ULBRA
CENTRO UNIVERSITÁRIO LUTERANO DE PALMAS
COMUNIDADE EVANGÉLICA LUTERANA "SÃO PAULO"
Recredenciado pela Portaria Ministerial nº 3.607 - D.O.U. nº 202 de 20/10/2005

ELIAS MELGAÇO CHAVES JÚNIOR

ALGORITMOS GENÉTICOS: COMPARAÇÃO ENTRE OS FRAMEWORKS AFORGE E JGAP

Palmas

2012

ELIAS MELGAÇO CHAVES JÚNIOR

ALGORITMOS GENÉTICOS: COMPARAÇÃO ENTRE OS FRAMEWORKS AFORGE E JGAP

Trabalho apresentado como requisito parcial da disciplina Trabalho de Conclusão de Curso (TCC) do curso de Sistemas de Informação, orientado pelo Professor Mestre Fabiano Fagundes.

Palmas

2012

ELIAS MELGAÇO CHAVES JÚNIOR

ALGORITMOS GENÉTICOS: COMPARAÇÃO ENTRE OS FRAMEWORKS AFORGE E JGAP

Trabalho apresentado como requisito parcial da disciplina Trabalho de Conclusão de Curso (TCC) do curso de Sistemas de Informação, orientado pelo Professor Mestre Fabiano Fagundes.

Aprovada em xxxxxxx de 2012.

BANCA EXAMINADORA

_____

Prof. M.Sc. Fabiano Fagundes

Centro Universitário Luterano de Palmas

_____

Prof. M.Sc. Madianita Bogo

Centro Universitário Luterano de Palmas

_____

Prof. M.Sc. Fernando Luiz de Oliveira

Centro Universitário Luterano de Palmas

Palmas

2012

SUMÁRIO

RESUMO

Os Algoritmos Genéticos (AGs) são algoritmos de otimização e busca baseados na teoria da evolução natural de Darwin, que indivíduos com melhores características genéticas têm maiores chances de sobrevivência, e de produzirem descendentes mais aptos Os AGs são indicados para solução de problemas de otimização complexos, como o do Problema do Caixeiro Viajante (PCV), utilizado neste trabalho, que consiste em visitar todas as cidades e retornar à cidade de início pelo menor caminho possível. Para solucionar um problema qualquer, podem-se caracterizar os AGs da forma: 1 – inicialização da população, 2 - avaliação, 3 – seleção, 4 – cruzamento, 5 – mutação. Dois *frameworks* foram utilizados para minimizar o esforço de criar todos os processos que levam à solução de um problema utilizando AGs: o *AFORGE*, na linguagem *C#* e o *JGAP*, na linguagem *Java*. A instalação e solução do PCV são explicadas para cada *framework* de forma que tornou possível compará-los de acordo com os seguintes critérios: tempo de execução, instalação, método de inicialização, método de seleção, método de aptidão, método de cruzamento, método de mutação, linguagem, quantidade de documentação disponível, disponibilidade de suporte e qualidade dos resultados.

PALAVRAS-CHAVE: Algoritmos Genéticos, *framework*, Inteligência Artificial.

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

AGs – Algoritmos Genéticos

API - *Application Programming Interface*

HTML – *Hyper Text Markup Language*

IA – Inteligência Artificial

IAC – Inteligência Artificial Conexionista

IAS – Inteligência Artificial Simbólica

IDE - Ambiente de Desenvolvimento Integrado

RNA - Redes Neurais Artificiais

PCV - Problema do Caixeiro Viajante

1 INTRODUÇÃO

O homem tem a capacidade única de raciocínio e durante milhares de anos, ele procurou “entender como pensamos: como um mero punhado de matéria pode compreender, perceber, prever e manipular um mundo muito maior e muito mais complexo que ele próprio”. Inteligência Artificial (IA) tenta compreender e construir entidades inteligentes, uma das ciências mais recentes que teve início de seus trabalhos após a segunda guerra mundial, e atualmente, envolve uma variedade de campos, como aprendizado e percepção, até tarefas específicas como jogos de xadrez (RUSSEL e NORVIG, 2004, p. 4).

Existem várias técnicas de IA que podem ser exploradas, como redes neurais, redes bayesianas e Algoritmos Genéticos (AGs), sendo este último o foco deste trabalho. AGs são baseados na teoria da evolução que descreve, segundo Darwin, que os organismos com melhores características genéticas têm mais chance de sobrevivência e de produzirem filhos com melhores características genéticas (REZENDE, 2003, p. 226).

Os AGs são utilizados em indivíduos de uma população, na tentativa de identificar uma solução ótima. Utilizam um processo que é chamado de geração ou ciclo. Nos indivíduos são aplicados processos de seleção, cruzamento e reprodução. Em todo ciclo, uma nova população é gerada a partir da anterior. A finalidade é o aumento da capacidade de adaptação dos indivíduos ao problema proposto.

O trabalho tem como objetivo analisar e comparar os *frameworks AFORGE* e *JGAP*, tendo como base estudos teóricos sobre Algoritmos Genéticos e o

desenvolvimento de uma aplicação utilizando os *frameworks* citados, auxiliando, assim, a resolver o Problema do Caixeiro Viajante (PCV).

Algoritmos Genéticos se tornam eficientes na resolução de problemas computacionais e também por ser reconhecido pela simplicidade de implementação e de encontrarem soluções satisfatórias.

Com o crescente uso de meios computacionais para o auxílio nas resoluções dos problemas, se faz necessário a existência de tecnologias que permitam o seu uso, tais como os *frameworks*, aqui abordados. Portanto, é de grande importância que haja um conhecimento sobre estas tecnologias que são disponibilizadas. Mas, com a existência de diferentes *frameworks* para o desenvolvimento de AGs, é necessária uma análise para que se possam conhecer as características de alguns deles bem como vislumbrar a que critérios eles respondem para melhor domínio das ações a serem realizadas.

Existem vários *frameworks* que podem ser utilizados para trabalhar com os AGs. Algumas opções de *frameworks open-source* são:

- *AFORGE* – na linguagem C#;
- *GAlib* - na linguagem C++;
- *GAUL* – na linguagem C;
- *JAGA* – na linguagem *Java*;
- *JGAP* – na linguagem *Java*;
- *Psyvolve* – na linguagem *Python*.

A partir do conhecimento da teoria de AGs, critérios foram estabelecidos para analisar *frameworks*, que se dispõe a auxiliar no seu desenvolvimento e, a partir destes critérios, foi realizada uma comparação entre dois *frameworks*: *AFORGE* e *JGAP.*

Os *frameworks* foram escolhidos por terem poucas referências e utilização. Consequentemente o trabalho terá muita importância nesse aspecto, por servir de referência para auxílio à decisão de qual *framework* utilizar ou mesmo verificar os pontos fortes de cada um.

O trabalho é estruturado. Primeiramente, apresentando a introdução, em seguida, os conceitos encontrados na seção de revisão de literatura sobre AGs, composto por: inteligência artificial, genética, algoritmos genéticos com seus respectivos operadores genéticos. Depois é apresentada a metodologia utilizada no desenvolvimento do projeto, assim como *hardware* e *softwares*. Posteriormente, na seção de resultados e discussão, composto por: descrição do PCV, implementação utilizando os dois *frameworks* e suas comparações. Em seguida, são apresentadas as referências bibliográficas utilizadas como fontes de pesquisa e estudo. E por fim, os apêndices.

2 REFERENCIAL TEÓRICO

2.1. Inteligência Artificial

Inteligência Artificial significa “a arte de criar máquinas com capacidade de realizar funções que quando realizadas por pessoas requerem inteligência” (RUSSELL e NORVIG, 2004, p. 5). Ou seja, IA é uma área de estudo que se preocupa em como imitar o ser humano, não só fisicamente (movimentos e biologia), mas, também, referente às suas faculdades mentais (inteligência e intelecto, por exemplo). Russel e Norvig citam algumas definições de IA:

- sistemas que pensam como seres humanos: “O novo e interessante esforço para fazer os computadores pensarem... máquinas com mentes, no sentido total e literal”. (HAUGELAND, 1985 *apud* RUSSEL e NORVIG, 2004, p. 5);
- sistemas que pensam racionalmente: “O estudo das faculdades mentais pelo uso de modelos computacionais”. (CHARNIAK e MCDERMOTT, 1985 *apud* RUSSEL e NORVIG, 2004, p. 5);
- sistemas que atuam como seres humanos: “A arte de criar máquinas que executam funções que exigem inteligência quando executadas por pessoas”. (KURZWEIL, 1990 *apud* RUSSEL e NORVIG, 2004, p. 5);
- sistemas que atuam racionalmente: “A Inteligência Computacional é o estudo do projeto de agentes inteligentes”. (POLE, 1998 *apud* RUSSEL e NORVIG, 2004, p. 5).

A fim de simular em um sistema computacional a maneira como os organismos vivos resolvem problemas, foram desenvolvidos alguns principais paradigmas da IA, dos quais se podem destacar: (BARRETO, 2001, p. 3):

- IA Simbólica (IAS): qualquer sistema, seja ele humano ou máquina, possa operar sua inteligência manipulando estruturas de dados compostos por símbolos. Um exemplo de uso é em Sistemas periciais e agentes;
- IA Conexionista (IAC): “aplica-se a problemas mal definidos, mas que são conhecidos através de exemplos”. Um exemplo de sua aplicação é em redes neurais artificiais (RNAs);
- IA Evolucionária (IAE): baseia-se na seleção natural, usados para solucionar problemas bem definidos de sobrevivência de uma espécie.

Os AGs encaixam-se no paradigma da IA Evolucionária, pois se baseiam na seleção natural. AGs são definidos como uma técnica de IA que tem como base as teorias da evolução e seleção natural, que foram inicialmente desenvolvidas por Charles Darwin.

Para melhor entendimento sobre AGs, uma breve introdução de genética será descrita na próxima seção.

2.2. Genética

Charles Robert Darwin (1809 – 1882), em cinco anos de estudos na fauna e flora em diferentes regiões, estabeleceu, na metade do século XIX, a teoria da seleção natural. Para Darwin, indivíduos da mesma espécie não são iguais geneticamente e, sim, apresentam pequenas variações em suas características.

Darwin defende a teoria de que indivíduos que tenham melhores características genéticas possuem mais chances de sobrevivência e de passar suas características genéticas para seus filhos. Consequentemente, indivíduos com características genéticas inferiores têm menos chance de sobrevivência e tenderão a desaparecer.

Um exemplo muito utilizado para o entendimento da teoria da seleção natural é o do crescimento do pescoço da girafa. Jean-Barptiste Lamarck defende a teoria de que cada espécie evolui de acordo com suas necessidades de sobrevivência. Segundo Lamarck, girafas eram todas iguais, com pescoços pequenos, mas com a necessidade de alcançar alimentos no topo das árvores, elas tinham que esticar o pescoço para conseguir se alimentar. Já Darwin defende a ideia de que girafas com características de pescoços pequenos não conseguiam se alimentar do topo das árvores e morriam de fome. Logo, girafas com características de pescoços longos conseguiam se alimentar, tendo assim mais chances de produzirem girafas com essa característica de sobrevivência.

Depois de exibidos os conceitos de IA e teorias básicas de genética, é possível entender melhor os AGs. No próximo tópico serão apresentadas definições de AGs.

2.3. Algoritmos Genéticos

Os AGs são algoritmos de otimização e busca baseados na teoria da evolução natural de Darwin, que indivíduos com melhores características genéticas têm maior chance de sobrevivência, e de produzirem filhos mais aptos e, consequentemente, os indivíduos menos aptos tenderão a desaparecer (REZENDE, 2003, p. 226). Ou

seja, AGs utilizam técnicas de evolução para alterar a população de indivíduos na tentativa de identificar uma solução ótima (COPPIN, 2010, p. 334).

Os AGs são indicados para a solução de problemas de otimização complexos, como o do PCV, utilizado neste trabalho, que envolvem um grande número de variáveis e, consequentemente, espaços de soluções de dimensões elevadas (MIRANDA, [s.a.], *online*).

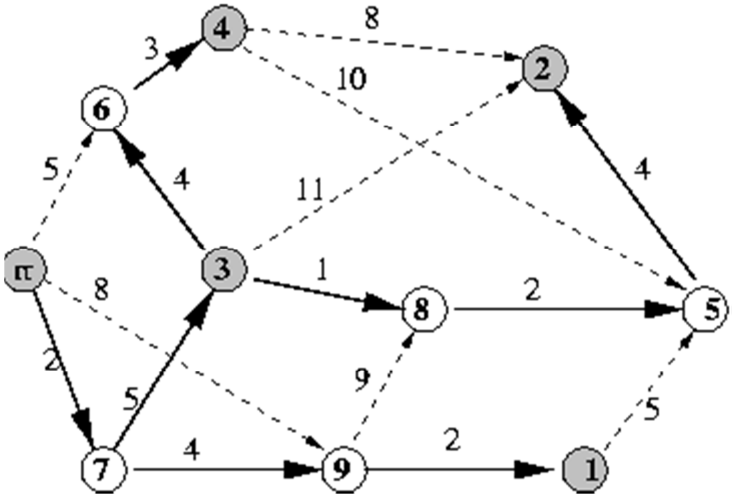

Figura 1 - O Problema do Caixeiro Viajante.

O PCV descreve um conjunto de *n* cidades *N = {1, 2, 3..., n}*, e uma matriz de distâncias ligando cada par de cidades de *N*. O objetivo do PCV é partir de uma cidade origem, visitar uma única vez cada uma das *n-1* cidades restantes, e retornar à cidade origem, minimizando a distância total percorrida (OCHI, [s.a.], *online*). A Figura 1 exibe as cidades e as distâncias entre elas.

Em geral, uma solução candidata é chamada de indivíduo ou cromossomo. O conjunto de indivíduos, simultaneamente avaliados, é chamado de população. Cada indivíduo é associado a um grau de aptidão, que mede a capacidade de solução representada pelo indivíduo, para resolver um dado problema (BRAGA, 2000, p.

132). Esses indivíduos possuem características extremamente importantes e interessantes. Tais características podem ser modificadas buscando tornar-se cada vez mais perfeitas, a fim de torná-las soluções ótimas. Essas modificações são realizadas por meio da mutação, que é capaz de gerar filhos para uma próxima geração, realizando sua reprodução e proporcionando que essas melhores características se perpetuem por várias gerações (REZENDE, 2003, p. 229).

AGs realizam um processo iterativo, chamado de geração. Cada geração utiliza funções de seleção e reprodução – cruzamento e mutação – aplicadas a cada indivíduo da população, O tamanho da população depende de cada problema e recursos computacionais disponíveis. “Ao final desse processo, é esperada uma solução ótima ou quase ótima”. (BRAGA, 2000, p. 132)

Para o melhor entendimento, na tabela 1 é mostrado um comparativo da relação entre a natureza e os AGs.

Tabela 1 – Relação Natureza x Algoritmos Genéticos (PACHECO, 1999, p. 1).

Natureza	Algoritmos Genéticos
Cromossomo	Palavra binária, vetor, etc
Gene	Característica do problema
Alelo	Valor da característica
Loco	Posição na palavra
Genótipo	Estrutura
Fenótipo	Estrutura submetida ao problema
Indivíduo	Solução
Geração	Ciclo

Cromossomo é um conjunto de genes, sendo que, alelo é referido a um valor dentro do conjunto de valores que um gene pode assumir. O caso mais simples é o

uso de valores binários (0 e 1). Na Figura 2, pode ser visto mais detalhada a relação entre eles.

Genótipo de um indivíduo é a informação codificada em um genoma (conjunto de genes de um determinado indivíduo). Fenótipo é a decodificação do genótipo, em um AGs é a solução. Indivíduos representam uma possível solução para o problema a ser tratado. Geração é cada passo do processo de evolução do AG, criada a partir da população anterior, e a ultima é atualizada (LUCAS, 2002, p. 60)

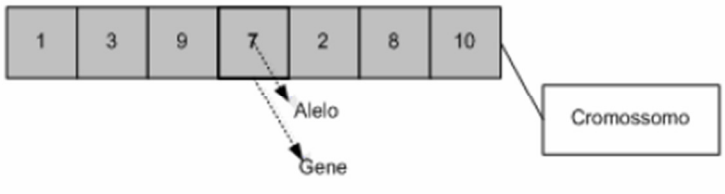

Figura 2 – Cromossomo.

Os AGs diferenciam-se dos métodos tradicionais de busca e otimização em quatro aspectos (BRAGA, 2000, p. 131):

- trabalham com uma codificação do conjunto de parâmetros e não com os próprios parâmetros;
- trabalham com uma população de soluções candidatas simultaneamente e não com uma única solução;
- utilizam informações de custo e recompensa, e não derivadas de funções;
- utilizam regras de transição probabilísticas, e não determinísticas;

A figura 3 exibe o ciclo de vida de um Algoritmo Genético.

```
1. t = 0;
2. Gerar População Inicial P(0);
3. para todo cada indivíduo i da população atual P(t) faça
4.     Avaliar aptidão do indivíduo i;
5. fim para
6. enquanto Critério de parada não for satisfeito faça
7.     t = t + 1;
8.     Selecionar população P(t) a partir de P(t – 1);
9.     Aplicar operadores de cruzamento sobre P(t);
10.          Aplicar operadores de mutação sobre P(t);
11.          Avaliar P(t);
12.      fim enquanto
```

Figura 3 - Ciclo de vida de um Algoritmo Genético (REZENDE, 2003, p. 230).

Como se pode observar na Figura 3, primeiramente, é definida uma variável para controlar o critério de parada; salvo se o critério de parada for pelo número de gerações. Posteriormente, a população inicial é gerada, a avaliação e definição de um valor de aptidão de cada indivíduo da população são realizadas e, depois de avaliado, uma iteração é iniciada. Após, os indivíduos são selecionados para a aplicação dos operadores de reprodução: operadores de cruzamento e operadores de mutação. Em seguida, os melhores indivíduos e dados estatísticos são colocados e armazenados para avaliação. Esse processo de selecionar indivíduos, aplicar operadores de cruzamento, operadores de mutação e avaliar são repetidos até que a condição de parada seja satisfeita.

2.3.1. Operadores Genéticos

Os AGs funcionam com uma série de passos denominados como operadores genéticos. Cada operador genético segue passos bem estruturados a fim de serem

aplicados em cromossomos e chegando a resultados mais satisfatórios. Os operadores mais utilizados são: inicialização, aptidão, seleção, cruzamento e mutação, explicados a seguir.

2.3.1.1. Inicialização

A inicialização gera automaticamente *n* cromossomos em uma população, criando assim uma população que será usada nos próximos operadores genéticos.

O operador de inicialização utiliza funções aleatórias para gerar seus indivíduos ou cromossomos, "sendo este um recurso simples que visa a fornecer maior biodiversidade" (GOLDBERG, 1989 e GEYER-SCHULTZ, 1997 *apud* LUCAS, 2002, p. 10). Ou seja, aumenta a diversidade genética dos indivíduos e, consequentemente, aumenta o alcance do espaço de pesquisa. Os operadores de inicialização mais utilizados são:

- **inicialização randômica uniforme:** cada gene do indivíduo receberá como valor um elemento do conjunto de alelos (diz respeito ao conjunto de valores que um gene pode assumir), sorteados de forma aleatoriamente uniforme. A Figura 4 exibe uma função de inicialização de forma randômica uniforme;
- **inicialização randômica não uniforme:** são inseridos na população indivíduos que têm determinados em seus genes valores que possuam uma maior probabilidade de serem escolhidos;
- **inicialização randômica com "dope":** em uma população são inseridos cromossomos gerado aleatoriamente, podendo assim apresentar o risco de fazer com que um ou mais super indivíduos tendam a dominar o processo de evolução (LUCAS, 2002, p. 15);

- **inicialização parcialmente enumerativa:** são inseridos na população indivíduos de forma a fazer com que esta comece o processo de evolução possuindo todos os esquemas possíveis de uma determinada ordem (LUCAS, 2002, p. 16);
- **distribuição de *cauchy*:** também chamado de função de distribuição ou *Lorentz*, possui a formula $f(x) = \frac{1}{\pi(1+((x-t))/s)^2)}$ com `t` é o parâmetro de localização e `s` é o parâmetro de escala. A função é usada para ter um valor mediano entre os valores de `t` e `s` (HANDBOOK, 2012, *online*);
- **distribuição normal:** também chamado de função *Gaussian* (ou gaussiana), possui a formula $\emptyset(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$, define-se a média representando o desvio padrão pela letra ∅ (SEDGEWICK, 2007, p. 47).

A fim de se obter o funcionamento de um AG, faz-se necessário criar uma função que inicie uma população de indivíduos. A seguir, descreve-se tal função na linguagem *Java*.

```
1. public static Chromosome[] inicializacao(int populationSize, int
   chromosomeSize){
2.     Chromosome[] population = new Chromosome[populationSize];
3.     for (int i = 0; i < populationSize; i++){
4.         Gene[] genes = new Gene[chromosomeSize];
5.         for (int j = 0; j < chromosomeSize; j ++){
6.             genes[j] = Gene();
7.         }
8.         Chromosome chromosome = new Chromosome();
9.         chromosome.setGenes(genes);
10.           population[i] = chromosome;
11.       }
12.       return population;
13.   }
```

Figura 4 - Função inicializacao.

Na Figura 4, a função de `inicializacao` tem como parâmetro duas variáveis *`int`*`: populationSize`, que informa a quantidade de indivíduos da população e `chromosomeSize`, que informa a quantidade de genes de cada cromossomo. Posteriormente na função:

- Na linha 2, cria-se um vetor do tipo `Chromosome`, nomeado como `population`, que recebe um vetor `Chromosome` do tamanho da quantidade de indivíduos da população;
- Nas linhas entre 3 e 11, cria-se um laço de repetição que percorre os indivíduos da população;
- Na linha 4, cria-se um vetor do tipo `Gene`, nomeado `genes`, que recebe um vetor do tamanho da quantidade de genes do cromossomo;
- Nas linhas entre 5 e 7, cria-se um laço de repetição que percorre a quantidade de genes do cromossomo;
- Na linha 6, o vetor `genes` recebe um gene que é retornado pela função externa `Gene()`;
- Na linha 8, cria-se uma nova variável do tipo `Chromosome`, nomeada `chromosome`, que recebe um construtor do objeto `Chromosome`;
- Na linha 9, determinam-se os genes da variável `chromosome` , recebendo os valores do vetor `genes`;
- Na linha 10, acrescenta-se o `chromossome` para o vetor `population`.

A fim de se obter a impressão dos cromossomos, primeiramente é executado o código da Figura 4.

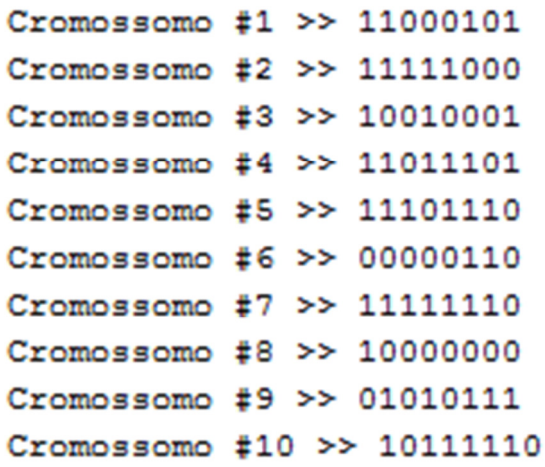

Figura 5 - Tela de impressão utilizando a função de inicialização.

A Figura 5 apresenta a tela de impressão de uma população de 10 cromossomos, gerada através da função de inicialização exibida na Figura 4.

Após gerar uma população de indivíduos pelo operador de inicialização, o operador de aptidão é acionado.

2.3.1.2. Aptidão

A função de aptidão, chamada também de objetivo, avaliação ou *fitness*, emprega o papel de avaliar todos os indivíduos da população, definindo uma nota de aptidão de acordo com a adaptação no ambiente em que está.

A teoria da evolução define que a chance de sobrevivência dos indivíduos mais aptos são maiores. Em AGs os indivíduos com o valor de aptidão alto têm maiores chances de serem selecionados.

Uma vez definido o valor de aptidão, a população é levada para o processo de seleção e reprodução (COPPIN, 2010, o. 336). “A função de aptidão é para um AGs o que o meio ambiente é para seres humanos”. (PACHECO, 1999, p. 3)

Para obter a aptidão da população, o próximo código é executado como exemplo utilizando a linguagem *Java.*

```
1. public static double[] aptidaoPopulacao(Chromosome[] chromosome){
2.     double[] ret = new double[chromosome.Length];
3.     for (int i = 0; i < chromosome.Length; i++){
4.         String val1 = "";
5.         String val2 = "";
6.         Gene[] genes = chromosome[i].getGenes();
7.         for (int f = 0; f < 4; f++)
8.             val1 += genes[f].getVal();
9.         for (int d = 4; d < genes.Length; d++)
10.                 val2 += genes[d].getVal();
11.             String valString = bin2dec(val1)+"."+bin2dec(val2);
12.             ret[i] = Double.Parse(valString);
13.         }
14.         return ret;
15.     }
```

Figura 6 - Função aptidão.

Na Figura 6, a função de `aptidaoPopulacao` tem como parâmetro um vetor de `Cromossomo`, que é repassada a população de cromossomos. Posteriormente na função:

- Na linha 2 cria-se um vetor do tipo `double`, nomeado *`ret`*, que recebe um vetor `double` do tamanho da quantidade de elementos do vetor `chromosome`;
- Nas linhas entre 3 e 13, cria-se um laço de repetição que tem como tamanho a quantidade de elementos do vetor `chromosome`;
- Na linha 4, cria-se uma variável do tipo `String`, nomeada como `val1`, que recebe uma *`String`* vazia;

- Na linha 5, cria-se uma variável do tipo `String`, nomeada como `val2`, que recebe uma `String` vazia;
- Na linha 6, cria-se um vetor do tipo `Gene`, onde recebe a quantidade de genes do vetor `chromosome` atual;
- Na linha 7, cria-se um laço de repetição que tem como tamanho 4;
- Na linha 8, a variável `val1` recebe ela mesma mais o gene atual;
- Na linha 9, cria-se um laço de repetição que começa do valor 4 e percorre até a quantidade de elementos do vetor `genes`;
- Na linha 10, a variável `val2` recebe o seu valor mais o gene atual;
- Na linha 11, cria-se uma variável do tipo `String`, nomeada `valString`, que recebe a conversão de binário para decimal das variáveis: *val1* e `val2`;
- Na linha 12, o vetor `ret` atual recebe a variável `valString` convertida em `double`;
- Na linha 14, a variável `ret` é retornada pela função.

A seguir, é utilizado o código da Figura 6 a fim de se obter o valor de aptidão para cada cromossomo da população.

```
Posicão> 0, Genes> 11010001, Aptidão> 13.1
Posicão> 1, Genes> 01111101, Aptidão> 7.13
Posicão> 2, Genes> 01000000, Aptidão> 4.0
Posicão> 3, Genes> 11010111, Aptidão> 13.7
Posicão> 4, Genes> 11011000, Aptidão> 13.8
Posicão> 5, Genes> 00011101, Aptidão> 1.13
Posicão> 6, Genes> 10110001, Aptidão> 11.1
Posicão> 7, Genes> 10011000, Aptidão> 9.8
Posicão> 8, Genes> 00011101, Aptidão> 1.13
Posicão> 9, Genes> 10111100, Aptidão> 11.12
```

Figura 7 - Tela de impressão utilizando a função de aptidão.

Na Figura 7, apresenta uma tela de impressão da função de aptidão, que exibe a posição, genes e aptidão de cada cromossomo.

Após definir um valor de aptidão para cada cromossomo da população, o operador de seleção é acionado. Tal operador é descrito na próxima seção.

2.3.1.3. Seleção

A partir da nota de aptidão, explicada anteriormente, é definida uma nota para todos os indivíduos da população. Indivíduos com maior aptidão têm maiores chances de serem selecionados.

Para garantir que indivíduos com valor de aptidão baixa também sejam proporcionalmente selecionados – afinal, eles podem carregar genes capazes de gerar ótimos filhos – a função de seleção é aplicada, emulando o processo de seleção natural. A quantidade de indivíduos que serão selecionados depende do que foi definido para o problema.

Existem alguns métodos para realizar a seleção dos indivíduos. Serão apresentados seis diferentes, mas com o mesmo objetivo:

- método roleta;
- método por classificação;
- método torneio;
- amostragem universal estocástica;
- método limiar;
- método melhor cromossomo.

Estes métodos serão explicados nas seções a seguir.

2.3.1.3.1. Método Roleta

No método roleta os indivíduos de uma população são separados pelo valor de aptidão. Depois de agrupados, são colocados em uma roleta e divididos de acordo com a quantidade de indivíduos de cada aptidão.

Dando sequência, o processo de seleção é iniciado, com a roleta sendo girada *n* vezes. Tem-se *n* como o número de indivíduos a serem selecionados até satisfazer a quantidade de população daquela geração. Durante esse processo, a cada giro da roleta o indivíduo selecionado é reservado em um local intermediário. Indivíduos com maior aptidão têm maiores chances de serem selecionados, pois ocupam maior área na roleta.

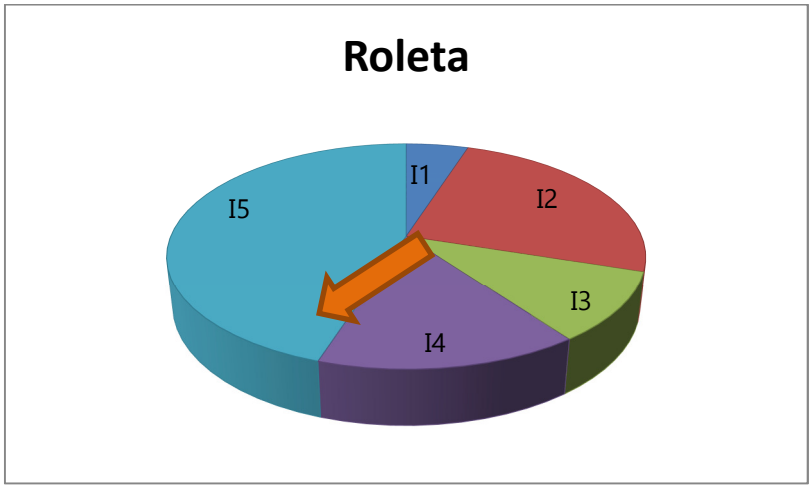

Figura 8 - Método roleta (BRAGA, 2000, p. 135).

A Figura 8 apresenta a roleta. A seta representa qual indivíduo será selecionado. E cada parte representa indivíduos com seu respectivo valor de aptidão (BRAGA, 2000, p. 135).

Para obter o cromossomo utilizando o método roleta, o código a seguir é executado como exemplo utilizando a linguagem *java.*

```
1. public static Chromosome select(Chromosome[] chromosome){
2.     double amountFitness = 0;
3.     double cumulativeFitness = 0;
4.     Chromosome ret = null;
5.     Collection<Object[]> roulettle = new ArrayList<Object[]>();
6.     for (int i = 0; i < chromosome.Length; i++){
7.         amountFitness += Util.individualFitness(chromosome [i]);
8.     }
9.     for (int i = 0; i < chromosome.Length; i++){
10.             roulettle.Add(new Object[] {new
   Double(cumulativeFitness), new Double(cumulativeFitness + Util.
   individualFitness(chromosome [i]) / amountFitness, chromosome [i]});
11.             cumulativeFitness += Util.individualFitness(chromosome
   [i]) / amountFitness
12.         }
13.         double sortition = Marth.random();
14.         Interator<Object[]> it3 = roulettle.iterator();
15.         while(it3.hasNext()){
16.             Object[] current = it3.next();
17.             double lowerLimit = ((Double)
   current[0]).doubleValue();
18.             double upperLimit = ((Double)
   current[1]).doubleValue();
19.             Chromosome crom = (Chromosome) current[2];
20.             if (sortition >= lowerLimit && sortition < upperLimit){
21.                 ret = crom;
22.             }
23.         }
24.         return ret;
25.     }
```

Figura 9 - Tela de impressão utilizando o método roleta.

A Figura 9 exibe uma função que retorna um cromossomo e como parâmetro um vetor do tipo `Chromosome`. O restante da função contém:

- Na linha 2, cria-se uma variável do tipo `double` nomeada `amountFitness` que recebe o valor 0;
- Na linha 3, cria-se uma variável do tipo `double`, nomeada `cumulativeFitness`, que recebe o valor 0;
- Na linha 4, cria-se uma variável do tipo `Chromosome`, nomeada `ret`, que recebe um valor nulo;
- Na linha 5, cria-se uma coleção nomeada `roulettle`;
- Nas linhas 6 a 8, somam-se os valores de aptidão de todos os cromossomos da população;
- Nas linhas 9 a 16, percorrem-se todos os cromossomos da população;
- Na linha 10, adiciona-se a média de aptidão dos indivíduos acumulado até o momento. O valor de todos os indivíduos, mais o valor do individuo atual, dividido pela soma total do valor de aptidão de todos os indivíduos, e por fim, o cromossomo atual;
- Na linha 11, faz-se a aptidão acumulada, que o valor de aptidão do individuo atual, dividido pelo valor de aptidão de todos os indivíduos;
- Na linha 13, cria-se uma variável do tipo `double`, que recebe um valor aleatório;
- Na linha 14, cria-se uma variável do tipo `interator`, podendo assim acessar cada elemento da variável `roulettle`;
- Nas linhas entre 15 e 23, cria-se um laço de repetição `while`, que percorre a variável `it`, chamando o próximo elemento `roulettle`;
- Na linha 16, cria-se um vetor que obtém o atual elemento `roulettle`;

- Na linha 17, cria-se um `double`, nomeado como `lowerLimit` (ou limite inferior), que obtém o valor em `double` da posição zero do atual elemento `roulettle`;
- Na linha 18, cria-se um `double`, nomeado como `uppeLimit` (ou limite superior), que obtém o valor em `double` da posição um do atual elemento `roulettle`;
- Na linha 19, cria-se uma variável do tipo *Chromosome*, nomeada *crom*, que recebe a posição dois do elemento *roulettle*;
- Na linha 20, cria-se uma condição de parada que o valor da variável `sortition` seja maior que a variável `lowerLimit` e menor que a variável `upperLimit`;
- Na linha 21, a variável de retorno `ret` recebe a variável `crom`.

  Na próxima seção é descrito o método por classificação.

2.3.1.3.2. Método por Classificação

"A aptidão sozinha não é o mais recomendado para se escolher o tamanho das fatias da roleta, pois pode levar a uma convergência prematura do AGs e também desfavorecer indivíduos menos aptos" (BRAGA, 2000, p. 135). Ou seja, a alocação dos indivíduos na roleta não é satisfeita com a nota de aptidão por poder originar uma solução não muito satisfatória e também não favorecer indivíduos com valor de aptidão baixo.

Para evitar a convergência prematura do AGs e desfavorecer os indivíduos menos aptos, Braga (2000, p. 135) menciona a utilização do método de *ranking* ou classificação. No método sem o *ranking*, a roleta é dividida de acordo com a nota de

aptidão de cada indivíduo. Já com ele, a roleta é repartida de acordo com o valor do *ranking*. É definida a fatia de cada indivíduo por meio de escolha de um valor real entre 0.0 e 1.0 e, em seguida, é calculada a nota normalizada. São colocados indivíduos em fatias maiores, de acordo com a sua posição no *ranking* (BRAGA, 2000, p. 135).

$$nota_j^{normalizada} = \frac{nota_j^{original}}{\sum_{i=1}^{N} nota_j^{original}}$$

Figura 10 - Fórmula nota normalizada (BRAGA, 2000, p. 133).

A nota original é definida na função de aptidão. Para melhor definição do *ranking*, as somas de todos os indivíduos não ultrapassam o valor um, que corresponde à totalidade da probabilidade. A fórmula para resolver a nota normalizada é ilustrada na Figura 10. Para cada indivíduo é obtida sua nota original e seu total dividido pela soma das notas originais de todos os indivíduos.

Para melhor entendimento da fórmula da nota normalizada, Braga (200, p.136) exemplifica com a definição de uma tabela com cinco indivíduos com suas respectivas nota original e nota normalizada.

Tabela 2 - Ranking com cinco indivíduos (BRAGA, 2000, p. 136).

Indivíduo	Nota original	Nota normalizada	*Ranking*
I1	0,15	0,05	1
I2	0,75	0,25	4
I3	0,30	0,10	2
I4	0,45	0,15	3
I5	1,35	0,45	5

Ao observar a Tabela 2, o indivíduo “I1” possui uma nota original “0,15” e a soma da nota original de todos os indivíduos é igual a 3. O cálculo para se chegar a nota normalizada é 0,15 dividido por 3, resultando, assim, no valor de “0,05”, ou seja, “0,05” é a nota normalizada de “I1”. Depois de calcular a nota normalizada de todos os indivíduos, é visível que a soma é idêntica a 1. A partir disso, o valor do *ranking* é formado.

2.3.1.3.3. Método Torneio

Com o método torneio, vários indivíduos são escolhidos aleatoriamente em uma população temporária. Dessa população, são escolhidas aleatoriamente *n* indivíduo, *n* é o total de indivíduos. Após, é escolhido um indivíduo com maior nível de aptidão entre eles.

Depois, são escolhidos *n* indivíduos resultantes desse processo descrito anteriormente, ou seja, o processo é feito *n* vezes, até que seja satisfeito o critério de preencher uma população intermediária, população essa que será utilizada durante o processo dos operadores de cruzamento e mutação.

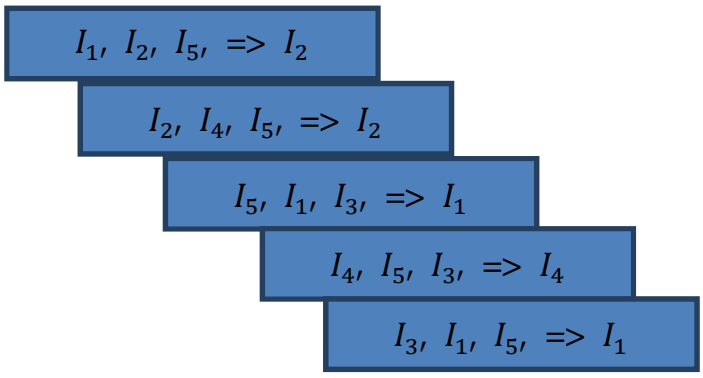

Figura 11 - Método torneio (REZENDE, 2003, p. 233).

No exemplo da Figura 11 é ilustrada como é feita a seleção. Primeiramente, são separados grupos de três indivíduos, escolhidos aleatoriamente. Em seguida, é escolhido o indivíduo com maior aptidão dentro do grupo. Este processo é repetido até que a população intermediária seja satisfeita.

2.3.1.3.4. Método de Amostragem Universal Estocástica

No Método de Amostragem Universal Estocástica, várias agulhas são posicionadas igualmente. O total de agulhas é definido pelo número de indivíduos que serão selecionados para geração. A roleta é girada uma única vez.

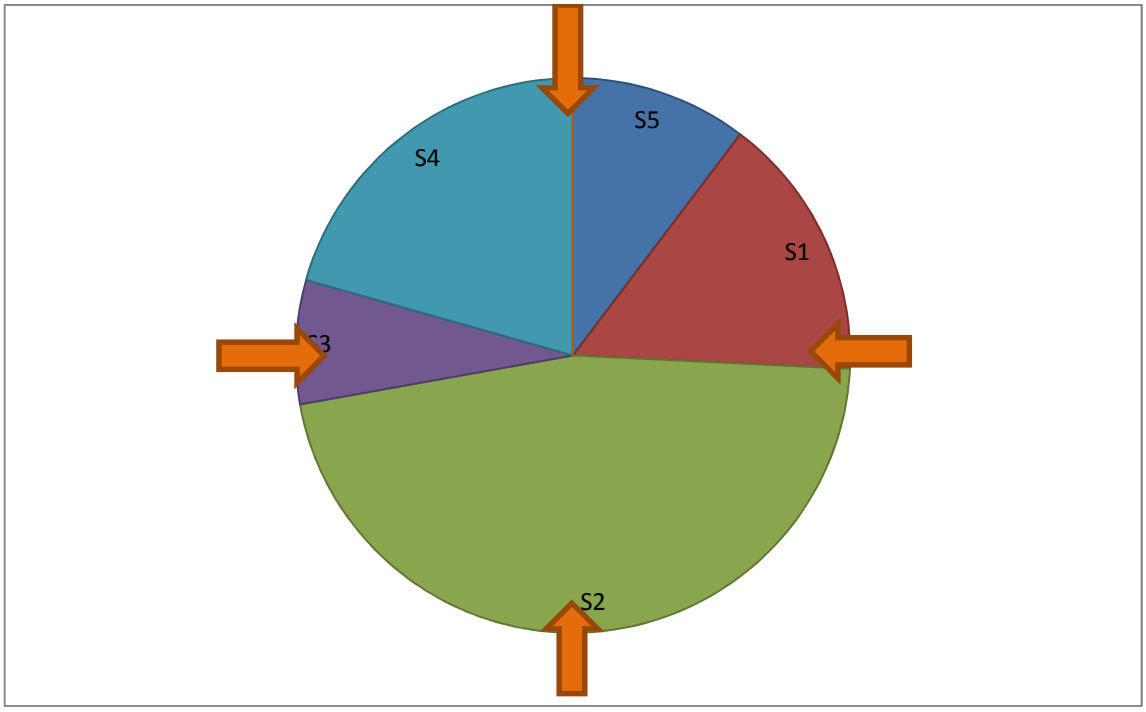

Figura 12 - Método amostragem universal estocástica (REZENDE, 2003, p. 233).

Este método é ilustrado na Figura 12 com o uso de um círculo. Cada indivíduo tem uma probabilidade de ser escolhido, de acordo com sua nota de aptidão. Um

círculo é dividido em regiões, cada uma dessas regiões corresponde ao valor de aptidão do indivíduo. É colocada sobre o círculo uma roleta que é dividida igualmente por *n* agulhas, *n* é número de indivíduos a ser selecionado para satisfazer a geração. Posteriormente, a roleta é girada uma única vez e os indivíduos que serão selecionados são apontados pelas agulhas.

2.3.1.3.5. Método Limiar

Boa parte de uma população possui nível alto e outras de nível baixo. Dois limites são utilizados para classificar estas partes em duas categorias, chamadas de valores de limiar. Um deles é o limiar superior e outro o limiar inferior. Os limiares são baseados no conhecimento prévio do problema, Isto depende do domínio da aplicação, e em que é possível determinar quais podem ser os valores possíveis para o limiar. Assim, esta estratégia de seleção é aplicável para a classe de problemas em que temos algum conhecimento prévio.

Os valores limiar também significam valores aceitáveis para a população, e cada geração pode ter indivíduos acima do limiar superior, já os indivíduos que estão abaixo do limiar superior, mas acima do limiar inferior são mantidos para o cruzamento. A população abaixo do limiar inferior é descartada. Assim, em cada geração poucas partes serão descartadas, poucas serão mantidas e outras serão aprovadas para a próxima geração, garantindo assim a possibilidade de ter cromossomos suficientes para a reprodução (KHANBARY, 2009, p. 2).

Supondo que a população de cromossomos `p0` está acima do limiar superior, definido com o valor 500, e `q0` é valor entre os limiares superior e inferior e `r0` é abaixo do limiar inferior, definido com o valor 250. Na primeira geração, a população

total (p0+q0+r0) aleatoriamente fazem o cruzamento. Na geração seguinte, a população de cromossomos p1 está acima do limiar superior, q1 tem o valor entre os limiares superior e inferior e r1 tem valor abaixo do limiar inferior. A população de cromossomos r1 será descartada da população atual, por possuir valor inferior ao limiar inferior. Já p1 e *q0* são aptos para acasalar e produzirem descendentes.

2.3.1.3.6. Método Melhor Cromossomo

O método de seleção de melhor cromossomo (ou *Best Chromosome*), consiste em selecionar *n* cromossomos com maior valor de aptidão da população corrente. A população de cromossomos é ordenada de acordo com o valor de aptidão das soluções, por exemplo, em ordem decrescente. A seleção de um número pré-definido de cromossomos mais aptos asseguram que a melhor solução da população esta sempre selecionada.

Após aplicar o operador de seleção, aplica-se o operador de cruzamento, descrito na próxima seção.

2.3.1.4. Cruzamento

O operador de cruzamento, também chamado de *crossover*, é responsável por transferir o material genético de indivíduos pais para seus descendentes. *Crossover* é considerado um dos operadores mais importantes dos AGs, pois faz parte do processo de reprodução e consequentemente responsável por gerar indivíduos mais capazes a cada geração (iteração).

Após o operador de seleção, é realizada a obtenção de uma população de indivíduos qualificados. O processo de cruzamento é acionado. É preciso que dois indivíduos sejam escolhidos aleatoriamente da população corrente, e gerados dois novos indivíduos, carregando seu material genético (PACHECO, 1999, p. 4).

Dessa forma, pode-se concluir que o *crossover* é o operador responsável para que melhores características de determinados indivíduos sejam herdadas para as próximas gerações.

Existem alguns tipos de *crossover*, como ponto único, multiponto, ponto uniforme e guloso, que serão descritos a seguir.

2.3.1.4.1. Ponto Único

No cruzamento de ponto único são escolhidos dois indivíduos aleatoriamente na população corrente, e escolhendo um ponto de cruzamento em cada, dividindo o indivíduo em duas partes. Posteriormente, são extraídas as informações genéticas para dois descendentes (REZENDE, 2003, p. 237).

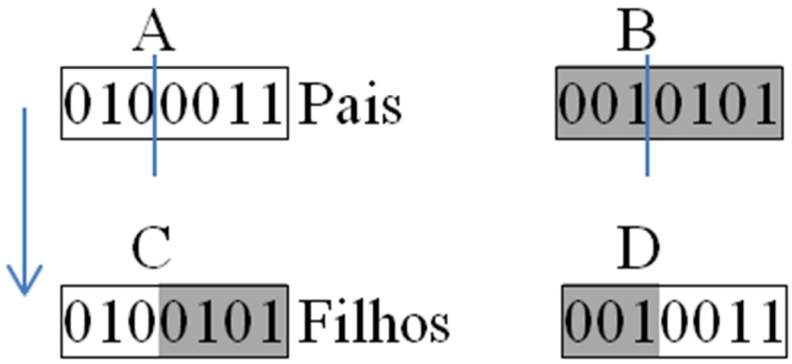

Figura 13 - Ponto único (REZENDE, 2003, p. 237).

A Figura 13 explica o conceito de cruzamento de ponto único. Existem dois pais, A e B, escolhidos aleatoriamente na população corrente. Um ponto é posicionado aleatoriamente entre os genes (linha azul), ficando com cada pai dois grupos de genes. Posteriormente, dois filhos ou proles são criados. Cada filho possui um grupo de gene de um pai e outro grupo de gene de outro pai.

A seguir é apresentado o código exemplo da implementação da função de cruzamento de ponto único.

```
1. public static Cromossomo[] cruzamentoPontoUnico(Cromossomo father1,
   Cromossomo father2, int limitGenes){
2.     int pointEnjoy = new Random().NextInt(8);
3.     Gene[] geneFather1 = new Gene[limitGenes];
4.     Gene[] geneFather2 = new Gene[limitGenes];
5.     for (int i = 0; i < pointEnjoy; i++){
6.         geneFather1[i] = new Gene(father2.getGenes()[i].getVal());
7.         geneFather2[i] = new Gene(father1.getGenes()[i].getVal ());
8.     }
9.     for (int i = pointEnjoy; i < limitGenes; i++){
10.              geneFather1[i] = new
   Gene(father1.getGenes()[i].getVal());
11.              geneFather2[i] = new
   Gene(father2.getGenes()[i].getVal());
12.          }
13.          Chromosome[] c = new Chromosome[2];
14.          Chromosome chromosome1 = new Chromosome();
15.          Chromosome chromosome2 = new Chromosome();
16.          chromosome1.setGenes(geneFather1);
17.          chromosome2.setGenes(geneFather2);
18.          c[0] = chromosome1;
19.          c[1] = chromosome2;
20.          return c;
21.      }
```

Figura 14 - Função cruzamentoPontoUnico.

Na Figura 14, a função de `cruzamentoPontoUnico` tem como parâmetro dois cromossomos e o limite de genes. A função é descrita a seguir:

- Na linha 2, cria-se uma variável do tipo `int`, nomeada `pointEnjoy`, que recebe um valor de ponto de corte aleatório entre os números 0 e 8;
- Na linha 3, cria-se um vetor do tipo `Gene`, nomeado `geneFather1`, que recebe um vetor de `Gene` com tamanho total do limite de genes;
- Na linha 4, cria-se um vetor do tipo *Gene* nomeado `geneFather2`, que recebe um vetor de `Gene` com tamanho total do limite de genes;
- Nas linhas entre 5 e 8, cria-se um laço de repetição do tamanho do limite de genes;
- Na linha 6, o vetor `geneFather1` na posição atual recebe um novo objeto do tipo `Gene`, que passa como parâmetro o gene atual da variável `geneFather2`;
- Na linha 7, o vetor `geneFather2` na posição atual recebe um novo objeto do tipo `Gene` que passa como parâmetro o gene atual da variável `geneFather1`;
- Na linha 9, cria-se um laço de repetição que seu início é o tamanho do ponto de corte e percorre até o limite de genes;
- Na linha 10, o vetor `geneFather1` na posição atual recebe um novo objeto do tipo `Gene` que passa como parâmetro o gene atual da variável `geneFather1`;
- Na linha 11, o vetor `geneFather2` na posição atual recebe um novo objeto do tipo `Gene` que passa como parâmetro o gene atual da variável `geneFather2`;

- Na linha 13, cria-se um vetor do tipo `Chromosome`, nomeado `c`, que recebe um vetor do tipo `Chromosome` com tamanho 2;
- Na linha 14, cria-se uma variável do tipo `Chromosome`, nomeada `chromosome1`, que recebe um novo objeto do tipo `Chromosome`;
- Na linha 15, cria-se uma variável do tipo `Chromosome`, nomeada `chromosome2`, que recebe um novo objeto do tipo *Chromosome*;
- Na linha 16, determina-se os genes do `chromosome1` recebendo a variável `geneFather1`;
- Na linha 17, determina-se os genes do `chromosome2` recebendo a variável `geneFather2`;
- Na linha 18, a variável `c` na posição zero recebe a variável `chromosome1`;
- Na linha 19, a variável `c` na posição um recebe a variável `chromosome2`;
- Na linha 19, a variável `c` é retornada pela função;

  Na próxima seção é descrito o tipo multiponto.

2.3.1.4.2. Multiponto

No cruzamento de multiponto, o processo é basicamente o mesmo do cruzamento de ponto único. A única diferença é que vários pontos de cruzamento podem dividir os indivíduos para serem usados na formação dos filhos. O exemplo é exibido na Figura 15. (REZENDE, 20038, p. 237).

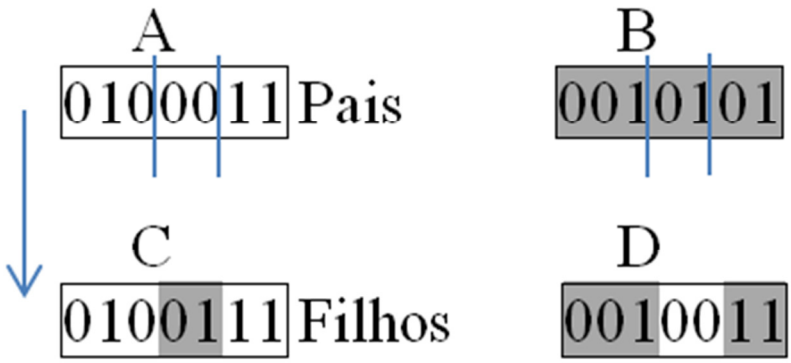

Figura 15 - Multiponto (REZENDE, 2003, p. 237).

A Figura 15 apresenta o cruzamento multiponto. Dois pais (A e B) são divididos por *n* pontos posicionados aleatoriamente entre os genes (linha azul), ficando com cada pai *n*+1 grupos de genes. Posteriormente, dois filhos ou proles são criados. Cada filho possui um grupo de gene de um pai e outro grupo de gene de outro pai.

```
1. public static Cromossomo[] cruzamentoDoisPontos(
2.     Cromossomo pai1,
3.     Cromossomo pai2,
4.     int limiteGenes){
5.         int[] pontosDeCortes = gerarDoisNumerosAleatorios();
6.         int ponto_de_corte1 = pontosDeCortes[0];
7.         int ponto_de_corte2 = pontosDeCortes[1];
8.         Gene[] gene_filho1 = new Gene[limiteGenes];
9.         Gene[] gene_filho2 = new Gene[limiteGenes];
10.              for (int i = 0; i < ponto_de_corte1; i++){
11.                  gene_filho1[i] = new
   Gene(pai1.getGenes()[i].getValor());
12.                  gene_filho2[i] = new
   Gene(pai2.getGenes()[i].getValor());
13.              }
14.              for (int i = ponto_de_corte1; i < ponto_de_corte2;
   i++){
15.                  gene_filho1[i] = new
   Gene(pai2.getGenes()[i].getValor());
16.                  gene_filho2[i] = new
   Gene(pai1.getGenes()[i].getValor());
17.              }
18.              for (int i = ponto_de_corte2; i < limiteGenes; i++){
19.                  gene_filho1[i] = new
   Gene(pai1.getGenes()[i].getValor());
20.                  gene_filho2[i] = new
   Gene(pai2.getGenes()[i].getValor());
21.              }
22.              Cromossomo[] c = new Cromossomo[2];
23.              Cromossomo cromossomo1 = new Cromossomo();
24.              Cromossomo cromossomo2 = new Cromossomo();
25.              cromossomo1.setGenes(gene_filho1);
26.              cromossomo2.setGenes(gene_filho2);
27.              c[0] = cromossomo1;
28.              c[1] = cromossomo2;
29.              return c;
30.      }
```

Figura 16 - Função cruzamentoDoisPontos.

Na Figura 16, a função de `cruzamentoDoisPontos` tem como parâmetro dois cromossomos e o limite de genes. Essa função é basicamente a função de `cruzamentoPontoUnico`, a diferença é que ao invés de ter dois laços de repetição, terá três laços, pois o primeiro percorre os pais da posição 0 até o primeiro ponto de corte. O segundo laço de repetição percorre do primeiro ponto de corte até o segundo ponto de corte. E o terceiro laço de repetição percorre do segundo ponto de corte até o limite dos genes.

2.3.1.4.3. Ponto Uniforme

No cruzamento uniforme é usado um método diferente do cruzamento de ponto único e multiponto, pois não são utilizados pontos de cruzamento e sim uma máscara. Esta máscara serve como referência para indicar que genes devem ser utilizados quando são criados os filhos, como se pode observar na Figura 17 (REZENDE, 2003, p. 237).

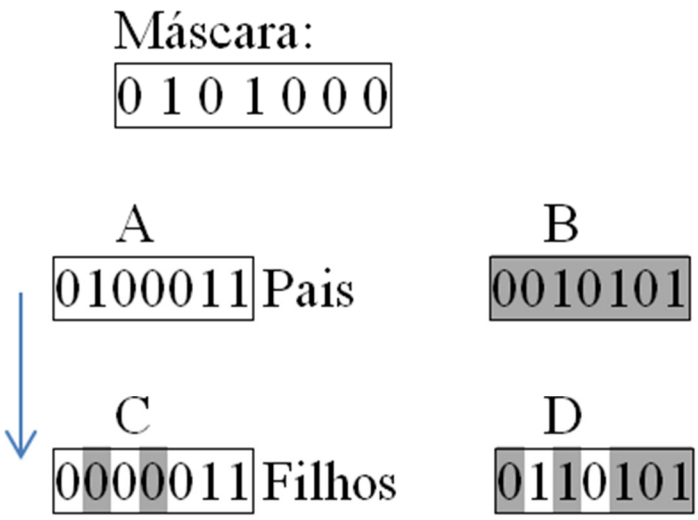

Figura 17 - Ponto uniforme (BRAGA, 2000, p. 137).

A Figura 17 exibe uma exemplificação do cruzamento de ponto uniforme. Dois pais baseiam-se em uma máscara para a criação de dois filhos. Para criar o filho C, foi definido que cada 0 da máscara referencia o pai A e cada 1 referencia o pai B na sua casa correspondente (posição). Em seguida, o filho D é criado, invertendo a definição da máscara, cada 0 corresponde ao pai B e cada 1 corresponde ao pai A.

```
1. public static Cromossomo[] cruzamentoPontoUniforme(
2.     Cromossomo mascara,
3.     Cromossomo pai1,
4.     Cromossomo pai2,
5.     int limiteGenes){
6.         Gene[] geneMascara = mascara.getGenes();
7.         Gene[] gene_filho1 = new Gene[limiteGenes];
8.         Gene[] gene_filho2 = new Gene[limiteGenes];
9.         for (int i = 0; i < limiteGenes; i++){
10.            byte temp = geneMascara[i].getValor();
11.            if (temp == 0) {
12.                gene_filho1[i] = new Gene(pai1.getGenes()[i].getValor());
13.                gene_filho2[i] = new Gene(pai2.getGenes()[i].getValor());
14.            }
15.            else{
16.                gene_filho1[i] = new Gene(pai2.getGenes()[i].getValor());
17.                gene_filho2[i] = new Gene(pai1.getGenes()[i].getValor());
18.            }
19.        }
20.        Cromossomo[] c = new Cromossomo[2];
21.        Cromossomo cromossomo1 = new Cromossomo();
22.        Cromossomo cromossomo2 = new Cromossomo();
23.        cromossomo1.setGenes(gene_filho1);
24.        cromossomo2.setGenes(gene_filho2);
25.        c[0] = cromossomo1;
26.        c[1] = cromossomo2;
27.        return c;
28.}
```

Figura 18 - Função cruzamentoPontoUniforme.

Na Figura 18, a função de `cruzamentoPontoUniforme` tem como parâmetro dois cromossomos pai, um cromossomo mascara e o limite de genes. No laço de repetição que se inicia na linha 9, percorre o vetor de 0 ate o limite de genes, em seguida, o valor do gene do cromossomo mascara é usado para definir qual herda o gene para seu filho.

2.3.1.4.4. Guloso

O cruzamento guloso é especialmente usado no PCV, garantindo a não repetição de genes no cromossomo resultante, ou seja, a não repetição de uma cidade na sua rota. Dados dois cromossomos pais:

- passo 1 - primeiro gene do cromossomo do primeiro pai é repassado diretamente para o cromossomo do filho.
- passo 2 - procura-se este gene no cromossomo de cada um dos pais e a partir dele calcula-se a distância até o gene seguinte.
- passo 3 - o filho irá receber o gene seguinte do pai em que a menor distância for encontrada. Se este gene já existir no cromossomo do filho, então é repassado o gene do outro pai (ou seja, com a maior distância). Caso este gene também já exista, é selecionado aleatoriamente um gene ainda não existente no filho.
- passo 4 - repetem-se os passos até que o cromossomo filho esteja completo (GRIGOLETTI, 2006, p. 18).

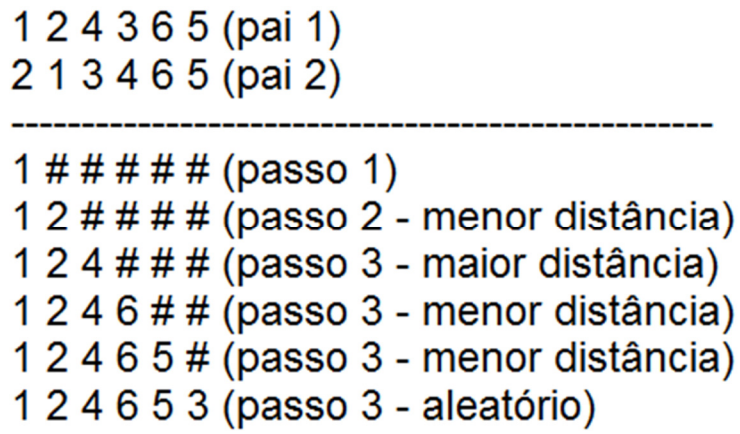

Figura 19 - Exemplo cruzamento guloso.

Na Figura 19 descreve um exemplo: são distribuídos dois cromossomos pais cujo cruzamento resulta em um cromossomo em que seus genes representem o menor caminho para percorrer todas as cidades.

Após o cruzamento, o operador de mutação é aplicado. Este operador é descrito na seção seguinte.

2.3.1.5. Mutação

Após a aplicação do operador de cruzamento, o operador de mutação é aplicado com uma probabilidade baixa. Geralmente utilizam-se valores para as taxas de mutação na faixa de 0,001 a 0,01% (REZENDE, 2003, p. 238). O objetivo é alterar um ou mais *bits* por outros de forma aleatória, a fim de “manter a introdução e manutenção da diversidade genética da população”. (BRAGA, 2010, p. 338).

Este operador simula a mutação em seres vivos, eles sofrem mutação alterando seu código genético naturalmente. Possui uma probabilidade baixa de mutação, mas ao longo do tempo podem causar efeitos importantes. Nos AGs esses efeitos são acelerados em pouco tempo.

O processo do operador de mutação segue uma lógica. Primeiramente, é escolhido aleatoriamente um conjunto de cromossomos iniciais da população corrente, posteriormente é repetida a mudança da propriedade genética do cromossomo atual até que são realizadas *n* mutações de cromossomos, *n* é o total de mutações definido (LUCAS, 2002, p.16).

Três operadores de mutação foram definidos para este trabalho:

- **mutação *flip*:** "cada gene a ser mutado recebe um valor sorteado do alfabeto válido". Na Figura 20 é apresentado um modelo.

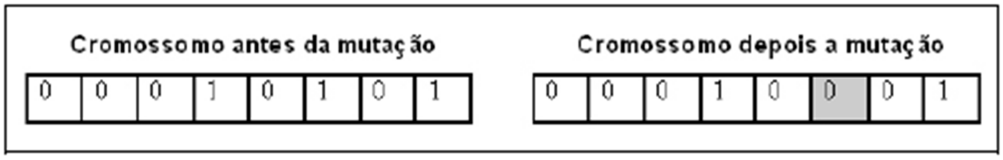

Figura 20 - Mutação *flip* (ROSA, 2009, p. 6).

A Figura 20 representa a mutação *flip*. É escolhido aleatoriamente o gene a ser mutado. Neste caso, o gene da posição 6 sofreu mutação, alterando seu valor de 1 para 0.

- **mutação por troca (*swap mutation*):** "são sorteados *n* pares de genes, e os elementos do par trocam valores entre si". Na Figura 21 é demonstrado um modelo.

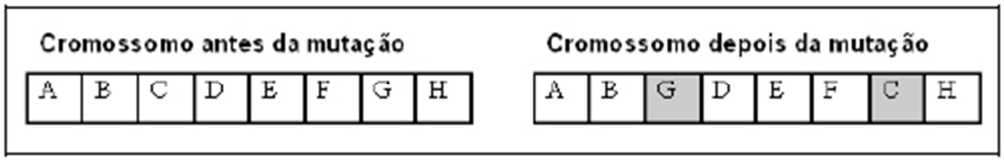

Figura 21 - Mutação *swap* (ROSA, 2009, p. 7).

A Figura 21 exemplifica a mutação por troca. Dois genes foram escolhidos para sofrer a mutação, estão eles: posição três com o valor “C”; e posição sete com o valor “G”. Após isso, os genes escolhidos trocam seus valores. “C” passa a ser “G”, e “G” passa a ser “C”.

- **mutação *creep*:** “um valor aleatório é somado ou subtraído do valor do gene”. Na Figura 22 é explicado um modelo.

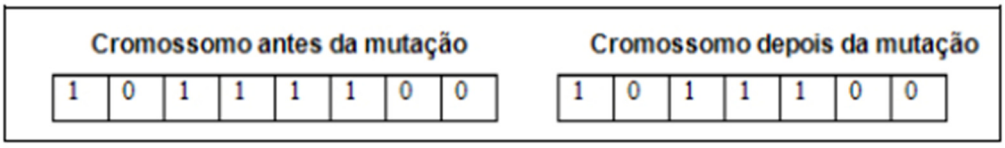

Figura 22 - Mutação *creep* (ROSA, 2009, p. 7).

Na Figura 22 pode ser verificada a mutação *creep*. O gene na posição seis foi removido.

2.3.2. Parâmetros Genéticos

Em cada geração, os AGs tentam encontrar indivíduos melhores do que a geração anterior, mas problemas no algoritmo podem apontar uma direção errada para a sua solução. “Nesses casos, provavelmente, está ocorrendo uma falha quantitativa ou qualitativa na ação dos operadores”. (SOARES, 1997, p. 41).

Quantitativamente refere-se ao tamanho da população, comprimento do cromossomo e às probabilidades de cruzamento e mutação. Já qualitativamente refere-se aos métodos de cruzamento e às estratégias de seleção (SOARES, 1997. p. 41).

O desempenho de um AGs é definido pelos parâmetros a serem utilizados:

- **tamanho da população:** a população pequena possui pouca cobertura do espaço de busca do problema. Já em uma população grande possui uma vasta cobertura do espaço de busca (REZENDE, 2003, p. 238).
- **taxa de cruzamento:** "quanto maior for essa taxa, mais rapidamente novas estruturas serão introduzidas na população. Mas se for multo alta, estruturas com boas aptidões poderão ser retiradas mais rapidamente que a capacidade de seleção em criar melhores estruturas. Se a taxa for muito alta, a busca pode estagnar". (REZENDE, 2003, p. 238).
- **taxa de geração:** "controla a porcentagem da população que será substituída para a próxima geração. Com um intervalo grande, a maior parte da população será substituída e isso pode levar à perda de estruturas de alta aptidão. Com um intervalo pequeno, o algoritmo pode ser tornar muito lento". (REZENDE, 2003, p. 238).
- **critério de parada:** diferentes critérios podem ser utilizados para terminar a execução de um AGs (REZENDE, 2003, p. 238).

Após compreender a definição, operadores genéticos e parâmetros genéticos de um AGs, é possível trabalhar com os frameworks *AFORGE* e *JGAP*, que serão trabalhados nas próximas seções.

3 MATERIAIS E MÉTODOS

A metodologia de desenvolvimento deste trabalho inicia-se com a realização de estudo sobre AGs. Posteriormente, um estudo mais aprofundado dos dois *frameworks*: o *AFORGE* e *JGAP*, buscando um maior aprendizado que envolve a instalar, preparar o ambiente de desenvolvimento, utilizar, buscar os operadores genéticos utilizados pelos *frameworks*, e em seguida, compilar.

Após o estudo dos *frameworks* citados, foi escolhido o PCV, um problema clássico de IA para ser implementado. O PCV consiste em um problema de menor caminho, começando de uma cidade e indo para outra cidade, visitando-a uma vez e voltando à cidade inicial para novamente procurar outro caminho até encontrar o menor caminho entre a cidade inicial e final. Após a implementação, os dois códigos são executados. Nesse processo, são coletadas algumas métricas:

- desempenho, que foi elaborado um algoritmo que calcula a velocidade de execução para resolver o PCV, utilizando para o *framework AFORGE*, o método de inicialização randômica uniforme, de seleção a classificação, de cruzamento o multiponto e mutação por troca. Para o *JGAP* utilizando o método de inicialização randômica uniforme, de seleção o limiar e de mutação por troca;
- instalação, que descrever o grau de dificuldade quanto a instalação dos *frameworks* começando da aquisição dos arquivos até sua instalação no IDE necessário;
- quantidade de documentação disponível foi pesquisada as documentações e tutoriais disponíveis na *web* para cada *framework*.

- quantidade de suporte, que foi contabilizado a quantidade de listas de discussões disponíveis na *web* para cada *framework*.
- qualidade dos resultados, foi definido 16 cidades para que os *frameworks* pudessem encontrar os menores caminhos.

A partir das métricas coletadas na etapa anterior, é construída uma tabela e gráficos comparativos baseando-se nos resultados teóricos e práticos, sobre os quais é realizada uma análise que sirva de referencial para trabalhos futuros.

3.1. Configurações

Para o desenvolvimento dos resultados com os *frameworks JGAP* e *AFORGE*, se fez o uso de *hardwares* e *Software* com as seguintes configurações:

- Processador AMD *Athlon II Dual-Core* M320 2.10 *GHz*;
- Memória *RAM* 4,00 *GB*;
- Sistema Operacional *Windows* 7 *Home Premium* 64 *Bits*;
- *Microsoft Visual Studio* 2010 *ultimate*;
- *NetBeans* 7.0.1.

As duas próximas seções descrevem as ferramentas de implementação para os *frameworks JGAP* e AGORGE.

3.2. Visual Studio

O *Visual Studio* é um pacote de ferramentas/programas de desenvolvimento nas linguagens *Visual Basic* (*VB*), *C*, *C++*, *C#* (*C Sharp*), *J#*. Disponíveis para aplicações na área *WEB*, aplicações *Desktop* e aplicações móveis.

O *Visual Studio* 2010 foi lançado em abril de 2010 pela empresa *Microsoft* com o código *.NET Framework 4.0*. O Visual Studio providencia um ambiente de desenvolvimento integrado (IDE), que ajuda os programadores a construir de uma forma rápida soluções utilizando características chave de produtividade acessíveis por qualquer linguagem .NET. O IDE é o ambiente totalmente customizado que permite um alto desempenho para os programadores.

Portanto, esta ferramenta permite um ambiente de desenvolvimento integrado que providencia uma interface consistente para todas as linguagens, incluindo c#. Utiliza uma arquitectura de alta escala e permite que o desenvolvimento da aplicação seja feita em qualquer linguagem.

3.3. NetBeans

O *NetBeans* é um produto *open* source que contém um conjunto de bibliotecas, módulos e APIs formando um ambiente integrado de desenvolvimento visual possibilitando ao desenvolvedor compilar, debugar, efetuar *deploying* de suas aplicações. Alguns recursos são: debugador e compilador de programas; auto completar avançado, depurador de erros; *Syntax highlighting* à XML, HTML, CSS, JSP, IDL; suporta linguagens Java, C, C++; Recursos para desenvolvimento EJBs, *Web Services*; Suporte a *Database*, *Data view*, *Connection wizzard*; etc (GUJ, [s.a], p. 2).

3.4. AFORGE

AFORGE é um *framework* na linguagem *C#* projetada para desenvolver aplicações de visões computacionais e inteligência artificial. Processamento de imagens, redes neurais, Algoritmos Genéticos, lógica fuzzy, aprendizado de máquina, a robótica, etc. O desenvolvedor e criador do *AFORGE* é *Andrew Kirillov*, do Reino Unido, iniciou seu projeto em 2008 e mantém suporte até os dias de hoje.

O *AFORGE* oferece classes e interfaces para representar:

- Genes (*IGPGene*);
- Cromossomos (*ChromosomeBase*);
- População (*Population*);
- Métodos de seleção (*EliteSelection*, *RankSelection* e *RouletteWheelSelection*);
- Função de *Fitness* (*IFitnessFunction*);

O *AFORGE* é um *framework* livre e pode ser distribuído sob a licença *GNU Lesses Public License* 2.0.0. As aplicações comerciais que não publicam seu código-fonte sejam distribuídas (*AFORGE*, *online*, 2011).

Para poder instalar e configurar o *framework AFORGE* necessita ter instalado o ambiente de desenvolvimento *Microsoft Visual Studio*.

3.4.1. Instalação

Para iniciar o processo de manuseio do *framework AFORGE* na ferramenta *Visual Studio* é precisa-se primeiramente ter acesso ao site do *AFORGE*, aonde é possível visualizar os *links* necessários para o *framework* funcionar, assim como, exemplos de aplicações, ferramentas, documentação, licença de uso e *download* que é o foco da seção. Após a realização o *download*, necessita-se descompactar os arquivos.

Posteriormente, cria-se um novo projeto utilizando o *Visual Studio*. Em seguida, procura-se o arquivo *Build All.sln* dentro da pasta *Source* nos arquivos que foram descompactados.

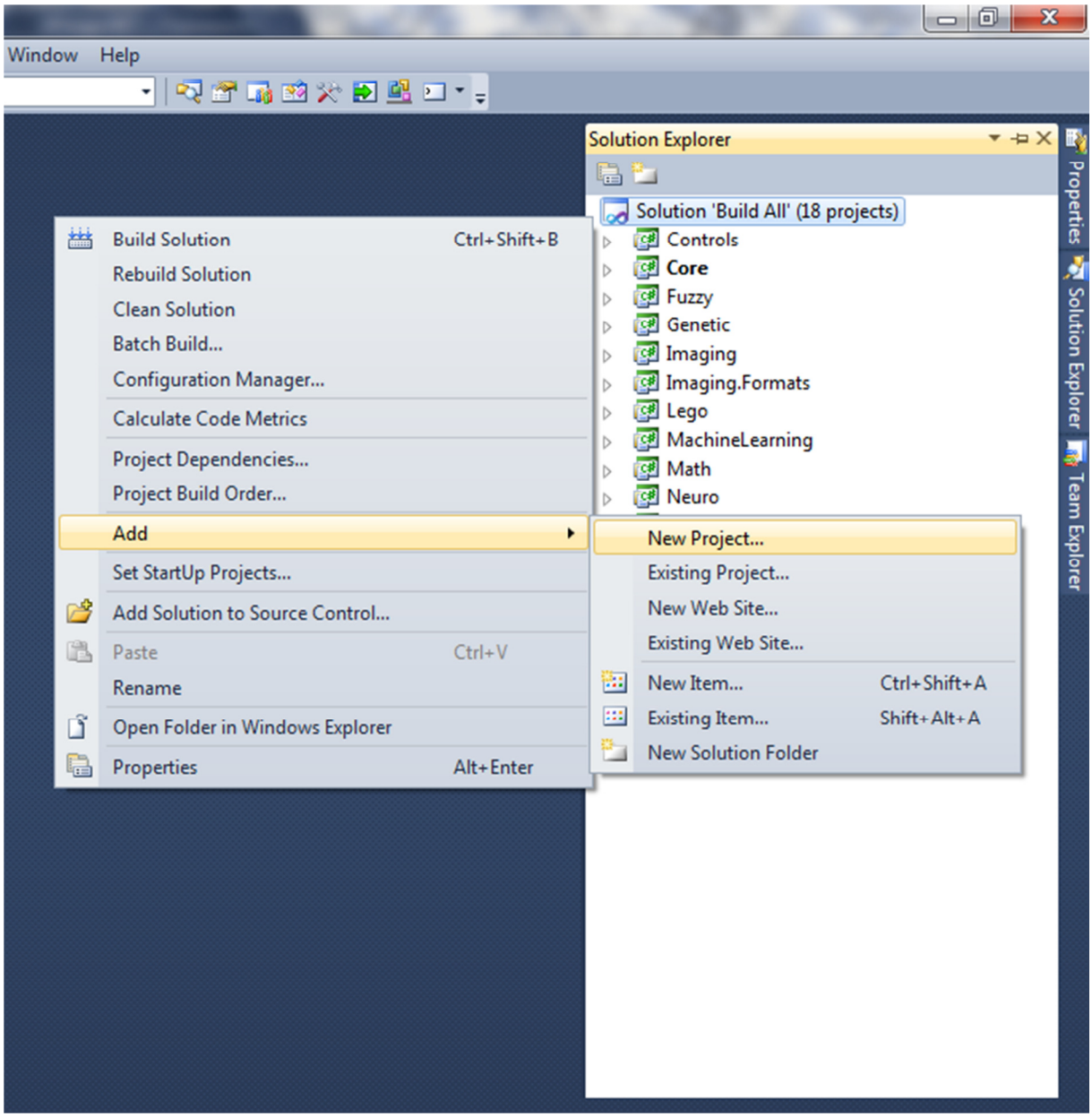

Figura 23 - Adicionar novo projeto para o *Traveling Salesman*

Os projetos serão exibidos na aba *Solution Explorer* do *Visual Studio*. Posteriormente cria-se um novo projeto para a resolução do PCV. Como pode ser

visto na Figura 23, precisa-se expandir o menu com o botão direito do *mouse*, em seguida, *Add* e *New Project*.

Em seguida, adiciona-se a referência dos Algoritmos Genéticos para que o PCV funcione. Na aba *Solution Explorer*, expandir o projeto *Salesman* clicando sobre o botão direito do *mouse* sobre *References*, depois, *Add Reference*. Deve-se clicar sobre a aba *Projects,* selecione *Genetic* na lista seguida do botão OK finalizando, assim, o processo de instalação do *framework AFORGE*.

3.5. JGAP

JGAP é um componente de programação de Algoritmos Genéticos que se utiliza como um *framework*, cuja sua sigla significa *Java Genetic Algorithms Package* (pacote de algoritmos genéticos para *Java*). Fundado por *Neil Rotstan*, o *JGAP* teve inicio de sua implementação em 2005. *Klaus Meffert* é responsável pela equipe que desenvolveu o *framework*, composta por *Javier Meseguer*, *Enrique D.Martí*, *Auddrius Meskauskas* e *Jerry Vos*.

O *JGAP* oferece classes e interfaces para representar:

- Genes (*Gene*);
- Cromossomos (*Chromosome*);
- Indivíduos (*IChromosome*);
- A população (*Genotype*);
- A função de *fitness* (*FitnessFunction*);
- Operadores Genéticos.

O *JGAP* é um *framework* livre, e pode ser distribuído sob a licença *GNU Lesses Public License* 2.1 ou posterior, que as aplicações comerciais que não

publicam seu código-fonte e seja distribuído sob a *Mozilla Public Licence* (*JGAP*, *online*, 2011).

Para poder instalar e configurar o *framework JGAP*, necessita ter instalado o *JDK Java* e o ambiente de desenvolvimento *NetBeans*

3.5.1. Instalação

O inicio da implementação do PCV é feito pela instalação do *framework JGAP* na ferramenta *NetBeans*. Ao acessar o site do *JGAP*, pode-se ter acesso à documentação, referência, *download*, andamento da próxima versão, contribuição, detalhes do projeto e *links* interessantes. Faz-se necessário o *download* do *framework*. Depois que o *download* é efetuado, abre-se o arquivo para ser descompactado em alguma pasta do sistema. Posteriormente cria-se um novo projeto para trabalhar com o *framework JGAP*. E executa-se o *NetBeans* e criar um novo projeto.

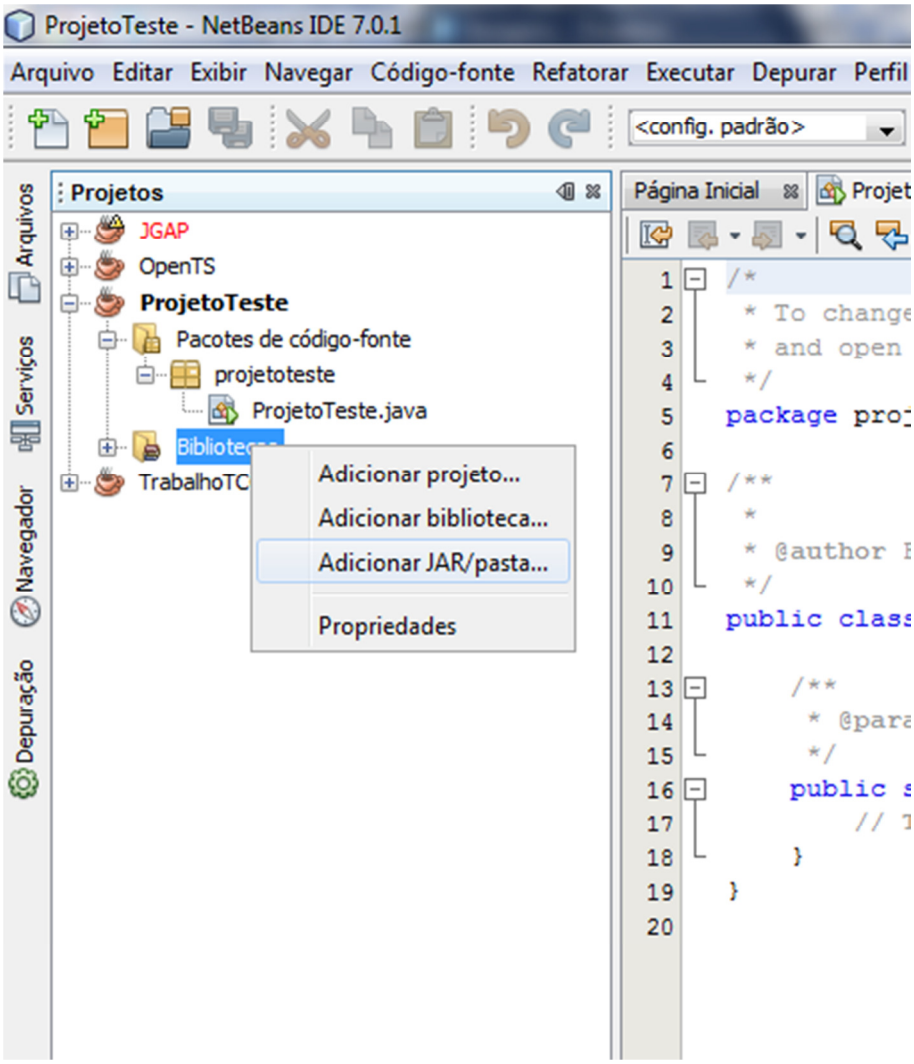

Figura 24 - Adicionar arquivos no *JGAP* ao projeto.

Na aba Projetos, pode-se visualizar os projetos existentes no *NetBeans*. Ao expandir o projeto recém-criado, clica-se sobre o botão direito do *mouse*, e depois, Adicionar JAR/pasta. Essa ação é importante para importações dos arquivos JAR do *framework JGAP*, necessário para o funcionamento do projeto. A Figura 24 ilustra essa etapa. Pode-se observar que uma janela é aberta, então, procura-se a pasta raiz dos arquivos do *framework* que foram descompactados. O arquivo '`jgap-exemples.jar`' carrega alguns exemplos de aplicações utilizando o *framework*. O `jgap-test.jar` são arquivos de teste dos desenvolvedores do *framework,* e por

fim, o `jgap.jar` que leva todos os arquivos necessários para o funcionamento do *framework JGAP*. Alguns arquivos de terceiros são necessários para que o *framework* funcione corretamente, sendo assim, necessita-se abrir a pasta `lib` que esta dentro da pasta raiz. E por fim, seleciona-se todos os arquivos com extensão .JAR.

4 RESULTADOS E DISCUSSÃO

Após a descrição do AGs e do problema a ser solucionado, no caso, o PCV, faz-se necessário compreender como é o seu funcionamento dos *frameworks AFORGE* e *JGAP*, para posteriormente avaliar os critérios de comparação definidos nas subseções seguintes.

4.1. Caixeiro Viajante

Para trabalhar o PCV com AGs, é necessário primeiramente modelá-lo de forma que ele possa ser trabalhado por esta técnica. Assim, para entender como as cidades serão modeladas de forma que possam ser utilizadas nos *frameworks*, algumas figuras são exibidas a seguir.

Cidade	X	Y
0	2	4
1	7	5
2	7	11
3	8	1
4	1	6
5	5	9
6	0	11

Figura 25 – Coordenadas das cidades.

A Figura 25 mostra, como exemplo, sete cidades nomeadas de 0 a 6 com seus respectivos valores de X e Y, o que define sua coordenada em um mapa.

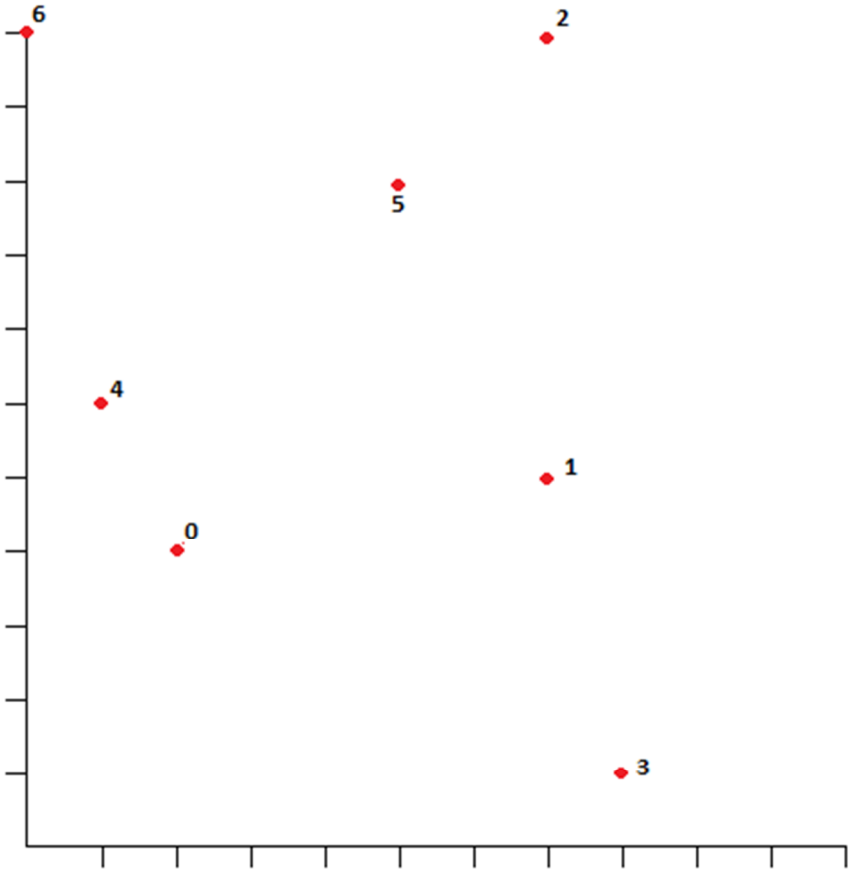

Figura 26 - Gráfico das coordenadas das cidades.

Com as coordenadas declaradas na Figura 25, observa-se na Figura 26 que é possível mostrar como tais coordenadas se comportam em um gráfico com eixo horizontal (coordenada X) e eixo vertical (coordenada Y).

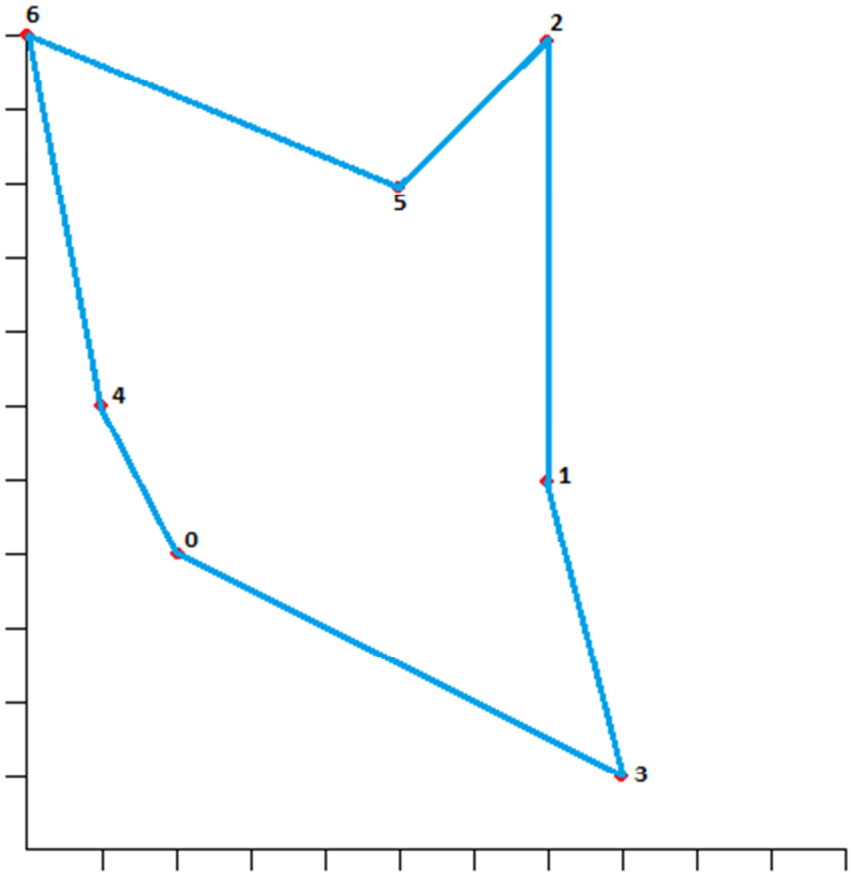

Figura 27 - Menor caminho entre as 7 cidades.

Utilizando os dados da Figura 25 e o exemplo da Figura 26, observa-se na Figura 27 o menor caminho entre as sete cidades foi traçado. Com esse gráfico é possível testar se o caminho definido por cada *framework* resulta o mesmo caminho do gráfico. Com as coordenadas das cidades agora é possível efetuar a instalação dos *frameworks AFORGE* e *JGAP*.

4.2. Implementação do Caixeiro Viajante utilizando o AFORGE

```
1. public double[,] CityArray = new double[7, 2] {
2.      { 2.0, 4.0 },
3.      { 7.0, 5.0 },
4.      { 7.0, 11.0 },
5.      { 8.0, 1.0 },
6.      { 1.0, 6.0 },
7.      { 5.0, 9.0 },
8.      { 0.0, 11.0 }
9. };
```

Figura 28 - Coordenadas das cidades no *AFORGE*.

A Figura 28 mostra que se deve declarar um vetor do tipo `double`, que define as coordenadas das cidades usadas para implementação do PCV no framework *AFORGE*.

Para calcular a distância entre duas cidades em um cromossomo faz-se necessário criar uma função específica, descrita a seguir.

```
1. public double Distance(int a_from, int a_to){
2.      int a = a_from;
3.      int b = a_to;
4.      double x1 = CityArray[a, 0];
5.      double y1 = CityArray[a, 1];
6.      double x2 = CityArray[b, 0];
7.      double y2 = CityArray[b, 1];
8.      double val = (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
9.      return Math.Sqrt(val);
10.    }
```

Figura 29 - Método que calcula distância entre duas cidades no *AFORGE*.

A Figura 29 mostra o método que calcula a distância entre duas cidades. Deve-se utilizar como parâmetro duas variáveis do tipo `int`, que correspondem aos valores de dois genes de um cromossomo. Em seguida, a descrição do código:

- Na linha 2, cria-se uma variável do tipo `int`, nomeada *a*, que recebe a variável `a_from`;
- Na linha 3, cria-se uma variável do tipo `int`, nomeada `b`, que recebe a variável `a_to`;
- Na linha 4, cria-se uma variável do tipo `double`, nomeada `x1` que recebe a coordenada do eixo X da primeira cidade;
- Na linha 5, cria-se uma variável do tipo `double`, nomeada `y1`, que recebe a coordenada do eixo Y da primeira cidade;
- Na linha 6, cria-se uma variável do tipo `double`, nomeada `x2`, que recebe a coordenada do eixo X da segunda cidade;
- Na linha 7, cria-se uma variável do tipo `double`, nomeada `y2`, que recebe a coordenada do eixo Y da segunda cidade;
- Na linha 8, cria-se uma variável do tipo `double`, nomeada `val`, que recebe os cálculos das variáveis `x1, y1, x2, y2`;
- Na linha 9, a função retorna a raiz quadrada do valor da variável `val`.

Com o método `Distance` criado, é possível avaliar o cromossomo, nesse caso, cria-se também um método para realizar tal avaliação.

```
1. public double Evaluate(IChromosome chromosome) {
2.     double s = 0;
3.     int[] genes = ((PermutationChromosome)chromosome).Value;
4.     for (int i = 0; i < genes.Count() - 1; i++){
5.         s += Distance(genes[i], genes[i + 1]);
6.     }
7.     s += Distance(genes[genes.Count() - 1], genes[0]);
8.     return int.MaxValue / 2 - s;
9. }
```

Figura 30 - Método que avalia o cromossomo calculando valor de fitness no *AFORGE*.

Com o método `Distance` ilustrado na Figura 29, é possível saber o valor de aptidão para cada dois genes do cromossomo. Cria-se um método para definir o valor de aptidão para o cromossomo, neste caso chamado de `Evaluate`, conforme descrito na Figura 30. Este método necessita de um cromossomo para obtenção dos genes Em seguida, seus genes são guardados temporariamente em um vetor que posteriormente será percorrido, calculando o valor de aptidão para cada par de genes (utilizando o método `Distance`), e acumulando seu total em uma variável. Após percorrer todos os genes, calcula-se a aptidão do último e do primeiro gene do cromossomo, a fim de se descobrir o custo (aptidão) de voltar para cidade inicial. A seguir, será descrito o código para criação do método `Evaluate` utilizando a linguagem C#:

- Na linha 1, cria-se o método passando como parâmetro um objeto do tipo `Chromosome`;
- Na linha 2, cria-se uma variável do tipo `double`, nomeada `s`, recebendo o valor zero;

- Na linha 3, cria-se um vetor do tipo `int`, nomeado *genes*, que recebe os valores dos genes do cromossomo;
- Nas linhas 4 a 6, cria-se um laço de repetição do tamanho do total da variável `genes`;
- Na linha 5, a variável `s` acumula o valor de aptidão para cada dois genes;
- Na linha 7, a variável `s` acumula o valor de aptidão do ultimo e primeiro gene;
- Na linha 8, é retornado pelo método o valor de aptidão do cromossomo, sendo o máximo valor `int` dividido por dois, e subtraído com o valor acumulado da variável `s`;

```
1. public void GA(){
2.      SalesmanFitnessFunction fitnessFunction = new
   SalesmanFitnessFunction(CityArray);
3.      Population population = null;
4.          population = new Population(getPopulationSize(), new
   PermutationChromosome(getCitiesSize()), fitnessFunction, new
   EliteSelection());
5.      IChromosome best = null;
6.      for (int i = 0; i < getMaxEvolution(); i++){
7.          population.RunEpoch();
8.          best = population.BestChromosome;
9.      }
10.   }
```

Figura 31 - Método solução Caixeiro Viajante no *AFORGE*.

O método da Figura 31 tem o papel de gerar a solução para o PCV utilizando o *framework AFORGE*. Primeiramente cria-se uma instância da função de *fitness* e

uma nova população. Em seguida, a população é preenchida utilizando um método que necessita o tamanho da população, quantidade de cidades, a função de fitness e o método de seleção (três métodos disponíveis). Após, cria-se um novo cromossomo e feito o processo de evolução para encontrar o melhor cromossomo. A seguir o código é descrito:

- Na linha 2, cria-se uma variável do tipo `SalesmanFitnessFunction` que é a classe que contém os métodos para calcular o valor de *fitness* ou aptidão.
- Na linha 3, cria-se uma variável do tipo `Population`, nomeada `population`, que recebe um valor nulo;
- Na linha 4, a variável `population` recebe uma nova população utilizando o método construtor da classe `Population`, que é passado como parâmetro a quantidade de indivíduos da população e o método `PermutationChromosome`, que se passa parâmetro a quantidade de cidades, a função de *fitness* e o método de seleção (três métodos de seleção disponíveis: `EliteSelection`, `RankSelection`, `RouletteWhellSelection`);
- Na linha 5, cria-se uma nova variável do tipo `IChromosome`, nomeada como `best`, responsável por guardar o melhor cromossomo;
- Nas linhas entre 6 e 9, cria-se um laço de repetição com tamanho do máximo de evoluções definido pelo usuário;
- Na linha 7, é executado o método do *framework* responsável pela evolução;
- Na linha 8, a variável `best` recebe o melhor cromossomo pelo método do *framework*.

A seguir, na próxima seção, a implementação do PCV utilizando o *framework JGAP*.

4.3. Implementação do Caixeiro Viajante utilizando o JGAP

Um problema clássico foi escolhido para que o *framework JGAP* ache a solução. A seguir serão exibidos os códigos que fizeram que o PCV fosse resolvido.

```
1. public static final int[][] CITYARRAY = new int[][] {
2.      { 2, 4 },
3.      { 7, 5 },
4.      { 7, 11 },
5.      { 8, 1 },
6.      { 1, 6 },
7.      { 5, 9 },
8.      { 0, 11 }
9. };
```

Figura 32 - Coordenadas das cidades no *JGAP.*

A Figura 32 exibe que primeiramente, precisa-se saber a localidade de cada cidade. As coordenadas de cada cidade são declaradas em uma matriz de `int`. O código é descrito a seguir, nas linhas entre 2 e 8, instancia-se a matriz com valores em `z`. Valores estes que definem as coordenadas de sete cidades.

Após declarar as coordenadas das cidades, faz-se necessário criar um método que calcule o valor de aptidão.

```
1. public double Distance(Gene a_from, Gene a_to) {
2.     IntegerGene geneA = (IntegerGene) a_from;
3.     IntegerGene geneB = (IntegerGene) a_to;
4.     int a = geneA.intValue();
5.     int b = geneB.intValue();
6.     int x1 = CITYARRAY[a][0];
7.     int y1 = CITYARRAY[a][1];
8.     int x2 = CITYARRAY[b][0];
9.     int y2 = CITYARRAY[b][1];
10.       double val = (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
11.       return Math.sqrt(val);
12.   }
```

Figura 33 - Método que calcula distância entre duas cidades no *JGAP.*

Vários cromossomos são gerados aleatoriamente pelo *framework JGAP* e posteriormente é preciso fazer o processo de *fitness* ou avaliação que define o valor de aptidão para cada dois genes do cromossomo para posteriormente outro método faça o valor de *fitness* para o cromossomo. A Figura 33 mostra o método que soma o valor da aptidão de cada dois genes, que se faz necessária criar duas variáveis `int` que receba o valor do gene, e em seguida, cria-se duas variáveis `int` para cada gene, a fim de guardar os valores da coordenada do gene correspondente a cidade. E por fim, deve-se retornar a raiz quadrada do valor calculado das variáveis. Para melhor entendimento, segue a descrição para criação do método *Distance*:

- Na linha 1, cria-se o método que tem como parâmetro duas variáveis do tipo `gene` nomeadas como `a_from` e `a_to`;
- Na linha 2, cria-se uma variável do tipo `IntergerGene`, nomeada `geneA`, que recebe a variável `a_from` convertida para `IntergerGene`;

- Na linha 3, cria-se uma variável do tipo `IntergerGene`, nomeada `geneB`, que recebe a variável `a_to` convertida para `IntergerGene`;
- Na linha 4, cria-se uma variável do tipo `int`, nomeada `a`, que recebe o valor em *int* da variável `geneA`*;*
- Na linha 5, cria-se uma variável do tipo `int`, nomeada `b`, que recebe o valor em `int` da variável `geneB`;
- Na linha 6, cria-se uma variável do tipo `int`, nomeada `x1`, que recebe o valor da matriz `CITYARRAY`*,* que guarda as coordenadas da cidade correspondente ao valor da variável *a* e a posição zero;
- Na linha 7, cria-se uma variável do tipo `int`, nomeada `y1`, que recebe o valor da matriz `CITYARRAY`, que guarda as coordenadas da cidade correspondente ao valor da variável *a* e a posição um;
- Na linha 8, cria-se uma variável do tipo `int`, nomeada `x2`, que recebe o valor da matriz `CITYARRAY`, que guarda as coordenadas da cidade correspondente ao valor da variável `b` e a posição zero;
- Na linha 9, cria-se uma variável do tipo `int`, nomeada `y2`, que recebe o valor da matriz `CITYARRAY`, que guarda as coordenadas da cidade correspondente ao valor da variável `b` e a posição um;
- Na linha 10, cria-se uma variável do tipo `double`, nomeada `val`, que recebe o calculo das variáveis `x1`, `y1`, `x2` e `y2`;
- Na linha 11, o método retorna a raiz quadrada da variável `val`.

Com o método *Distance,* faz-se necessário criar o método que calcule o valor de aptidão para o cromossomo;

```
1. protected double Evaluate(final IChromosome chromosome) {
2.     double s = 0;
3.     Gene[] genes = chromosome.getGenes();
4.     for (int i = 0; i < genes.length - 1; i++) {
5.         s += Distance(genes[i], genes[i + 1]);
6.     }
7.     s += Distance(genes[genes.length - 1], genes[0]);
8.     return Integer.MAX_VALUE / 2 - s;
9. }
```

Figura 34 - Método que avalia o cromossomo calculando valor de fitness no *JGAP*.

A Figura 34 demonstra o método que define o valor de aptidão para o cromossomo. Primeiramente necessita-se de um cromossomo, em seguida, se obtém os valores dos genes para posteriormente calcular o valor de aptidão para cada par de genes. Logo após, calcula-se o valor da ultima e primeira cidade, a fim de se descobrir o custo (aptidão) de voltar para cidade inicial. A seguir, será descrito o código para criação do método `Evaluate` utilizando a linguagem *Java:*

- Na linha 1, faz-se necessário obter uma variável do tipo `IChromosome` nomeado `chromosome`, ou seja, um cromossomo para calcular o valor de *fitness*;
- Na linha 2, cria-se uma variável do tipo `s` que recebe o valor zero;
- Na linha 3, cria-se um vetor do tipo `Gene` que recebe os genes do cromossomo atual;
- Nas linhas entre 4 e 6, cria-se um laço de repetição que tem como tamanho a quantidade de genes do cromossomo;

- Na linha 5, a variável `s` acumula seu valor juntamente com o calculo da distância de cada dois *genes*;
- Na linha 7, a variável `s` acumula seu valor juntamente com o calculo do ultimo e primeiro *gene*;
- Na linha 8, o método retorna o limite da variável `interger` convertido em `double`, dividido por 2 e subtraindo o valor de aptidão do cromossomo.

Em seguida, faz-se necessário criar um método que cria as configurações necessárias para o funcionamento do *framework JGAP*;

```
1. public Configuration createConfiguration(Object a_initial_data){
2.     Configuration config = new Configuration();
3.     ThresholdSelector thres = new ThresholdSelector(config, 0.1);
4.     config.addNaturalSelector(thres, true);
5.     config.setRandomGenerator(new StockRandomGenerator());
6.     config.setMinimumPopSizePercent(0);
7.     config.setEventManager(new EventManager());
8.     config.setFitnessEvaluator(new DefaultFitnessEvaluator());
9.     config.addGeneticOperator(new GreedyCrossover(config));
10.        config.addGeneticOperator(new SwappingMutationOperator(config,
   20));
11.        return config;
12.   }
```

Figura 35 - Criar configuração do *JGAP.*

O *framework JGAP* necessita configurar uma variável do tipo `Configuration`, essa variável irá carregar as configurações necessárias para o funcionamento do *framework*. Com a variável do tipo `Configuration` criada, faz-se necessário informar o método de seleção, método de inicialização, método de *fitness*, operador de cruzamento e mutação. A Figura 35 exibe como essa configuração é feita:

- Na linha 1, cria-se um método que tem como parâmetro uma variável do tipo `Object`, nomeada `a_initial_data`, necessária para sincronizar com outros métodos da classe;
- Na linha 2, cria-se uma variável do tipo `Configuration`, nomeada `config`;
- Na linha 3, cria-se uma variável do tipo `Thereshold Selector`, nomeada `thres`, instanciada pelo construtor da classe passando como parâmetro a variável `config`. E a probabilidade de se escolher o melhor indivíduo (de 0.0 à 1.0);
- Na linha 4, define-se o método de seleção, passando como parâmetro a variável `thres`;
- Na linha 5, define-se o operador de inicialização;
- Na linha 6, define-se a porcentagem mínima do tamanho da população;
- Na linha 7, define-se o `event mananger`, necessário para o funcionamento do *framework*;
- Na linha 8, define-se a função padrão de *fitness* (função essa que será alterada posteriormente);
- Na linha 9, define-se o operador de cruzamento, passando como parâmetro a variável `config`;
- Na linha 10, define-se o operador de mutação, passando como parâmetro a variável `config`;
- Na linha 11, o método retorna a variável `config`.

A seguir, com o *framework JGAP* faz-se necessário criar o cromossomo, para isso, cria-se o método `createchromosome`.

```
1. public IChromosome createChromosome(Object a_initial_data) {
2.         Gene[] genes = new Gene[CITIES];
3.         for (int i = 0; i < genes.length; i++) {
4.             genes[i] = new IntegerGene(getConfiguration(), 0, CITIES -
   1);
5.             genes[i].setAllele(new Integer(i));
6.         }
7.         IChromosome sample = new Chromosome(getConfiguration(), genes);
8.         return sample;
9. }
```

Figura 36 - Criar população no *JGAP.*

O método `createChromosome` descrito na figura 36, faz-se necessário sua criação para se obter um cromossomo. Primeiramente, cria-se um vetor de `genes` do tamanho da quantidade de cidades existentes, posteriormente cada posição do vetor é preenchida, e por fim, retornado o cromossomo. A seguir é descrito o código:

- Na linha 1, cria-se um método que tem como parâmetro uma variável do tipo `Object` nomeada `a_initial_data`, responsável por sincronizar com outros métodos;
- Na linha 2, cria-se um vetor de `Gene`, nomeado `genes`, que recebe a quantidade de cidades;
- Nas linhas entre 3 e 6, cria-se um laço de repetição com o tamanho total de genes do vetor `genes`;
- Na linha 4, o vetor `genes` na posição atual recebe uma variável do tipo `IntergerGene` pela construtor da classe, que passa como parâmetro o arquivo de configuração (criado na Figura 35), a cidade inicial, e a cidade final;
- Na linha 5, o vetor `genes` na posição atual define o alelo;

- Na linha 7, cria-se uma variável do tipo `IChromosome`, nomeada `samples`, que recebe o construtor da classe `Chromosome` que passa como parâmetro o arquivo de configuração e a variável `genes`;
- Na linha 8, a classe retorna a variável `sample`.

Posteriormente, depois do método de criação do cromossomo pronto, faz-se necessário evoluir a população a fim de se obter o melhor cromossomo da população.

```
1. private Configuration m_config;
2. public IChromosome findOptimalPath(final Object a_initial_data) {
3.     m_config = createConfiguration(a_initial_data);
4.     FitnessFunction myFunc = createFitnessFunction(a_initial_data);
5.     m_config.setFitnessFunction(myFunc);
6.     IChromosome sampleChromosome = createChromosome(a_initial_data);
7.     m_config.setSampleChromosome(sampleChromosome);
8.     m_config.setPopulationSize(getPopulationSize());
9.     IChromosome[] chromosomes = new
   IChromosome[m_config.getPopulationSize()];
10.        Gene[] samplegenes = sampleChromosome.getGenes();
11.        for (int i = 0; i < chromosomes.length; i++) {
12.            Gene[] genes = new Gene[samplegenes.length];
13.            for (int k = 0; k < genes.length; k++) {
14.                genes[k] = samplegenes[k].newGene();
15.                genes[k].setAllele(samplegenes[k].getAllele());
16.            }
17.            chromosomes[i] = new Chromosome(m_config, genes);
18.        }
19.        Genotype population = new Genotype(m_config, new
   Population(m_config, chromosomes));
20.        IChromosome best = null;
21.        for (int i = 0; i < getMaxEvolution(); i++) {
22.            population.evolve();
23.            best = population.getFittestChromosome();
24.        }
25.        return best;
26.    }
```

Figura 37 - Procurar solução no *JGAP.*

A Figura 37 mostra o método que procura a solução para o PCV no *framework JGAP*, que se faz necessário informar o arquivo de configuração, a função de *fitness*, a amostra de cromossomo, o tamanho da população, para posteriormente criar uma população de cromossomos e após, evolui-la. A fim de se obter o melhor cromossomo, resolvendo assim o PCV. Em seguida, pode-se observar a descrição do código:

- Na linha 1, cria-se uma variável do tipo `Configuration`, nomeada `m_config`;
- Na linha 2, cria-se um método que passa como parâmetro uma variável do tipo `Object`, nomeada `a_initial_data`;
- Na linha 3, a variável `m_config` recebe a variável `a_initial_data`;
- Na linha 4, cria-se uma variável do tipo `FitnessFunction`, nomeada `myFunc`, que recebe a variável `a_initial_data`;
- Na linha 5, define-se a função de *fitness* passando a variável `myFunc`*;*
- Na linha 6, cria-se uma variável do tipo `IChromosome`, nomeada `sampleChromosome`;
- Na linha 7, define-o cromossomo amostra passando como parâmetro a variável `sampleChromosome`;
- Na linha 8, define-se o tamanho da população;
- Na linha 9, cria-se um vetor do tipo `IChromosome`, nomeado `chromosomes`, que recebe o construtor da classe que passa como parâmetro o tamanho da população;
- Na linha 10, cria-se um vetor do tipo *Gene*, nomeado `samplegenes`, que recebe o tamanho dos genes do cromossomo criado na variável `sampleChromosome`;
- Nas linhas entre 11 e 18, cria-se um laço de repetição do tamanho da quantidade cromossomos da variável `chromosomes`;
- Na linha 12 cria-se um vetor de *Gene*, nomeado *genes*, que recebe o tamanho da quantidade de genes da variável `samplegenes`;

- Nas linhas entre 13 e 16, cria-se um laço de repetição do tamanho da quantidade de genes da variável `genes`;
- Na linha 14, a variável `genes` na posição atual recebe um novo gene;
- Na linha 15, define-se o alelo da variável `genes`;
- *Na linha 17*, a variável `chromosomes` na posição atual recebe uma nova variável do tipo `Chromosome` pelo construtor da classe, passando como parâmetro a variável `m_config` e o vetor `genes`;
- Na linha 19, cria-se uma variável do tipo `Genotype`, nomeada `population`, que recebe um novo `genotype` pelo construtor da classe, que passa como parâmetro a variável `m_config` e uma variável `Population` pelo construtor da classe que passa como parâmetro a variável `m_config` e o vetor `chromosomes`
- Na linha 20, cria-se uma variável do tipo `IChromosome`, nomeada `best`, que recebe um valor nulo;
- Nas linhas entre 21 e 24, cria-se um laço de repetição que tem o tamanho do total de evoluções definido pelo usuário;
- Na linha 22, faz-se necessário evoluir a população, chamando o método `evolve`;
- Na linha 23, a variável `best` recebe o melhor cromossomo da evolução;
- Na linha 25, o método retorna o melhor cromossomo.

Faz-se necessário a comparação dos *frameworks* a fim de descobrir os pontos mais fortes de cada um, para auxiliar na escolha do *framework*. A próxima seção descreve tais comparações.

4.4. Comparação

Como proposto, faz-se necessário descrever os critérios de comparação, onde são descritos nas subseções seguintes.

4.4.1. Tempo de execução

O tempo de execução foi medido utilizando algoritmos feitos em cada *framework* para saber a velocidade de execução para resolução do PCV. Para obter estes valores definiu-se o número de evoluções igual a 512 e tamanho da população igual a 50. O código a seguir não é fornecido pelo *framework*.

O próximo código é executado utilizando o *framework AFORGE*, obtendo assim, o tempo de execução e guardar os dados resultantes em um arquivo de texto, tendo como método de seleção a classificação.

```
1. using System.IO;
2. using System.Diagnostics;
3. private static FileStream fs = new
   FileStream(@"C:\Users\Usuario\Desktop\TempoExecucaoAForge.txt",
   FileMode.OpenOrCreate, FileAccess.Write);
4. private static StreamWriter saida = new StreamWriter(fs);
5. Stopwatch sw = new Stopwatch();
6. sw.Start();
7.     ...EXECUÇÃO DO ALGORITMO DE RESOLUÇÃO DO PROBLEMA DO PCV...
8. Sw.Stop();
9. saida.WriteLine("{0} {1}", DateTime.Now.ToLongTimeString(),
   DateTime.Now.ToLongDateString() + " -- Tempo de Execução -> " +
   sw.ElapsedMilliseconds.ToString() + " Milissegundos \n");
10.   saida.WriteLine();
11.   saida.Flush();
```

Figura 38 – Algoritmo de tempo de execução para o *AFORGE*.

A Figura 38 descreve o algoritmo do tempo de execução para o *framework AFORGE*. Em seguida faz-se necessário descrever o código:

- Na linha 3, é instanciada uma variável do tipo `FileStream`, recebendo uma nova instância do tipo `FileStream` (responsável por criar um novo arquivo de texto do tipo *.txt*), passando como parâmetro o tipo de acesso do arquivo e dando permissão de escrita sobre ele;
- Na linha 4 é criada uma variável do tipo `StreamWriter`, responsável por preencher o arquivo *.txt;*
- Na linha 5 a variável do tipo `Stopwath` é criada a fim de controlar o tempo de execução do algoritmo de resolução do PCV;
- Na linha 6, é iniciada a contagem de tempo;
- Na linha 7 o algoritmo responsável por solucionar o PCV é executado;
- Na linha 8 é finalizada a contagem de tempo;

- Na linha 9 é escrita a primeira linha no arquivo de texto, que guarda a data e hora atual;
- Na linha 10 é escrita uma linha em branco;
- A linha 10 finaliza, limpa todos os *buffers* da variável `StreamWriter`.

O código a seguir é executado utilizando o *framework JGAP*, que contabiliza o tempo de execução e gravar os dados resultantes em um arquivo de texto, tendo como método de seleção a classificação e de inicialização randomicamente uniforme.

```
1. import java.io.*;
2. import java.util.Date;
3. long tempoInicial = System.currentTimeMillis();
4.     ...EXECUÇÃO DO ALGORITMO DE RESOLUÇÃO DO PCV...
5. long tempoFinal = System.currentTimeMillis();
6. Date date = new Date();
7. BufferedWriter saida = new BufferedWriter(new
   FileWriter("C:/Users/Fox/Desktop/TempoExecucaoJGAP.txt", true));
8. saida.newLine();
9. saida.write(date.toGMTString());
10.   saida.write(" -- Tempo de Execução -> " + (tempoFinal - tempoInicial)
   + " Milissegundos");
11.   saida.newLine();
12.   saida.close();
```

Figura 39 - Algoritmo de tempo de execução para o *JGAP*.

Faz-se necessário criar um método que calcule o desempenho do *framework JGAP*. O código da Figura 39 é descrito a seguir:

- Nas linhas 1 e 2, deve-se importar as bibliotecas `java.io` e `java.util.Date`;

- Na linha 3, cria-se uma variável `long` que recebe a hora atual em milissegundo;
- Na linha 4, é executado o algoritmo que soluciona o PCV;
- Na linha 5, cria-se uma variável do tipo `long` que recebe a hora atual em milissegundos;
- Na linha 6, cria-se uma variável do tipo `Date`;
- Na linha 7, cria-se uma variável `BufferedWriter` responsável por criar um novo arquivo de texto *.txt*;
- Na linha 8, cria-se uma nova linha no arquivo de texto;
- Na linha 9, cria-se uma linha no arquivo de texto, que preenche a data atual;
- Na linha 10, é continuado na mesma linha do arquivo de texto, adicionando o tempo de execução em milissegundos;
- Na linha 11, cria-se uma nova linha no arquivo de texto;
- Na linha 12, é fechado o fluxo da variável `BufferedWriter`.

Ao executar o algoritmo exibido na Figura 38, é dado como resultado a hora, data e tempo de execução da resolução do PCV utilizando o *framework AFORGE*. Como se pode observar na Figura 40, que cada linha corresponde a uma execução do algoritmo. A média obtida das cinco execuções é de 709,6 milissegundos.

```
08:31:45 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 658 Milissegundos
10:51:54 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 643 Milissegundos
12:34:34 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 666 Milissegundos
16:23:42 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 703 Milissegundos
18:34:51 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 878 Milissegundos
```

Figura 40 - Tempo de execução no *AFORGE.*

O algoritmo da Figura 39 é executado e obtém-se como resultado a hora, data e tempo de execução da resolução do PCV (Figura 41), mas utilizando o *framework JGAP*. A média obtida das cinco execuções é de 3143,4 milissegundos.

```
08:13:11 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 2956 Milissegundos
12:16:45 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 3401 Milissegundos
14:10:37 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 3856 Milissegundos
19:02:00 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 2982 Milissegundos
19:02:05 quarta-feira, 11 de abril de 2012 -- Tempo de Execução -> 2522 Milissegundos
```

Figura 41 - Tempo de execução no *JGAP.*

A seguir, a Figura 42 exibe um gráfico comparativo do desempenho dos dois *frameworks*.

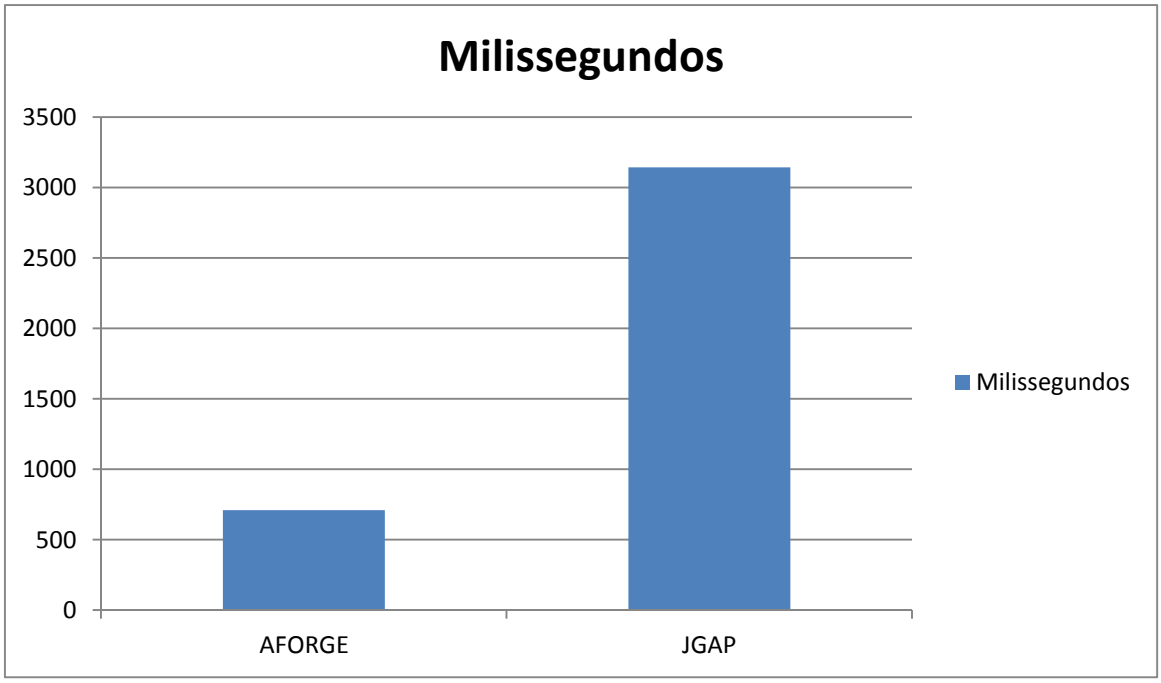

Figura 42 - Comparação de desempenho entre os *frameworks*.

Com a média de tempo de execução para resolução do PCV de cada *framework*, podendo agora compará-los. A Figura 40 mostra que o *AFORGE* teve um tempo de execução bem inferior ao *JGAP* (Figura 41).

4.4.2. Instalação

A seção 3.4.1 descreve a instalação do *framework AFORGE*, que se deve primeiramente efetuar o *download* do pacote de arquivos, em seguida, criar um novo projeto no *Visual Studio*, posteriormente importar o arquivo de configuração do projeto. Após isto, criar um novo projeto entre os projetos existentes, a fim de iniciar a implementação do PCV.

Já a instalação do *framework* JGAP, descrita na seção 3.5.1, tem sua instalação mais complexa, por necessitar de, primeiramente, se obter os arquivos necessários, em seguida, executar o *NetBeans* e criar um novo projeto, posteriormente abrir a aba bibliotecas e adicionar um novo arquivo *.JAR*, que se encontra as classes para o funcionamento do *framework*, escolher também bibliotecas de terceiros encontradas em outras pastas do arquivo do *framework*.

4.4.3. Métodos de seleção

Os *frameworks* possuem vários de métodos de seleção, que é possível a utilização de todos eles na implementação do PCV, eles são divididos da seguinte forma:

- No *AFORGE*:
    - ***EliteSelection***: equivale ao método de melhor cromossomo;
    - ***RanSelection***: equivale ao método por classificação;

- ***RouletteWhellSelection***: equivale ao método roleta.

- No *JGAP*:
  - ***BestChromosomesSelector***: equivale ao método melhor cromossomo;
  - ***ThresholdSelector***: equivale ao método limiar;
  - ***TournamentSelector***: equivale ao método torneio;
  - ***WightedRouletteSelector***: equivale ao método por classificação.

Os dois *frameworks* disponibilizam os métodos de seleção de melhor cromossomo e seleção por classificação. O *AFORGE* possui, além destes, apenas o método de classificação roleta, porém, como visto anteriormente, este método pode ocasionalmente levar a uma convergência prematura do AG, o que pode ser contornado utilizando-se o método por classificação que este *framework* disponibiliza. Já o *JGAP* oferece a possibilidade de utilizar mais dois métodos de seleção: torneio e limiar.

Em uma análise quantitativa, o *JGAP* se sobressai por possuir um número maior de métodos de seleção em relação ao *AFORGE*, o que na prática pode ser interpretado como uma vantagem, pois possibilita ao AG uma maior variedade de possibilidades de selecionar a próxima geração.

Partindo para uma análise dos métodos disponibilizados pelos dois *frameworks* além dos métodos que ambos possuem (melhor cromossomo e classificação), o *AFORGE* disponibiliza apenas o método de seleção roleta, que tem a conhecida característica de, em determinados cenários, contribuir para a convergência prematura do AG. Já o *JGAP* oferece o método torneio e limiar. O método torneio pode vir a ser interessante, visto que escolhe o cromossomo mais apto em amostragens menores escolhidas aleatoriamente, impedindo assim que

o AG convirja de forma prematura. O método limiar necessita o conhecimento prévio do problema, e deve ser utilizado conscientemente a fim de evitar a definição errônea dos limiares superior e inferior, o que acabaria por gerar uma população com características insuficientes para a solução do problema; este método aliado ao conhecimento prévio do problema torna-se interessante em determinados cenários.

Tais características apresentam o *JGAP* como tendo os métodos mais apropriados para o PCV proposto, pois além de permitir a seleção de indivíduos baseando-se no seu valor de aptidão e em fatores probabilísticos, oferece uma alternativa que une esses dois fatores.

4.4.4. Métodos de inicialização

O *framework AFORGE* não possui opções de métodos de inicialização, ficando apenas com a inicialização randômica uniforme, que define um valor aleatoriamente para os genes do cromossomo.

O *framework JGAP* possui opções de métodos de inicialização:

- **CauchyRandomGenerator**: equivale ao método de distribuição de *cauchy*;
- **GaussianRandomGenerato**r: equivale ao método de distribuição normal;
- **StockRandomGenerator**: equivale ao método de inicialização randômica uniforme.

Em análise quantitativa, o *JGAP* tem vantagem sobre o *AFORGE*, já que possui um número maior de métodos de inicialização. Essa vantagem possibilita uma maior abrangência de escolha de criar uma nova população.

Já para análise de métodos disponibilizados, o *AFORGE* fica preso ao método de inicialização randômica uniforme - o método de inicialização mais utilizado por autores como HANDLOOK (2012) e LUCAS (2002) - que distribui aleatoriamente, um elemento no conjunto de alelos para os genes de um indivíduo. Já o *JGAP*, além do método de inicialização randômica uniforme, fornece os métodos de distribuição de *cauchy* e distribuição normal.

Com as análises acima, pode-se verificar que o *JGAP* apresenta mais viabilidade para sua escolha, por utilizar de maior número de métodos inicialização.

4.4.5. Métodos de aptidão

Para cada problema envolvendo AGs, é necessário que cada um dos métodos seja adaptado a fim de garantir que o problema seja solucionado pelo algoritmo e, além disso, que a solução seja feita da melhor forma possível. Já a função de aptidão deve ser implementada especificamente para o problema cuja solução está sendo criada, visto que para cada problema a aptidão do cromossomo será calculada de acordo com os critérios previamente estabelecidos.

Porém, diferentemente dos demais, o método de aptidão deve ser implementado especificamente para o problema cuja solução está sendo buscada, visto que não há uma forma genérica de se calcular a aptidão de um cromossomo. No caso do PCV, precisa-se definir um valor de aptidão para cada um par de genes (que refere a duas cidades) para descobrir a distância entre as duas cidades.

Por isso os dois *frameworks* não se propõem a definir métodos para se calcular a aptidão, deixando a responsabilidade pela implementação por conta do usuário do *framework* para que este, após análise do problema, encarregue-se de

codificá-lo; em contrapartida, ambos os *frameworks* provêm as Interfaces que deverão ser implementadas, sendo elas *IFitnessFunction* e *FitnessFunction*, do *AFORGE* e *JGAP* respectivamente.

Criou-se a função de aptidão para o *framework AFORGE*, descrita no Apêndice B, e o Apêndice D descrito para o framework JGAP.

Como a implementação do cálculo da aptidão fica sob responsabilidade do usuário do *framework*, não há como avaliar os *frameworks* nesse quesito, porém ressalte-se que ambos oferecem o devido suporte para sua implementação.

4.4.6. Métodos de cruzamento

O *framework AFORGE* não possui opções de métodos de cruzamento, dispondo unicamente do método de cruzamento multiponto, que se utiliza de dois pontos de corte e dois cromossomos pais para gerar dois cromossomos filhos.

O *framework JGAP* dispõe do método de cruzamento guloso, que é definido para o uso na solução do PCV. Necessitam-se, neste método, de dois cromossomos pais para se obter um cromossomo filho.

O método de cruzamento multiponto, disponibilizado pelo *AFORGE*, tem a vantagem de gerar proles com blocos de genes dos pais, blocos estes definidos de forma aleatória. Esta vantagem é evidente, pois possibilita a geração de indivíduos com características mescladas, ou seja, indivíduos com genes dos pais, mas organizados de forma diferente, o que pode resultar em um indivíduo mais apto a ser a solução do problema, ou até mesmo o indivíduo solução. Porém esta vantagem passa a ser uma desvantagem no problema proposto, pois há a possibilidade de existirem genes iguais nos indivíduos-pais, o que no problema proposto seriam as

cidades; como o problema proposto exige a não repetição de cidades, esse método se torna inviável.

Em contrapartida, o *JGAP*, com o método de cruzamento guloso, contorna essa característica do cruzamento multiponto, por ser usado especialmente para resolver o PCV, onde garante a não repetição dos genes resultantes. Por este motivo o *JGAP* se apresenta como o *framework* com o método de cruzamento apropriado para o problema, obtendo assim vantagem em relação ao *AFORGE*.

4.4.7. Métodos de mutação

O *AFORGE* não possui opções de métodos de mutação, mas utiliza do método mutação por troca (ou *swap*), que troca dois genes entre dois cromossomos.

O *JGAP* também se utiliza do método de mutação por troca, feito pelo método *SwappingMutationOperator*, disponível no *framework*.

No que diz respeito aos métodos de mutação, os *frameworks* poderiam disponibilizar mais opções, como mutação *flip* ou *creep*, que são de fácil implementação, mas que enriqueceriam um pouco mais cada uma das bibliotecas.

O fato de os *frameworks* disponibilizarem apenas uma opção de método de mutação acaba sendo um ponto negativo para ambos; isso pode ser contornado com implementações próprias no algoritmo, porém deve-se avaliar a necessidade de um método de mutação adicional de acordo com o resultado desejado.

4.4.8. Linguagem

A linguagem de programação utilizada pelos *frameworks* pode influenciar no momento da escolha. O site ALIOTH, especializado em medir e comparar desempenho de 24 linguagens de programação utilizando um computador com sistema operacional Linux Ubuntu, processador Intel Q6600 *one core*, disponibiliza uma tabela comparativa de uso de tempo de execução, uso de memória e quantidade de código usado.

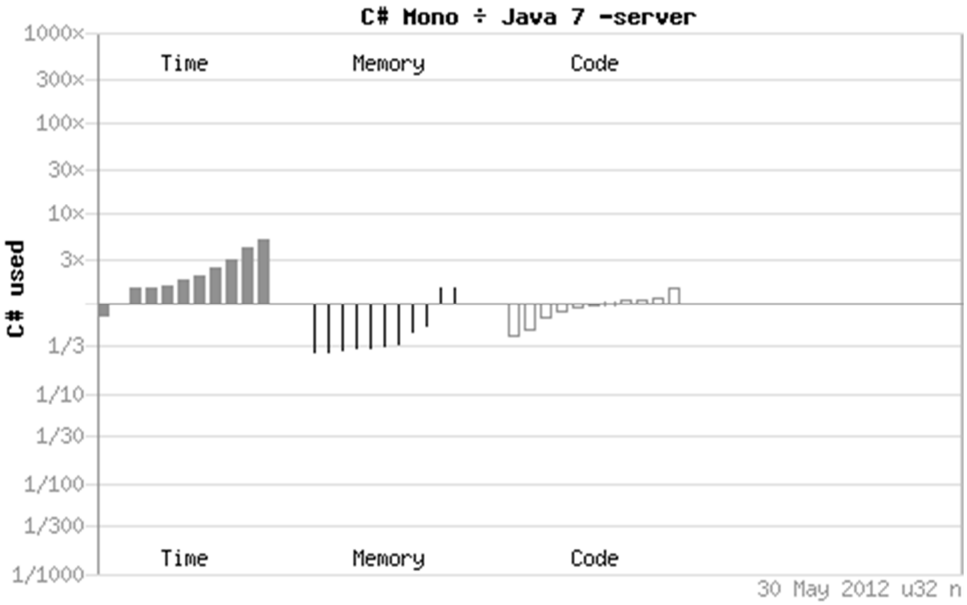

Figura 43 - Comparação desempenho do C# *versus* JAVA.

ALIOTH também a Figura 43, que exibe graficamente a comparação de desempenho do C# em relação ao Java. O tempo de execução chega a ser cinco vezes mais rápido, uso de memória ¼ menor e ½ do uso de código.

C# Mono used what fraction? **used** how many times more?			
Benchmark	Time	Memory	Code
pidigits	±	1/3	±
spectral-norm	±	1/4	±
n-body	±	1/3	±
fannkuch-redux	±	1/4	1/2
reverse-complement	2×	1/3	±
fasta	2×	1/3	±
fasta-redux	2×	1/3	±
mandelbrot	2×	1/2	±
k-nucleotide	3×	±	±
regex-dna	4×	±	1/2
binary-trees	5×	1/2	±

C# Mono used what fraction? **used** how many times more?							
Time-used	|-	|---	25%	**median**	75%	---|	-|
(Elapsed secs)	±	±	±	2×	3×	5×	5×

Figura 44 - Comparação detalhada do desempenho do C# *versus* JAVA.

A Figura 44 mostra as informações detalhadas do desempenho do C# *versus* JAVA. A solução de uma árvore binária chega a ser cinco vezes mais rápido e ½ menos uso de memória.

4.4.9. Quantidade de documentação disponível

Há duas formas de verificar a disponibilidade de documentação disponível: averiguando as documentações disponíveis no *site* oficial ou verificando o número de tutoriais disponíveis na *web*.

O *AFORGE* disponibiliza uma página em seu site com documentação *on-line* e *off-line* em inglês. A documentação *off-line* está em formato HTML, conhecida como CHM (formato padrão para documentação de *software*). Tais arquivos podem

ser obtidos na página de *download* do site *AFORGE*. A versão *on-line* está disponível para versão 2.0 do *framework*. Já a versão mais recente do *framework* é a 2.2.4. Esta diferença de versões pode causar falhas de referências de algumas classes.

Os tutoriais para o *AFORGE* não estão disponíveis, pois ao utilizar a ferramenta de busca *Google* não retorna tutorial ou sites disponíveis na *web*. Não há também disponibilidade em seu *site*.

O *JGAP* disponibiliza também em seu site a documentação *on-line* e *off-line* em inglês. A documentação *off-line* possui o formato HTML e pode ser obtido na página *documentation* do site *JGAP*. A versão *on-line* está disponível na ultima versão do *framework*, a 3.6.

O *JGAP* possui em seu *site* um tutorial básico para instalação e configuração. Realizaram-se também, pesquisas referentes a tutoriais disponíveis para o *JGAP*, retornando um trabalho disponível por Michay (2011) com uma breve introdução ao *framework*.

As documentações de ambos os *frameworks* ficam vinculados apenas à descrição de parâmetros de métodos de classes, e deixam a desejar sobre informações que realmente venham a suprir a necessidade do usuário, como uma boa descrição da utilidade da classe e método.

4.4.10. Quantidade de suporte

Os *frameworks AFORGE* e *JGAP* possuem comunidades ativas que são utilizadas como lista de discussão para o suporte gratuito. Para contabilizar a disponibilidade

de suporte dos *frameworks*, realizou-se uma pesquisa para contabilizar a quantidade de informações trocadas nas listas de discussão.

No período de 2009 a maio de 2012, contabilizam-se 376 mensagens trocadas entre usuários ao fórum de suporte do *JGAP*. Tais mensagens têm agilidade de resposta da comunidade sendo, na maioria das vezes, respondida por *Klaus Meffert*, responsável pela equipe de programadores do *framework*.

Já o *AFORGE*, no mesmo período, contabiliza-se 342 mensagens trocadas entre usuários no fórum de suporte. As mensagens são respondidas pelo criador *Andrew Kirillov* e com ajuda de colaboradores que fazem parte de sua equipe. O tempo de resposta é rápido e na maioria das vezes o problema apresentado é resolvido.

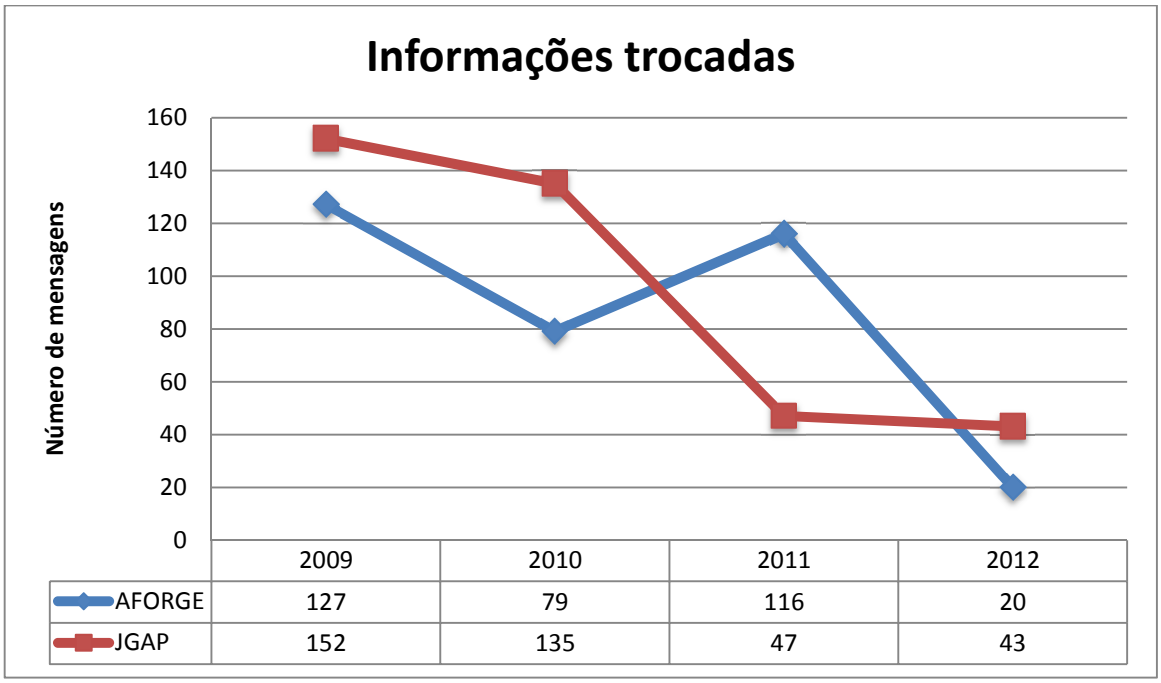

Figura 45 - Comparação de informações trocadas.

A Figura 45 exibe a comparação entre a quantidade de informações trocadas nos dois fóruns. Os *frameworks* possuem comunidades com um total de mensagens

trocadas consideravelmente iguais. Conclui-se assim, que os fóruns são de grande ajuda para quem necessita de sanar suas dúvidas. As diferenças de informações trocadas nos fóruns dos *frameworks* não influenciam na sua escolha.

4.4.11. Qualidade dos resultados

Faz-se necessário comparar a qualidade dos resultados entre os *frameworks*. Primeiramente, definiram-se as coordenadas das cidades para cada *framework*, a Figura 46 exibe um mapa com tais coordenadas.

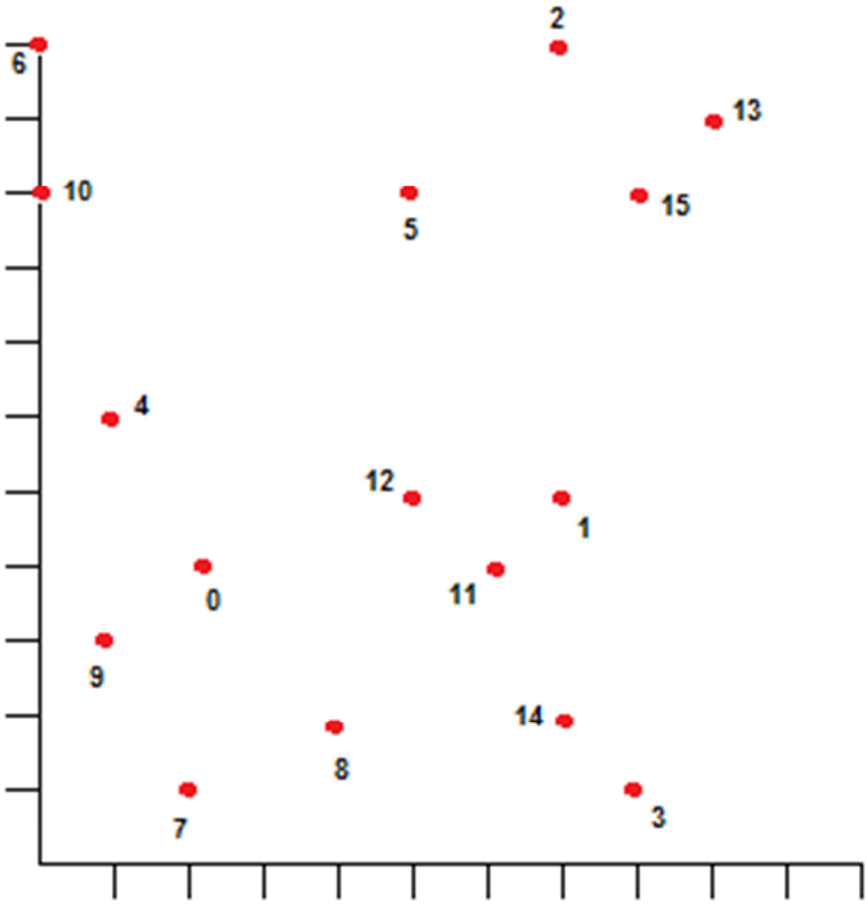

Figura 46 - Coordenadas cidades para comparação da qualidade dos resultados.

Após, necessitou-se definir os menores caminhos de modo que percorre todas as cidades:

- o menor caminho: 14, 3, 8, 7, 9, 0, 4, 10, 6, 5, 2, 13, 15, 1, 12, 11 e retornando a cidade 14, resultando com o valor de *fitness* de 40,459. Ilustrado como exemplo, na Figura 47;
- o segundo menor caminho: 14, 3, 11, 12, 1, 15, 13, 2, 5, 6, 10, 4, 0, 9, 7, 8 e retornando a cidade 14, resultando o valor de *fitneess* 40,705;
- o Terceiro menor caminho: 14, 11, 12, 8, 7, 9, 0, 4, 10, 6, 5, 2, 13, 15, 1, 3 e retornando a cidade 14, resultando o valor de *fitness* 41,621.

Para

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0															
1	5,099	0														
2	8,602	6	0													
3	6,708	4,123	10,049	0												
4	2,236	6,082	7,81	8,602	0											
5	5,83	4,472	2,828	8,544	5	0										
6	7,28	9,219	7	12,806	5,099	5,385	0									
7	3	6,403	11,18	6	5,099	8,544	10,198	0								
8	2,828	4,242	9,486	4,123	5	7,071	9,848	2,236	0							
9	1,414	6,324	10	7,28	3	7,211	8,062	2,236	3,162	0						
10	5,385	8,062	7,28	11,313	3,162	5	2	8,246	8,062	6,082	0					
11	4	1,414	7,071	3,605	5,385	5,099	9,219	5	2,828	5,099	7,81	0				
12	3,162	2	6,324	5	4,123	4	7,81	5	3,162	4,472	6,403	1,414	0			
13	9,219	5,385	2,236	9,055	8,944	4,123	9,055	11,401	9,433	10,63	9,055	6,708	6,403	0		
14	5,385	3	9	1,414	7,211	7,28	11,401	5,099	3	6,082	9,899	2,236	3,605	8,246	0	
15	7,81	4,123	2,236	8	7,615	3	8,246	10	8,062	9,219	8	5,385	5	1,414	7,071	0

Figura 47 - Tabela de distancia entre as cidades.

Esses caminhos são definidos a partir do valor de *fitness* do cromossomo, que representam as cidades do PCV. Precisa-se utilizar o método *distance* (ilustado

na Figura 29), que calcula a distancia entre duas cidades no mapa, resultando em uma tabela que pode visto na Figura 47.

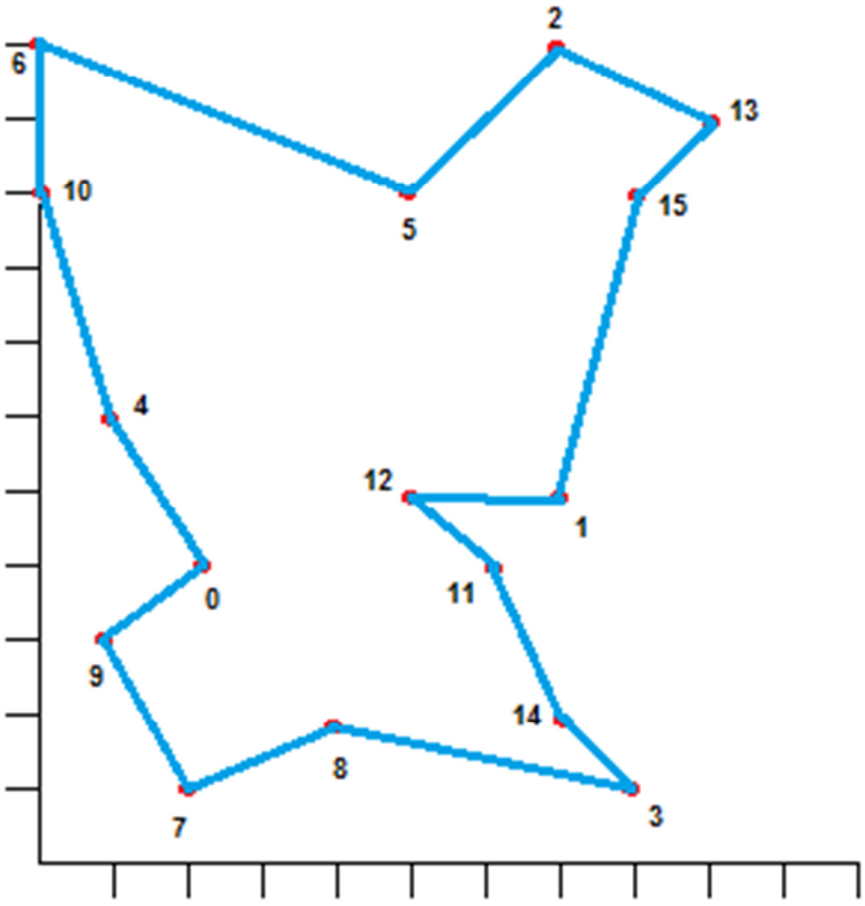

Figura 48 - Menor caminho entre as cidades.

Ao executar o código do *AFORGE* para resolver o PCV (Apêndice A), que se utilizou do método de inicialização randômica uniforme, de seleção o melhor cromossomo, de cruzamento multiponto, de mutação por troca, máximo de evoluções igual a vinte e tamanho da população igual a oitocentos, resultou no terceiro menor caminho.

Já ao executar o código do *JGAP* (Apêndice C), que se utilizou do método de inicialização randômica uniforme, de seleção o limiar, de cruzamento o guloso, de

mutação por troca, máximo de evolução igual a vinte e tamanho da população igual a oitocentos, resultou no segundo menor caminho.

Os resultados utilizando os *frameworks* podem variar. A ressaltar para fins de comparação, executou-se uma única vez seu código, e quanto mais evoluções e quantidade de população, mais chance de achar o menor caminho.

4.4.12. Resumo da comparação

A seguir será apresentada uma tabela que resume as comparações feitas nas subseções anteriores.

Tabela 3 - Tabela Comparativa.

	AFORGE	*JGAP*
Tempo de execução	709,6 milissegundos	3143,4 milissegundos
Instalação	Simples	Simples
Métodos de seleção	• Método melhor cromossomo • Método por classificação • Método roleta	• Método melhor cromossomo • Método limiar • Método torneio • Método por classificação
Método inicialização	Inicialização randômica uniforme	• Distribuição de *Cauchy*. • Distribuição normal • Inicialização randômica uniforme
Método de aptidão	Personalizado	Personalizado
Método de cruzamento	Multiponto	Guloso
Método de mutação	Mutação por troca	Mutação por troca
Linguagem	C#	Java
Quantidade de documentação disponível	Baixo	Baixo
Disponibilidade de suporte	Sim	Sim
Qualidade dos resultados	Boa	Boa

Após a tabela comparativa, uma sinopse da comparação é feita a seguir:

- Pontos fortes do *framework AFORGE*:
    - **tempo de execução**: 709,5 contra 3143,4 milissegundos do *JGAP*;
    - **instalação**: simples, necessitando apenas da ferramenta *Visual Studio* e o arquivo para abrir todos os arquivos necessários para seu funcionamento;
    - **quantidade de linhas de código**: no *AFORGE* necessitam-se menos linhas de códigos para solucionar o PCV em comparação ao *JGAP*.
- Pontos fortes do *framework JGAP*:
    - **método de seleção**: apesar de a diferença ser pouca em relação ao *AFORGE*, a maioria de opões de método de seleção, torna um ponto forte;
    - **método de inicialização**: com três opões de escolha contra apenas um método de inicialização do *AFORGE*;

Em geral, os dois *frameworks* possuem pontos fracos na quantidade de métodos de cruzamento, mutação e quantidade de documentação. Já um ponto forte que deve ser levado em consideração é a disponibilidade de suporte, pois possuem comunidades ativas que são de grande ajuda quando algum problema ou dúvida venha a surgir e que os resultados para solução do PCV foi satisfatório.

5 CONSIDERAÇÕES FINAIS

Este trabalho apresentou os conceitos básicos da inteligência artificial, com foco em uma de suas técnicas, os Algoritmos Genéticos. Os estudos destes conceitos é muito importante para a compreensão do trabalho, como também para realizar a instalação, configuração e implementação dos *frameworks AFORGE* e *JGAP,* como também para resolver o Problema do Caixeiro Viajante (PCV).

Foi realizada a comparação dos *frameworks* utilizados neste trabalho, o *AFORGE* e *JGAP*. Os critérios de comparação foram divididos da seguinte forma. Primeiramente, foi verificado o tempo de execução para resolver o PCV, utilizando um algoritmo que quantifica esse tempo que resultou com o ganho do *framework AFORGE*. Em seguida, foi descrito a instalação, partindo da aquisição dos arquivos e finalizando com a configuração dos *frameworks* nos seus respectivos ambiente de trabalho.

Após foram feitos comparações dos métodos de seleção. Em uma análise quantitativa, o *JGAP* possui vantagem, e em análise dos métodos disponibilizados, além de ambos os *frameworks* possuírem os métodos de melhor cromossomo e classificação, o *AFORGE* possui o método por roleta. Já o *JGAP* possui o método torneio e limiar. No método de inicialização, ambos possuem o método de inicialização randômica uniforme. O *JGAP* possui também os métodos *cauchy* e distribuição normal. No método de aptidão, pela necessidade de cada problema possuir sua forma de calcular o valor dos genes de um cromossomo. No caso do PCV, foi necessário que o método calculasse o valor de cada par de genes, referentes a um par de cidades, a fim de descobrir a distancia entre eles.

No método de cruzamento, o *AFORGE* disponibiliza do método multiponto, já o *JGAP* o método guloso, que se torna mais viável se for utilizado para resolver o PCV. No método de mutação, os dois *frameworks* possuem somente o método por troca. A linguagem, avaliada pelo site ALIOTH, disponibiliza resultados sobre comparações das duas linguagens. A linguagem C# se torna mais eficiente na análise de tempo de execução.

A quantidade de documentação foi verificada pelas documentações disponíveis no *site* oficial ou verificando o número de tutoriais disponíveis na *web*. Ambos os *frameworks* possuem documentações *on-line* e *off-line* e o *JGAP* possui uma referência de tutoriais disponível. A quantidade de suporte foi medida, contabilizando a quantidade de informações trocadas nos fóruns de cada *framework*, que obtiveram as quantidades consideravelmente iguais. A qualidade dos resultados foi definida utilizando um método que calcula o valor de *fitness* do cromossomo, tendo como resultados satisfatórios.

As comparações dos *frameworks AFORGE* e *JGAP* servirão de auxilio para futuros projetos que necessitam do uso de um *framework*, verificando o que vão encontrar ao utiliza-los.

Como sugestão para trabalhos futuros, propõe-se utilizar de novos critérios de comparação, utilizar outros *frameworks* e possivelmente em outras linguagens de programação e criação de uma interface. Dessa forma, as comparações ficam mais disponíveis a quem está acostumado a um tipo de linguagem ou necessitam de mais critérios de comparação.

6 REFERÊNCIAS BIBLIOGRÁFICAS

AFORGE. Disponível em: <http://www.aforgenet.com/framework>. Acessado em 20/09/2011.

ALIOTH. Disponível em: <http://shootout.alioth.debian.org>. Acessado em 31/05/2011.

BARRETO, J. M. **Inteligência Artificial no limiar do Século XXI:** abordagem Híbrida Simbólica, conexionista e Evolucionária. 3. ed. Florianópolis: ρρρ Edições, 2001. 183 p.

BRAGA, Antônio de Pádua. **Redes Neurais Artificiais**: teorias e aplicações. 1. ed. Rio de Janeiro: LTC, 2000. 262 p.

COPPIN, Ben. **Inteligência Artificial**. 1. ed. Rio de Janeiro: LTC, 2010. 636 p.

FERREIRA, Aurélio Buarque de Holanda. **Dicionário Aurélio Básico da Língua Portuguesa**. 1. ed. Rio de Janeiro: Nova Fronteira, 1988. 214 p.

GRIGOLETTI, Pablo Souza. **Algoritmos Genéticos em Otimização Combinatória:** Teoria e Prática. Rio Grande do Sul, 2006, 35 p. (Pós-Graduação em Computação). Disponível em:<http://www-usr.inf.ufsm.br/~andrezc/ia/swap_mutate_greedy_crossover.pdf>. Acesso em: 03 de Maio de 2012.

GUJ. Disponível em: <http://guj.com.br/content/articles/netbeans/netbeans_guj.pdf>. Acessado em 05/06/2012.

HANDBOOK, Disponível em <http://www.itl.nist.gov/div898/handbook/eda/section3/eda3663.htm >. Acessado em 20/04/2012.

LUCAS, Diogo C. **Algoritmos Genéticos:** uma Introdução. Rio Grande do Sul: UFRGS, 2002, 47 p. Disponível em: <http://www.inf.ufrgs.br/~alvares/INF01048IA/ApostilaAlgoritmosGeneticos.pdf>. Acesso em: 24 de outubro de 2011.

JGAP. Disponível em: <http://jgap.sourceforge.net>. Acessado em 20/09/2011.

KHANBARY, Lufi Mohammed Ommer. **Modified Genetic Algorithm with Threshold Selection**. India: School of Computer and Systems Sciences, Jwahalal Nehru University, 2009, 34 p. Disponível em: <http://www.ceser.in/ceserp/index.php/ijaii/article/view/784>. Acessado em 23/05/2012.

MIRANDA, Marcio Nunes de. **Algoritmos Genéticos:** fundamentos e aplicações: Rio de Janeiro: UFRJ, [s.a.], online, departamento de teleinformática e automação. Disponível em: <http://www.nce.ufrj.br/GINAPE/VIDA/alggenet.htm>. Acesso em: 29 de outubro de 2011.

OCHI, Luiz Satoru. **Algoritmos Genéticos:** origem e evolução: Rio de Janeiro: UFF, [s.a.], online, departamento de ciência da computação do instituto de matemática. Disponível em: <http://www.sbmac.org.br/bol/bol-2/artigos/satoru/satoru.html#bibliografia>. Acesso em: 24 de novembro de 2011.

PACHECO, Marco Aurélio Calvalvanti. **Algoritmos Genéticos:** princípios e aplicações. Rio de Janeiro: PUC-RIO, 1999, 9 p. departamento de laboratório de inteligência computacional aplicada. Disponível em: <http://www.ica.ele.puc-rio.br/Downloads/38/CE-Apostila-Comp-Evol.pdf>. Acesso em: 24 de outubro de 2011.

REZENDE, Solange Oliveira. **Sistemas Inteligentes**: fundamentos e aplicações. 1 ed. Barueri: Editora Manole Ltda, 2003. 525 p.

ROSA, Thatiane de Oliveira. **Aplicações de algoritmos genéticos para o alinhamento de sequências biológicas**. Palmas, CEULP, 2007. 72 p. Dissertação (Trabalho de Conclusão de Curso I em sistemas de informação).

RUSSEL, Stuart; NORVIG, Peter. **Inteligência Artificial**. 2. ed. Rio de Janeiro: Campos, 2004. 1020 p.

SEDGEWICK, Robert: **Introduction to Programming in Java:** An interdisciplinary approach. 1 ed. Addison-Wesley Publishing Company, 2007. 736p.

SOARES, Gustavo Luís. **Algoritmos Genéticos:** estudo, novas técnica e aplicações: Belo Horizonte: UFMG, 1997, 145 p, Dissertação (pós-graduação em engenharia elétrica). Disponível em: <http:// cpdee.ufmg.br/~joao/TesesOrientadas/VAS1997_1.pdf>. Acesso em: 24 de outubro de 2011.

7 APÊNDICES

APÊNDICE A – Resolução do PCV utilizando o *framework AFORGE*.

```
1. using System;
2. using System.Collections.Generic;
3. using System.ComponentModel;
4. using System.Data;
5. using System.Drawing;
6. using System.Linq;
7. using System.Text;
8. using System.Windows.Forms;
9. using AForge.Genetic;
10.
11.   namespace CaixeiroViajante
12.   {
13.       public partial class Form1 : Form
14.       {
15.           int maxEvolucao = 128;
16.           int quantPopulacao = 512;
17.           int  quantCidades = 7;
18.           double[,] arrayCidades = new double[7, 2] {
19.               { 2, 4 },
20.               { 7, 5 },
21.               { 7, 11 },
22.               { 8, 1 },
23.               { 1, 6 },
24.               { 5, 9 },
25.               { 0, 11 }
26.           };
27.
28.
```

```
29.             public double distance(ushort a_from, ushort a_to)
30.             {
31.                 int a = Convert.ToInt32(a_from.ToString());
32.                 int b = Convert.ToInt32(a_to.ToString());
33.                 double x1 = arrayCidades[a,0];
34.                 double y1 = arrayCidades[a,1];
35.                 double x2 = arrayCidades[b,0];
36.                 double y2 = arrayCidades[b,1];
37.                 double val = (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
    y2);
38.                 return Math.Sqrt(val);
39.             }
40.
41.             public void main()
42.             {
43.                 SalesmanFitnessFunction fitnessFunction = new
    SalesmanFitnessFunction(this);
44.                 Population populacao = new Population(quantPopulacao, new
    PermutationChromosome(quantCidades), fitnessFunction, new
    EliteSelection());
45.                 for (int i = 0; i < maxEvolucao; i++)
46.                 {
47.                     populacao.RunEpoch();
48.                 }
49.                 String melhorSolucao =
    populacao.BestChromosome.ToString();
50.                 listBoxCromossomos.Items.Add(melhorSolucao);
51.         }
```

APÊNDICE B – Função de *Fitness* do *framework AFORGE*.

```
1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Text;
5. using AForge.Genetic;
6. namespace CaixeiroViajante
7. {
8.     class SalesmanFitnessFunction : IFitnessFunction
9.     {
10.           CaixeiroViajante.Form1 caixeiroViajante;
11.           public SalesmanFitnessFunction(CaixeiroViajante.Form1
   caixViajante) {
12.               caixeiroViajante = caixViajante;
13.           }
14.
15.           public double Evaluate(IChromosome cromossomo)
16.           {
17.               double s = 0;
18.               ushort[] genes = ((PermutationChromosome)cromossomo).Value;
19.               for (int i = 0; i < genes.Count() - 1; i++) {
20.                   s += caixeiroViajante.distance(genes[i], genes[i + 1]);
21.               }
22.               s += caixeiroViajante.distance(genes[genes.Count() - 1],
   genes[0]);
23.               return Int32.MaxValue / 2 - s;
24.           }
25.       }
26.   }
```

APÊNDICE C – Resolução do PCV utilizando o *framework JGAP*.

```
1. import org.jgap.*;
2. import org.jgap.event.*;
3. import org.jgap.impl.*;
4. import java.io.*;
5. import java.util.Date;
6. public class CaixeiroViajante {
7.     Configuration config;
8.     int maxEvolucao = 128;
9.     int quantPopulacao = 512;
10.       static final int  quantCidades = 7;
11.       static final int[][] arrayCidades = new int[][] {
12.           { 2, 4 },
13.           { 7, 5 },
14.           { 7, 11 },
15.           { 8, 1 },
16.           { 1, 6 },
17.           { 5, 9 },
18.           { 0, 11 }
19.       };
20.
21.       public IChromosome criarCromossomo(final Object a_initial_data) {
22.           try {
23.             Gene[] genes = new Gene[quantCidades];
24.             for (int i = 0; i < genes.length; i++) {
25.               genes[i] = new IntegerGene(config, 0, quantCidades - 1);
26.               genes[i].setAllele(new Integer(i));
27.             }
28.             IChromosome exemplo = new Chromosome(config, genes);
29.             return exemplo;
30.           }
```

```
31.             catch (InvalidConfigurationException iex) {
32.               throw new IllegalStateException(iex.getMessage());
33.             }
34.         }
35.
36.         public     FitnessFunction     criarFuncaoFitness(final     Object
    a_initial_data){
37.             return new SalesmanFitnessFunction(this);
38.         }
39.
40.         public     Configuration     criarConfiguracao     (final     Object
    a_initial_data) throws InvalidConfigurationException{
41.             Configuration configuracao = new Configuration();
42.             configuracao.removeNaturalSelectors(true);
43.             configuracao.addNaturalSelector(new
    ThresholdSelector(configuracao, 0.1), false);
44.             configuracao.setRandomGenerator(new StockRandomGenerator());
45.             configuracao.addGeneticOperator(new
    GreedyCrossover(configuracao));
46.             configuracao.setMinimumPopSizePercent(0);
47.             configuracao.setEventManager(new EventManager());
48.             configuracao.setFitnessEvaluator(new
    DefaultFitnessEvaluator());
49.             configuracao.addGeneticOperator(new
    SwappingMutationOperator(configuracao, 20));
50.             return configuracao;
51.         }
52.
53.         public    IChromosome    procurarMelhorSolucao    (final    Object
    a_initial_data) throws InvalidConfigurationException{
54.             config = criarConfiguracao(a_initial_data);
```

```
55.           FitnessFunction                funcaoFitness                =
   criarFuncaoFitness(a_initial_data);
56.           config.setFitnessFunction(funcaoFitness);
57.           IChromosome                exemploCromossomo                =
   criarCromossomo(a_initial_data);
58.           config.setSampleChromosome((exemploCromossomo));
59.           config.setPopulationSize(quantPopulacao);
60.           IChromosome[]           cromossomos           =           new
   IChromosome[config.getPopulationSize()];
61.           Gene[] exemploGenes = exemploCromossomo.getGenes();
62.           for (int i = 0; i < cromossomos.length; i++) {
63.               Gene[] genes = new Gene[exemploGenes.length];
64.               for (int j = 0; j < genes.length; j++) {
65.                   genes[j] = exemploGenes[j].newGene();
66.                   genes[j].setAllele(exemploGenes[j].getAllele());
67.               }
68.               cromossomos[i] = new Chromosome(config, genes);
69.           }
70.           Genotype    populacao    =    new    Genotype(config,    new
   Population(config, cromossomos));
71.           IChromosome melhorCromossomo = null;
72.           for (int i = 0; i < maxEvolucao; i++) {
73.               populacao.evolve();
74.               melhorCromossomo = populacao.getFittestChromosome();
75.           }
76.           return melhorCromossomo;
77.       }
78.       public double distance(Gene a_from, Gene a_to) {
79.           IntegerGene geneA = (IntegerGene) a_from;
80.           IntegerGene geneB = (IntegerGene) a_to;
81.           int a = geneA.intValue();
```

```
82.             int b = geneB.intValue();
83.             int x1 = arrayCidades[a][0];
84.             int y1 = arrayCidades[a][1];
85.             int x2 = arrayCidades[b][0];
86.             int y2 = arrayCidades[b][1];
87.             double val = (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2);
88.             return Math.sqrt(val);
89.         }
90.
91.         public static void main (String[] args){
92.             try{
93.                 long tempoInicial = System.currentTimeMillis();
94.                 CaixeiroViajante c = new CaixeiroViajante();
95.                 IChromosome               cromossomoExemplo               =
    c.procurarMelhorSolucao(null);
96.                 System.out.println("Solucao: ");
97.                 Gene[] genes = cromossomoExemplo.getGenes();
98.                 for(int i = 0; i < cromossomoExemplo.size(); i++)
99.                 {
100.                    System.out.print(genes[i].getAllele().toString()+"
    ");
101.                }
102.                System.out.println("");
103.                long tempoFinal = System.currentTimeMillis();
104.                Date data = new Date();
105.                BufferedWriter    saida    =    new    BufferedWriter(new
    FileWriter("C:/Users/Fox/Desktop/TempoExecucaoJGAP.txt", true));
106.                saida.newLine();
107.                saida.write(data.toGMTString());
108.                saida.write("  --  Tempo  de  Execução  ->  "+(tempoFinal-
    tempoInicial)+" Milissegundos");
```

```
109.                saida.newLine();
110.                saida.close();
111.            }
112.            catch(Exception ex){
113.                ex.printStackTrace();
114.            }
115.        }
116.    }
```

APÊNDICE D – Função de *Fitness* do *framework JGAP*.

```
1.  import org.jgap.*;
2.  public class SalesmanFitnessFunction extends FitnessFunction {
3.    private final CaixeiroViajante caixeiroViajante;
4.
5.    public SalesmanFitnessFunction(final CaixeiroViajante caixViajante) {
6.      caixeiroViajante = caixViajante;
7.    }
8.
9.    protected double evaluate(final IChromosome chromosome) {
10.     double s = 0;
11.     Gene[] genes = chromosome.getGenes();
12.     for (int i = 0; i < genes.length - 1; i++) {
13.       s += caixeiroViajante.distance(genes[i], genes[i + 1]);
14.     }
15.     s += caixeiroViajante.distance(genes[genes.length - 1], genes[0]);
16.     return Integer.MAX_VALUE / 2 - s;
17.   }
18. }
```