (ou limitando) dessa forma a entrada de indivíduos ruins nas próximas gerações. Ainda segundo os autores a pressão seletiva (*selective pressure*)[15] é um parâmetro que influencia a performance de um GA, estando baixa a convergência para uma solução otimizada se torna lenta, estando alta ocasiona perda de diversidade da população fazendo com que o GA venha a convergir muito rápido, frequentemente para um máximo local. Miller e Goldberg (1995, p. 194) caracterizam o mecanismo de seleção ideal como: simples de implementar, eficiente (em arquiteturas paralelas ou não) e com uma pressão seletiva passivel de configuração.

Além das características já citadas Grefenstette (1986, p. 3), introduziu um novo conceito ao processo de seleção de GAs: o *generation gap*. Este conceito define o percentual de indivíduos da população a serem substituídos a cada geração, simulando assim duas características existentes na natureza: a não substituição de todos os indivíduos de uma

[13] Blocos são considerados bons quando produzem pais com *fitness* elevado.

[14] Segundo Beyer e Schwefel (2002, p. 19) essa hipótese, apesar de intuitiva, se mostrou extremamente difícil de ser demonstrada na prática, sendo que apenas recentemente foram apresentados, por Jansen e Wegener (2005), resultados concretos sobre sua importância.

[15] A pressão seletiva é um parâmetro das técnicas de seleção que são apresentadas a seguir, seu valor ideal varia conforme o problema a ser resolvido não havendo um mecanismo de calculo.

população simultaneamente e a competição entre indivíduos de diferentes gerações (pais e filhos) pelos recursos.

Nas próximas seções são apresentados alguns operadores de seleção clássicos sendo a seleção por truncamento (*Truncation* ou *Breeding*), na subseção 2.1.3.1, a seleção por roleta (*Roulette Wheel*), na subseção 2.1.3.2, a seleção por amostragem universal estocástica (*Stochastisc Universal Samplig*), na subseção 2.1.3.3, a seleção por ranking, na subseção 2.1.3.4, e a seleção por torneiro, na subseção 2.1.3.5. Para a apresentação desses operadores devem ser formalizados alguns conceitos, para tal serão utilizadas as definições formais apresentadas por Blickle e Thiele (1995, p. 6-13) onde:

- **Indivíduo**: Um indivíduo em um GA é denotado J_i sendo que $J_i \in J$ onde J é o conjunto de todos os indivíduos possíveis;
- **Tamanho da população**: A quantidade de soluções candidatas (indivíduos) é denotada N;
- ***Filhos***: O conjunto de filhos gerados após o processo de *crossover* é denotada por γ;
- **Operação de seleção**: Uma operação de seleção é definida como uma função $\omega : J^N \mapsto J^N$;
- **Fluxo de execução**: Uma vez inicializado o GA, as operações de seleção, mutação e de *crossover* são executadas repetidamente até que uma condição de parada pré-estabelecida seja satisfeita;
- ***Fitness***: A qualidade de um indivíduo é determinada através de uma função definida como $f : J \mapsto R$, sendo que existe uma quantidade finita de valores $(f_1, \ldots, f_n) \mid n \leq N$;

2.1.3.1 Seleção por truncamento

Este método de seleção baseia-se, segundo Beyer e Schwefel (2002, p. 12), em selecionar os M melhores indivíduos (com *fitness* mais elevado) em uma população e possui duas variações.

Uma variação, conhecida como *comma selection* e denotada pelos autores como (J, γ), diferencia-se pelo fato dos cromossomos pais não competirem com os cromossomos filhos, sendo descartados ao final operação de *crossover*. Essa operação, visando a manutenção do tamanho da população, implica em todos os indivíduos da população ($M = N$) serem selecionados para reprodução o que resulta em um procedimento de busca aleatória.

A segunda variação, conhecida como *plus selection* e denotada pelos autores como $(J + \gamma)$, consiste na seleção dos m (onde $m < N$) melhores indivíduos para reprodução e

posterior seleção de N indivíduos em um conjunto composto por $J \cup \gamma$, cujo tamanho é maior que N, implicando assim na seleção dos melhores indivíduos. Este procedimento é exemplificado na Figura 9 através da utilização de uma população inicial de 5 indivíduos e da seleção dos 4 ($m = 4$ neste exemplo) melhores indivíduos para reprodução.

Figura 9 – Seleção por truncamento $(J + \gamma)$ - População inicial.

Cromossomos pais

Cromossomo	***Fitness***
A	5
B	4
C	3
D	2
E	1

Cromossomos filhos

Cromossomo	***Fitness***
Filho A e B	1
Filho A e D	6
Filho B e C	4
Filho B e D	2

Os cromossomos filhos, resultantes do processo de reprodução, são adicionados à população resultando em uma nova população intermediária contendo os N indivíduos originais e os m indivíduos resultantes de reprodução, este situação é apresentada na Figura 10. Por último os indivíduos com maior *fitness* desta população são selecionados para compor a nova população resultando na população apresentada na Figura 11.

Figura 10 – Seleção por truncamento $(J + \gamma)$ - Conjunto de pais e filhos.

Cromossomos pais e filhos, ordenados por *fitness*

Cromossomo	***Fitness***
Filho A e D	6
A	5
Filho B e C	4
B	4
C	3
Filho B e D	2
D	2
Filho A e B	1
E	1

Figura 11 – Seleção por truncamento $(J + \gamma)$ - Nova população.

Nova população

Cromossomo	***Fitness***
Filho A e D	6
A	5
Filho C e E	4
B	4
C	3

2.1.3.2 Seleção por roleta

A seleção por roleta foi originalmente proposta por Holland (1992) e consiste em simular a criação de uma "roleta viciada" onde certos números (cromossomos no caso de GAs) possuam maior possibilidade de serem escolhidos. Esse procedimento é explicado por Sivaraj e Ravichandran (2011, p. 3793) como um conjunto de 4 passos.

Primeiramente é calculado o *fitness* total da população (F_t) através do somatório do *fitness* de cada indivíduo, conforme apresentado na Equação 2.1, fazendo com que cada indivíduo possua uma probabilidade de seleção (sel_i) proporcional ao seu *fitness*, essa proporcionalidade é calculada através da Equação 2.2.

$$F_t = \sum_{i=1}^{N} f_i \tag{2.1}$$

$$sel_i = \frac{f_i}{F_t} \tag{2.2}$$

Calculada a nova probabilidade de seleção é construído um vetor com todos os cromossomos, onde a quantidade de ocorrências de um cromossomo seja proporcional sua probabilidade de seleção. O último passo consiste em gerar aleatoriamente N números entre 1 e F_t e selecionar os indivíduos da respectiva posição para compor a nova população.

Considerando a população inicial de 5 cromossomos descrita na Figura 12, ao calcular a probabilidade de seleção dos indivíduos obtêm-se valores entre 7% e 33%. A partir dessas probabilidades é criada uma lista (abstração da roleta) de 15 indivíduos contendo 5 cromossomos "A", 4 cromossomos "B", 3 cromossomos "C", 2 cromossomos "D" e 1 cromossomo "E"[16], esta lista é apresentada na Figura 13.

Figura 12 – Seleção por roleta - População inicial.

População

Cromossomo	***Fitness***	***Probabilidade seleção***
A	5	33%
B	4	27%
C	3	20%
D	2	13%
E	1	7%

Uma vez criada a roleta são gerados aleatoriamente 5 números entre 1 e 15. Na Figura 14, por exemplo, foram gerados os números 1, 2, 4, 8 e 13. Por último os indivíduos

[16] Foram realizados arredondamentos nos valores das probabilidades para exibir os valores em uma lista de 15 indivíduos, caso fosse necessário utilizar os valores exatos uma solução possível seria a criação de uma lista de 100 indivíduos com 33 "A", 27 "B", 20 "C", 13 "D" e 7 "E".

Figura 13 – Seleção por roleta - Criação da roleta.

Cromossomos em quantidade proporcional

A	A	A	A	A	B	B	B	B	C	C	C	D	D	E

das posições 1, 2, 4, 8 e 13 são selecionados resultando na nova população apresentada na Figura 15.

Figura 14 – Seleção por roleta - Seleção dos indivíduos.

Seleção aleatória de indivíduos (roleta)

↓	↓		↓				↓					↓		
A	A	A	A	A	B	B	B	B	C	C	C	D	D	E

Figura 15 – Seleção por roleta - Nova população.

Nova população

Cromossomo	***Fitness***
A	5
A	5
A	5
B	4
D	2

2.1.3.3 Seleção por amostragem universal estocástica

Esse método assemelha-se muito ao método da seleção por roleta, apresentado na subseção 2.1.3.3, buscando apenas a redução do elemento de aleatoriedade. Segundo Sivaraj e Ravichandran (2011, p. 3794) este método consiste em realizar os três passos iniciais descritos no método de seleção por roleta (passos que tratam da construção da roleta) e utilizar um novo mecanismo para a seleção dos indivíduos.

Enquanto no método da seleção da roleta são gerados N números aleatórios para seleção (simulando N giros da roleta) neste método é calculado um valor de espaçamento uniforme S na roleta, então é gerado um único número aleatório entre 1 e F_t como posição do primeiro marcador, após isso são inseridos $N-1$ marcadores com uma distância de S entre si[17].

O mesmo exemplo apresentado na subseção 2.1.3.2 para o método de seleção por roleta pode ser utilizado na explicação da seleção por amostragem universal estocástica.

[17] A roleta normalmente é implementada computacionalmente como um vetor, entretanto deve-se considerar que o sucessor do último elemento do vetor é o primeiro elemento, simulando um anel, para a aplicação desse método.

Novamente considerando a população inicial apresentada na Figura 12 e sua respectiva roleta, apresentada na Figura 13, calcula-se um espaçamento S uniforme (nesse caso $S = \frac{\text{comprimento da roleta}}{N} = 3$) e gera-se um número aleatório entre 1 e 15 como ponto inicial. Na Figura 16 o número aleatório gerado é 5, resultando na seleção dos indivíduos da posições 5, 8, 11, 14 e 2 e na população apresentada na Figura 17.

Figura 16 – Seleção por amostragem universal - Seleção dos indivíduos.

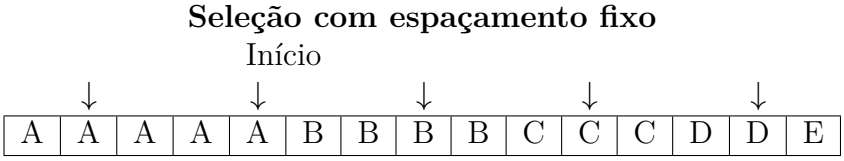


Figura 17 – Seleção por amostragem universal - Nova população.

Nova população

Cromossomo	***Fitness***
A	5
A	5
B	4
C	3
D	2

2.1.3.4 Seleção por ranking

A seleção por ranking é um método implementado para superar os problemas referentes a variação do *fitness* na população[18] como evitar a convergência prematura e prevenir que o algoritmo fique estagnado. Blickle e Thiele (1995, p. 27) subdividem essa seleção em duas categoria, baseando-se na forma de criação do ranking, a seleção por ranking linear e a seleção por ranking exponencial enquanto Pohlheim (2007 apud POHLHEIM, 1995) inclui a essa lista um mecanismo de seleção não linear.

Todas as técnicas de seleção por ranking supracitadas consistem inicialmente em ordenar os cromossomos conforme o valor de seu *fitness*[19], sendo que cada posição no ranking deverá conter um único cromossomo. Em seguida deve-se posicionar os cromossomos em um ranking onde as posições são alocadas de maneira decrescente e linear baseadas em seu *fitness*, onde o melhor cromossomo possuiria um valor de *fitness* N e o pior cromossomo possuiria um valor de *fitness* 1 (SIVARAJ; RAVICHANDRAN, 2011, p. 3794).

[18] Métodos como a roleta e a amostragem universal estocástica são distorcidos quando existem indivíduos cujo *fitness* seja muito superior ao *fitness* dos demais indivíduos da população e podem demorar a convergir caso todos os *fitness* sejam muito próximos.

[19] Caso existam dois ou mais cromossomos com o mesmo valor de *fitness* deve-se escolher a ordem baseada em outro critério, a ser decidido pelo projetista.

Após o posicionamento dos cromossomos no ranking deve-se recalcular o valor do *fitness* de cada indivíduo através da Equação 2.3, no caso da utilização de ranking linear, da Equação 2.4, no caso da utilização de ranking exponencial, e da Equação 2.5, quanto utilizado o ranking não linear. Este método explicita aos GAs o conceito de *selective pressure* (SP) que representa a intensidade com a qual ocorre o processo de seleção, esse valor é definido pelo projetista do GA devendo ser otimizado para cada problema a ser solucionado. Finalizado o recalculo dos *fitness* é realizado o procedimento de seleção do método da roleta (subseção 2.1.3.2)[20].

$$fitness_i = 2 - SP_1 + 2.(SP_1 - 1).\frac{i-1}{N-1} \tag{2.3}$$

$$fitness_i = \frac{SP_2 - 1}{SP_2^N - 1}.SP_2^{N-1} \tag{2.4}$$

$$fitness_i = \frac{N.X^{i-1}}{\sum_{i=1}^{N} f_i.X^{i-1}} \tag{2.5}$$

Onde:

SP_1: a pressão seletiva desejada, aceitando a mesma valores entre [1.0, 2.0].

SP_2: o coeficiente que determinará a intensidade da pressão seletiva aceitando valores no intervalo $]0, 1[$.

X: a solução da Equação 2.6 sendo SP_3 a pressão seletiva desejada, aceitando a mesma valores entre [1.0, N - 2].

$$(SP_3 - N).X^{N-1} + (SP_3.X^{N-2}) + \cdots + (SP_3.X) + SP_3 = 0 \tag{2.6}$$

Considerando a população inicial hipotética de 4 indivíduos com uma variação elevada de *fitness* apresentada na Figura 18, para a aplicação de qualquer mecanismo de seleção por ranking é necessário primeiramente que todos os cromossomos sejam ordenados, conforme apresentado na Figura 19.

Figura 18 – Seleção por ranking linear - População inicial.

População

Cromossomo	***Fitness***
A	2
B	7
C	11
D	1

[20] Embora o algoritmo tradicionalmente utilize a seleção por roleta é possível aplica-lo utilizando o mecanismo de seleção por amostragem universal estocástica.

Figura 19 – Seleção por ranking linear - População inicial ordenada.

População em ranking de *fitness*

Cromossomo	***Fitness***	**Ranking**
D	1	1
A	2	2
B	7	3
C	11	4

Uma vez ordenada a população é realizado o recalculo do *fitness*, no presente exemplo será utilizado como mecanismo de seleção um ranking linear e um valor de SP de 1,5, resultando nos novos valores de *fitness* apresentados na Figura 20, após esse procedimento é aplicado o mecanismo de seleção por roleta.

Figura 20 – Seleção por ranking linear - Recalculo do *fitness*.

População com *fitness* recalculado

Cromossomo	***Fitness***
D	0,50
A	0,75
B	1,00
C	1,25

É possível observar neste exemplo que embora a população original possuam grandes diferenças de *fitness* (11 para o cromossomo "C" e 1 para o cromossomo "D") após a aplicação desse método o cromossomo "C" possui um *fitness* apenas 2,5 vezes melhor que "D" implicando na criação de uma roleta mais equilibrada, caso fosse aplicado diretamente o mecanismo da roleta a proporção seria de 11 para 1.

2.1.3.5 Seleção por torneio

O método de seleção por torneio foi um dos primeiros métodos de seleção propostos e tem como princípio a competição direta entre indivíduos pelo direito de se reproduzir, da mesma forma de machos disputam uma parceira na natureza. Sivaraj e Ravichandran (2011) subdividem este método em torneios binários (compostos por dois indivíduos), "torneios maiores"[21] (compostos por mais de dois indivíduos) e torneios de Boltzmann (BTS).

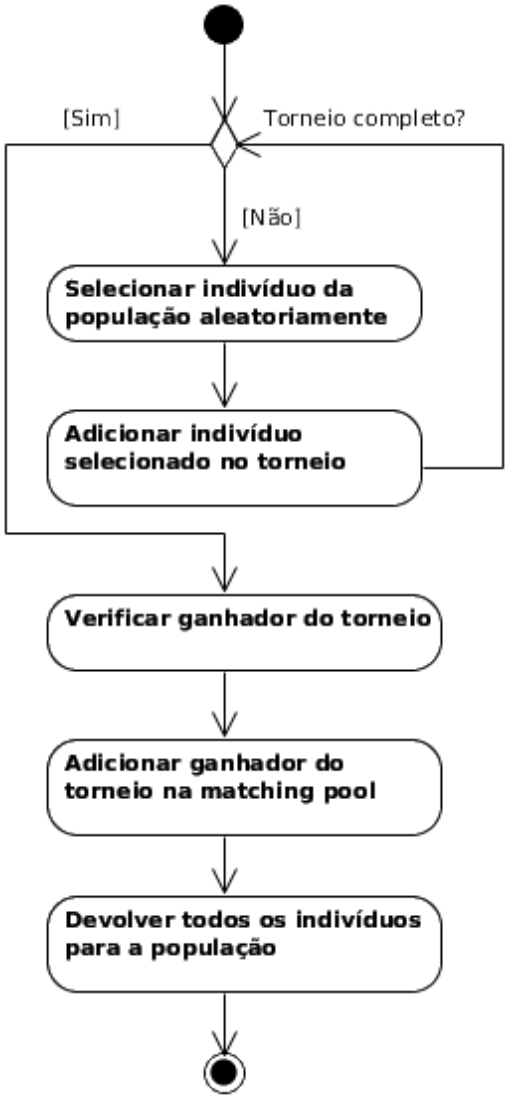
Os torneios de tipo binário ou "maior" variam somente na quantidade de indivíduos e possuem o seguinte procedimento: seleciona-se β[22] indivíduos aleatoriamente na

[21] Tradução livre do original "Larger Tournament selection" apresentado em (SIVARAJ; RAVICHANDRAN, 2011, p. 3794).

[22] Nesse algoritmo a pressão seletiva exercida pelo algoritmo é diretamente proporcional a β, ou seja, quanto maior o torneio menor a possibilidade de seleção de um indivíduo com baixo *fitness*.

população e elabora-se um torneiro onde o vencedor[23] irá para a *matching pool* se reproduzir. Após a realização do torneio todos os indivíduos retornam à população para participar de novos torneios. Repetem-se esses passos até que uma quantidade m (onde $m <= N$) de indivíduos seja selecionada, o fluxo de execução deste procedimento encontra-se ilustrado na Figura 21.

Figura 21 – Procedimento de geração seleção utilizando o mecanismo de torneio.



O torneio Boltzmann é apresentado por (SIVARAJ; RAVICHANDRAN, 2011, p. 3794) como um tipo de torneio motivado por *simulated annealing*[24] onde a seleção dos indivíduos do torneio segue regras, fazendo com que sua seleção ocorra de maneira semelhante a uma distribuição de Boltzmann. Esse procedimento consiste em criar um torneio de três indivíduos, o primeiro indivíduo é escolhido aleatoriamente, o segundo

[23] O cromossomo com o maior *fitness* no torneio é considerado o vencedor.

[24] Simulated annealing é uma heurística de busca fundamentada em uma analogia com a termodinâmica, esse tópico entretanto não faz parte do presente estudo sendo explicado com detalhes em (DAVIS, 1987).

indivíduo deve ser selecionado dentre os indivíduos que possuem um *fitness* distante do *fitness* dos primeiro indivíduo até um limite Δ pré estabelecido. O terceiro competidor deve ser solucionado de duas formas, em 50% dos casos deve-se selecionar o terceiro competidor através de *strict choice*, possuindo o mesmo um valor de *fitness* distante dos valores dos demais competidores em até Δ, as outras vezes deve-se utilizar *relaxed choice* cuja operação é igual a do segundo indivíduo.

2.1.4 Operadores de *crossover*

Srinivas e Patnaik (1994a, p. 656) afirmam que GAs realizam uma busca direcionada, embora aleatória, explorando regiões potencialmente boas (conjunto de esquemas que possuem um *fitness* melhor) no *fitness landscape*. Enquanto o processo de seleção é responsável pela identificação das regiões promissoras, os métodos de *crossover* são os principais responsáveis pela exploração desse espaço. Ainda segundo os autores a importância do processo de *crossover* é reflexo dos modelos formais apresentados na subseção 2.1.2 devido a possibilidade de criação de novos *building blocks* bons através da troca de material genético entre dois indivíduos compostos por blocos bons resultando, possivelmente, em indivíduos melhores.

Os mecanismos de *crossover* utilizados em GAs baseiam-se no princípio biológico da reprodução sexuada (AGRAWAL, 2001, p. 692) onde são necessários dois indivíduos (pais) para o surgimento de um novo indivíduo (filho). Entretanto, diferentemente dos processos reprodutivos que ocorrem na natureza, o processo de *crossover* de GAs sempre gerará dois indivíduos. De maneira semelhante à natureza, nem todos os indivíduos selecionados para a *matching pool*[25] se reproduzem, um dos parâmetros de configuração de um GA é o *crossover rate* que representa o percentual de chance que um indivíduo possui de se reproduzir, Beasley et al. (1993a, p. 3) afirma que esse percentual normalmente fica entre 60% e 100%. Os indivíduos que não foram selecionados para reprodução são copiados para a próxima geração.

Nas próximas seções serão apresentados alguns operadores de *crossover* clássicos sendo o *crossover* de um ponto (*One-Point crossover*) apresentado na subseção 2.1.4.1, o *crossover* de dois pontos (*Two-Point crossover*) na subseção 2.1.4.2 e o *crossover* uniforme (*Uniform crossover*) na subseção 2.1.4.3.

2.1.4.1 *Crossover* de um ponto

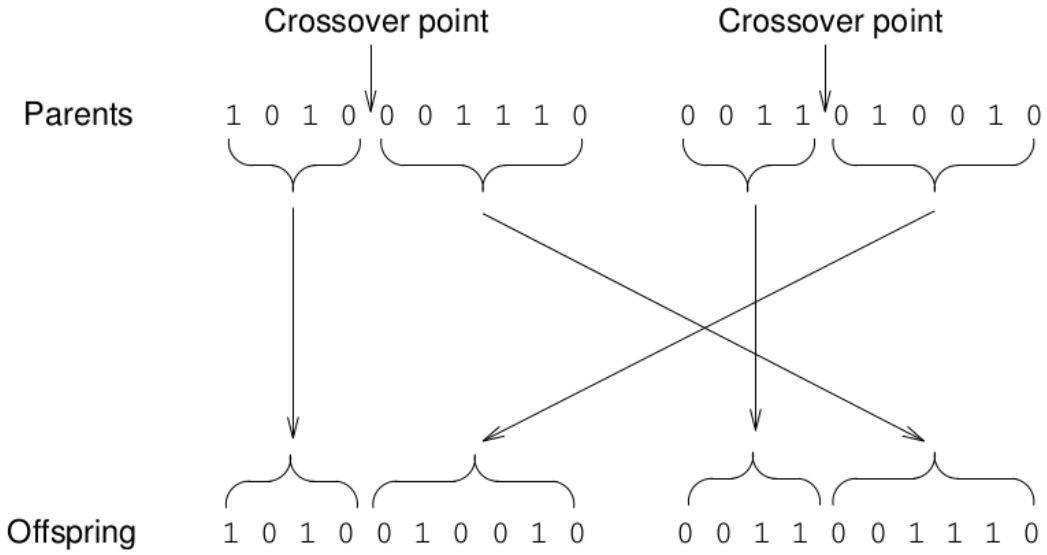
O *crossover* de um ponto (*One-Point crossover*) é o mecanismo original de reprodução apresentado por Holland (1992, p. 68). Para a aplicação desse operador o código

[25] *Matching pool* é o espaço reservado para os indivíduos que foram escolhidos através do processo de seleção para terem a chance de se reproduzir.

genético do cromossomo é visualizado como uma lista de l posições, onde cada posição equivale a um número (0 ou 1) existente no código genético.

Este método consiste em gerar aleatoriamente um número no intervalo $[1, l[$ como ponto de corte (*crossover point*) e segmentar o código genético dos cromossomos pais neste ponto, resultando em dois segmentos *head* e dois segmentos *tail* chamados respectivamente de H_1, H_2, T_1 e T_2. Os filhos serão gerados pela justaposição dos segmentos H_1 com T_2 (*head* do cromossomo 1 e *tail* do cromossomo 2) e H_2 com T_1 (*head* do cromossomo 2 e *tail* do cromossomo 1), esse procedimento é apresentado na Figura 22 utilizando como ponto de corte a posição 4.

Figura 22 – Aplicação do método de *One-Point crossover*.



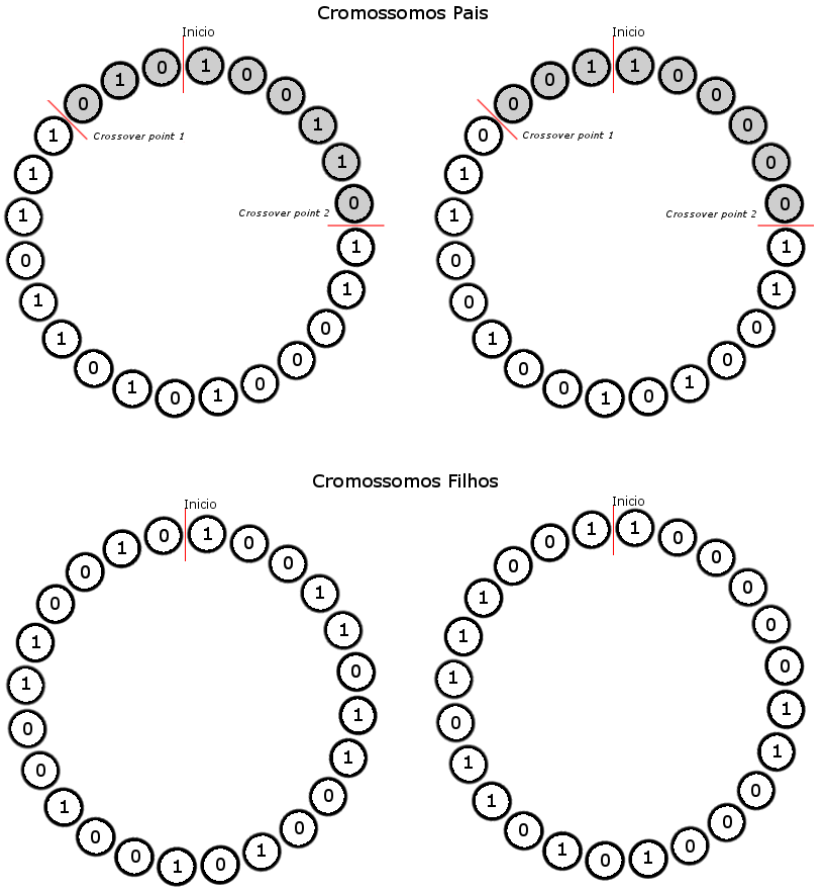
Fonte: (BEASLEY et al., 1993a, p. 4)

2.1.4.2 *Crossover* de dois pontos

Diferentemente do *One-Point crossover*, onde o código genético de um cromossomo é visualizado como uma lista, o *crossover* de dois pontos (*Two-Point crossover*) trata o código genético como um anel (DEJONG, 1975). Esta forma de visualização necessita de dois *crossover points* para a extração de um segmento (ponto de inicio e fim), esses pontos são selecionados de maneira aleatória, da mesma forma que no *One-Point crossover*. O segmento resultante entre os pontos de corte será então permutado entre os cromossomos pais para a produção dos cromossomos filhos.

A Figura 23 apresenta o *Two-Point crossover* tendo como pontos de corte as posições 21 e 6, através dessa imagem é possível verificar que, ao se tratar o cromossomo como um anel, não há necessidade de obrigar que o primeiro ponto de corte seja inferior ao segundo.

Figura 23 – Aplicação do método de *Two-Point crossover*.



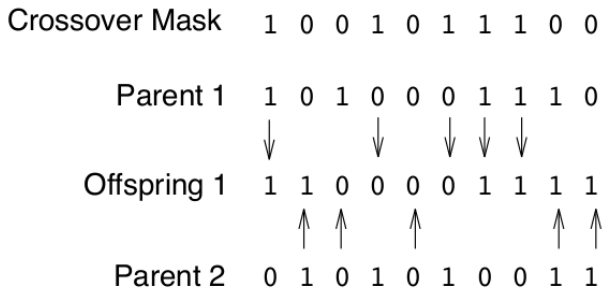
2.1.4.3 *Crossover* uniforme

Afirma Beasley et al. (1993b, p. 2) que o *crossover* uniforme difere completamente dos *crossovers* baseados em pontos de corte. Segundo o autor, considerando um cromossomo com código genético de comprimento l, esse método consiste na geração de uma máscara aleatória (*crossover mask*) de comprimento l composta por dígitos binários. Cada posição dessa mascara definirá de qual pai o gene será copiado. Quando a posição da mascara contiver o valor 1 o gene da posição equivalente será copiado do primeiro pai, caso contrário o valor será copiado do segundo pai, esse procedimento encontra-se ilustrado na Figura 24. Após a criação do primeiro filho os cromossomos pais são invertidos e é gerado o segundo filho[26] através do mesmo procedimento utilizado para a geração do primeiro filho. Uma nova máscara deve ser gerada para cada par de cromossomos que participar da operação.

Segundo Whitley (1994, p. 74) uma outra característica do *crossover* uniforme é que a quantidade de filhos passiveis de geração é superior a quantidade gerada através de operações como *One-Point crossover*. Considerando como código genético dos cromossomos pais os valores 0000 e 1111 é possível verificar na Figura 25 que todas as 2^4 strings

[26] Também é possível manter a posição dos cromossomos pais e utilizar o complemento da máscara para a criação do segundo filho.

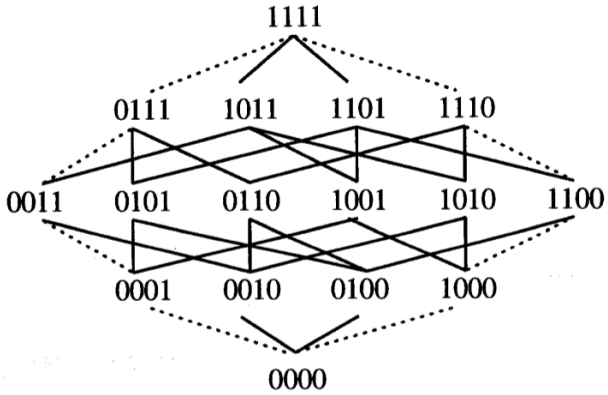
Figura 24 – Aplicação do método de *Uniform crossover*.



Fonte: (BEASLEY et al., 1993b, p. 2).

de comprimento 4 que podem ser obtidas através da combinação de 0s e 1s são acessíveis através da operação de *crossover*. A mesma figura apresenta, através de tracejados, os valores que uma operação de *One-Point crossover* poderia gerar.

Figura 25 – Filhos possíveis dos cromossomos 0000 e 1111 por *Uniform crossover*.



Fonte: (WHITLEY, 1994, p. 74).

2.1.5 Operadores de mutação

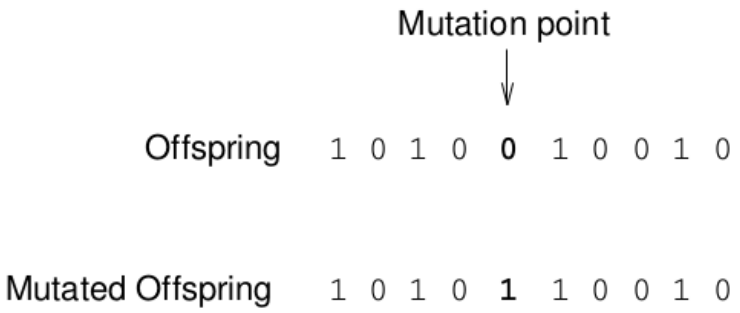
Na biologia mutações são ruídos decorrentes do processo de reprodução (BEYERA; SENDHOFF, 2007, p. 3207). Nos trabalhos de Srinivas e Patnaik (1994b, p. 18) e Cantú-Paz (1998, p. 4) o processo de mutação é descrito como um operador secundário cujo principal objetivo é restaurar a diversidade genética que pode ser perdida no decorrer do processo evolutivo, prevenindo assim que o algoritmo pare em um máximo local.

A influência que a mutação deve exercer em um GA entretanto não é um consenso, baseando-se na natureza Cantú-Paz (1998, p. 4) afirma que a incidência de mutações (p_m) é baixa, principalmente se comparada a incidência de reproduções (p_c), Grefenstette (1986, p. 124) por sua vez afirma que um valor elevado de mutação resultaria em uma busca aleatória. Entretanto Srinivas e Patnaik (1994b, p. 23) afirmam que o tamanho da população impacta diretamente na influência que a mutação deverá exercer, para

grandes populações (100^{16} indivíduos) a mutação deve ser tratada como um operador secundário[27] enquanto para populações menores (30^{8} indivíduos) a mutação deve ter um papel maior[28]. Trabalhos para verificar a validade de alterar a probabilidade de mutação durante a execução do algoritmo já foram realizados, entre eles o de Srinivas e Patnaik (1994a) onde é apresentado o AGA (*Adaptative Genetic Algorithm*), uma variação do SGA onde a taxa de mutações é variável no decorrer da execução.

Existem diferentes abordagens na realização da mutação, Guo et al. (2010, p. 2991) realiza essa operação negando todos os bits do código genético de um cromossomo. Grefenstette (1986, p. 124) por sua vez realiza a operação conhecida como *bit-flit* onde, com probabilidade p_m, cada bit de cada cromossomo na população decorrente da aplicação dos operadores de seleção e *crossover* sofre mutação. A aplicação desse método resulta em $p_m * L * N$, sendo L o comprimento do código genético de um cromossomo, mutações a cada geração. A Figura 26 ilustra a aplicação deste mecanismo de mutação em um *bit* de um cromossomo, que tem seu valor alterado de 0 para 1.

Figura 26 – Aplicação da mutação utilizando a técnica de *bit-flip*.



Fonte: (BEASLEY et al., 1993a, p. 4)

2.1.6 Algoritmo genético com população variável

Inicialmente propostos por Arabas et al. (1994), os GAs com população variável visam reduzir a influência do tamanho da população no resultado obtido por um GA. Eiben et al. (2004, p. 41-42) afirmam que em ambientes naturais o tamanho da população altera-se até encontrar um ponto de equilíbrio com o meio ambiente (exercido normalmente pela pressão seletiva), sendo mais flexível que as taxas de reprodução ou mutação. Ainda segundo os autores essa variação no tamanho da população pode ser utilizada na sua auto-calibração. Algumas variações ao SGA com inclusão de população variável foram propostas dentre as quais citam-se o GAVaPS (*Genetic Algorithm with Variable Population Size*), proposto por Arabas et al. (1994), o PLGA (*Parameter-less Genetic*

[27] Segundo os autores são valores típicos dessa categoria $p_c = 0,6$ e $p_m = 0,001$.
[28] Segundo os autores são valores típicos dessa categoria $p_c = 0,9$ e $p_m = 0,01$.

Algorithm) proposto por Harik e Lobo (1999) e o APGA (*Adaptative Population Size Genetic Algorithm*), apresentado por Bäck et al. (2000).

O GAVaPS introduz ao SGA os conceitos de idade e tempo de vida dos cromossomos. Neste método todo cromossomo tem seu tempo de vida calculado no nascimento, assim como, por definição, sua idade inicial é 0. Arabas et al. (1994, p. 75) afirma que uma vez que todos os cromossomos possuem a mesma probabilidade de seleção para reprodução (não existe mecanismo de seleção) a seleção dos melhores indivíduos deve ser feita através do cálculo do seu tempo de vida, os autores definem o objetivo da função de *lifetime* como: reforçar os indivíduos mais aptos e ajustar o tamanho da população prevenindo assim o seu crescimento exponencial. Para atingir os objetivos propostos os autores apresentam três mecanismos de cálculo do tempo de vida, o proporcional, o linear e o bi-linear, definidos respectivamente nas equações 2.7, 2.8 e 2.9. Eiben et al. (2004, p. 42-43) por sua vez propõem o APGA, uma alteração do GAVaPS onde é reintroduzida a operação de seleção, visando dar mais foco ao processo evolutivo do algoritmo, e definido que o melhor indivíduo na população[29] não envelhece, ou seja, sua idade não é incrementada ao final de cada iteração do algoritmo. Ainda segundo os autores essas variações não removem a necessidade de se definir um tamanho de população inicial.

$$LT_i = min(MinLT + \eta \frac{f_i}{AvgFit}, MaxLT) \tag{2.7}$$

$$LT_i = MinLT + 2\eta \frac{f_i - |MinFitness|}{|MaxFitness| - |MinFitness|} \tag{2.8}$$

$$LT_i = \begin{cases} MinLT + \eta \frac{f_i - MinFitness}{AvgFitness - MinFitness} & \text{se } AvgFit \geq f_i \\ \frac{1}{2}(MinLT + MaxLT) + \eta \frac{f_i - AvgFit}{MaxFit - AvgFit} & \text{se } AvgFit < f_i \end{cases} \tag{2.9}$$

Onde:
$\eta = \frac{1}{2}(MaxLT - MinLT)$
MinLT = Quantidade mínima de geração em que um cromossomo viverá.
MaxLT = Quantidade máxima de geração em que um cromossomo viverá.

Lobo e Lima (2006, p. 1242) demonstram que o tamanho da população durante a execução do algoritmo variam conforme o valor máximo estabelecido de MaxLT ficando implicitamente limitados pela inequação 2.10, além disso, segundo os autores, o tamanho médio da população é $\overline{N} = MinLT + MaxLT + 1$.

$$N \leq 2MaxLT + 1 \tag{2.10}$$

[29] É considerado como melhor indivíduo o cromossomo com o maior *fitness*

O PLGA apresenta um algoritmo diferente visando eliminar todos os parâmetros de um GA, dentre eles o *crossover rate*, *selective pressure* e o tamanho da população inicial. No presente trabalho será utilizada somente a alteração do algoritmo que remove a necessidade da definição de um tamanho de população inicial.

Harik e Lobo (1999, p. 3-7) propõe que seja realizada uma corrida entre múltiplos GAs com tamanhos de população diferentes. Na prática os autores criaram uma estrutura de controle (UC) para a execução simultânea de diversos SGAs, essa estrutura é responsável pela criação e destruição de novos SGAs e pela definição da ordem de execução de suas iterações. O algoritmo é iniciado definindo-se o tempo t como 0 e criando um SGA sga_0 com população de tamanho N. A cada execução de uma iteração de qualquer SGA t será incrementado em 1. Em seguida inicia-se a execução das iterações de sga_0 até que t seja múltiplo de 4[30], nesse momento é criado um SGA sga_1 com população de tamanho $2N$ e executa-se uma iteração do mesmo. Após essa ação executa-se novamente iterações de sga_0 até que t seja novamente múltiplo de 4, então executa-se uma iteração de sga_1. Quando t atingir 16 cria-se e executa-se um novo SGA sga_3 de tamanho populacional $4N$. Sinteticamente, a cada 4 execuções de um SGA sga_n é executada uma iteração de um SGA sga_{n+1}, esse fluxo de execução encontra-se ilustrado na Figura 27 e é utilizado para dar preferência a execução de algoritmos com populações menores.

Durante a execução do PLGA, um SGA pode ser removido caso possua pouca possibilidade de sucesso, os autores dividem esse momento em duas situações: quando sua população convergir ou quando um SGA de população maior possuir um indivíduo com *fitness* superior a seu melhor indivíduo. Em ambos os casos o tamanho da população é considerado insuficiente para a solução do problema, o mesmo é descartado e a posição dos SGAs subsequentes é decrementada em 1, mantendo a sequência de contadores dos SGAs contínuos. Essa abordagem faz com que sejam executados diversos SGAs até que a condição de parada do algoritmo seja satisfeita fazendo com que o tamanho da população necessário para alcançar o resultado não seja mais definido pelo desenvolvedor.

2.2 FRAMEWORKS ORIENTADOS A OBJETOS

Essa seção apresenta a fundamentação teórica sobre *frameworks*[31]. Os conceitos básicos sobre reuso de artefatos de software, *frameworks* e suas principais classificações são abordados na subseção 2.2.1. O processo de análise de domínio necessário à elaboração de *frameworks* é apresentado na subseção 2.2.2, enquanto padrões de projeto e sua aplicabilidade em *frameworks* são apresentados na subseção 2.2.3. Por último a subseção 2.2.4 apresenta os princípios da geração de aplicações utilizando *frameworks*.

[30] Os autores trabalham com um valor fixo de 4 iterações do SGA sga_n para cada iteração do SGA sga_{n+1}, porém esse valor pode ser alterado.

[31] No presente trabalho a expressão *framework* se refere a *framework* orientado a objetos

Figura 27 – Fluxo de execução de um PLGA.

t (base 4)	Ação
0	Executar 1 iteração da população p_0
1	Executar 1 iteração da população p_0
2	Executar 1 iteração da população p_0
3	Executar 1 iteração da população p_0
10	Executar 1 iteração da população p_1
11	Executar 1 iteração da população p_0
12	Executar 1 iteração da população p_0
13	Executar 1 iteração da população p_0
20	Executar 1 iteração da população p_1
21	Executar 1 iteração da população p_0
22	Executar 1 iteração da população p_0
23	Executar 1 iteração da população p_0
30	Executar 1 iteração da população p_1
31	Executar 1 iteração da população p_0
32	Executar 1 iteração da população p_0
33	Executar 1 iteração da população p_0
100	Executar 1 iteração da população p_2
101	Executar 1 iteração da população p_0
⋮	⋮

Fonte: (HARIK; LOBO, 1999, p. 6).

2.2.1 Conceitos básicos

A Engenharia de Software é uma disciplina da Ciência da Computação cujos objetivos principais são "a melhora da qualidade do software e o aumento da produtividade da atividade de desenvolvimento de software" (SILVA, 2000 apud FAIRLEY, 1985, p. 21), uma abordagem para atingir esses objetivos é a reutilização de elementos de software como funções, classes e bibliotecas. Dentre as técnicas desenvolvidas com essa finalidade estão os *frameworks* que, segundo Markiewicz e Lucena (2001, p. 1), se tornaram um dos pilares da Engenharia de Software moderna. Na literatura existem diversas definições de frameworks:

Markiewicz e Lucena (2001, p. 3) definem *frameworks* como geradores de aplicação para uma família específica de problemas. Johnson e Foote (1988, p. 27) definem-os como arquiteturas abstratas para tipos específicos de aplicação, normalmente sendo constituídos de diversas classes. Carneiro (2003 apud VILJAMAA, 2001, p. 16) define-os como "um conjunto de objetos reutilizáveis que engloba conhecimento de determinadas áreas e se aplica a um domínio específico" sendo que uma aplicação completa, ou parte significante dela, pode ser especializada dessa estrutura fazendo-se as adaptações necessárias ou adicionando-se novas características. Barreto Junior (2006 apud CRESPO; PINTO,

2000, p. 34) os define como um software incompleto que determina uma arquitetura para uma família de subsistemas, entretanto a definição utilizada no presente trabalho será a de Silva (2000 apud WIRFS-BROCK et al., 1991, p. 31) que descrevem-os de forma mais detalhada como conjuntos de classes inter-relacionadas que proveem não somente o reuso de código-fonte mas também de arquitetura e de informações de projeto, o autor ressalta ainda que *frameworks* não são aplicações completas e sim esqueletos de aplicações ou subsistemas, abstrações de uma família de problemas onde o desenvolvedor deve incluir a lógica necessária para seu funcionamento.

Tratando-se de esqueletos para uma família de problemas, *frameworks* devem ser flexíveis. Seus pontos flexíveis são chamados de *hot spots* e são métodos ou classes abstratas que devem ser implementadas pelo desenvolvedor para uma aplicação. Os pontos do *framework* que não devem ser alteradas e constituem seu núcleo são chamados de *frozen spots*. Os *frozen spots* são trechos de código pré implementados no *framework* e que fazem a chamada do código acoplado aos *hot spots* (MARKIEWICZ; LUCENA, 2001, p. 3).

2.2.1.1 Reuso tradicional x Reuso por *frameworks*

Segundo Yang et al. (1998, p. 211) o aumento da complexidade dos sistemas impacta diretamente no custo do processo de desenvolvimento dos mesmos. Isso torna cada vez mais importante o reuso de artefatos de software. As abordagens iniciais de reuso foram bibliotecas de funções e procedimentos, estas proviam somente o reuso de trechos de código-fonte, nestas abordagens a aplicação realiza a chamada à funções da biblioteca escolhendo quais funções utilizar ou ignorar.

Com o surgimento do paradigma OO foi possível o reuso de artefatos de software mais complexos, o que resultou nos *frameworks* (YANG et al., 1998, p. 211). Sparks et al. (1996, p. 52-53) afirmam que *frameworks* são caracterizados principalmente pela inversão do fluxo de controle da aplicação, isto é: o código do desenvolvedor (aplicação), após a etapa de inicialização do *framework*, passa o controle da aplicação para o *framework* e aguarda invocações, essa abordagem posteriormente passou a ser conhecida como princípio de *Hollywood*[32]. Essa arquitetura de *framework* traz percentuais mais elevados de reuso e produtividade que as apresentadas anteriormente pois introduzem o reuso da arquitetura e de informações de projeto, caso disponíveis, além do reuso de código-fonte.

2.2.1.2 Classificação por estrutura

Um *framework* pode ser classificado de acordo com a forma com a qual deve ser utilizado. Quando projetado para que uma aplicação seja gerada a partir do processo

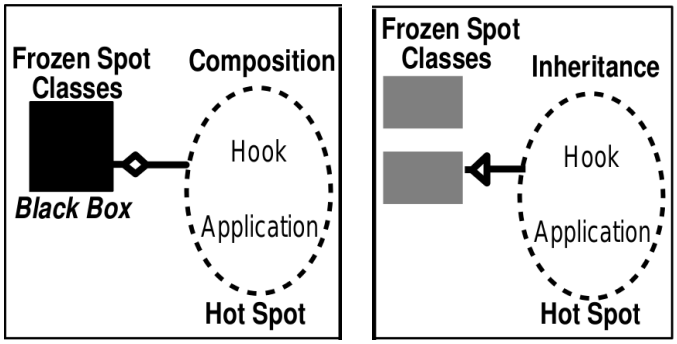
[32] Princípio de *Hollywood* for o termo utilizado por Bosch et al. (2000, p. 5) para descrever a abordagem "não nos chame, nós chamamos você" (SPARKS et al., 1996, p. 53).

de herança[33] o *framework* é classificado como dirigido a arquitetura ou de caixa-branca (SILVA, 2000, p. 33), segundo Parsons et al. (1999, p. 142), ao utilizar um *framework* de caixa-branca os desenvolvedores devem conhecer sua arquitetura interna para adaptá-lo aplicações concretas implicando em uma curva de aprendizado maior.

Quando um *framework* é projetado para que aplicações sejam desenvolvidas através da diferente composição de objetos gerados a partir de suas classes ele é classificado como dirigido a dados ou de caixa-preta (SILVA, 2000, p. 33). Diferentemente dos *frameworks* de caixa-branca, os *frameworks* de caixa-preta escondem sua estrutura interna o que, segundo Parsons et al. (1999, p. 142), reduz a curva de aprendizado para utilização do mesmo uma vez que os desenvolvedores necessitam apenas de uma descrição do mesmo e informações sobre os *hot spots*.

Por último *frameworks* podem ser classificados como caixa-cinza quando são híbridos entre *frameworks* de caixa-branca e caixa-preta, ou seja, alguns *hot spots* são desenvolvidos visando a utilização através de herança e outros através de composição. Na prática existem poucos *frameworks* puramente caixa-branca ou caixa-preta (PARSONS et al., 1999, p. 142). A Figura 29(a) e a Figura 29(b) ilustram respectivamente *frameworks* de caixa-preta e de caixa-branca sendo possível observar a diferença entre o mecanismo utilizado para implementação dos *hot spots* e a visibilidade entre as estruturas.

Figura 28 – *Frameworks* de caixa-preta e caixa-branca.



(a) Caixa-preta (b) Caixa-branca

Fonte: (PARSONS et al., 1999, p. 142)

2.2.1.3 Classificação por finalidade

Frameworks possuem finalidades diferentes e, em alguns casos, não destinam-se ao desenvolvimento de uma aplicação completa e sim de funcionalidades ou subsistemas de aplicações completas. Taligent (1995, p. 6-7) classificou-os conforme sua finalidade em: *frameworks* em nível de sistema (*system-level frameworks*)[34], *frameworks* em nível de

[33] Herança é um processo de hierarquização de classes onde subclasses são geradas especializando o comportamento de superclasses (BOOCH, 1986, p. 216).

[34] Alguns autores como Carneiro (2003, p. 16) traduzem essa expressão como *frameworks* de suporte.

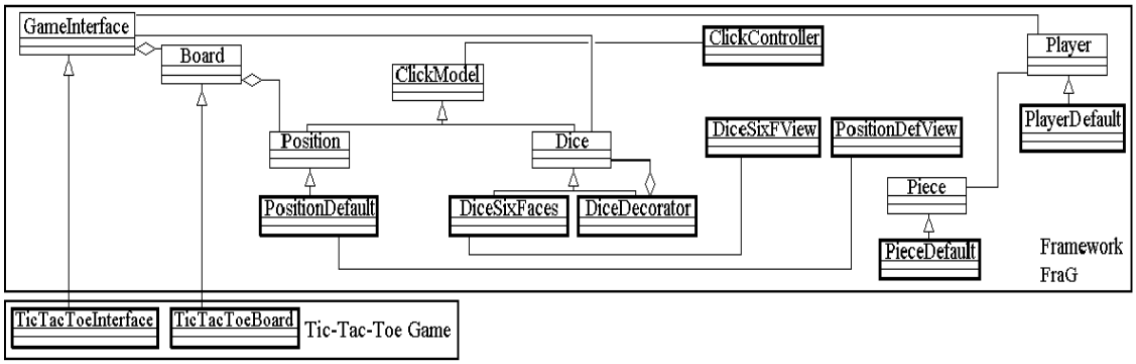
aplicação (*application-level frameworks*) e *frameworks* de domínio (*domain-specific frameworks*).

Segundo Carneiro (2003, p. 16) os *frameworks* em nível de sistema disponibilizam serviços a nível de S.O. (sistema operacional) como acesso a arquivos e memória. Esse tipo de *framework* é frequentemente encontrado em ambientes distribuídos como o *Hadoop Distributed File-System* (SHVACHKO et al., 2010) e o *G-Hadoop: MapReduce* (WANG et al., 2013).

São considerados *frameworks* em nível de aplicação, segundo Taligent (1995, p. 6), os que encapsulam conceitos aplicáveis a diversos tipos de *softwares* sendo utilizados para a implementação de somente parte de uma aplicação. Um exemplo dessa categoria são os *frameworks* de interface gráfica (GUI) como o SEKS (YU et al., 2007) e os *frameworks* de persistência de dados como o Hibernate (BAUER; KING, 2006).

Os *frameworks* de domínio por sua vez encapsulam conhecimento de um domínio podendo gerar grande parte de aplicações relativas aos mesmos. São exemplos de *frameworks* de domínio o Unidraw para a geração de editores gráficos (VLISSIDES; LINTON, 1990) e o FraG para a implementação de jogos de tabuleiro (SILVA, 2000). A Figura 29 apresenta o diagrama de classes de um jogo *Tic-Tac-Toe* desenvolvido utilizando o *framework* FraG onde coube ao desenvolvedor somente a especialização das classes *GameInterface* e *Board* com as regras específicas do jogo.

Figura 29 – Jogo da velha implementado utilizando o *framework* FraG.



Fonte: (SILVA, 2000, p. 35)

Outra classificação de *frameworks* segundo sua finalidade é feita por Fayad e Schmidt (1997, p. 34-35) onde os mesmos são classificados em *frameworks* de infraestrutura (*system infrastructure frameworks*), *frameworks* de integração (*middleware integration frameworks*) e *frameworks* empresariais (*enterprise application frameworks*).

Os *frameworks* de infraestrutura, segundo os autores, são utilizados para o desenvolvimento de elementos de infraestrutura como S.O., mecanismos de comunicação, processamento de linguagens e interfaces gráficas podendo ser utilizados para a geração

de aplicações em diversas áreas.

Fayad e Schmidt (1997, p. 35) afirmam que *frameworks* de integração são projetados para facilitar o desenvolvimento de soluções modulares, sendo frequentemente utilizados para o desenvolvimento de soluções distribuídas. Essa categoria de *frameworks* é exemplificada pelos mesmos com um *message-oriented middleware* (MOM).

Na última categoria de *frameworks* proposta pelos autores, os *frameworks* empresariais, enquadram-se os *frameworks* que englobam diversos conceitos de um mesmo domínio de aplicação como aviação, telecomunicações, engenharia, visando o desenvolvimento de múltiplas soluções em uma mesma área. Essa categoria de *frameworks* difere das demais pois o *framework* possui elementos do domínio do problema e não elementos utilizados em subsistemas como persistência ou comunicação em rede.

2.2.2 Análise de domínio

Na Ciência da Computação, enquanto a etapa de análise de requisitos de um problema identifica os elementos de uma aplicação, a análise de domínio, segundo Mattsson (2000, p. 12-13), trabalha com um nível de abstração mais elevado explorando requisitos de uma família de aplicações em uma área específica.

Na literatura encontram-se diversas definições complementares sobre esse tema. Guizzardi (2000 apud NEIGHBORS, 1980, p. 40) define análise de domínio como "uma tentativa de identificar os objetos, operações e relações entre o que peritos em um determinado domínio percebem como importante".

Arango (1989, p. 153) define-a como a identificação, aquisição e evolução de informação reutilizável sobre um domínio de problema para ser utilizado na especificação e construção de softwares. O autor subdivide análise de domínio em duas fases, a análise conceitual e a análise construtiva. A análise conceitual é o processo de identificação e aquisição das informações necessárias para a especificação de um sistema em um domínio enquanto a análise construtiva é identificação e aquisição das informações necessárias para a implementação dessa especificação.

Sendo um *framework* um tipo de modelo de domínio, o processo de análise de domínio que será apresentado no decorrer desta seção é, de maneira abstrata, um processo para a elaboração de um *framework*. Entretanto para a apresentação desse processo alguns conceitos devem ser apresentados:

- **Domínio de problema:** informações do mundo real inter-relacionadas e que pertençam a uma mesma classe de problemas (ARANGO, 1989, p. 153);
- **Modelo do domínio:** sistema formal que define entidades, operações, eventos e relações que abstraem um domínio de problema específico visando formar um modelo

capaz de servir como fonte unificada de referência e repositório de conhecimento comum (GUIZZARDI, 2000, p. 40-41);

- **Análise e modelagem do domínio:** processo que visa reduzir a complexidade de um determinado domínio organizando dados adquiridos por experimentos, engenharia reversa de sistemas existentes e dados de experimentos (GUIZZARDI, 2000, p. 41).

2.2.2.1 Um processo para Análise de Domínio

Segundo Guizzardi (2000) a complexidade de identificação, captura e organização dos elementos relevantes para representação do conhecimento embutido em uma classe de problemas impõe a necessidade da definição de um processo bem estruturado de análise de domínio englobando técnicas de engenharia de software, engenharia de requisitos e modelagem conceitual. Diversos processos de análise de domínio foram analisados em (GUIZZARDI, 2000 apud ARANGO, 1994) dentre eles:

- *McCain's Product-Oriented Paradigm*;
- *Prieto-Díaz Domain Analysis for reusability*;
- *Simon's Domain Analysis for building a Organon*;
- *SEI's Feature-Oriented Domain Analysis*;
- *Software Productivity Consortium Domain Analysis*;
- *Lubar's Domain Analysis in Intelligent Design Aid*;
- *Vitaletti and Guerrieri*;
- *Bailin.*

A partir da análise desses métodos Guizzardi (2000, p. 42-44) descreve um processo comum de desenvolvimento composto de três fases: planejamento, aquisição e seleção dos dados e análise dos dados e modelagem do domínio as quais serão apresentadas abaixo.

- **Planejamento:** esta fase é composta por atividades como análise do negócio e análise de risco. São avaliadas questões como a relação custo-benefício da análise de domínio e se o domínio é estável e conhecido para que a análise possa ser realizada. Concluídas essas avaliações realiza-se a definição do escopo e das métricas de avaliação do processo resultando em uma especificação de requisitos do domínio.

- **Aquisição e seleção dos dados:** procede-se com a identificação das fontes de dados disponíveis como: livros, artigos, revistas científicas, entrevistas com especialistas e engenharia reversa de projetos existentes. Guizzardi (2000 apud CORNWELL, 1996, p. 43) sugere que utilize-se um mínimo de três fontes de dados e que imagine-se um mínimo de três aplicações futuras que poderão ser produzidas a partir do modelo gerado, Guizzardi (2000 apud ARANGO, 1994, p. 43) por sua vez defende que as fontes de dados devem complementar umas às outras.

- **Análise dos dados e modelagem do domínio:** nesta etapa avalia-se os dados capturados quanto a fatores como consistência e completude. Em seguida realiza-se a identificação de entidades, relações e funções produzindo um modelo conceitual onde estão representadas as entidades de domínio e seus relacionamentos e um dicionário do domínio que apresenta as definições dos elementos que compõem o modelo conceitual.

2.2.3 Padrões de projeto

Silva (2000, p. 56) define que padrões de projeto (*design patterns*) são microarquiteturas constituídas por classes, suas responsabilidades e sua forma de cooperação, originadas a partir da observação de que diferentes partes de *frameworks* possuíam estruturas de classes semelhantes, evidenciando a existência de soluções padrões para problemas semelhantes, ou seja, cada padrão apresenta uma solução reutilizável para um problema recorrente. O uso dessas microarquiteturas ajudam a manter um baixo acoplamento entre os diferentes subsistemas em um *software*, além de facilitar sua extensão e documentação (SRINIVASAN, 1999, p. 25).

Apesar das similaridades entre *design patterns* e *frameworks*, quanto a ambas serem abordagens voltadas ao reuso, elas são abordagens diferentes, embora inter-relacionadas. Guizzardi (2000, p. 29) relaciona-os da seguinte maneira:

- **Implementação:** Alguns *frameworks* foram implementados diversas vezes, em plataformas e linguagens distintas, neste caso eles podem ser considerados também como *design patterns*, um exemplo disso é o *framework Model View Controller* (MVC) que é apresentado por Buschmann et al. (1996) como um padrão;

- **Nível de abstração:** Padrões são microarquiteturas para problemas de projeto e não problemas concretos, ou seja, são independentes de linguagem. *Frameworks* por sua vez proporcionam o reuso de projeto e código fonte o que os torna dependentes de linguagem e menos abstratos que *patterns*;

- **Tamanho:** *Frameworks* são maiores (compostos por mais elementos arquitetônicos) que *patterns*, normalmente *frameworks* são compostos por diversos padrões, o

framework MVC por exemplo é composto principalmente pelos padrões *Observer*, *Composite* e *Strategy*;

Uma classificação para os padrões de projeto é feita por Srinivasan (1999, p. 25) onde os mesmos são agrupados em duas categorias. A primeira categoria refere-se a *design patterns* voltadas ao projeto de soluções OO enquanto a segunda categoria volta-se para o desenvolvimento de soluções para tratamento de problemas de maneira eficiente, concorrente e distribuída, padrões dessa segunda categoria também podem ser utilizados no projeto de soluções OO. Schmidt et al. (1996, p. 38) cita como exemplos dessa segunda categoria os trabalhos de McKenney (1996), Islam e Devarakonda (1996) e Aarsten et al. (1996) apresentando respectivamente padrões para sistemas multiprocessados eficientes, confiabilidade em sistemas distribuídos e sistemas distribuídos para sistemas de controle em larga escala. Mais recentemente outros padrões para ambientes distribuídos visando confiabilidade foram apresentados por Babaoglu et al. (2006) inspirados em sistemas biológicos.

Outra classificação é a criada por Gamma et al. (1995) onde os *patterns* são categorizados em padrões de criação (relacionados a instanciação de objetos), estruturais (referentes ao relacionamento entre classes) e comportamentais (referentes a divisão de responsabilidades entre as classes). A classificação de Gamma et al. (1995) foi publicada no primeiro catalogo de padrões de projeto, livro que é conhecido atualmente como GoF (*Gang of four*), onde são apresentados 23 padrões genéricos e independentes de domínio. Outro catálogo de padrões importante foi publicado por Larman (2012) onde são apresentados 9 padrões para a atribuição de responsabilidade entre classes, seus padrões hoje são conhecidos como GRASP (*General Responsibility Assignment Software Patterns*).

Além dos padrões de projeto, o catalogo de Gamma et al. (1995) definiu um formato de apresentação de padrões de projeto que é utilizado até hoje, esse formato é composto pelas principalmente pelas seções: nome, categoria, objetivo, outros nomes, motivação, aplicabilidade, estrutura e participantes. A Figura 30 apresenta as seções estrutura e participantes do padrão de projeto *Abstract Factory* apresentado pelos autores.

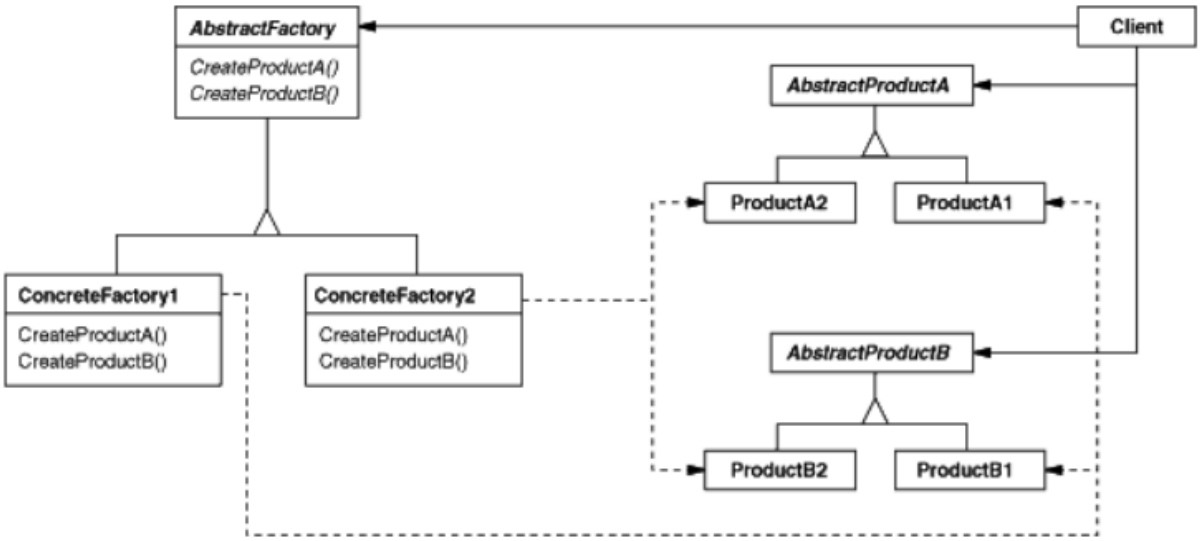
2.2.4 Geração de aplicações a partir de frameworks

Uma aplicação concreta pode ser desenvolvida estendendo classes do *framework* ou conectando classes específicas a cada *hot spot* (YANG et al., 1998, p. 211).

Markiewicz e Lucena (2001, p. 3) utilizam uma metáfora para explicar o processo de utilização de um *framework*. Eles consideram o *framework* como um motor que necessita de energia para funcionar, entretanto diferentemente dos motores tradicionais esse motor possui vários conectores de energia. Cada conector de energia deve estar conectado para que o motor possa funcionar, o motor por sua vez utiliza a energia provida pelas conexões

Figura 30 – Seções estrutura e participantes do padrão de projeto *Abstract Factory*.

Structure



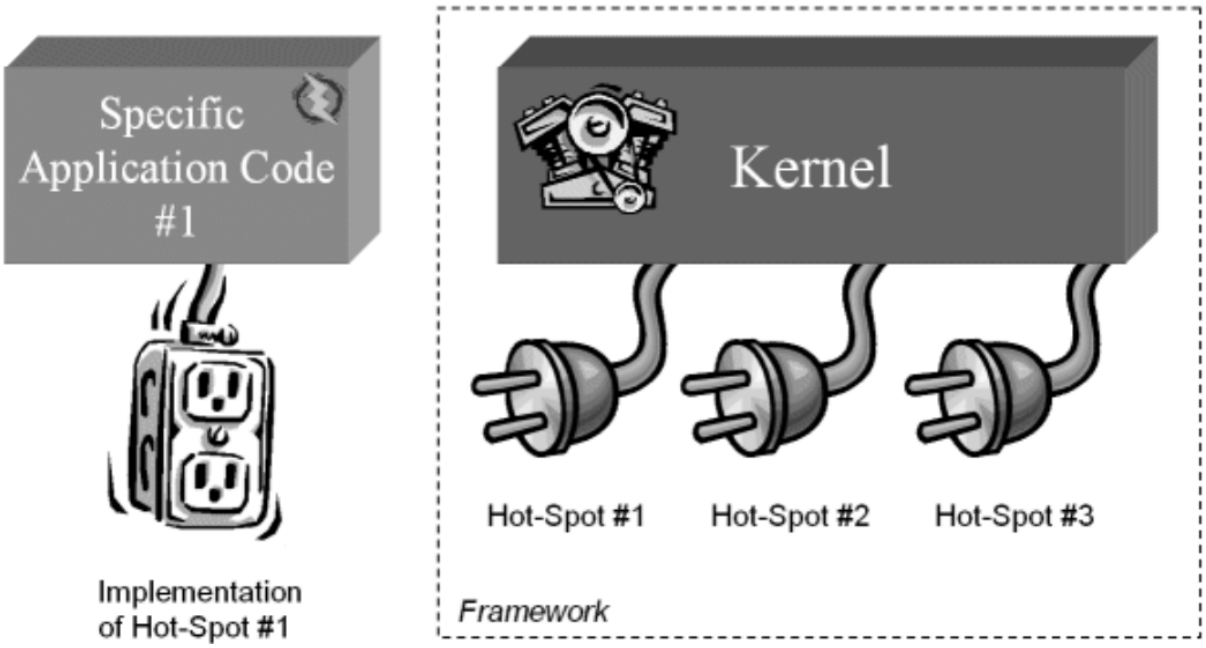
Participants

- **AbstractFactory** (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- **AbstractProduct** (Window, ScrollBar)
  - declares an interface for a type of product object.
- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Fonte: (GAMMA et al., 1995, p. 101)

para executar as suas tarefas. Nesta metáfora, ilustrada na Figura 31, os geradores de energia são os trechos de código específicos da aplicação, enquanto os conectores de energia são os *hot spots* do *framework*. Ou seja, para a geração de uma aplicação, ou subsistema, a partir de um *framework* cabe ao desenvolvedor incluir o código específico de sua aplicação nos *hot spots* do *framework* e escrever um código de inicialização para transferir o controle fluxo de execução para o *framework*.

Figura 31 – Metáfora da conexão entre aplicação e *framework*.



Fonte: (MARKIEWICZ; LUCENA, 2001, p. 3)

2.3 *MAPREDUCE*

Essa seção apresenta a fundamentação teórica referente ao modelo *MapReduce*. Na subseção 2.3.1 são abordados os conceitos básicos sobre o modelo, sua motivação e principais características. A forma como deve ser realizada a programação utilizando esse modelo é apresentada na subseção 2.3.2 enquanto o fluxo de execução de um algoritmo desenvolvido sobre ele é apresentado na subseção 2.3.3. Por último é apresentada a implementação *open source* mais popular desse modelo na subseção 2.3.4.

2.3.1 Conceitos básicos

MapReduce é um modelo de programação distribuída para o processamento de grandes massas de dados baseado em conceitos da programação funcional. Esse modelo foi inicialmente desenvolvido pela Google em 2003 para utilização interna, onde haviam

sido implementados diversos algoritmos para o processamento de quantidades massivas de dados, como *logs* de requisições web e páginas indexadas, resultando em informações como, por exemplo, os termos de busca mais utilizados em um período. Embora os algoritmos executados sobre a massa de dados sejam conceitualmente simples, necessidades como a paralelização das operações e realização de tratamento de falhas aumentavam a complexidade do desenvolvimento. Devido a esse aumento de complexidade foi proposta uma abstração que possibilitasse a realização de computações simples de maneira distribuída escondendo os detalhes da paralelização, tolerância a falhas, distribuição dos dados e balanço de carga dos computadores (DEAN; GHEMAWAT, 2008, p. 107).

Esse modelo é baseado nas primitivas *map* e *reduce* das linguagens funcionais, onde uma mesma operação é realizada para cada registro da massa de dados. Nele o desenvolvedor especifica uma função *map*, que recebe uma chave e um valor de entrada e gera uma nova chave e um valor intermediário, e uma função *reduce*, que recebe as chaves e valores intermediários e une todos os elementos com as mesmas chaves (DEAN; GHEMAWAT, 2010, p. 72).

Na programação funcional a primitiva *map* é definida como uma função de dois argumentos, o primeiro é uma função de mapeamento $f(a) = b$ que converte um elemento de a em um elemento de b, o segundo é uma lista de elementos do tipo a. A utilização dessa primitiva realiza a aplicação a função f a cada elemento da lista recebida como parâmetro, produzindo como saída uma nova lista contendo elementos do tipo b. A primitiva *reduce* por sua vez é definida como uma função que recebe uma lista de elementos do tipo b e sumariza os resultados produzindo uma nova lista de elementos do tipo c, ou seja, é responsável pela união dos valores resultantes da primitiva *map*, resumindo os resultados (HUDAK, 1989, p. 377).

2.3.2 Modelo de programação

Segundo Dean e Ghemawat (2008, p. 107), neste modelo de programação o dado de entrada de um programa é um conjunto de pares *key-value* e o resultado do mesmo é um novo conjunto de pares *key-value*. Ao utilizar esse modelo de programação o desenvolvedor é responsável somente pela programação das funções *map* e *reduce*.

A função *map* é escrita pelo usuário e recebe um único par *key-value* e produz um conjunto de pares *key-value* intermediários que são automaticamente agrupados e enviados para a função de *reduce*. A função *reduce* também é escrita pelo usuário e recebe uma chave intermediária e um conjunto de valores gerados durante o *map*, essa função tem como objetivo realizar a união desses valores para produzir pares *key-value* de saída. (DEAN; GHEMAWAT, 2008, p. 107)

Um dos propósitos originais do modelo de programação *MapReduce* foi a simpli-

ficação do algoritmo de elaboração da lista dos termos de busca mais utilizados em um período, esse problema pode ser abstraído como um problema de contar a quantidade de ocorrências de cada palavra em um conjunto de documentos. Segundo Dean e Ghemawat (2010, p. 72), para atingir esse objetivo utilizando o modelo *MapReduce* o desenvolvedor deve implementar as funções *map* e *reduce* como apresentadas nos algoritmos Algoritmo 1 e Algoritmo 2.

Algoritmo 1 Algoritmo da função *map*

```
1: function MAP(chave, valor)        ▷ chave: nome do documento, valor: conteúdo
2:     for all palavra in valor do
3:         EmitIntermediate(w, 1);
```

Traduzido de (DEAN; GHEMAWAT, 2010, p. 72)

Algoritmo 2 Algoritmo da função *reduce*

```
1: function REDUCE(chave, valores)   ▷ chave: palavra, valores: lista de quantidades
2:     resultado = 0
3:     for all v in valores do
4:         resultado += v
5:     Emit(resultado);
```

Traduzido de (DEAN; GHEMAWAT, 2010, p. 72)

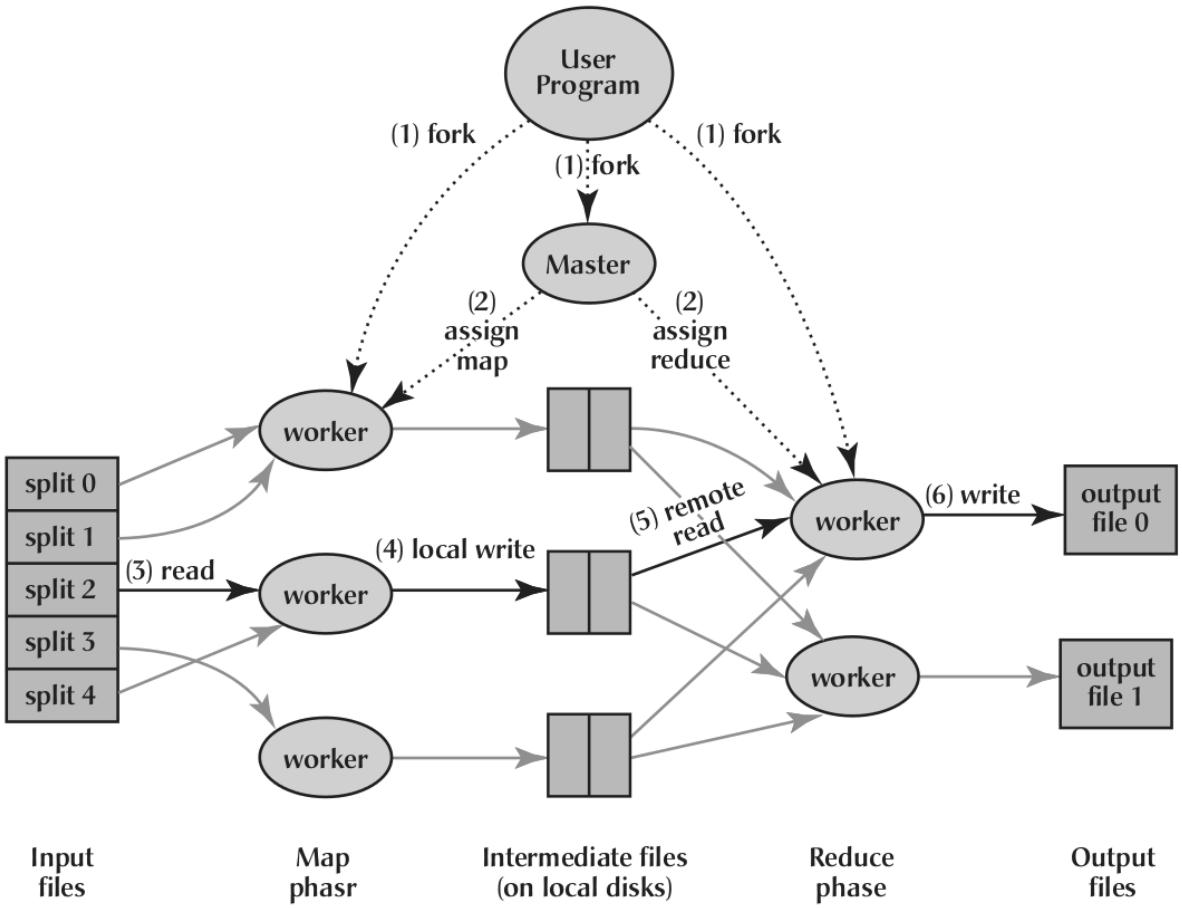
Nesses algoritmos a função *map* emite cada palavra associada com a sua quantidade de ocorrências (no exemplo sempre 1) como um valor intermediário. A função *reduce* por sua vez realiza a soma de todas as quantidades emitidas para uma palavra em particular. O modelo de programação *MapReduce* provê internamente os mecanismos de paralelização necessários a execução dessa operação em *clusters* possibilitando que usuários sem nenhuma experiência em programação distribuída possam utilizar os recursos computacionais disponíveis (DEAN; GHEMAWAT, 2010, p. 72).

Esse modelo de programação, embora simples, abrange uma grande diversidade de tarefas e algoritmos e, apesar de focado no processamento em lote de grandes volumes de dados, pode ser utilizado na realização de análises que exigem grande capacidade computacional como *machine learning* (TAYLOR, 2010, p. 2).

2.3.3 Fluxo de execução

A execução de uma aplicação através do modelo *MapReduce* segue uma ordem padrão constituída de 7 passos. A Figura 32 apresenta esses passos, os números identificados na figura representam as etapas descritas a seguir, essas etapas foram apresentadas por Dean e Ghemawat (2008, p. 108-109):

Figura 32 – Fluxo de execução padrão utilizando *MapReduce*.



Fonte: (DEAN; GHEMAWAT, 2008, p. 18)

- 1. Dividir os arquivos de entrada em M partes, normalmente com 16-64MB por parte e inicia diversas cópias do programa em um *cluster* de computadores.
- 2. Uma das cópias (chamada de *master*) é responsável pelo distribuição das tarefas de *map* e *reduce* através das demais cópias (chamadas de *workers*). Existem M tarefas *map* e R tarefas *reduce* a serem distribuídas.
- 3. Ao receber uma tarefa de *map* uma *thread worker* lê sua parte de dados, converte-a em um conjunto de pares *key-value* e envia a função de *map* escrita pelo usuário. Os valores intermediários produzidos são temporariamente armazenados em memória.
- 4. Periodicamente os valores intermediários que estavam armazenados em memória são escritos em discos e particionados em R regiões. Após a conclusão das tarefas de *map* a localização de cada região é passada a *thread master* que fará a distribuição das tarefas de *reduce* para as *threads workers*.
- 5. Ao receber da *thread master* uma tarefa de *reduce* a *thread worker* lê a partição de

dados que lhe foi atribuída e realiza a ordenação dos dados através de suas chaves, este procedimento é necessário pois normalmente diversas chaves encontram-se na mesma partição.

- 6. A *thread worker* segmenta os dados intermediários criando um grupo para cada chave encontrada. A *thread* então passa a chave e o grupo de dados à função *reduce* escrita pelo usuário. O resultado da função reduce é então escrito em disco.

Após a execução o resultado é disponibilizado em R partes no disco, uma parte para cada partição criada no processo de *reduce*.

2.3.4 Hadoop

Hadoop é um *framework* orientado a objetos escrito em linguagem Java que pode ser instalado em um *cluster* de computadores possivelmente heterogêneos para possibilitar a análise de grandes volumes de dados sem a necessidade de nenhuma alteração de hardware. Em 2008 o Hadoop tornou-se um projeto da *Apache Software Foundation* (TAYLOR, 2010, p. 1).

O projeto Hadoop é composto por diversos componentes dentre os quais citam-se o Hadoop Distributed File System (HDFS), um sistema de arquivos robusto e tolerante a falhas, uma API Java, para desenvolvimento, e o Hadoop Streaming, para possibilitar a execução de códigos em outras linguagens como C e Python além do *framework* para execução de aplicações no paradigma *MapReduce*. Segundo Taylor (2010, p. 1-2), as principais características do *framework* são:

- Proximidade de dados: O *framework* tenta minimizar a distância entre os dados e as tarefas que os utilizaram, isto é, ele tenta criar e distribuir as tarefas de *map* nos nodos onde encontram-se os dados ou, caso não seja possível, no mesmo *rack*. Essa abordagem visa aumentar a performance pois reduz a necessidade de tráfego de dados pela rede.

- Tolerância a falhas: O *framework* cria e gerencia as tarefas de maneira independente, ou seja, um problema com uma das tarefas não afeta as demais. Além disso, como o controle entre as etapas de *map* e *reduce* é feito pelo *framework*, ele detecta automaticamente falhas na execução e reinicia a execução das tarefas defeituosas. O problema nessa abordagem é pertencente ao HDFS, pois o nodo principal do sistema de arquivos é um ponto de falha.

- Confiabilidade: Os dados são replicados através de múltiplos nodos, dispensando a utilização de armazenamento RAID.

- Modelo MapReduce: O *framework* utiliza o modelo MapReduce para o processamento de grandes volumes de dados armazenados no sistema de arquivos HDFS.

Existem entretanto alguns problemas na utilização do *framework* devido ao HDFS. Segundo Taylor (2010, p. 2) os principais são:

- O gerenciamento atualizações de registros é realizado de forma menos eficiente que bancos de dados relacionais;
- Não é possível montar o sistema de arquivos diretamente no disco sendo necessário utilizar a API para acessá-los.

3 O FRAMEWORK

Este capítulo apresenta o *framework* desenvolvido. A seção 3.1 apresenta o processo de análise de domínio realizado para o seu desenvolvimento. Suas funcionalidades e fluxo de execução são apresentados na seção 3.2. Na seção 3.3 é apresentada sua modelagem utilizando UML, nesta modelagem estão inclusos os diagramas de atividades, sequência, classes e pacotes.

3.1 ANÁLISE DE DOMÍNIO

A análise de domínio para o desenvolvimento do *framework* foi iniciada pela revisão bibliográfica de material técnico sobre o assunto. Dentre os materiais pesquisados encontravam-se algoritmos genéticos simples (HOLLAND, 1975; GREFENSTETTE, 1986; FORREST, 1993; SANTARELLI et al., 2006; ROEVA et al., 2012), multiobjetivos (MEUNIER et al., 2000; BOLCHINI et al., 2010) e com população variável (ARABAS et al., 1994; HARIK; LOBO, 1999; BÄCK et al., 2000; EIBEN et al., 2004). Outra fonte de pesquisa foram outros *frameworks* para a execução de GAs (CHUANG; WU, 2000; ESCUELA et al., 2007). Analisando os diferentes GAs mencionados acima é possível identificar uma estrutura fixa composta por etapas de inicialização, seleção e reprodução. O resultado da análise de domínio foi o projeto do *framework* que é, por definição, um modelo de domínio.

3.2 FUNCIONALIDADES E FLUXO DE EXECUÇÃO

Um algoritmo genético possui como funcionalidade básica a otimização da solução de um problema. Esta ação entretanto é realizada através da execução iterativa de um processo de evolução, composto por etapas de seleção e reprodução. O diagrama de casos de uso exposto na Figura 33 apresenta os casos de uso identificados, os quais são descritos abaixo.

- ***Optimize***: a otimização de um problema é a funcionalidade do GA desenvolvido. Esta etapa inclui a verificação da configuração do algoritmo genético, definição da população inicial e a execução iterativa do processo evolutivo até que uma condição de parada pré estabelecida seja alcançada;

- ***Evolve***: este caso de uso representa a evolução de uma geração (iteração) do algoritmo genético, nesta etapa são aplicados os mecanismos de seleção e reprodução.

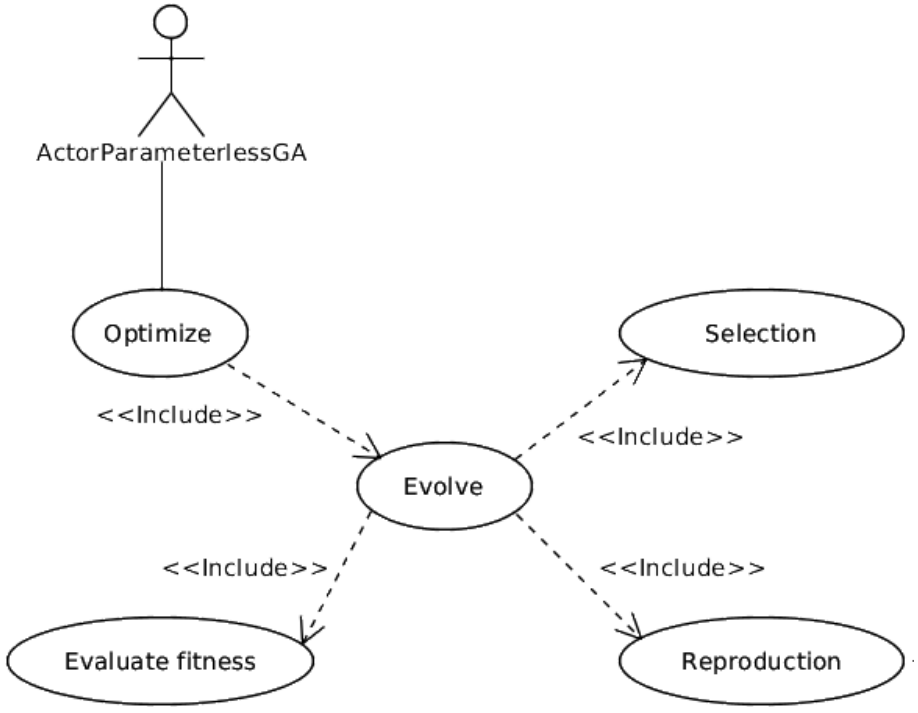
Nesta etapa ocorre também a avaliação do *fitness* dos cromossomos de maneira distribuída utilizando o *framework* Hadoop;

- ***Selection***: este caso de uso consiste na aplicação do mecanismo de seleção (subseção 2.1.3) selecionado pelo usuário no momento da configuração do *framework*;

- ***Reproduction***: este caso de uso consiste no processo de aplicação dos mecanismos de *crossover* (subseção 2.1.4) e mutação (subseção 2.1.5) selecionados pelo usuário no momento da configuração do *framework*;

- ***Evaluate fitness***: este caso de uso representa o processo de distribuição dos elementos necessários ao Hadoop para a realização do cálculo do *fitness* dos cromossomos de maneira distribuída. A função de cálculo do *fitness* deve ser implementada pelo usuário.

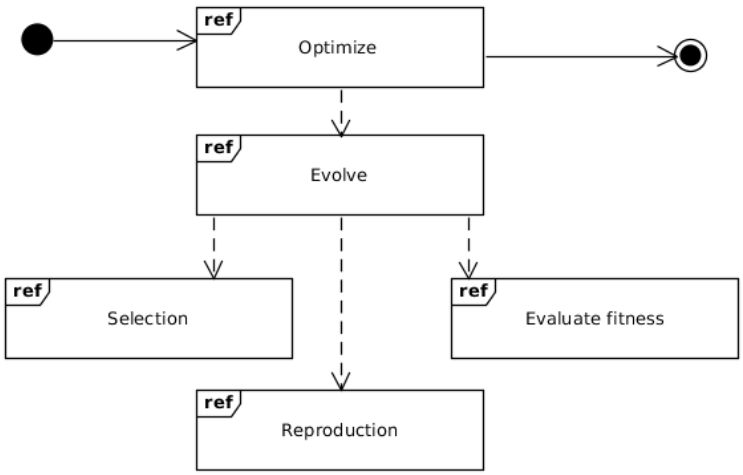
Figura 33 – Diagrama de casos de uso.



A ordem com que os casos de uso do *framework* são executadas é apresentada na Figura 34, sendo que é possível relacionar diretamente as etapas do fluxo de execução com as etapas da execução do algoritmo genético apresentadas na Figura 2[1].

[1] Diferentemente do diagrama apresentado na Figura 2, onde havia um nodo para reprodução e outro para mutação o diagrama de visão geral de iteração possui um único nodo *Reproduction* que engloba estas duas etapas.

Figura 34 – Diagrama de visão geral de interação.



3.3 MODELAGEM

A modelagem do *framework* foi dividida em um conjunto de etapas. Inicialmente foi realizada a análise de domínio para identificação dos elementos em comum entre diversos modelos de algoritmos genéticos. Do modelo gerado, foram extraídas as funcionalidades e o fluxo de execução. A partir do modelo de domínio e das funcionalidades identificadas, iniciou-se a modelagem computacional do *framework*. A modelagem foi desenvolvida utilizando a linguagem UML e contemplando os seguintes requisitos, apresentados por (SILVA, 2009, p. 21):

- modelagem estrutural de sistema;
- modelagem dinâmica de sistema;
- modelagem estrutural de parte;
- modelagem dinâmica de parte.

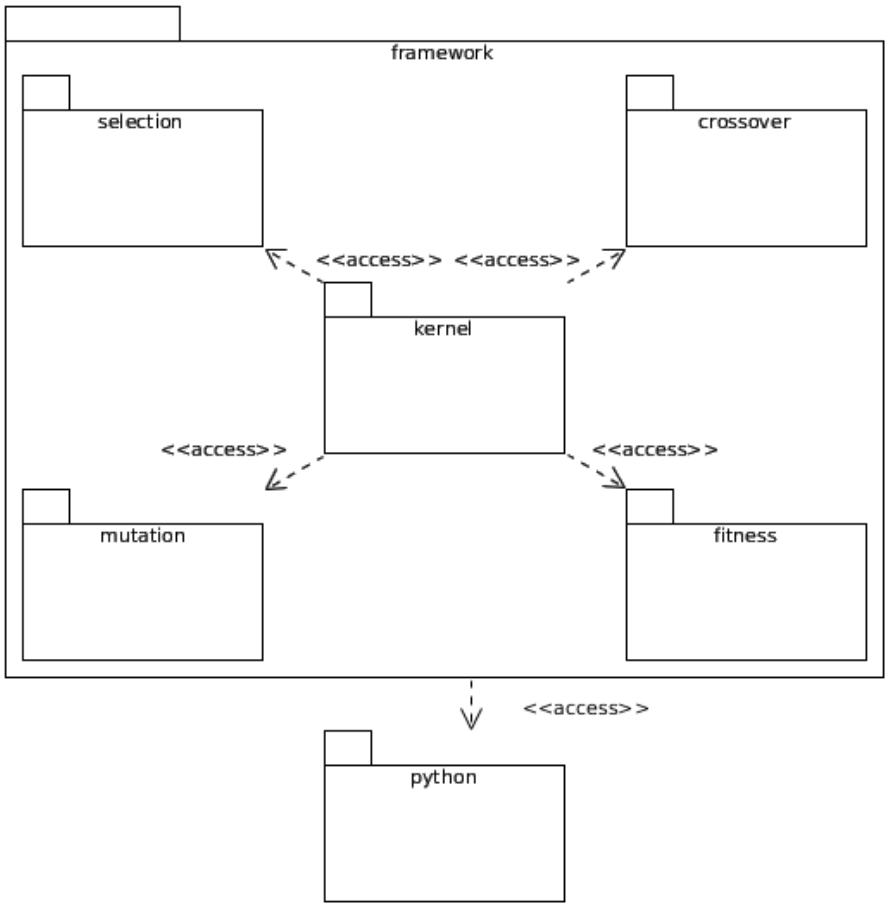
3.3.1 Diagramas de pacotes

Para o desenvolvimento do *framework*, as classes foram distribuídas conforme suas funcionalidades, além disso foi criado um pacote *kernel* que possui as classes básicas como *Chromosome* e *AbstractGeneticAlgorithm*. Todos os pacotes comunicam-se exclusivamente com o pacote *kernel*. Esta organização funcional é apresentada na Figura 35.

3.3.2 Diagramas de classes

Uma vez definidas as funcionalidades e pacotes foram modeladas as classes que compõe o *framework*. A partir da identificação das classes necessárias, realizou-se um

Figura 35 – Diagrama de pacotes.



processo de refinamento dos diagramas visando a extração de interfaces e incremento na utilização de herança, mantendo assim um baixo acoplamento entre classes. Em seguida, as classes foram novamente refinadas para que utilizem padrões de projeto. As próximas seções apresentam as classes identificadas em cada um dos pacotes.

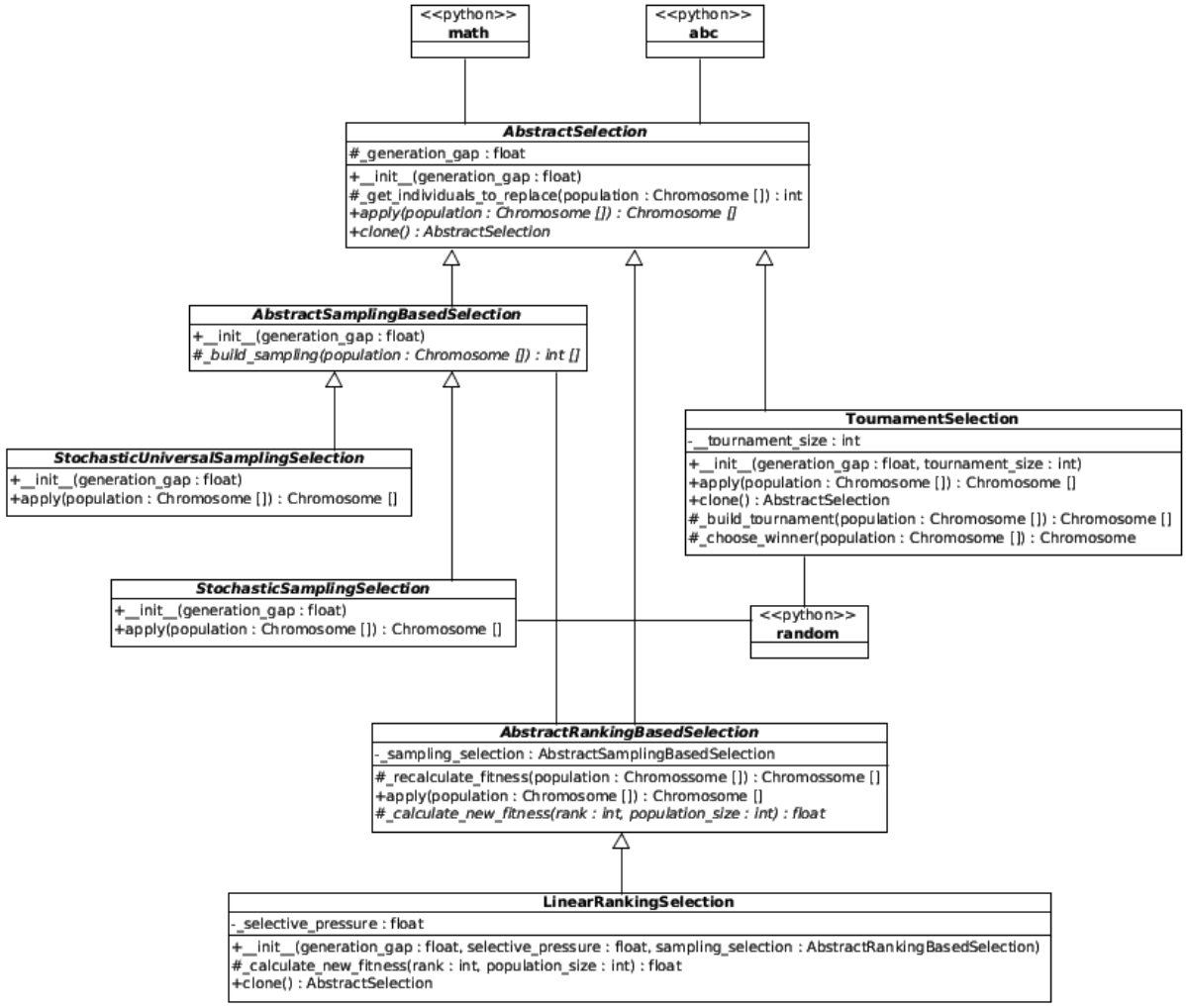
3.3.2.1 Pacote *selection*

Este pacote contém os classes que abstraem os métodos de seleção apresentados durante a revisão bibliográfica. As classes *LinearRankingSelection* e *TournamentSelection* implementam completamente os mecanismos de seleção por ranking linear (subseção 2.1.3.4) e por torneio (subseção 2.1.3.5). Os métodos de seleção por roleta (subseção 2.1.3.2) e amostragem universal estocástica (subseção 2.1.3.3) são respectivamente implementados pelas classes abstratas *StochasticSamplingSelection* e *StochasticUniversalSamplingSelection*, cabendo ao desenvolvedor a implementação do método de construção da roleta. Além dos mecanismos pré implementados, o *framework* possui as classes abstratas *AbstractSelection* e *AbstractRankingBasedSelection* possibilitando a implementadoção de novos métodos de seleção.

O diagrama de classes deste pacote é apresentado pela Figura 36. Todas as classes

apresentadas baseiam-se nos padrões de projeto *Strategy* e *Prototype* estando seus papeis em cada um dos padrões descritos nas Figuras 37 e 38.

Figura 36 – Diagrama de classes do pacote *selection*.



3.3.2.2 Pacote *crossover*

As classes responsáveis pela abstração dos mecanismos de *crossover* estão contidas neste pacote. O mecanismo de *crossover* de um ponto (subseção 2.1.4.1) é implementado pela classe *OnePointCrossover*, o *crossover* de dois pontos (subseção 2.1.4.2) é implementado pela classe *TwoPointCrossover* e o *crossover* uniforme (subseção 2.1.4.3) é implementado pela classe *UniformCrossover*. Assim como no pacote de seleção o pacote de *crossover* também provê uma classe base para a derivação de novos mecanismos de *crossover*.

A organização das classes deste pacote é apresentada na Figura 39, todas as classes contidas nele seguem os padrões de projeto *Strategy* e *Prototype* estando seus papeis em cada um dos padrões descritos nas Figuras 40 e 41.

Figura 37 – Diagrama de estrutura composta do *design pattern strategy* no pacote *selection*.

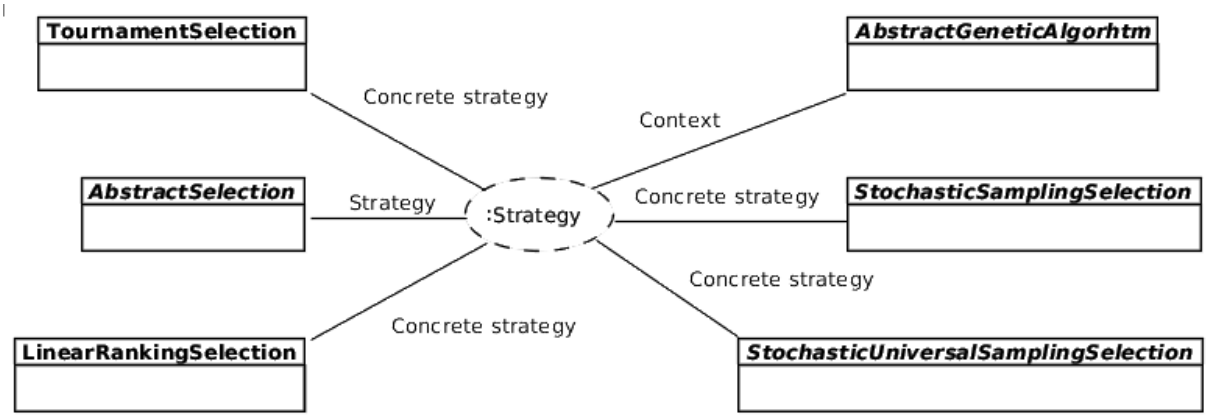
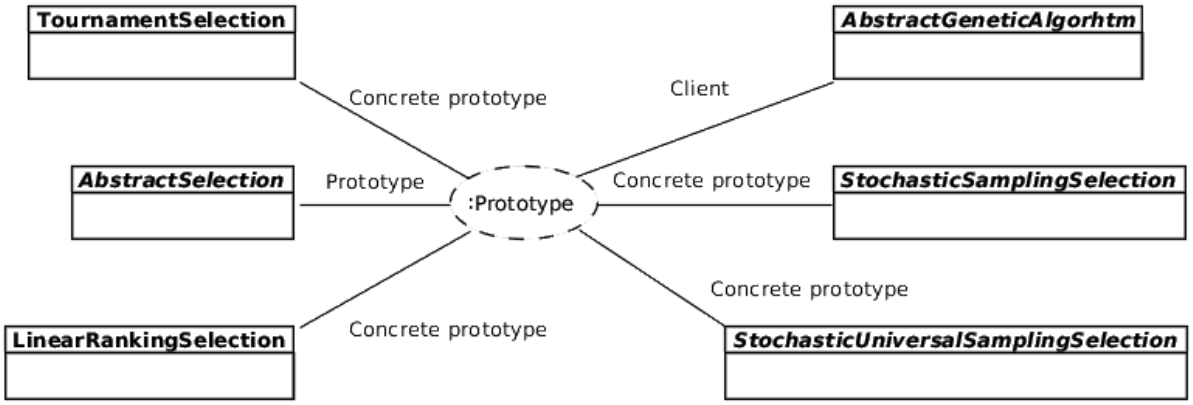


Figura 38 – Diagrama de estrutura composta do *design pattern prototype* no pacote *selection*.



3.3.2.3 Pacote *mutation*

Os mecanismos de mutação encontram-se agrupados no pacote *mutation*. O único mecanismo de mutação nativo pelo *framework* é o *bit-flit* (subseção 2.1.5), implementado pela classe *BitMutation*. Entretanto, assim como nos pacotes de seleção e *crossover*, o *framework* possui uma classe abstrata que possibilita ao desenvolvedor implementar novos mecanismos de mutação.

As classes deste pacote seguem os padrões de projeto *Strategy* e *Prototype*, estando seus papeis em cada um dos padrões descritos nas Figuras 42 e 43, e são apresentadas pelo diagrama da Figura 44.

3.3.2.4 Pacote *fitness*

GAs podem ser utilizados tanto para a identificação de máximos quanto mínimos, além disso podem ser utilizados para a localização de valores específicos em um espaço de

Figura 39 – Diagrama de classes do pacote *crossover*.

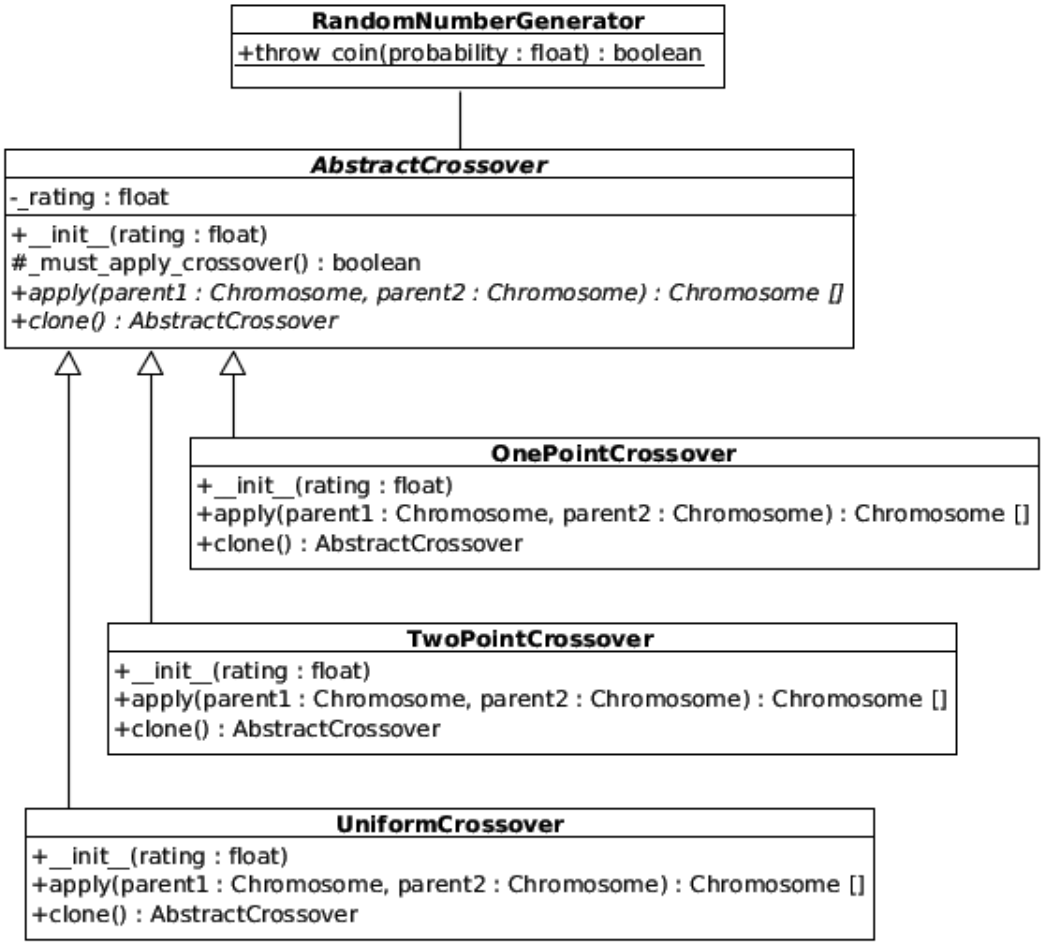


Figura 40 – Diagrama de estrutura composta do *design pattern strategy* no pacote *crossover*.

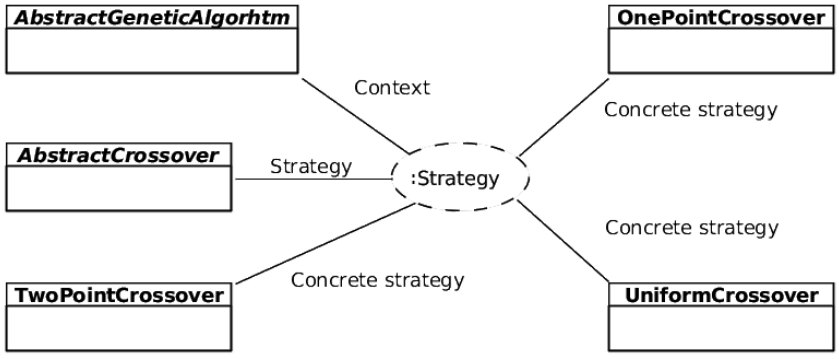


Figura 41 – Diagrama de estrutura composta do *design pattern prototype* no pacote *crossover*.

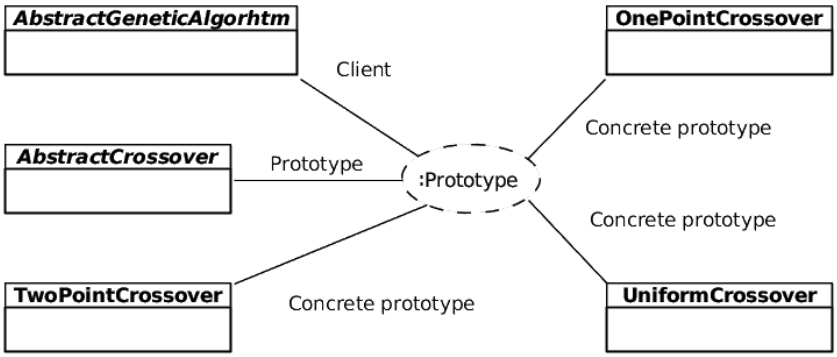


Figura 42 – Diagrama de estrutura composta do *design pattern strategy* no pacote *mutation*.

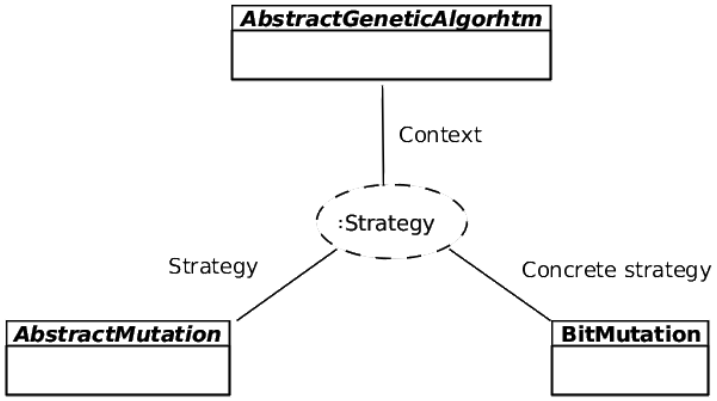


Figura 43 – Diagrama de estrutura composta do *design pattern prototype* no pacote *mutation*.

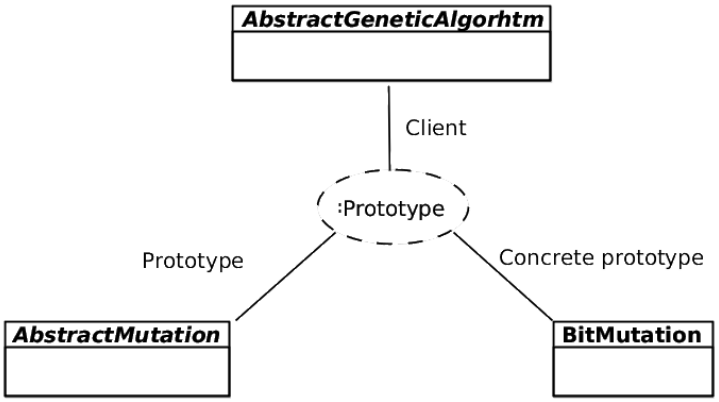
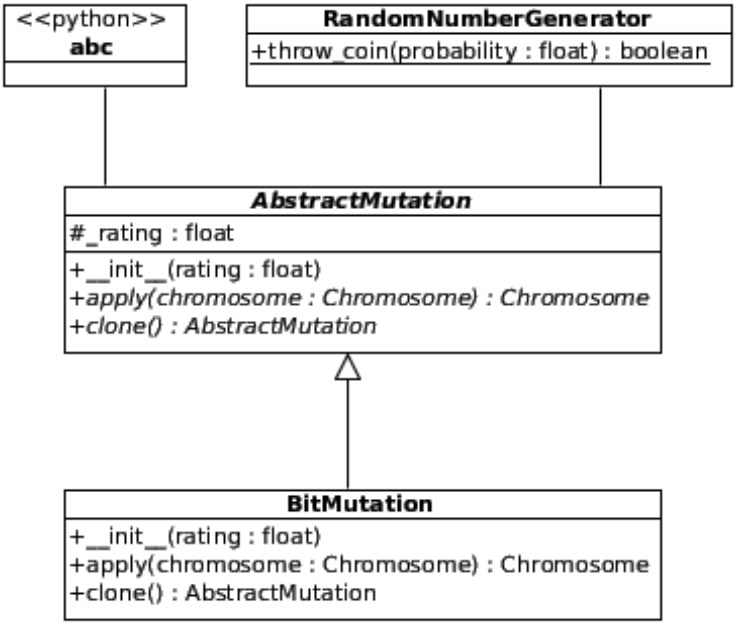


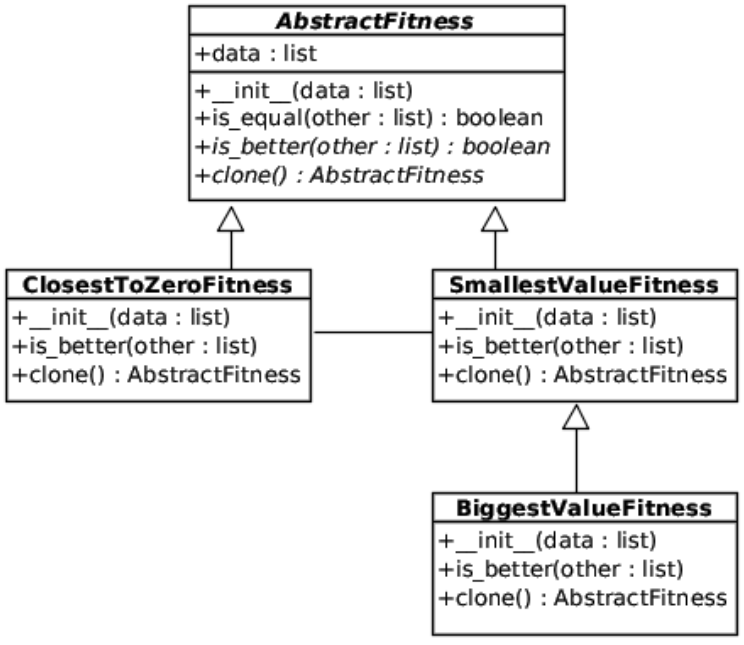
Figura 44 – Diagrama de classes do pacote *mutation*.



busca. Neste pacote encontram-se as classes que abstraem a forma de comparação entre os cromossomos[2], definindo se indivíduos com um *fitness* maior, menor ou mais próximo de zero são considerados melhores.

O diagrama da Figura 45 apresenta as classes deste pacote, as quais seguem os padrões *Strategy* e *Prototype*, conforme papeis apresentados nas Figuras 46 e 47.

Figura 45 – Diagrama de classes do pacote *fitness*.



[2] Internamente a comparação entre cromossomos é realizada utilizando sobrecarga de operadores e métodos de ordenação.

Figura 46 – Diagrama de estrutura composta do *design pattern strategy* no pacote *fitness*.

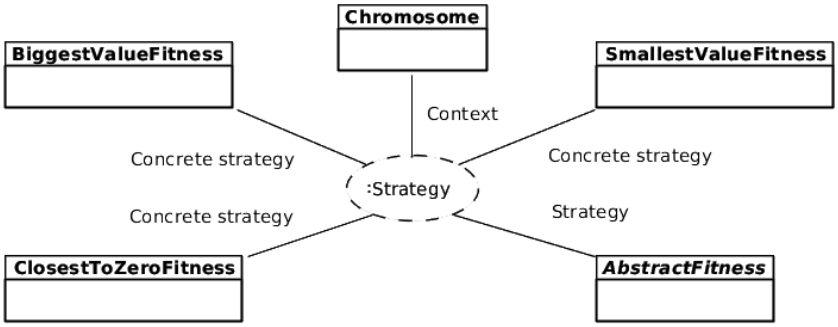
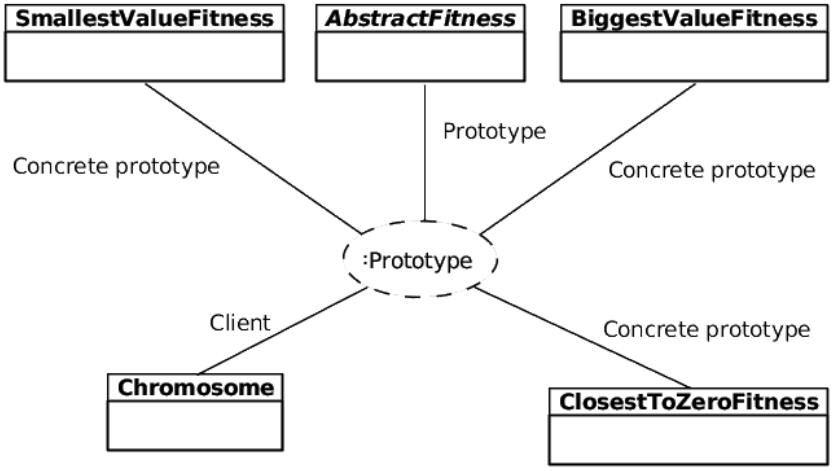


Figura 47 – Diagrama de estrutura composta do *design pattern prototype* no pacote *fitness*.



3.3.2.5 Pacotes *kernel* e *framework*

O pacote *kernel* é o núcleo do *framework*, possuindo classes que abstraem os conceitos de cromossomos e GAs, além de classes auxiliares. A Figura 48 apresenta o diagrama de classes deste pacote. As classes deste pacote seguem os padrões de projeto *Observer* e *Prototype*, estando seus papeis descritos nas Figuras 49 e 50.

Dentre as classes deste pacote encontram-se a classe *GrayCode* que não é utilizada internamente por nenhuma outra, esta classe tem como objetivo pré implementar ao usuário métodos para a conversão de números inteiros e de ponto flutuante para sua notação utilizando o código de *Gray*. O pacote *framework* por sua vez é a base do *framework*, nele estão contidos todos os demais pacotes.

3.3.3 Diagramas de atividades

Os diagramas de atividades foram desenvolvidos para o detalhamento dos casos de uso, sendo especificado um diagrama para cada caso de uso. Dentre os diagramas de atividades desenvolvidos encontram-se os diagramas referentes aos casos de uso *optimize* e *evolve* respectivamente apresentados nas Figuras 51 e 52.

Figura 48 – Diagrama de classes do pacote *kernel*.

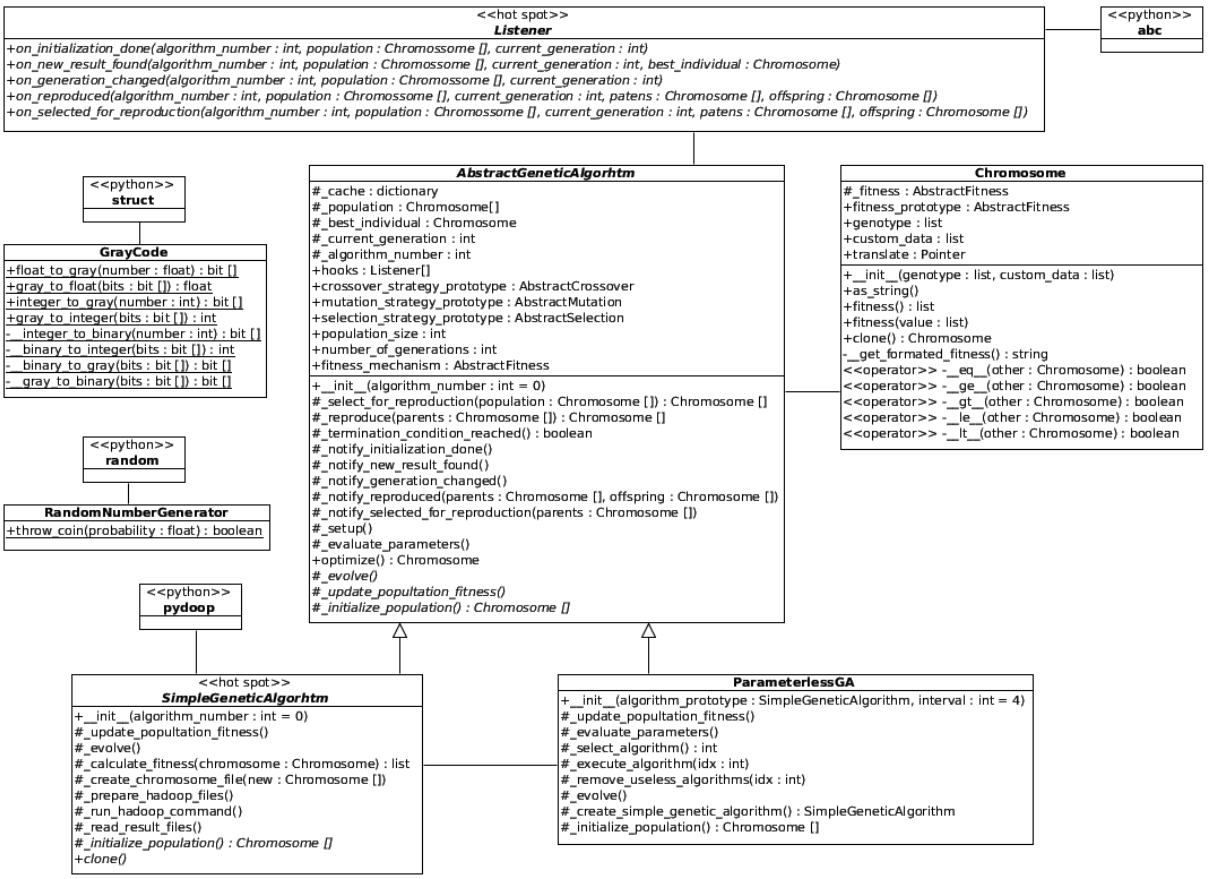


Figura 49 – Diagrama de estrutura composta do *design pattern observer* no pacote *kernel*.

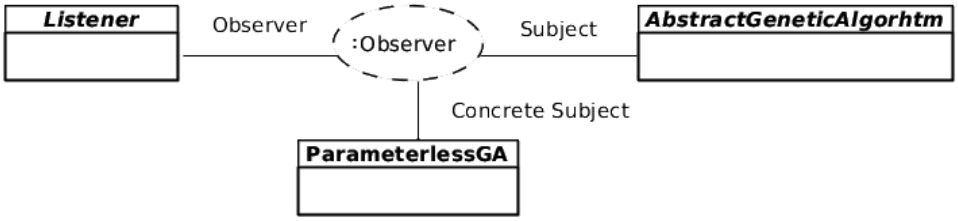


Figura 50 – Diagrama de estrutura composta do *design pattern prototype* no pacote *kernel*.

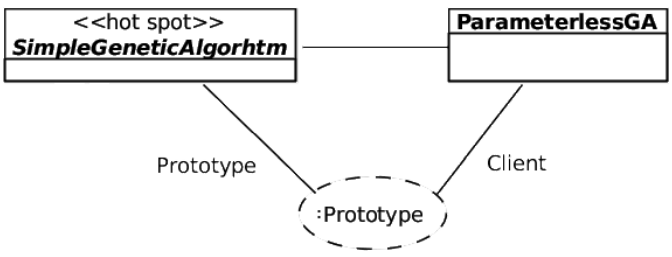
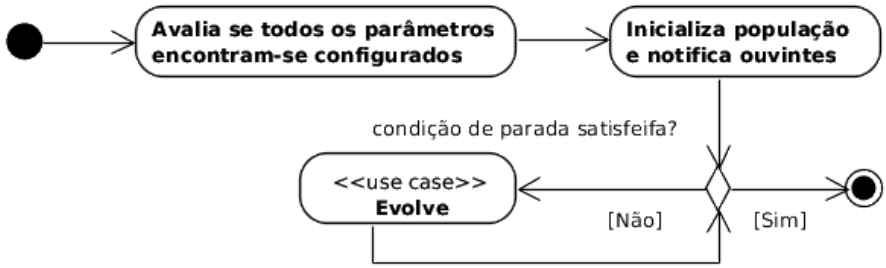
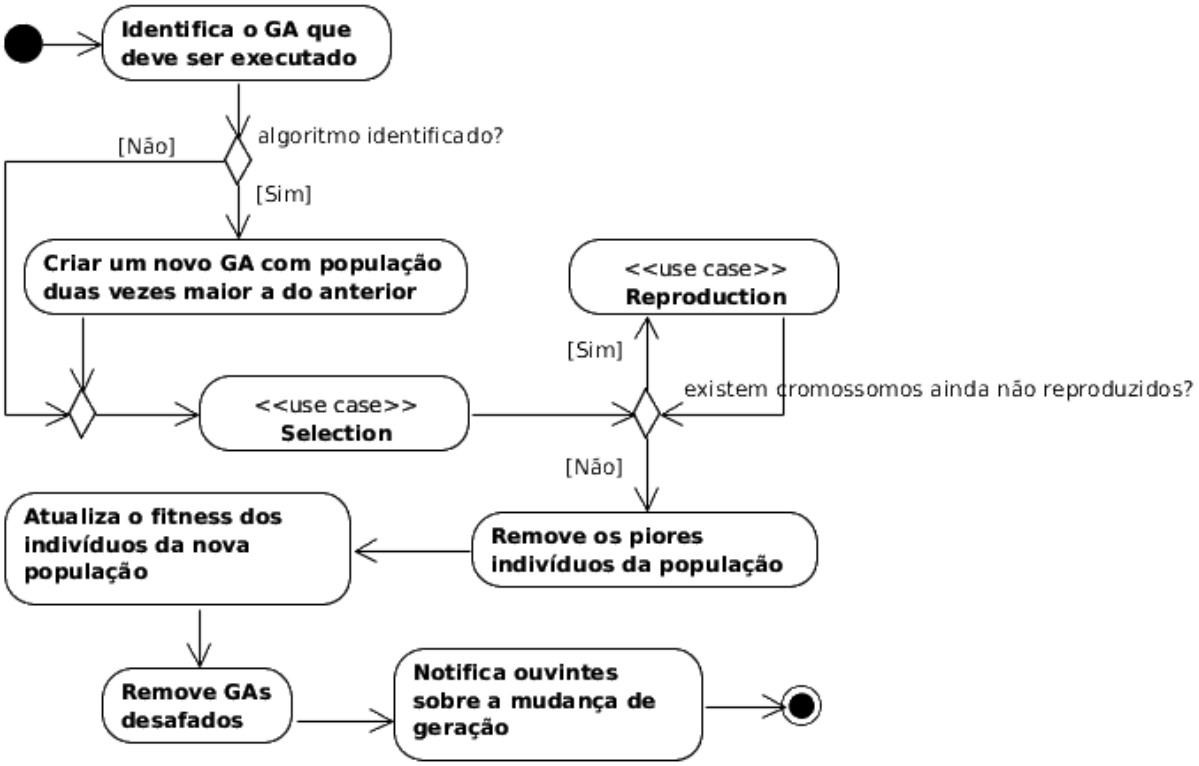


Figura 51 – Diagrama de atividades do caso de uso *optimize*



Além dos diagramas de atividades referentes aos casos de uso *optimize* e *evolve*, foram desenvolvidos também diagramas de atividades referentes as estratégias de seleção, *crossover* e mutação pré implementadas pelo *framework*, estes diagramas, assim como os demais diagramas de atividades referentes aos casos de uso, podem ser visualizados no Apêndices A e B.

Figura 52 – Diagrama de atividades do caso de uso *evolve*

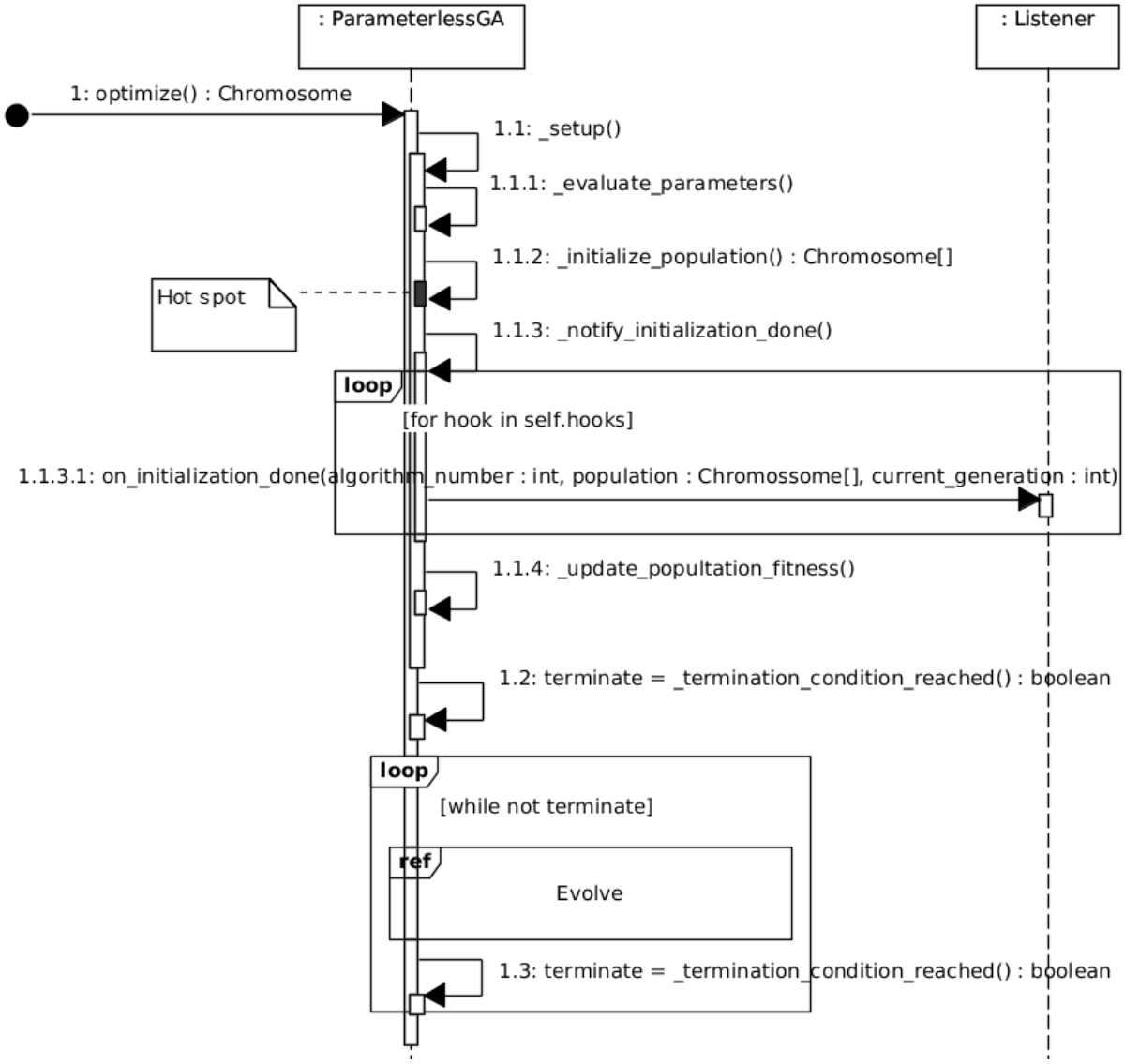


3.3.4 Diagramas de sequência

Para a auxiliar na identificação dos métodos de cada classe, assim como aumentar o grau de detalhamento na especificação dos métodos já identificados foram produzidos os diagramas de sequência. Da mesma forma que para os diagramas de atividade, foi produzido um diagrama de sequência para cada caso de uso. A Figura 53 apresenta o diagrama

de sequência do caso de uso *optimize*. Nesta figura encontra-se também uma anotação *hot spot*, representando um ponto não modelado pelo *framework* uma vez que varia conforme a aplicação a ser desenvolvida. Os demais diagramas de sequência encontram-se nos Apêndices C e D.

Figura 53 – Diagrama de sequência do caso de uso *optimize*



4 APLICAÇÕES DE AVALIAÇÃO

O *framework* foi avaliado através do desenvolvimento de três aplicações distintas, as quais são apresentadas nas próximas seções. O protótipo do *framework* e as aplicações de testes foram implementados utilizando a linguagem Python3 com o pacote PyDoop[1] (LEO; ZANETTI, 2010). As execuções foram realizadas em um *cluster* Hadoop em sua configuração *Single Node*. Para critérios de comparação, cada aplicação foi desenvolvida duas vezes, uma utilizando o código do *framework* e outra implementando todas as funcionalidades manualmente.

4.1 MAXIMIZAÇÃO DE FUNÇÃO

A primeira aplicação desenvolvida foi a maximização de lucro utilizando funções de custo e receita, esta aplicação foi baseada no exercício apresentado em (BUELOW, 2014). As funções de custo e receita deste problema são apresentadas respectivamente nas Equações 4.1 e 4.2.

$$Custo(x) = 16000 + 500x + 1.6x^2 + 0.004x^3 \tag{4.1}$$

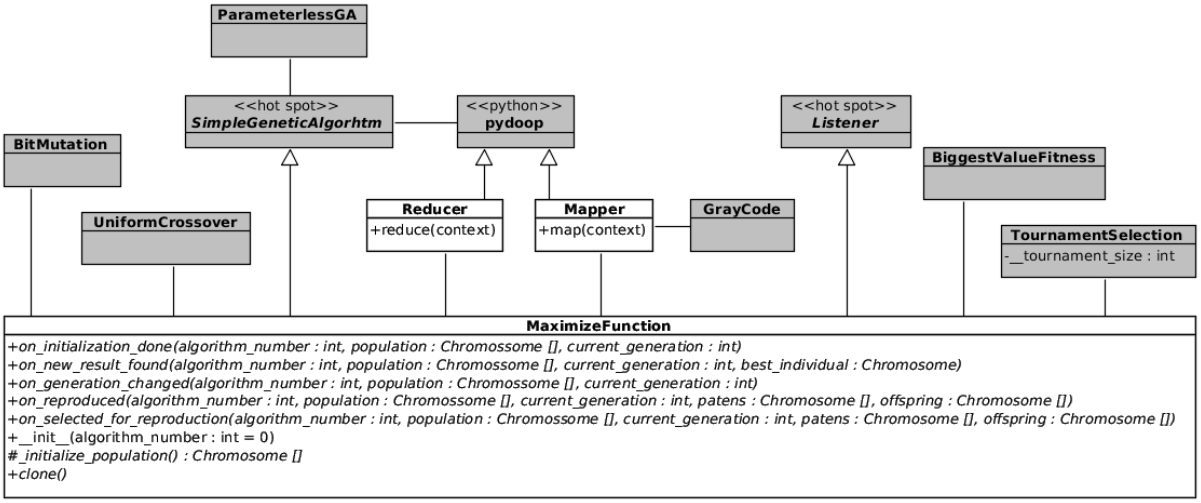
$$Receita(x) = 1700x - 7x^2 \tag{4.2}$$

Embora este problema específico possa ser solucionado através de métodos matemáticos, como *gradient descent* e programação linear, o mesmo serve para ilustrar a otimização de uma função matemática utilizando o *framework*. De maneira análoga a solução desde problema outras funções podem ser otimizadas, mesmo que, por exemplo, as mesmas não sejam diferenciáveis.

Na Figura 54 é apresentado o diagrama de classes da aplicação de testes, nesta imagem as classes reutilizadas do *framework* estão representadas na cor cinza e as classes específicas da aplicação são representadas na cor branca. É possível verificar neste diagrama que 9 classes do *framework*[2] são utilizadas, entre elas as estratégias de seleção, mutação e torneio pré implementadas, cabendo ao desenvolvedor apenas a implementação das classes *MaximizeFunction*, *Mapper* e *Reducer*. Neste problema o genótipo é uma *string* binária de 32 bits, utilizando *Gray code*, que representa um número inteiro.

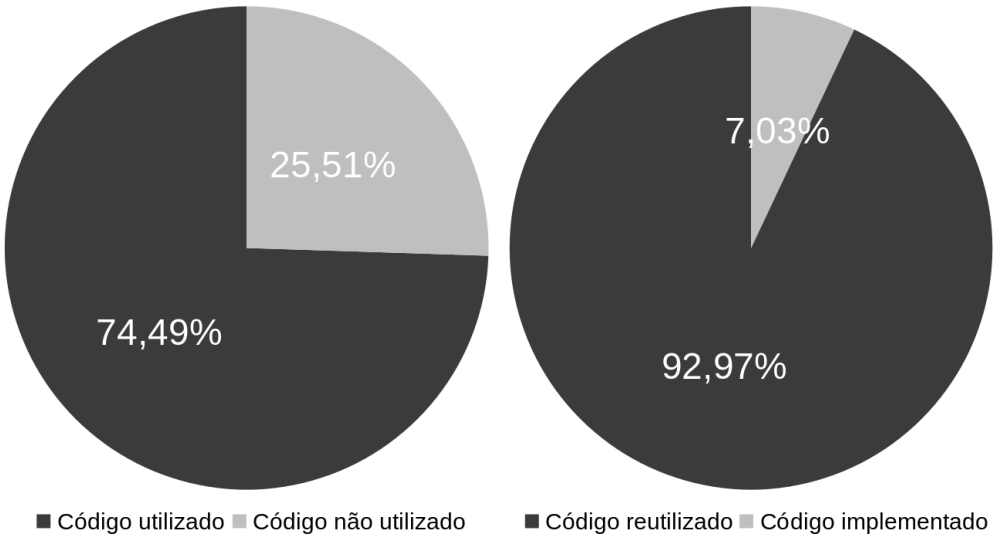
[1] PyDoop é uma API para utilização do Hadoop MapReduce e HDFS através da linguagem Python.
[2] Nas classes do *framework* estão inclusas as classes da biblioteca PyDoop acessada internamente.

Figura 54 – Diagrama de classes da aplicação de testes que maximiza uma função de lucro.



Analisando quantitativamente o reuso proporcionado pelo *framework*, foi observado que esta aplicação reutiliza 1282 linhas de código do *framework*, cabendo ao desenvolvedor implementar 97 linhas de código para produzir a aplicação. Sem a utilização do *framework* foi possível desenvolver a mesma aplicação com 873 linhas de código, entretanto desta forma não foram reutilizadas linhas de código, implicando em um acréscimo de 800% na quantidade de linhas de código implementadas. A Figura 55(a) apresenta o percentual de linhas de código do *framework* que são utilizadas nesta aplicação e a Figura 55(b) apresenta a o percentual de linhas de código implementadas pelo desenvolvedor.

Figura 55 – Linhas de código reutilizadas na aplicação de maximização de função.



(a) Percentual de utilização do *framework* (b) Percentual de implementação

4.2 IDENTIFICAÇÃO DE PARÂMETROS EM ADIÇÃO DE IMAGENS

A segunda aplicação de validação foi projetada para verificar os benefícios da utilização do *framework* Hadoop, esta aplicação consiste na identificação dos valores utilizados em um procedimento de adição de imagens.

A adição é um processamento realizado sobre um conjunto de imagens para a criação de uma nova imagem composta deste conjunto, nesta operação o valor de cada pixel de uma imagem é adicionado ao valor do pixel equivalente em outra imagem. Dependendo do conjunto de imagens utilizadas a imagem resultante pode ser uma combinação de duas imagens totalmente diferentes, ou uma adição de várias imagens semelhantes tiradas em tempos distintos.

A partir da operação de adição o problema a ser resolvido é identificar o percentual de cada imagem utilizado na composição da imagem resultante, tendo como dados de entrada somente as imagens de base e a imagem resultante[3]. Neste problema o genótipo dos cromossomos é tratado como uma *string* binária de 128 bits, utilizando *Gray code*, representando dois números de ponto flutuante, com 64 bits cada, cujos valores encontram-se no intervalo (0, 1). A Figura 56 apresenta o diagrama de classes desta aplicação de testes, nesta figura as classes reutilizadas do *framework* são apresentadas em cinza e as classes implementadas pelo usuário são apresentadas em azul.

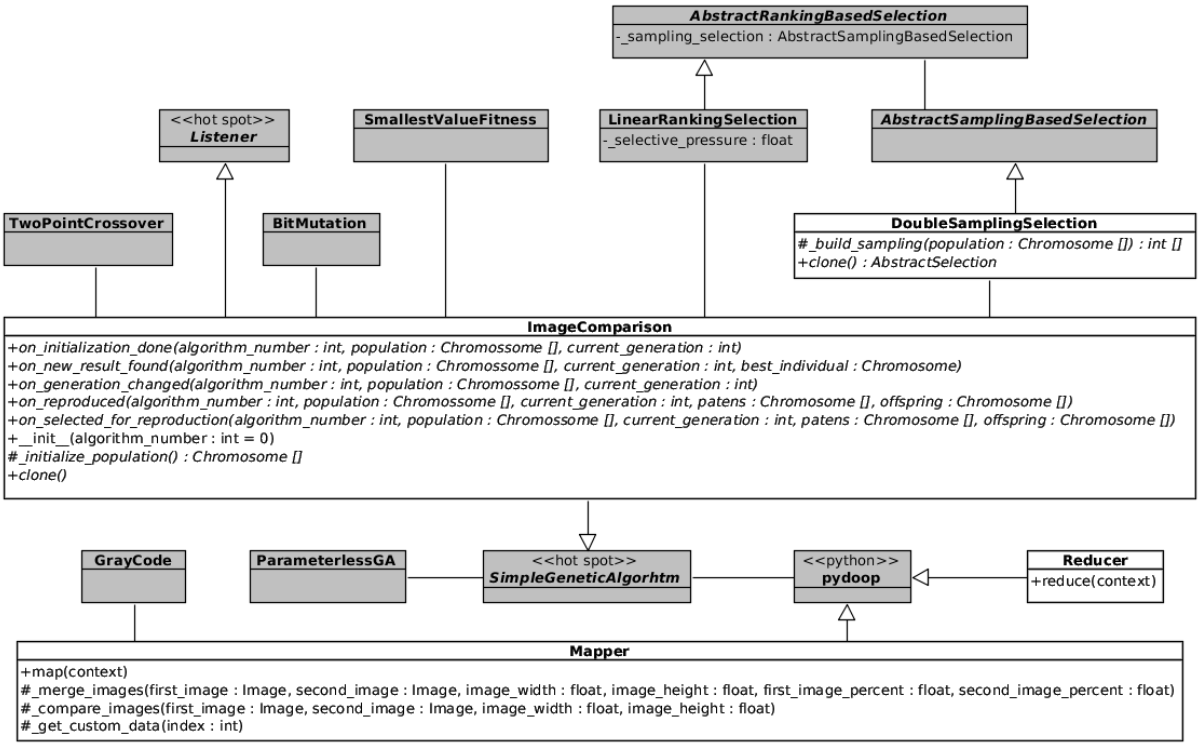
Analisando quantitativamente o reuso proporcionado pelo *framework* foi observado que esta aplicação reutiliza 1301 linhas de código do *framework*, cabendo ao desenvolvedor implementar 249 linhas de código. Na aplicação desenvolvida sem a utilização do *framework* houve um acréscimo de 300,40% na quantidade de linhas de código implementadas pelo desenvolvedor, resultando no desenvolvimento de 997 linhas de código. As Figuras 58(a) e 58(b) apresentam respectivamente o percentual de linhas de código do *framework* que são utilizadas nesta aplicação e o percentual de linhas de código implementadas pelo desenvolvedor.

4.3 OTIMIZAÇÃO DE SEGMENTAÇÃO PLANIMÉTRICA DE RODOVIAS

Informações sobre a geometria de uma rodovia são necessárias para a realização de estudos na área de operação e planejamento de transporte rodoviário. Embora tais informações tenham origem em projetos geométricos, elaborados por diferentes entida-

[3] Devido a limitações do PyDoop quanto a passagem de objetos complexos durante operações de MapReduce é necessário realizar o upload dos arquivos de dados para o sistema de arquivos HDFS previamente.

Figura 56 – Diagrama de classes da aplicação de testes de comparação de adição de imagens.



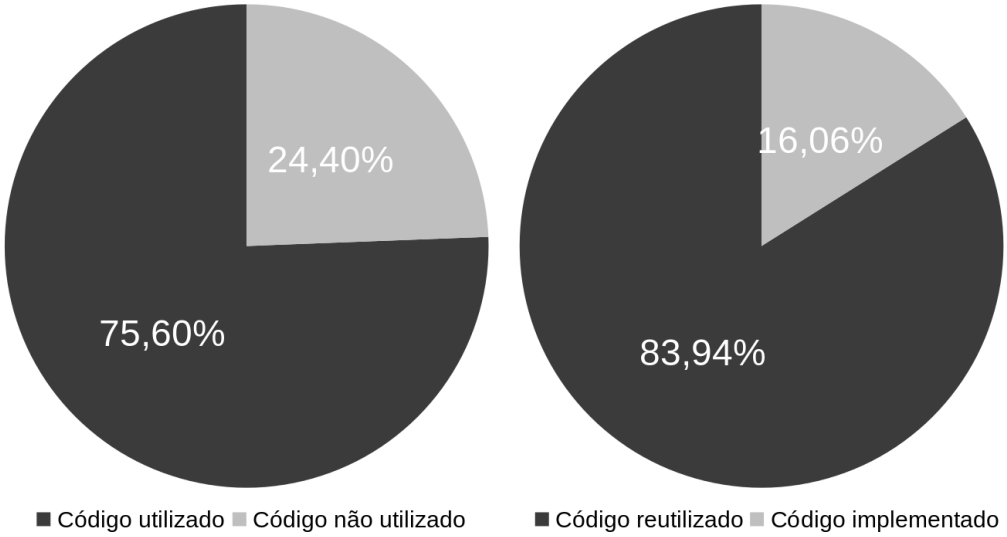
des, grande parte destas informações está, de forma aproximada, intrinsecamente contida em poligonais geográficas que são criadas durante o levantamento da malha federal utilizando GPS. Estas poligonais geográficas podem ser submetidas a um processamento de segmentação e classificação, para identificação de segmentos horizontais. Dos segmentos em curva podem ser automaticamente extraídos os valores dos raios e ângulos centrais, de posse desses valores é possível calcular o índice de geometria horizontal (IGH) utilizado pelo HDM-4 (WORLD ROAD ASSOCIATION, 2000).

Um dos métodos de segmentação é apresentado em (WORRAL; NEBOT, 2007), este método segmenta as rodovias em tangentes e curvas circulares[4] através de um processamento utilizando 3 parâmetros, um *coarse threshold*, para remoção de ruído, um *fine threshold*, para identificação das curvas, e um α limite para classificar um segmento como tangentes e curvas circulares. Os autores afirmam entretanto que os valores ideais destes parâmetros devem ser otimizados para cada poligonal.

Para a representação deste problema o genótipo do cromossomo é uma *string* binária, utilizando *Gray code*, com 192 bits, representando três números de pontos flutuante com 64 bits cada. Como função de *fitness* é utilizado o Linear Matching Model (SHEHU; DIKA, 2012) que mede a dissimilaridade entre dois conjuntos de pontos, quanto menor o valor de dissimilaridade melhor foi a segmentação realizada. A Figura 58 apresenta o

[4] Em projeto rodoviários existem também as curvas de transição modeladas por clotóides.

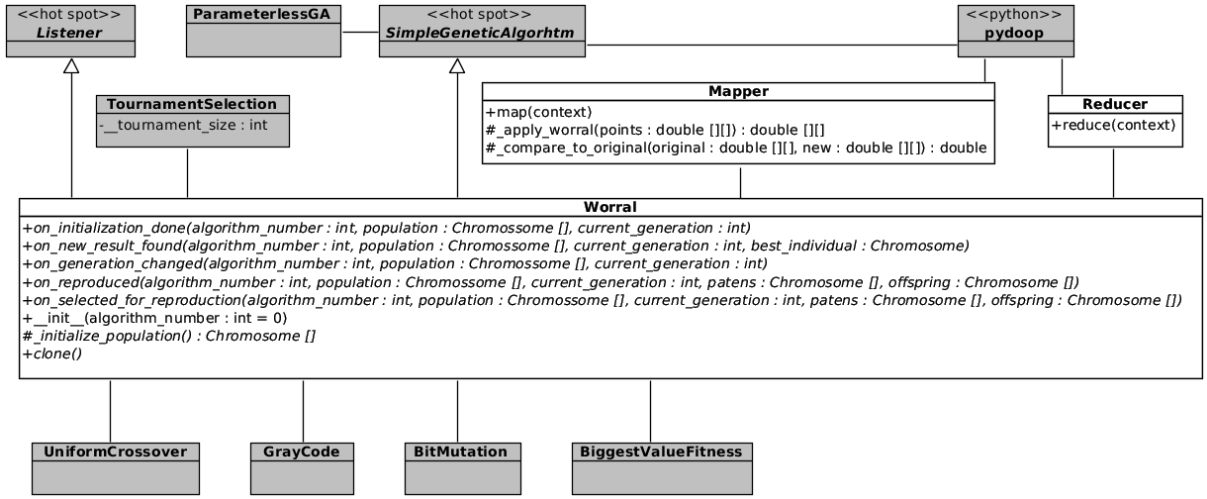
Figura 57 – Linhas de código reutilizadas na aplicação de identificação de parâmetros em adição de imagens.



(a) Percentual de utilização do *framework* (b) Percentual de implementação

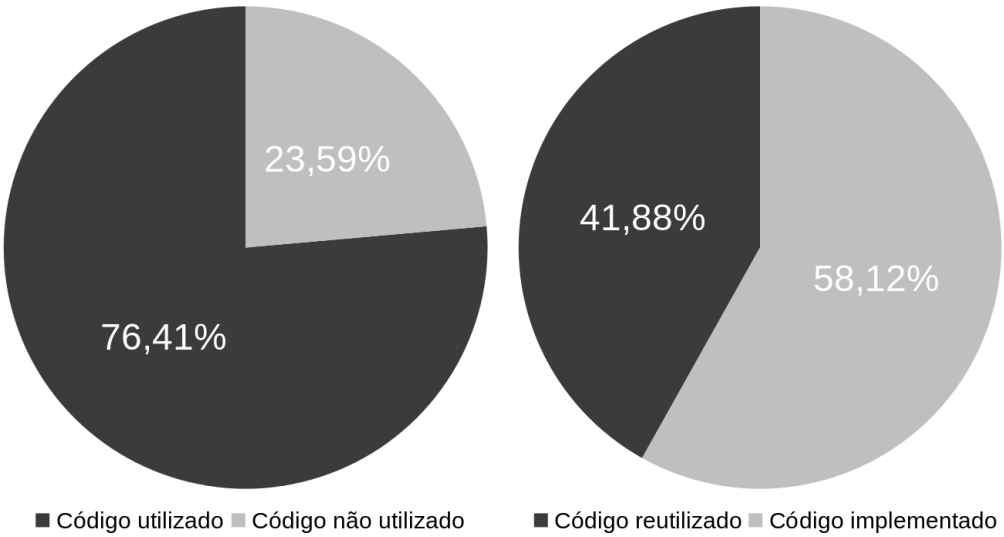
diagrama das classes produzidas para a implementação deste método, em cinza estão as classes reutilizadas do *framework* e em azul as classes implementadas pelo usuário.

Figura 58 – Diagrama de classes da aplicação de testes de segmentação de poligonais.



O desenvolvimento desta aplicação sem a utilização do *framework* necessitou da escrita de 2694 linhas de código. Ao utilizar o *framework*, foram reutilizadas 1315 linhas de código, cabendo ao desenvolvedor implementar apenas 1825 linhas de código, uma redução de 32,26%. A Figura 60(a) apresenta o percentual de linhas de código do *framework* que são utilizadas nesta aplicação e a Figura 60(b) apresenta a o percentual de linhas de código implementadas pelo desenvolvedor.

Figura 59 – Linhas de código reutilizadas na aplicação de otimização de segmentação planimétrica.



(a) Percentual de utilização do *framework* (b) Percentual de implementação

5 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Com a popularização de ambientes multi-computados e dos modelos de negócio de software como serviço (SaaS), plataforma como serviço (PaaS) e infraestrutura como serviço (IaaS) é possível alugar grandes *clusters*. Visando beneficiar-se dessa facilidade na utilização de grandes quantidades recursos computacionais para a otimização de problemas, o presente trabalho apresentou alguns modelos de GAs presentes na literatura, os quais foram utilizados no processo de análise de domínio. A partir do modelo de domínio produzido, foi elaborada a modelagem de um *framework* orientado a objetos, utilizando UML2, visando proporcionar o reuso de artefatos de software como classes e modelagem. Por último foi realizada sua implementação utilizando a linguagem Python e a biblioteca PyDoop, mantendo correspondência direta entre a modelagem e o código.

De forma a avaliar se o *framework* cumpre com a função proposta foram implementadas três aplicações de avaliação, uma maximização de função e duas minimizações de erro, uma utilizando um único parâmetro e outra utilizando múltiplos parâmetros. Em todos os casos foi verificado que as aplicações obtiveram os mesmos resultados. Entretanto com a utilização do *framework* foi possível reutilizar, em média, 75% das classes necessárias à solução dos problemas, implicando em uma redução mínima de 32,26% na quantidade de linhas de código implementadas pelo desenvolvedor.

Durante a produção das aplicações de avaliação foi constatado que a escolha de uma arquitetura de caixa cinza mostrou-se adequada, possibilitando ao *framework* pré implementar mecanismos tradicionais de reprodução, seleção e *crossover*, porém permitindo que o desenvolvedor produza novos mecanismos conforme sua necessidade. Nesta etapa também foi verificado que a utilização do Hadoop, através da biblioteca PyDoop, mostrou-se prejudicial à performance da aplicação. Esse fato é decorrente do tamanho do *cluster* Hadoop e da complexidade do problema a ser otimizado.

5.1 TRABALHOS FUTUROS

Existem diversos aspectos que podem ser aprimorados na atual implementação do *framework* de forma a torna-lo mais maduro e adequado a uso. Dentre os aprimoramentos possíveis incluem-se:

- Verificar os benefícios da implementação do *framework* em Java, visando reduzindo o *overhead* causado pela utilização da biblioteca PyDoop;

- Avaliar a utilização do *framework* em problemas com alto custo computacional e em *clusters* de tamanhos variados, visando identificar o ponto a partir do qual a utilização do Hadoop torna-se vantajosa;
- Adicionar suporte a execução de GAs multiobjetivos como, por exemplo, NSGA-II.

REFERÊNCIAS

AARSTEN, A.; BRUGALI, D.; MENGA, G. Designing concurrent and distributed control systems. *Communications of the ACM*, v. 39, n. 10, p. 50–58, Outubro 1996. Disponível em: <http://doi.acm.org/10.1145/236156.236168>. Acesso em: 18.3.2014. Citado na página 53.

AGRAWAL, A. F. Sexual selection and the maintenance of sexual reproduction. *Nature*, v. 411, p. 692–695, Junho 2001. Disponível em: <http://dx.doi.org/10.1038/35079590>. Acesso em: 11.11.2013. Citado na página 39.

ANGELOVA, M.; TZONKOV, S.; PENCHEVA, T. Genetic algorithms based parameter identification of yeast fed-batch cultivation. In: *Numerical Methods and Applications*. Springer Berlin Heidelberg, 2011. v. 6046, p. 224–231. Disponível em: <http://link.springer.com/chapter/10.1007%2F978-3-642-18466-6_26>. Acesso em: 25.9.2013. Citado na página 20.

ARABAS, J.; MICHALEWICZ, Z.; MULAWKA, J. Gavaps - a genetic algorithm with varying population size. In: *Proceedings of the First IEEE Conference on Evolutionary Computation*. Orlando, FL: [s.n.], 1994. (999, v. 1), p. 73–78. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=350039>. Acesso em: 30.8.2013. Citado 5 vezes nas páginas 17, 26, 43, 44 e 61.

ARANGO, G. Domain analysis: From art form to engineering discipline. In: *Proceedings of the 5th International Workshop on Software Specification and Design*. New York, NY, USA: ACM, 1989. (IWSSD '89), p. 152–159. ISBN 0-89791-305-1. Disponível em: <http://doi.acm.org/10.1145/75199.75224>. Acesso em: 25.02.2014. Citado na página 50.

ARANGO, G. Domain analysis methods. In: *Workshop on Software Acrchitecture*. USC Center for Software Engineering, Los Angeles: [s.n.], 1994. Citado 2 vezes nas páginas 51 e 52.

BABAOGLU, O.; CANRIGHT, G.; DEUTSCH, A.; DUCATELLE, F.; GAMBARDELLA, L. M.; GANGULY, N.; JELASITY, M.; MONTEMANNI, R.; MONTRESOR, A.; URNES, T. Design patterns from biology for distributed computing. *ACM Transactions on Autonomous and Adaptive Systems*, v. 1, n. 1, p. 26–66, Setembro 2006. Disponível em: <http://dl.acm.org/citation.cfm?id=1152937>. Acesso em: 26.8.2013. Citado na página 53.

BÄCK, T.; EIBEN, A. E.; VAART, N. A. L. van der. An empirical study on gas "without parameters". In: *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*. [s.n.], 2000. (PPSN VI), p. 315–324. Disponível em: <http://dl.acm.org/citation.cfm?id=645825.669092>. Acesso em: 26.9.2013. Citado 4 vezes nas páginas 17, 26, 44 e 61.

BAKIRTZIS, A.; PETRIDIS, V.; KAZARLIS, S. Genetic algorithm solution to the economic dispatch problem. *IEE Proceedings - Generation, Transmission and Distribution*, v. 141, p. 377–382, July 1994. ISSN 1350-2360. Disponível em:

<http://digital-library.theiet.org/content/journals/10.1049/ip-gtd_19941211>. Acesso em: 11.4.2014. Citado na página 20.

BARRETO JUNIOR, C. G. *Agregando frameworks de infra-estrutura em uma aplicação baseada em componentes: Um estudo de caso no amibente AulaNet*. Dissertação (Mestrado) — Pontifícia Universidade Católica do Rio de Janeiro, Março 2006. Disponível em: <http://groupware.les.inf.puc-rio.br/public/papers/dissertacaoBarreto/DissertacaoBarreto.pdf>. Acesso em: 19.2.2014. Citado na página 46.

BAUER, C.; KING, G. *Java Persistence with Hibernate*. Greenwich, CT, USA: Manning Publications Co., 2006. ISBN 1932394885. Citado na página 49.

BEASLEY, D.; BULL, D. R.; MARTIN, R. R. An overview of genetic algorithms: Part 1, fundamentals. *University Computing*, v. 15, p. 58–69, 1993. Disponível em: <http://www.geocities.ws/francorbusetti/gabeasley1.pdf>. Acesso em: 22.8.2013. Citado 6 vezes nas páginas 23, 25, 30, 39, 40 e 43.

BEASLEY, D.; BULL, D. R.; MARTIN, R. R. An overview of genetic algorithms: Part 2, research topics. *University Computing*, v. 15, n. 4, p. 170–181, 1993. Disponível em: <http://ralph.cs.cf.ac.uk/papers/GAs/ga_overview2.pdf>. Acesso em: 22.8.2013. Citado 2 vezes nas páginas 41 e 42.

BEYER, H.-G.; SCHWEFEL, H.-P. Evolution strategies – a comprehensive introduction. *Natural Computing*, v. 1, p. 3–52, 2002. Disponível em: <http://link.springer.com/article/10.1023/A:1015059928466>. Acesso em: 2.9.2013. Citado 2 vezes nas páginas 30 e 31.

BEYERA, H.-G.; SENDHOFF, B. Robust optimization – a comprehensive survey. *Computer Methods in Applied Mechanics and Engineering*, v. 196, p. 3190–3218, Julho 2007. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0045782507001259>. Acesso em: 27.8.2013. Citado na página 42.

BLICKLE, T.; THIELE, L. *A comparison of selection schemes used in Genetic Algorithms*. [S.l.], 1995. Disponível em: <ftp://129.132.2.212/pub/publications/TIK-Report11.pdf>. Acesso em: 23.8.2013. Citado 2 vezes nas páginas 31 e 35.

BOLCHINI, C.; LANZI, P. L.; MIELE, A. A multi-objective genetic algorithm framework for design space exploration of reliable fpga-based systems. In: *IEEE Congress on Evolutionary Computation*. Barcelona: [s.n.], 2010. p. 1–8. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5586376>. Acesso em: 22.8.2013. Citado 2 vezes nas páginas 19 e 61.

BOOCH, G. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12, p. 211–221, 1986. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6312937>. Acesso em: 22.8.2013. Citado na página 48.

BOOKER, L.; GOLDBERG, D.; HOLLAND, J. Classifier systems and genetic algorithms. *Artificial Intelligence*, v. 40, n. 1-3, p. 235–282, Setembro 1989. Disponível em: <http://www.sciencedirect.com/science/article/pii/0004370289900507>. Acesso em: 22.8.2013. Citado na página 27.

BOONE, G.; CHIANG, H.-D. Optimal capacitor placement in distribution systems by genetic algorithm. *International Journal of Electrical Power & Energy Systems*, v. 15, n. 3, p. 155–161, Junho 1993. Disponível em: <http://www.sciencedirect.com/science/article/pii/014206159390030Q>. Acesso em: 11.4.2014. Citado na página 20.

BOSCH, J.; MOLIN, P.; MATTSSON, M.; BENGTSSON, P. Object-oriented framework-based software development: problems and experiences. *ACM Computing Surveys*, v. 32, p. 3–8, Março 2000. Disponível em: <http://dl.acm.org/citation.cfm?id=351939>. Acesso em: 25.8.2013. Citado na página 47.

BUELOW, R. *Sample Problems, Lesson 4.7*. 2014. Disponível em: <http://faculty.wlc.edu/buelow/calc/ex4-7.html>. Acesso em: 8.10.2014. Citado na página 75.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. *Pattern-Oriented Software Architecture: A System of Patterns*. West Sussex, Inglaterra: [s.n.], 1996. Citado na página 52.

CANTÚ-PAZ, E. A survey of parallel genetic algorithms. *Calculateurs Paralleles*, v. 10, 1998. Disponível em: <http://tracer.uc3m.es/tws/cEA/documents/cant98.pdf>. Acesso em: 1.9.2013. Citado na página 42.

CARNEIRO, C. *Frameworks de aplicações orientadas a objeto - Uma abordagem interativa e incremental*. Dissertação (Mestrado) — Universidade Salvador, Abril 2003. Disponível em: <http://tede.unifacs.br/tde_arquivos/2/TDE-2007-01-05T180322Z-52/Publico/DissertacaoMestradoCristianeCarneiroTextocompleto.pdf>. Acesso em: 6.3.2014. Citado 3 vezes nas páginas 46, 48 e 49.

CHUANG, A. S.; WU, F. An extensible genetic algorithm framework for problem solving in a common environment. *IEEE Transactions on Power Systems*, v. 1, p. 269–275, Fevereiro 2000. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=852132>. Acesso em: 27.8.2013. Citado 2 vezes nas páginas 19 e 61.

CORNWELL, P. C. Hp domain analysis: Producing useful models for reusable software. *Hewlett-Packard Journal*, v. 47, p. 46–49, 1996. Citado na página 52.

CRESPO, S.; PINTO, S. *Composição em webframeworks*. Tese (Doutorado) — Pontifícia Universidade Católica do Rio de Janeiro, 2000. Citado 2 vezes nas páginas 46 e 47.

DAVIS, L. *Genetic algorithms and simulated annealing*. [s.n.], 1987. Disponível em: <http://www.osti.gov/scitech/servlets/purl/5037281>. Citado na página 38.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM - 50th anniversary*, ACM, New York, NY, USA, v. 51, n. 1, p. 107–113, Janeiro 2008. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1327452.1327492>. Acesso em: 8.5.2014. Citado 3 vezes nas páginas 56, 57 e 58.

DEAN, J.; GHEMAWAT, S. Mapreduce: A flexible data processing tool. *Communications of the ACM*, ACM, New York, NY, USA, v. 53, n. 1, p. 72–77, Janeiro 2010. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1629175.1629198>. Acesso em: 8.5.2014. Citado 2 vezes nas páginas 56 e 57.

DEJONG, K. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. Tese (Doutorado) — University of Michigan, 1975. Disponível em: <http://deepblue.lib.umich.edu/handle/2027.42/4507>. Acesso em: 11.11.2013. Citado na página 40.

DIAZ-GOMEZ, P. A.; HOUGEN, D. F. Initial population for genetic algorithms: A metric approach. In: *Proceedings of the International Conference on Genetic and Evolutionary Methods*. [s.n.], 2007. p. 43–49. Disponível em: <http://www.researchgate.net/publication/220862320_Initial_Population_for_Genetic_Algorithms_A_Metric_Approach/file/79e4150e5dd55b727a.pdf>. Acesso em: 23.8.2013. Citado na página 26.

EIBEN, A. E.; MARCHIORI, E.; VALKÓ, V. A. Evolutionary algorithms with on-the-fly population size adjustment. In: *Parallel Problem Solving from Nature - PPSN VIII*. Springer Berlin Heidelberg, 2004. v. 3242, p. 41–50. Disponível em: <http://link.springer.com/chapter/10.1007/978-3-540-30217-9_5>. Acesso em: 23.8.2013. Citado 3 vezes nas páginas 43, 44 e 61.

EIBEN, A. E.; SCHOENAUER, M. Evolutionary computing. *Information Processing Letters*, v. 82, n. 1, p. 1–6, Abril 2002. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0020019002002041>. Acesso em: 2.9.2013. Citado na página 19.

ESCUELA, G.; CARDINALE, Y.; GONZÁLEZ, J. A java-based distributed genetic algorithm framework. In: *19th IEEE International Conference on Tools with Artificial Intelligence*. [s.n.], 2007. v. 1, p. 437–441. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4410317>. Acesso em: 27.8.2013. Citado 2 vezes nas páginas 23 e 61.

FAIRLEY, R. *Software engineering concepts*. [S.l.]: McGraw-Hill, Inc., 1985. Citado na página 46.

FAYAD, M. E.; SCHMIDT, D. C. Object-oriented application frameworks. *Communications of the ACM*, v. 40, p. 32–38, 1997. Disponível em: <http://dl.acm.org/citation.cfm\?id=262798>. Acesso em: 1.9.2013. Citado 2 vezes nas páginas 49 e 50.

FOGEL, D. B. An introduction to simulated evolutionary optimization. *IEEE Transactions on Neural Networks*, v. 5, n. 1, p. 3–14, Janeiro 1994. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=265956>. Acesso em: 27.8.2013. Citado 2 vezes nas páginas 23 e 24.

FORREST, S. Genetic algorithms: principles of natural selection applied to computation. *Science*, v. 261, n. 5123, p. 872–878, Agosto 1993. Disponível em: <http://www.sciencemag.org/content/261/5123/872.short>. Acesso em: 26.8.2013. Citado 4 vezes nas páginas 24, 27, 28 e 61.

GAMMA, E.; JOHNSON, R.; HELM, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley Professional, 1995. Citado 2 vezes nas páginas 53 e 54.

GOLDBERG, D. E.; HOLLAND, J. H. Genetic algorithms and machine learning. *Machine learning*, v. 3, n. 2, p. 95–99, 1988. Citado 2 vezes nas páginas 27 e 29.

GREFENSTETTE, J. J. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, v. 16, n. 1, p. 122–128, 1986. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4075583>. Acesso em: 22.8.2013. Citado 5 vezes nas páginas 27, 30, 42, 43 e 61.

GUIZZARDI, G. *Desenvolvimento para e com Reuso: Um Estudo de Caso no Domínio de Vídeo sob Demanda*. Dissertação (Mestrado) — Universidade Federal do Espirito Santo, Julho 2000. Disponível em: <http://www.inf.ufes.br/~falbo/download/aulas/tengsoft/2006-1/DissertacaoGian.pdf>. Acesso em: 24.2.2014. Citado 3 vezes nas páginas 50, 51 e 52.

GUO, P.; WANG, X.; HAN, Y. The enhanced genetic algorithms for the optimization design. In: *3rd International Conference on Biomedical Engineering and Informatics*. Yantai: [s.n.], 2010. v. 7, p. 2990–2994. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5639829>. Acesso em: 22.8.2013. Citado 3 vezes nas páginas 23, 27 e 43.

HARIK, G.; LOBO, F. A parameter-less genetic algorithm. In: *IEEE Transactions on Evolutionary Computation*. [s.n.], 1999. p. 258–265. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.5188>. Citado 5 vezes nas páginas 17, 44, 45, 46 e 61.

HOLLAND, J. H. *Adaptation in natural and artificial systems*. [S.l.]: Ann Arbor, Michigan, 1975. Citado 2 vezes nas páginas 27 e 61.

HOLLAND, J. H. Genetic algorithms. *Scientific american*, v. 267, n. 1, p. 66–72, 1992. Disponível em: <http://faculty.samford.edu/~sfdonald/Courses/cosc407/Papers/Geneticalgorithms(Holland).pdf>. Acesso em: 22.8.2013. Citado 2 vezes nas páginas 33 e 39.

HORNBY, G.; GLOBUS, A.; LINDEN, D. Automated antenna design with evolutionary algorithms. In: AMERICAN INSTITUTE OF AERONAUTICS AND ASTRONAUTICS. *SPACE 2006*. San José, California, 2006. Citado na página 20.

HUDAK, P. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, ACM, New York, NY, USA, v. 21, n. 3, p. 359–411, Setembro 1989. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/72551.72554>. Acesso em: 9.5.2014. Citado na página 56.

HUTTENLOCHER, D.; KLANDERMAN, G.; RUCKLIDGE, W. Comparing images using the hausdorff distance. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, v. 15, n. 9, p. 850–863, Setembro 1993. ISSN 0162-8828. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=232073&tag=1>. Acesso em: 1.5.2014. Citado na página 25.

ISLAM, N.; DEVARAKONDA, M. An essential design pattern for fault-tolerant distributed state sharing. *Communications of the ACM*, v. 39, n. 10, Outubro 1996. Disponível em: <http://doi.acm.org/10.1145/236156.236172>. Acesso em: 18.3.2014. Citado na página 53.

JANSEN, T.; WEGENER, I. Real royal road functions - where crossover provably is essential. *Discrete Applied Mathematics*, v. 149, n. 1–3, p. 111–125, 2005. ISSN

0166-218X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0166218X05000727>. Acesso em: 2.9.2013. Citado na página 30.

JOHNSON, R. E.; FOOTE, B. Designing reusable classes. *Journal of object-oriented programming*, v. 1, n. 2, p. 22–35, 1988. Disponível em: <http://www.laputan.org/drc.html>. Acesso em: 17.2.2014. Citado na página 46.

KEARNEY, J. K.; THOMPSON, W. B.; BOLEY, D. L. Optical flow estimation: An error analysis of gradient-based methods with local optimization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9, n. 2, p. 229–244, Março 1987. ISSN 0162-8828. Citado na página 19.

LARMAN, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3/e*. [S.l.]: Pearson Education India, 2012. Citado na página 53.

LEO, S.; ZANETTI, G. Pydoop: A python mapreduce and hdfs api for hadoop. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. New York, NY, USA: ACM, 2010. (HPDC '10), p. 819–825. ISBN 978-1-60558-942-8. Disponível em: <http://doi.acm.org/10.1145/1851476.1851594>. Citado na página 75.

LINDEN, R. *Algoritmos Genéticos*. [S.l.: s.n.], 2006. Citado na página 21.

LOBO, F. G.; LIMA, C. F. Revisiting evolutionary algorithms with on-the-fly population size adjustment. In: *Proceedings of the 8th annual conference on genetic and evolutionary computation*. Seattle, Washington, USA: ACM New York, NY, USA, 2006. (GECCO '06), p. 1241–1248. Disponível em: <http://dl.acm.org/citation.cfm?id=1144192>. Acesso em: 26.8.2013. Citado 2 vezes nas páginas 27 e 44.

MARKIEWICZ, M. E.; LUCENA, C. J. P. de. Object oriented framework development. *Crossroads*, v. 7, p. 3–9, Junho 2001. Disponível em: <http://dl.acm.org/citation.cfm?id=372771>. Acesso em: 2.9.2013. Citado 5 vezes nas páginas 21, 46, 47, 53 e 55.

MATTSSON, M. *Evolution and Composition of Object-Oriented Frameworks*. Tese (Doutorado) — University of Karlskrona/Ronneby - Department of Software Engineering and Computer Science, Suécia, 2000. Disponível em: <http://btu.se/fou/Forskinfo.nsf/Sok/73c98b952598b035c12573c90033e1b1/$file/T02.OO.Evolution.Composition.OO.Frameworks.pdf>. Acesso em: 6.3.2014. Citado na página 50.

MCKENNEY, P. E. Selecting locking primitives for parallel programming. *Communications of the ACM*, v. 39, n. 10, p. 75–82, Outubro 1996. Disponível em: <http://doi.acm.org/10.1145/236156.236174>. Acesso em: 18.3.2014. Citado na página 53.

MEHTA, H.; OWENS, R. M.; IRWIN, M. J. Some issues in gray code addressing. In: *Proceedings Sixth Great Lakes Symposium on VLSI, 1996.* Ames, IA: [s.n.], 1996. p. 178–181. ISSN 1066-1395. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=497616&tag=1>. Acesso em: 1.5.2014. Citado na página 29.

MEUNIER, H.; TALBI, E.; REININGER, P. A multiobjective genetic algorithm for radio network optimization. In: *Proceedings of the 2000 Congress on Evolutionary Computation*. La Jolla, CA: [s.n.], 2000. v. 1, p. 317–324. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=870312>. Acesso em: 1.5.2014. Citado 2 vezes nas páginas 21 e 61.

MILLER, B. L.; GOLDBERG, D. E. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, v. 9, n. 3, p. 193–212, 1995. Citado na página 30.

MIRANDA, V.; RANITO, J. V.; PROENCA, L. M. Genetic algorithms in optimal multistage distribution network planning. *IEEE transactions on power systems*, v. 9, p. 1927–1933, 1994. Citado na página 20.

MONTESINOS, F. G.; ARNOSO, J.; VIEIRA, R. Using a genetic algorithm for 3-d inversion of gravity data in fuerteventura (canary islands). *International Journal of Earth Sciences*, v. 94, n. 2, p. 301–316, Abril 2005. Disponível em: <http://link.springer.com/article/10.1007/s00531-005-0471-6>. Acesso em: 16.4.2014. Citado na página 21.

MUNAWAR, A.; WAHIB, M.; MUNETOMO, M.; AKAMA, K. A survey: Genetic algorithms and the fast evolving world of parallel computing. In: *10th IEEE International Conference on High Performance Computing and Communications*. Dalian: [s.n.], 2008. p. 897–902. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4637800>. Acesso em: 22.8.2013. Citado na página 23.

NEIGHBORS, J. M. *Software Construction Using Components*. Tese (Doutorado) — University of California, Irvine, 1980. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download\?doi=10.1.1.21.4397&rep=rep1&type=pdf>. Acesso em: 25.2.2014. Citado na página 50.

PAPADOPOULOS, G.; WIGGINS, G. A genetic algorithm for the generation of jazz melodies. In: *Proceedings of Step 98*. [S.l.: s.n.], 1998. p. 7–9. Citado na página 21.

PARSONS, D.; RASHID, A.; SPECK, A.; TELEA, A. A "framework" for object oriented frameworks design. In: *Proceedings of Technology of Object-Oriented Languages and Systems*. Nancy: [s.n.], 1999. p. 141–151. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=779007>. Acesso em: 2.9.2013. Citado na página 48.

PETIT, E.; SWIGGER, K. M. An analysis of genetic-based pattern tracking and cognitive-based component tracking models of adaptation. In: *National Conference on Al*. [S.l.: s.n.], 1983. Citado na página 27.

POHLHEIM, H. Ein genetischer algorithmus mit mehrfachpopulationen zur numerischen optimierung. *Automatisierungstechnik*, v. 3, p. 127–135, 1995. Disponível em: <http://cat.inist.fr/?aModele=afficheN&cpsidt=3448014>. Citado na página 35.

POHLHEIM, H. *GEATbx - The Genetic and Evolutionary Algorithm Toolbox for Matlab*. 2007. Disponível em: <http://www.geatbx.com/docu/algindex-02.html>. Citado na página 35.

ROEVA, O.; SHANNON, A.; PENCHEVA, T. Description of simple genetic algorithm modifications using generalized nets. In: *6th IEEE International Conference Intelligent Systems*. [s.n.], 2012. p. 178–183. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6335212>. Acesso em: 22.8.2013. Citado 2 vezes nas páginas 23 e 61.

ROSÁRIO, R. R. L. do. *Algoritmos Evolutivos Adaptados para Problemas de Programação de Pessoal*. Tese (Doutorado) — Universidade Federal de Santa Catarina, 2011. Disponível em: <https://repositorio.ufsc.br/handle/123456789/94967>. Acesso em: 1.5.2014. Citado na página 21.

SANTARELLI, S.; YU, T.-L.; GOLDBERG, D. E.; ALTSHULER, E.; O'DONNELL, T.; SOUTHALL, H.; MAILLOUX, R. Military antenna design using simple and competent genetic algorithms. *Mathematical and Computer Modelling*, v. 43, n. 9-10, p. 990–1022, Maio 2006. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0895717705005315>. Acesso em: 16.4.2014. Citado 2 vezes nas páginas 19 e 61.

SCHMIDT, D. C.; FAYAD, M.; JOHNSON, R. E. Software patterns. *Communications of the ACM*, v. 39, n. 10, p. 37–39, Outubro 1996. Disponível em: <http://doi.acm.org/10.1145/236156.236164>. Acesso em: 18.3.2014. Citado na página 53.

SHEBLE, G.; BRITTIG, K. Refined genetic algorithm-economic dispatch example. *Power Systems, IEEE Transactions on*, v. 10, n. 1, p. 117–124, Feb 1995. ISSN 0885-8950. Citado na página 20.

SHEHU, V.; DIKA, A. Curve similarity measurement algorithms for automatic gesture detection systems. *MIPRO, 2012 Proceedings of the 35th International Convention*, p. 973–976, May 2012. Citado na página 78.

SHI, X.; LIANG, Y.; LEE, H.; LU, C.; WANG, L. An improved ga and a novel pso-ga-based hybrid algorithm. *Information Processing Letters*, v. 93, n. 5, p. 255–261, Março 2005. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0020019004003254>. Acesso em: 28.8.2013. Citado na página 23.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The hadoop distributed file system. In: *26th Symposium on Mass Storage Systems and Technologies (MSST)*. Incline Village, NV: [s.n.], 2010. p. 1–10. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5496972>. Acesso em: 6.3.2014. Citado na página 49.

SILVA, R. P. e. *Suporte ao desenvolvimento e uso de frameworks e componentes*. Tese (Doutorado) — Universidade Federal do Rio Grande do Sul, Março 2000. Disponível em: <http://www.inf.ufsc.br/~ricardo/download/tese.pdf>. Acesso em: 25.8.2013. Citado 5 vezes nas páginas 46, 47, 48, 49 e 52.

SILVA, R. P. e. *Como Modelar com UML2*. [S.l.]: Visual Books, 2009. 320 p. Citado na página 63.

SIVARAJ, R.; RAVICHANDRAN, T. A review of selection methods in genetic algorithm. *International Journal of Engineering Science and Technology*, v. 3, n. 5, p. 3792–3797, 2011. Disponível em: <http://www.ijest.info/docs/IJEST11-03-05-190.pdf>. Acesso em: 2.10.2013. Citado 6 vezes nas páginas 30, 33, 34, 35, 37 e 38.

SPARKS, S.; BENNER, K.; FARIS, C. Managing object oriented framework reuse. *Computer*, v. 29, p. 52–61, 1996. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=536784>. Acesso em: 1.9.2013. Citado na página 47.

SRINIVAS, M.; PATNAIK, L. M. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, v. 24, n. 4, p. 656–667, Abril 1994. Acesso em: 25.8.2013. Citado 2 vezes nas páginas 39 e 43.

SRINIVAS, M.; PATNAIK, L. M. Genetic algorithms: a survey. *Computer*, v. 27, p. 17–26, 1994. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp\?arnumber=294849>. Acesso em: 24.8.2013. Citado 4 vezes nas páginas 27, 29, 30 e 42.

SRINIVASAN, S. Design patterns in object-oriented frameworks. *Computer*, v. 2, p. 24–32, Fevereiro 1999. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=745717>. Acesso em: 25.8.2013. Citado 2 vezes nas páginas 52 e 53.

SUNDHARARAJAN, S.; PAHWA, A. Optimal selection of capacitors for radial distribution systems using a genetic algorithm. *Power Systems, IEEE Transactions on*, v. 9, n. 3, p. 1499–1507, Agosto 1994. ISSN 0885-8950. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=336111>. Acesso em: 11.4.2014. Citado na página 20.

TALIGENT. *Leveraging object oriented frameworks*. [S.l.], 1995. Disponível em: <http://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/leveragingoo.pdf>. Acesso em: 5.3.2014. Citado 2 vezes nas páginas 48 e 49.

TAYLOR, R. C. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. In: *Proceedings of the 11th Annual Bioinformatics Open Source Conference (BOSC) 2010*. [s.n.], 2010. v. 11. Disponível em: <http://www.biomedcentral.com/1471-2105/11/S12/S1>. Acesso em: 8.5.2014. Citado 3 vezes nas páginas 57, 59 e 60.

VEREL, S. *Fitness Landscapes and Graphs - Multimodularity, Ruggedness and Neutrality*. INRIA Lille-Nord Europe, 2013. Disponível em: <http://www-lisic.univ-littoral.fr/~verel/talks/2tut16-verel.pdf>. Acesso em: 6.10.2013. Citado 3 vezes nas páginas 25, 27 e 28.

VILJAMAA, A. *Pattern-Based Framework Annotation and Adaptation - A Systematic Approach*. Tese (Doutorado) — University of Helsinki, Finlandia, 2001. Citado na página 46.

VLISSIDES, J. M.; LINTON, M. A. Unidraw: A framework for building domain-specific graphical editors. *ACM Trans. Inf. Syst.*, ACM, New York, NY, USA, v. 8, n. 3, p. 237–268, jul. 1990. ISSN 1046-8188. Disponível em: <http://doi.acm.org/10.1145/98188.98197>. Acesso em: 6.3.2014. Citado na página 49.

WANG, L.; TAO, J.; RANJAN, R.; MARTEN, H.; STREIT, A.; CHEN, J.; CHEN, D. G-hadoop: Mapreduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems*, v. 29, n. 3, p. 739–750, 2013. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167739X12001744>. Acesso em: 6.3.2014. Citado na página 49.

WEN, F.; HAN, Z. Fault section estimation in power systems using a genetic algorithm. *Electric Power Systems Research*, v. 34, n. 3, p. 165–172, Setembro 1995. Disponível em: <http://dx.doi.org/10.1016/0378-7796(95)00974-6>. Acesso em: 11.4.2014. Citado na página 20.

WHITLEY, D. A genetic algorithm tutorial. *Statistics and Computing*, v. 4, n. 2, p. 65–85, Junho 1994. Disponível em: <http://link.springer.com/article/10.1007/BF00175354>. Acesso em: 22.8.2013. Citado 7 vezes nas páginas 19, 25, 26, 27, 28, 41 e 42.

WIRFS-BROCK, A.; VISSADES, J.; CUNNINGHAM, W.; JOHNSON, R.; BOLLETTE, L. Designing reusable designs (panel session): experiences designing object-oriented frameworks. In: *Proceedings of the European conference on Object-oriented programming addendum : systems, languages, and applications*. New York, NY, USA: ACM, 1991. (OOPSLA/ECOOP '90), p. 19–24. Disponível em: <http://doi.acm.org/10.1145/319016.319035>. Acesso em: 24.8.2013. Citado na página 47.

WORLD ROAD ASSOCIATION. *Highway Development & Management (HDM-4)*. 2000. Citado na página 78.

WORRAL, S.; NEBOT, E. Automated process for generating digitised maps through GPS data compression. In: *Proceedings of the 2007 Australasian Conference on Robotics & Automation*. Brisbane, Australia: [s.n.], 2007. Citado na página 78.

YANG, Y. J.; KIM, S. Y.; CHOI, G. J.; CHO, E. S.; KIM, S. D.; KIM, S. D. A uml-based object-oriented framework development methodology. In: *Proceedings of Software Engineering Conference*. Taipei: [s.n.], 1998. p. 211–218. Disponível em: <http://ieeexplore.ieee.org/xpl/abstractKeywords.jsp?arnumber=733722>. Acesso em: 2.9.2013. Citado 2 vezes nas páginas 47 e 53.

YIN, X. Application of genetic algorithms to multiple load flow solution problem in electrical power systems. In: *Proceedings of the 32nd IEEE Conference on Decision and Control*. San Antonio, TX: [s.n.], 1993. v. 4, p. 3734–3739. Disponível em: <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=325915>. Acesso em: 11.4.2014. Citado na página 20.

YIN, X.; GERMAY, N. Investigations on solving the load flow problem by genetic algorithms. *Electric Power Systems Research*, v. 22, n. 3, p. 151–163, Dezembro 1991. ISSN 0378-7796. Disponível em: <http://www.sciencedirect.com/science/article/pii/0378779691900014>. Acesso em: 11.4.2014. Citado na página 20.

YU, H.-L.; KOLOVOS, A.; CHRISTAKOS, G.; CHEN, J.-C.; WARMERDAM, S.; DEV, B. Interactive spatiotemporal modelling of health systems: the seks–gui framework. *Stochastic Environmental Research and Risk Assessment*, v. 21, n. 5, p. 555–572, Agosto 2007. Disponível em: <http://link.springer.com/article/10.1007/s00477-007-0135-0>. Acesso em: 6.3.2014. Citado na página 49.

Apêndices

APÊNDICE A – DIAGRAMAS DE ATIVIDADES - CASOS DE USO

Esta seção apresenta os diagramas de atividades desenvolvidos para o refinamento dos casos de uso.

Figura 60 – Diagrama de atividades do caso de uso *evaluate fitness*

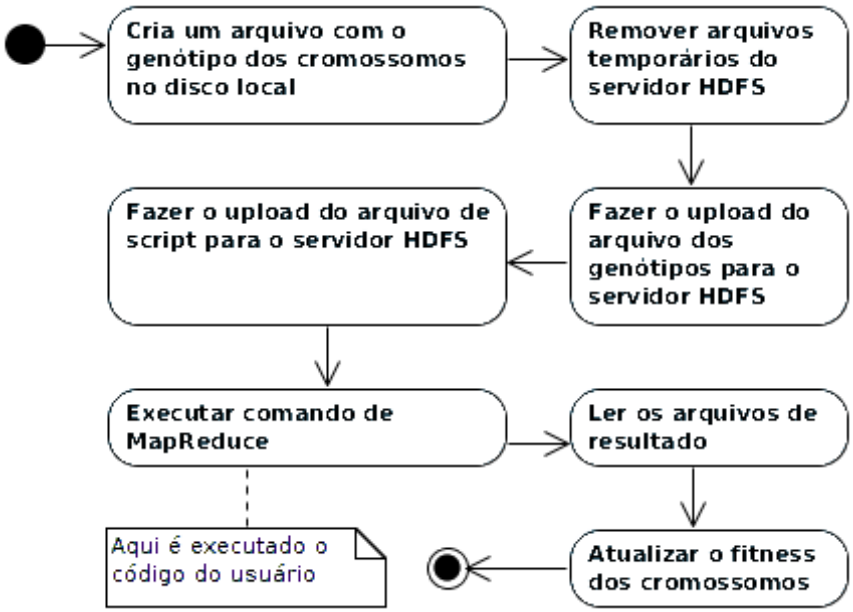


Figura 61 – Diagrama de atividades do caso de uso *reproduction*

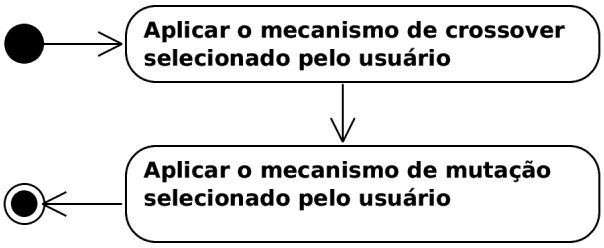
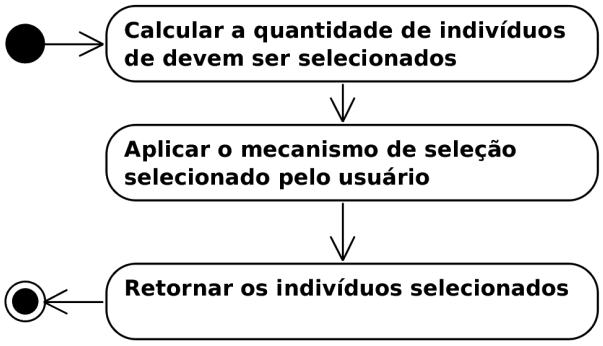


Figura 62 – Diagrama de atividades do caso de uso *selection*



APÊNDICE B – DIAGRAMAS DE ATIVIDADES - ESTRATÉGIAS PRÉ IMPLEMENTADAS

Esta seção apresenta os diagramas de atividades desenvolvidos para especificar as estratégias de seleção, mutação e *crossover* pré implementadas.

Figura 63 – Diagrama de atividades da estratégia de seleção por amostragem universal estocástica

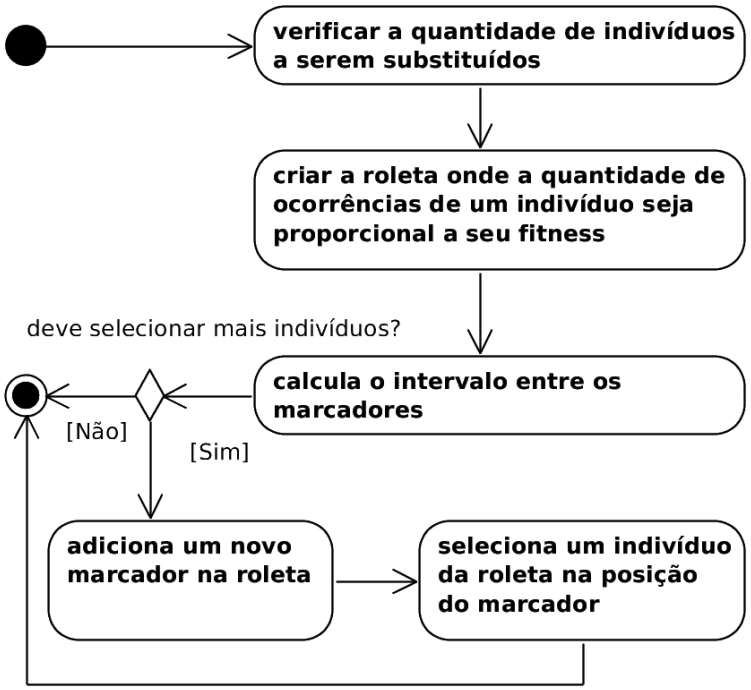


Figura 64 – Diagrama de atividades da estratégia de seleção por torneio

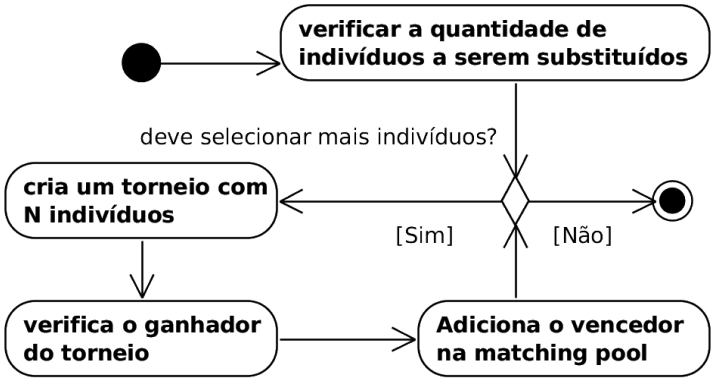


Figura 65 – Diagrama de atividades da estratégia de seleção por roleta

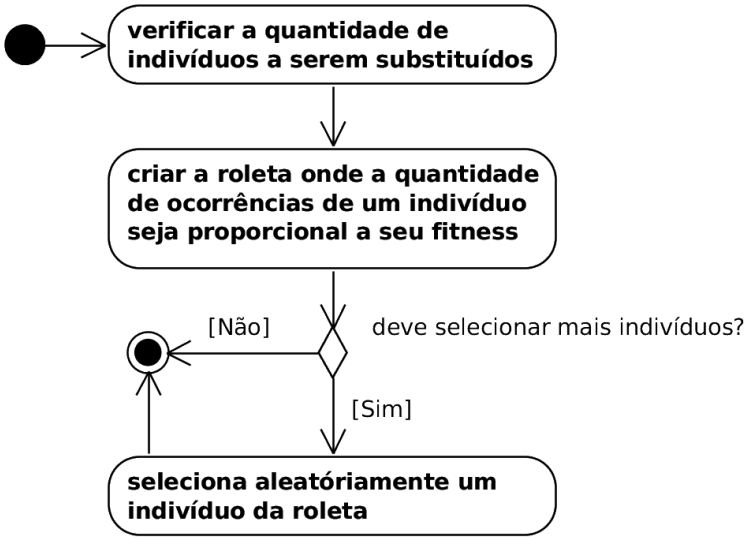


Figura 66 – Diagrama de atividades da estratégia de seleção por ranking

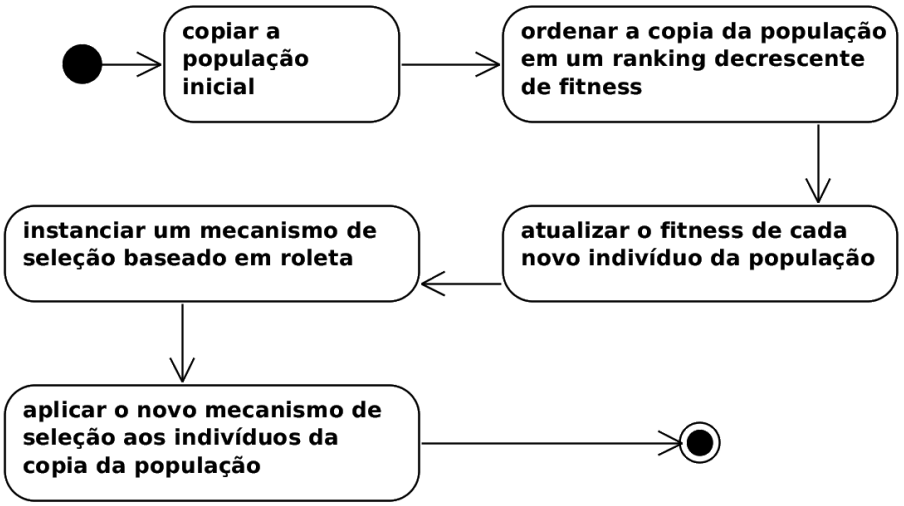


Figura 67 – Diagrama de atividades da estratégia de *crossover* de 1 ponto

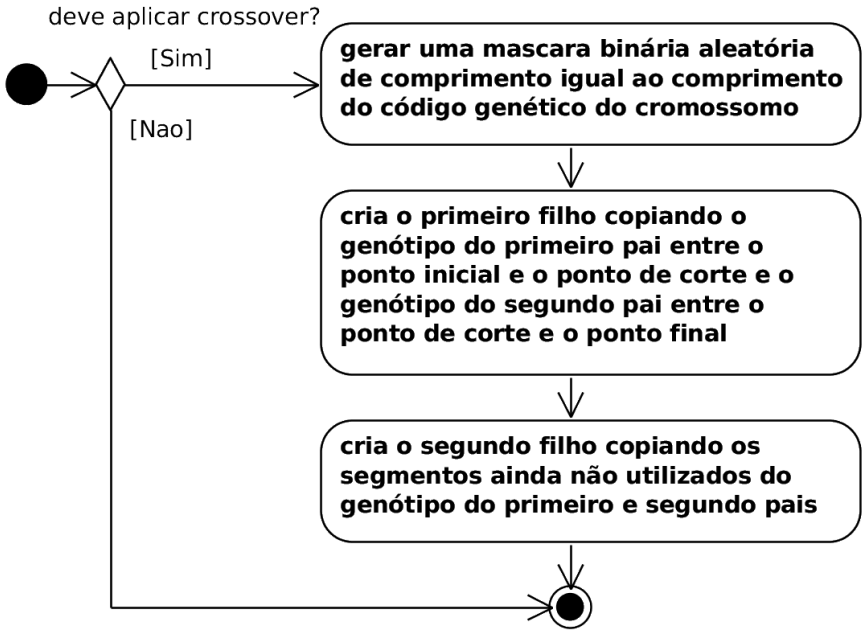


Figura 68 – Diagrama de atividades da estratégia de *crossover* de 2 pontos

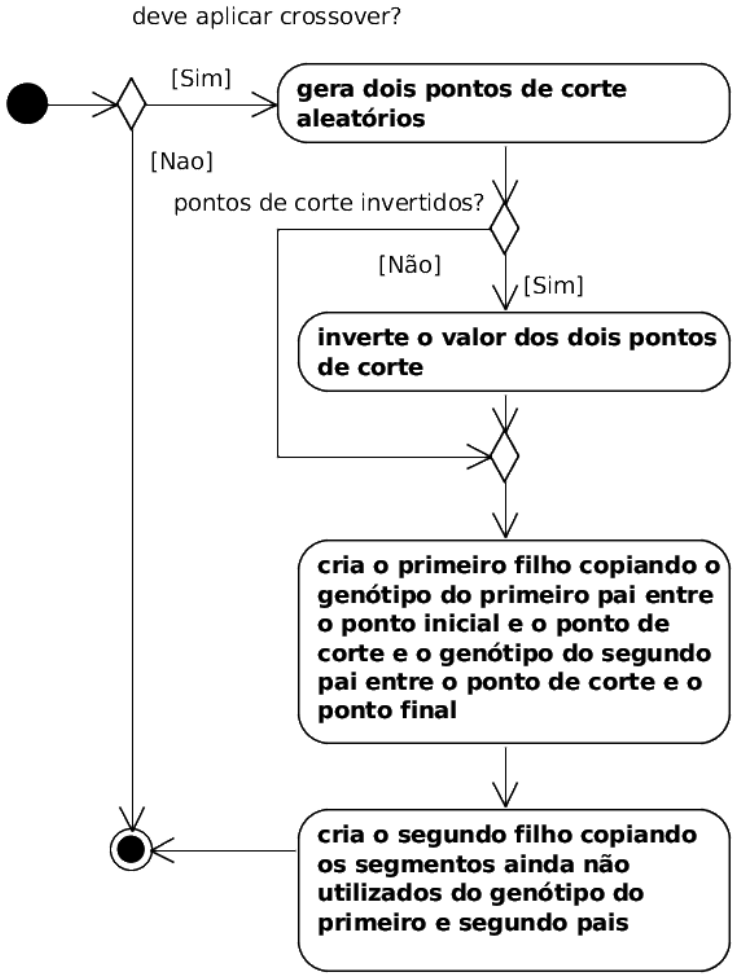


Figura 69 – Diagrama de atividades da estratégia de *crossover* uniforme

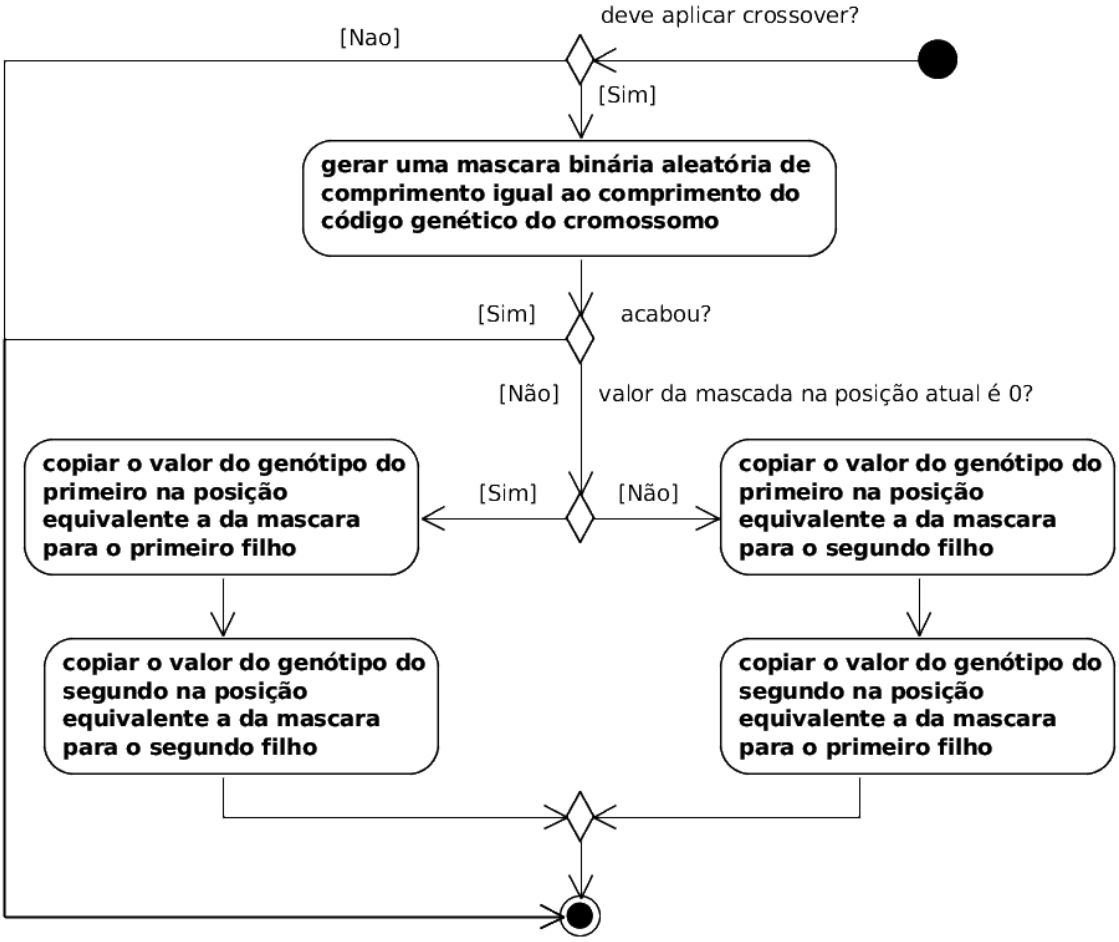
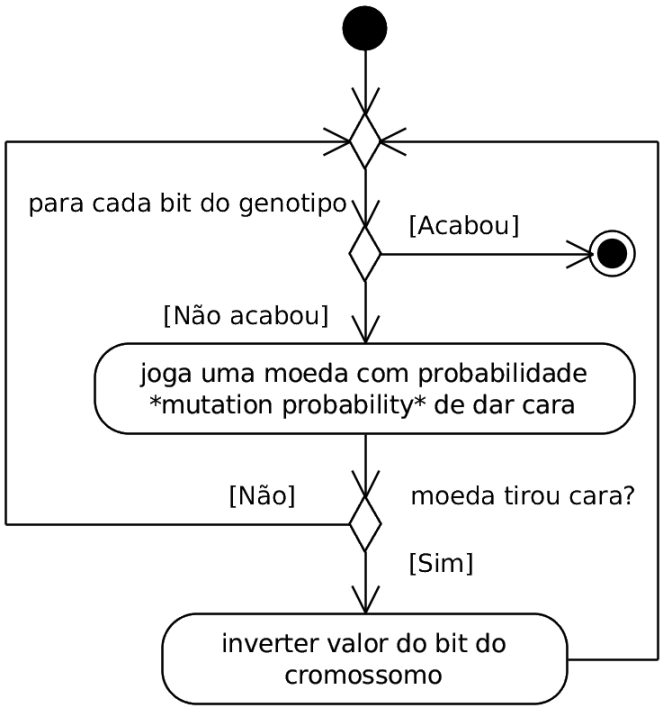


Figura 70 – Diagrama de atividades da estratégia de *mutação* de bit



APÊNDICE C – DIAGRAMAS DE SEQUÊNCIA - CASOS DE USO

Esta seção apresenta os diagramas de sequência desenvolvidos para o refinamento dos casos de uso.

Figura 71 – Diagrama de sequência do caso de uso *evolve*

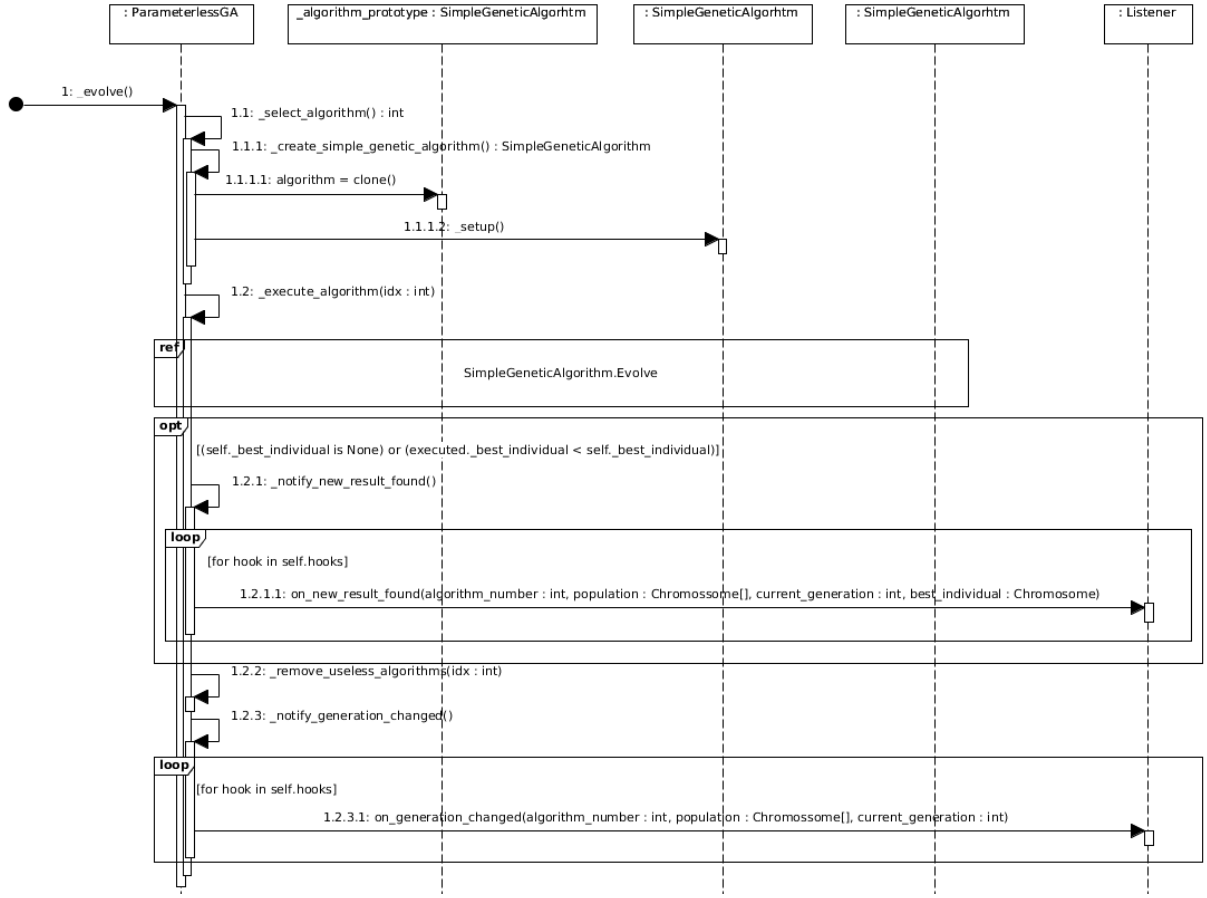


Figura 72 – Diagrama de sequência do caso de uso *evaluate fitness*

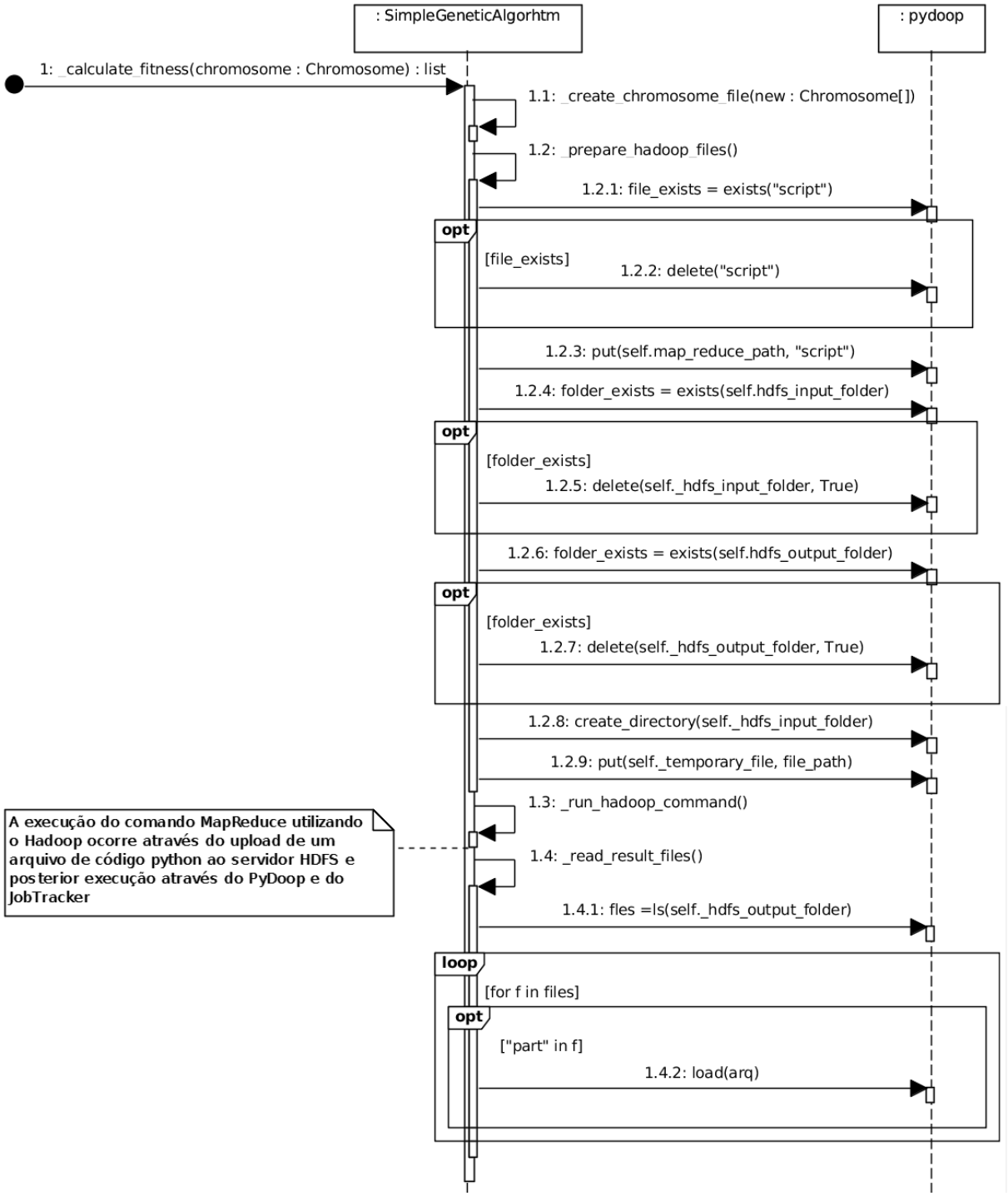


Figura 73 – Diagrama de sequência do caso de uso *selection*

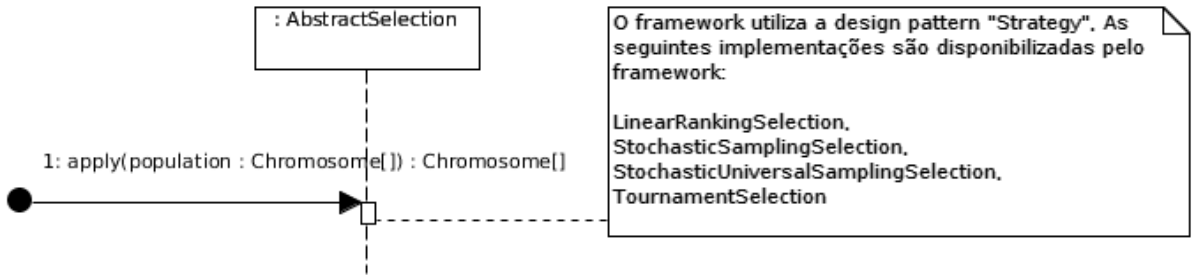


Figura 74 – Diagrama de sequência do caso de uso *reproduction*

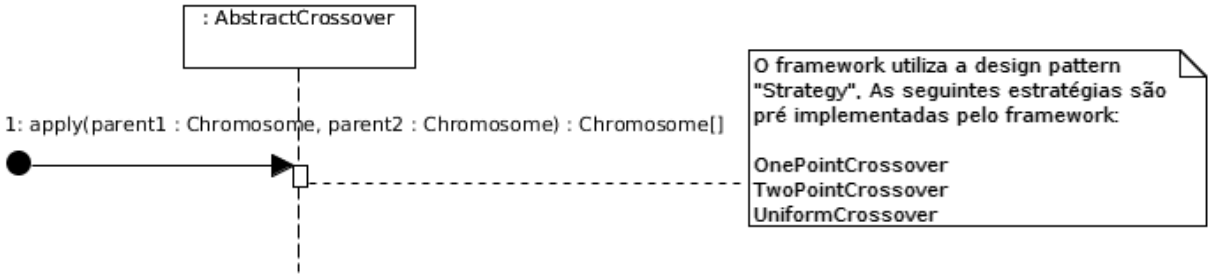
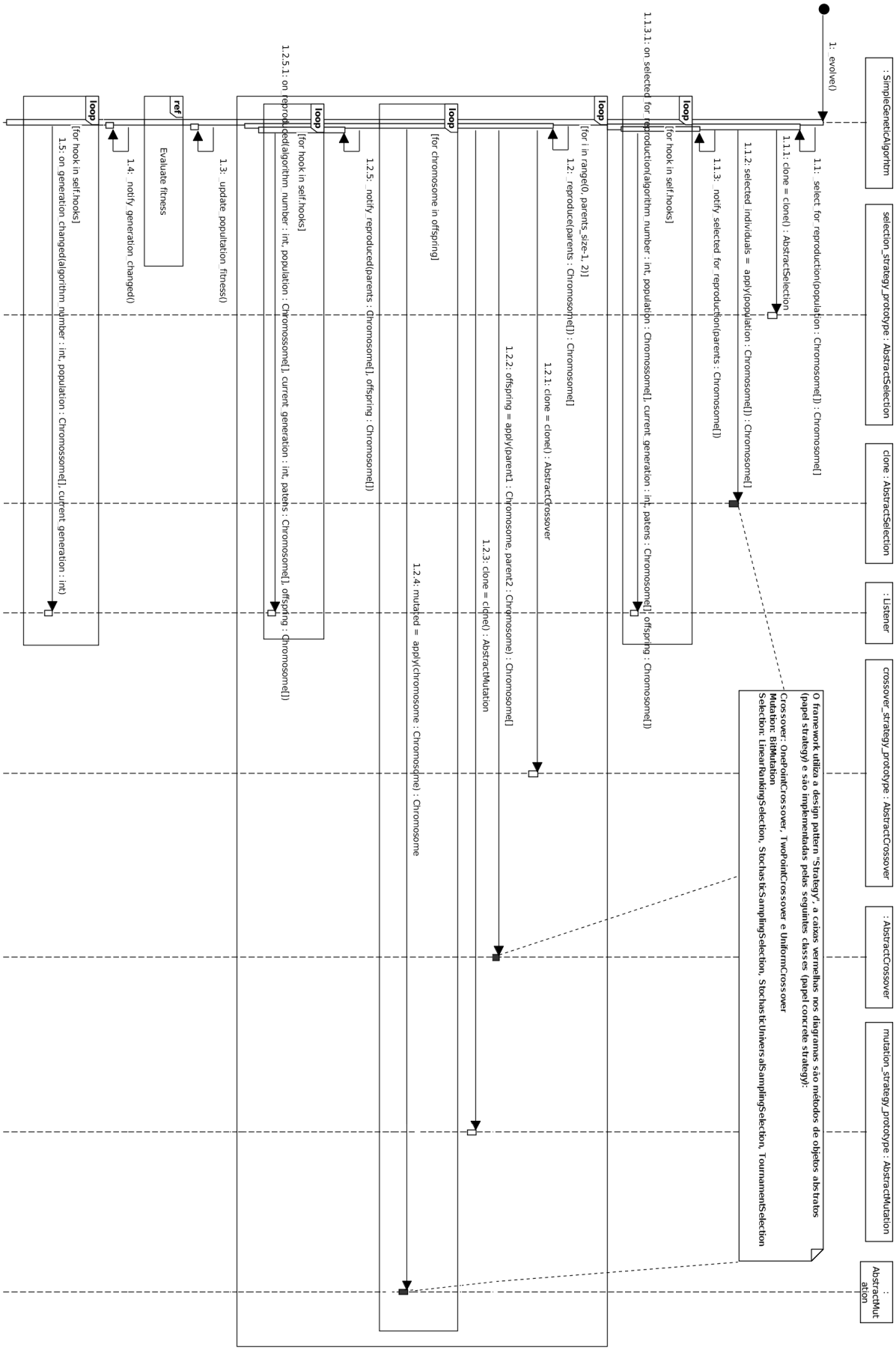


Figura 75 – Diagrama de sequência para refinamento do uso de iteração “SimpleGeneticAlgorithm.Evolve” do diagrama *evolve*



APÊNDICE D – DIAGRAMAS DE SEQUÊNCIA - ESTRATÉGIAS PRÉ IMPLEMENTADAS

Esta seção apresenta os diagramas de sequência desenvolvidos para especificar as estratégias de seleção, mutação e *crossover* pré implementadas.

Figura 76 – Diagrama de sequência da estratégia de seleção por torneio

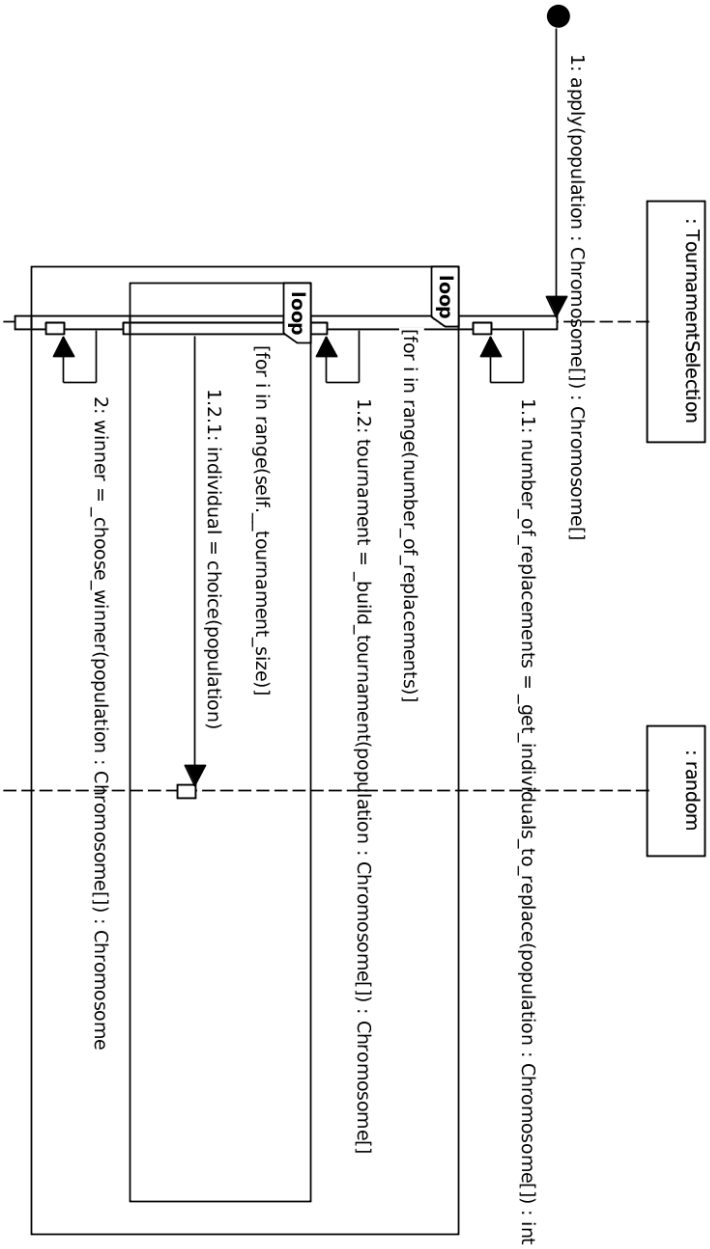


Figura 77 – Diagrama de sequência da estratégia de seleção por amostragem universal estocástica

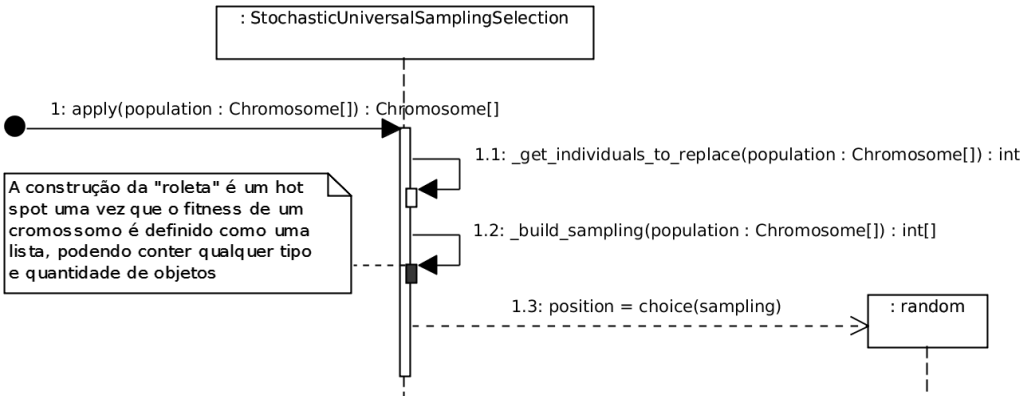


Figura 78 – Diagrama de sequência da estratégia de seleção por roleta

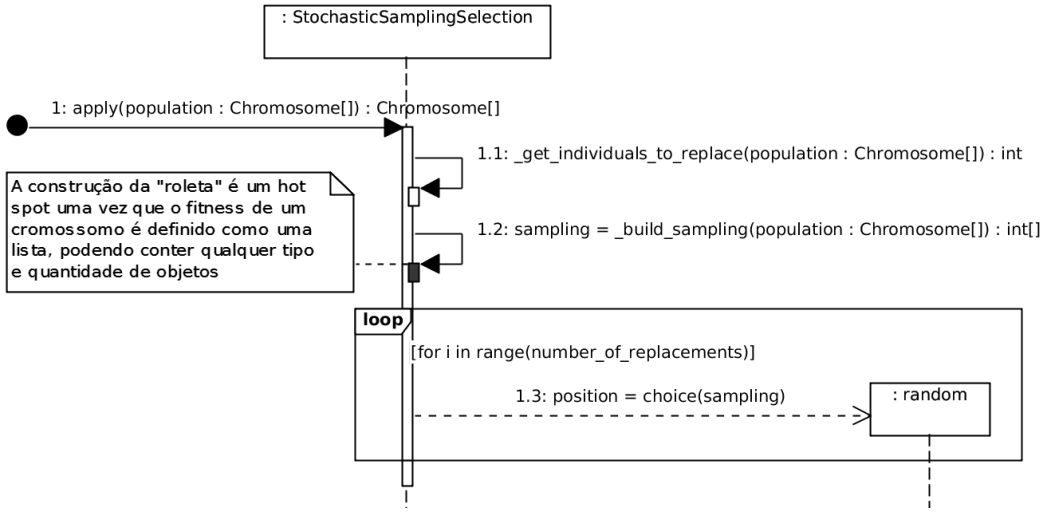


Figura 79 – Diagrama de sequência da estratégia de seleção por ranking

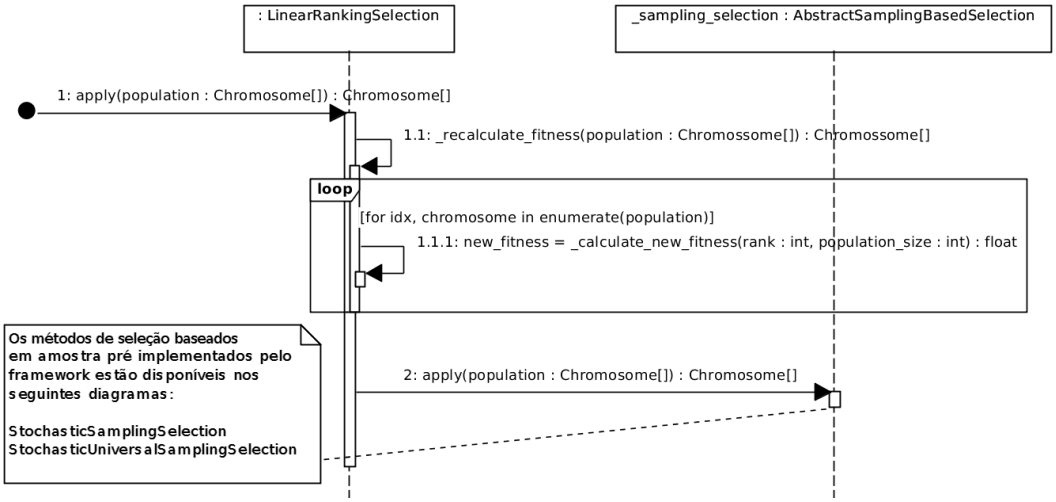


Figura 80 – Diagrama de sequência da estratégia de *crossover* de 1 ponto

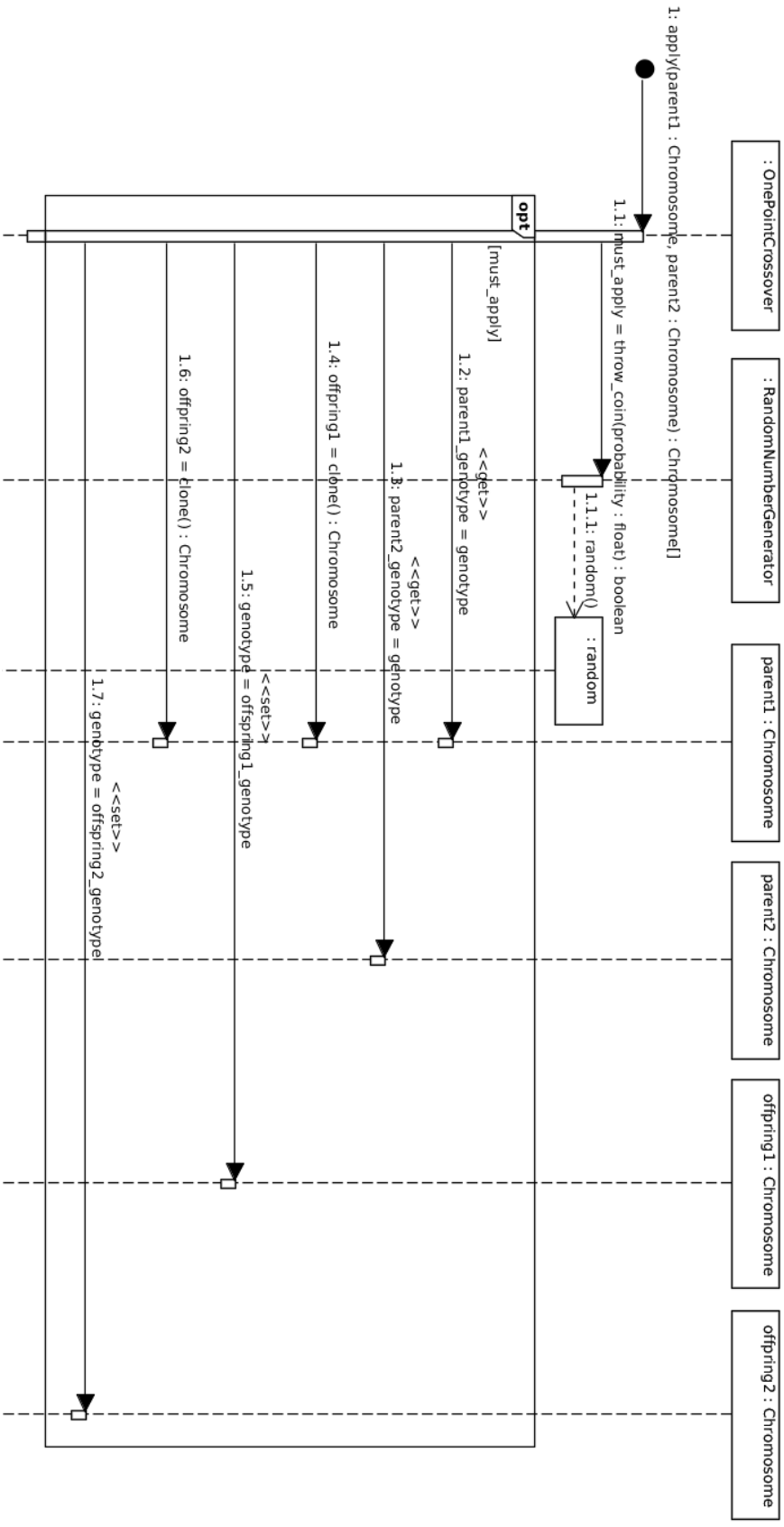


Figura 81 – Diagrama de sequência da estratégia de *crossover* de 2 pontos

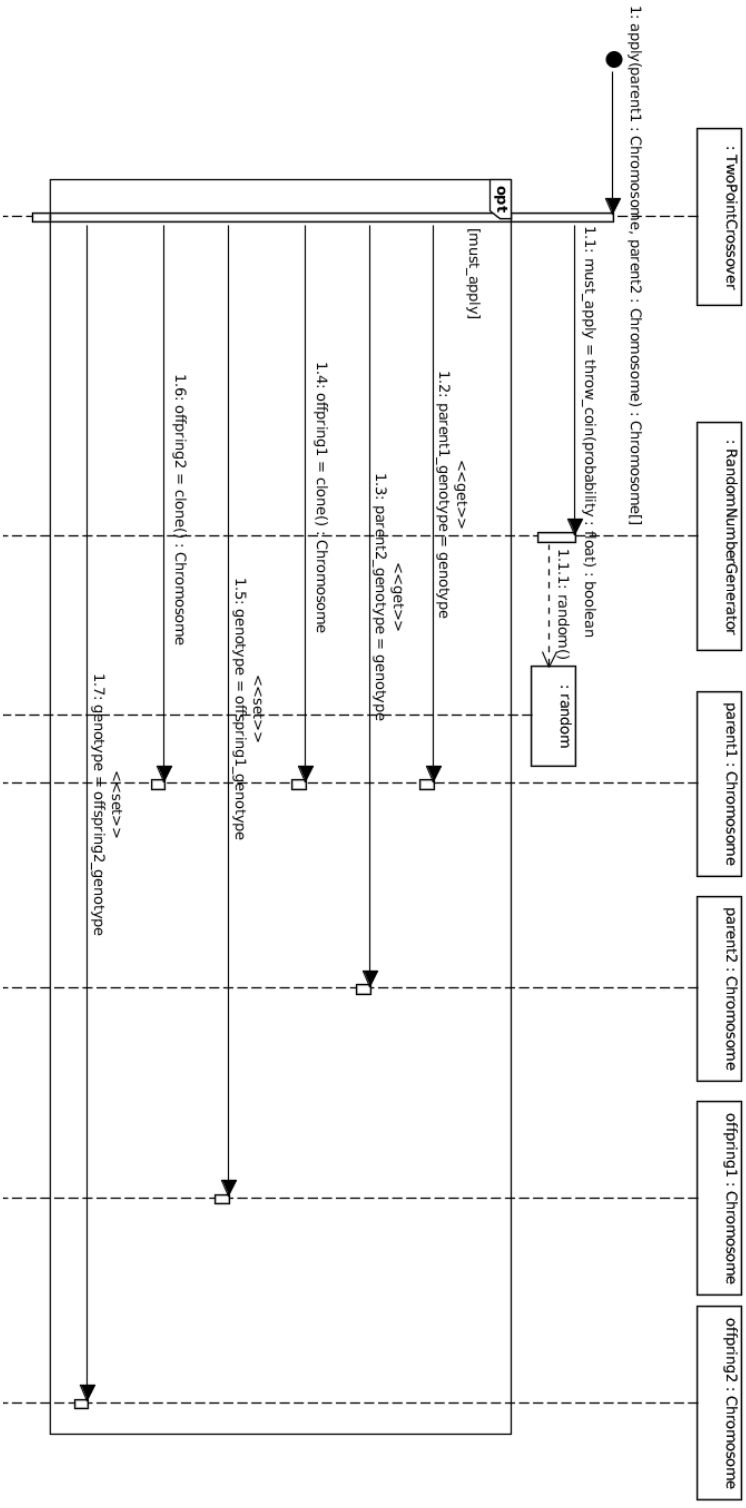


Figura 82 – Diagrama de sequência da estratégia de *crossover* uniforme

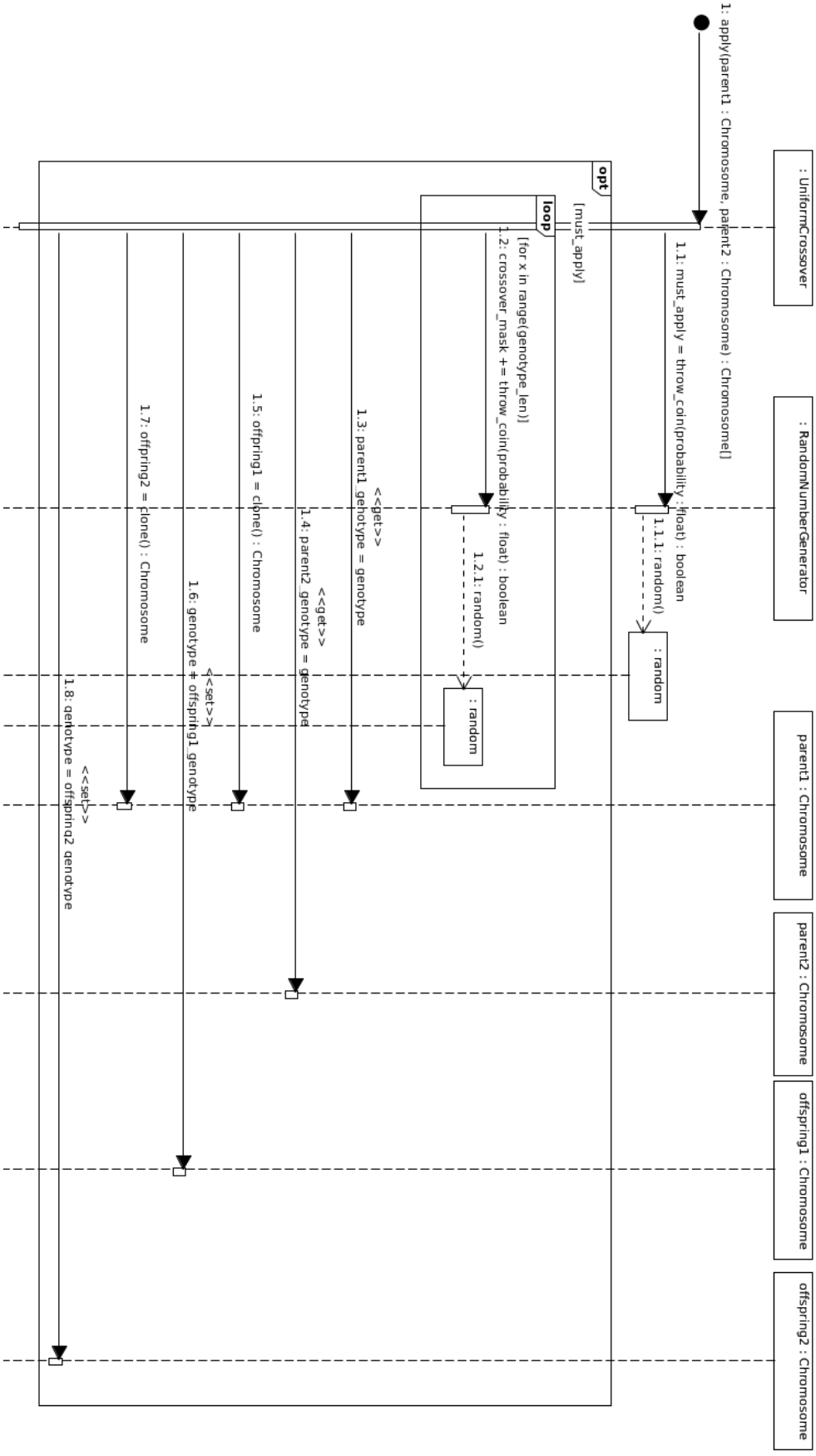


Figura 83 – Diagrama de sequência da estratégia de *mutação* de bit