ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES – EACH USP

SISTEMAS DE INFORMAÇÃO

Disciplina:

[1]ACH2016 – Inteligência Artificial

Professor:

Clodoaldo A M Lima

Atividade:

Exercício Trabalho - Algoritmo Genético:

O Caixeiro Viajante

Componentes:	**NUSP:**
André Thomaz de Mello	**7137440**
Antonio Carlos Mateus da Silva	**8516031**
Luís Felipe de Almeida da Silva	**8516775**
Marcelo Kazuya Kajiwara	**8516903**

[1]Em busca do Selo de excelência "Relatório Responsa" Clodoaldo

1. Sobre implementação

O código do Algoritmo Genético para resolver o problema do Caixeiro Viajante foi implementado em Java e acompanha o documento no diretório src/. O código foi dividido nos seguintes pacotes:

- br.com.ia : contém as classes:
  - Caixeiro: é a classe principal para ser executada via linha de comando com os parâmetros correspondentes (explicados na seção Parâmetros Via Linha de Comando). O arquivo .java da classe (Caixeiro.java) que deve ser editado caso haja necessidade de editar os parâmetros que são configurados via código (explicados na seção Parâmetros Via Código); e
  - Leitor_Arquivo_Entrada: lê um arquivo de nome passado como parâmetro com cada linha representando uma cidade e o primeiro *token* sendo o id da cidade, o segundo a coordenada x e o terceiro a coordenada y. O método retornando uma matriz da classe *Matrix* (do pacote Jama, explicado na seção Bibliotecas Utilizadas) com cada linha representando uma cidade e com duas colunas, a primeira coluna representado a coordenada x e a segunda coluna y.
- br.com.ia.ga : contém as classes:
  - AlgoritmoGenetico: essa classe é chamada pela classe Caixeiro para manipular as classes Fitness, Seleção, Crossover e Mutacao durante as gerações sem a utilização de múltiplas threads. O principal método dela é o get_melhor_caminho() que retorna uma matriz com todos os melhores cromossomos em cada época executada (explicado na seção Saída). Além disso, essa classe contém métodos como calcula_distancias(), que retorna uma matriz que contém as distâncias entre todas as cidades, e a classe avalia() que retorna, dentre a matriz de fitness passada como parâmetro, uma vetor com o melhor, a média entre os fitness e a diversidade da população; a classe distancia_euclidiana() que retorna a distancia euclidiana entre duas cidades; e a classe gera_populacao_aleatoria() que retorna uma matriz que representa uma população aleatória com o tamanho determinado por parâmetro, onde cada linha representa um cromossomo (a representação do cromossomo será explicada na seção Codificação do Cromossomo) ;

- Fitness: a principal função dessa classe é calcular o fitness de cada um dos cromossomos passados na população e das distâncias entre as cidades, ambos passados por parâmetros para a função fitness() (o cálculo do Fitness é explicado com mais detalhes na seção Função Fitness). A tarefa de ordenar o fitness em ordem decrescente e de calcular o fitness acumulado (como é necessário para a implementação da Seleção via Roleta) é dessa classe também a partir do método ordena_fitness() e calcula_fitness_acumulado(), respectivamente;
- Seleção: é a classe que tem os métodos para aplicar os diferentes tipos de seleção implementados para o algoritmo genético. Ao chamar o método estático seleciona_candidatos() dessa classe, é necessário passar como parâmetro um operador de seleção que direciona a população (também passada como parâmetro) para ser submetida a algum dos métodos de seleção implementados pelo grupo que são explicados na seção Seleção;
- Crossover: tem os métodos para aplicar os diferentes tipos de crossover implementados para o algoritmo genético. Ao chamar o método estático aplica_crossover() dessa classe, é necessário passar como parâmetro o tipo de crossover que direciona a população (também passada como parâmetro) para ser submetida a algum dos tipos de crossover implementados pelo grupo que são explicados na seção Crossover; e
- Mutacao: tem os métodos para aplicar os diferentes tipos de mutação implementados para o algoritmo genético. Ao chamar o método estático aplica_mutacao() dessa classe, é necessário passar como parâmetro o tipo de mutação que direciona a população (também passada como parâmetro) para ser submetida a algum dos tipos de mutação implementados pelo grupo que são explicados na seção Mutação.

- br.com.ia.ga.threads: contém as classes:
  - AlgoritmoGeneticoThreads: assim como a classe br.com.ia.ga.ia.AlgoritmoGenetico, essa classe manipula durante as gerações a população submetendo os seus cromossomos a determinados tipos de seleção, crossover e mutação. Mas ao contrário da classe do AlgoritmoGenetico, o método get_melhor_caminho() utiliza uma ou mais threads (distintas da thread *main*) para executar o crossover e a mutação (o recurso *multi-threading* utilizado é detalhado na seção Threads). Todos os outros métodos da classe AlgoritmoGenetico são herdados.

- CrossoverThread: essa classe herda os métodos da classe java.lang.Thread e que, quando o método .start é chamado, executa o método Crossover.aplica_crossover(), passando para este, os parâmetros que foram passados ao construtor da classe. Ou seja, em *background* (em paralelo com a classe *main*), essa classe submete a um determinado número de cromossomos um determinado tipo de crossover; e
- MutacaoThread: essa classe herda os métodos da classe java.lang.Thread e que, quando o método start() é chamado, executa o método Mutacao.aplica_mutacao(), passando para este, os parâmetros que foram passados ao construtor da classe. Ou seja, em *background* (em paralelo com a classe *main*), essa classe submete a um determinado número de cromossomos um determinado tipo de mutação.

a. Codificação do Cromossomo

Cada cromossomo da população é a representação do caminho que o caixeiro viajante deve percorrer. Os genes do cromossomo são as cidades que farão parte desse caminho e a sequência dos genes representa exatamente a sequência das cidades percorridas pelo caixeiro durante a sua viagem (considerando que após a última cidade no cromossomo, o caixeiro percorre o caminho desta para a primeira cidade do cromossomo para completar o problema).

Cada um dos genes são, então, inteiros que representam cada uma dessas cidades (o *id* de cada cidade). Portanto, consideramos um alfabeto de [1,N], sendo N o número de cidades totais contidas no problema. A sequência é armazenada numa matriz (da classe *Matrix*) linha.

a.i. A População

A população é uma matriz (da classe *Matrix*), onde cada linha é um cromossomo, ou seja, cada linha é um caminho que representa uma solução possível para o problema. Os primeiros cromossomos (que fazem parte da população inicial) são gerados pelo método AlgoritmoGenetico.gera_populacao_aleatoria() a partir de um quantidade inicial da população e da quantidade de cidades que envolvem o problema.

Esse método utiliza a classe java.util.Random (assim como em muitos trechos do algoritmo) para gerar números aleatórios dentro do alfabeto possível para o problema. A cada novo inteiro gerado aleatoriamente, o método confere se o *id* gerado já foi incluso nesse cromossomo. Se essa cidade já foi inclusa, uma busca sequencial a partir deste *id* é feita para buscar uma cidade que não foi inclusa ainda.

Embora isso evite gerar soluções infactíveis, isso não evita que o método gere soluções ruins. No decorrer do algoritmo (em crossover e mutação, mais especificamente), a ideia de não gerar em nenhum momento soluções infactíveis, já que a função de Fitness não pune esse tipo de cromossomo, como mais explicado na seção Função Fitness mais adiante.

b. Função Fitness

Cada cromossomo na população tem um valor de *fitness* que em suma representa o quão bom é aquela solução. A modelagem do problema e do fenótipo dos cromossomos foi feita de forma a tornar o problema do Caixeiro Viajante um problema de maximização. Portanto, o fitness de cada cromossomo é o inverso da distância percorrida em todo o caminho, ou seja, o inverso da soma das distâncias entre as cidades.

O método Fitness.calcula_fitness() é o responsável por calcular o *fitness* de cada um dos cromossomos da população passada como parâmetro. Além de cada um dos cromossomos, o método deve receber como parâmetro uma matriz que contém todas as distancia entre as cidades, um para um (o elemento da linha 1 e coluna 2, representa a distância da cidade 1 para a cidade 2, assim como o elemento da linha 3 e coluna 5, representa a distância da cidade 3 e a cidade 5, e assim por diante).

Cada um dos *fitness* calculados por esse método é armazenado em uma matriz coluna, onde o *fitness* contido na posição 1 é o *fitness* calculado para o cromossomos da linha 1 da população passada como parâmetro e assim por diante. Embora seja óbvio para alguns, esse fato deve estar bem claro já que o método Fitness.ordena_fitness() tem a tarefa de ordenar a matriz de *fitness* na ordem decrescente juntamente com os índices que (muito provavelmente) foram "embaralhados" nessa ordenação. Portanto, o *fitness* da primeira linha não necessariamente é do cromossomo da linha 1 na população em questão. Esse método retorna uma matriz cujo elemento na primeira coluna representa um *fitness,* na ordem decrescente, e o elemento da segunda coluna representa o índice do cromossomo na matriz da população do qual o *fitness* da mesma linha pertence.

c. Seleção

A cada geração, os cromossomos da população corrente são submetidos a uma seleção para determinar quais indivíduos serão os candidatos a realizar crossover. Foram

implementados dois tipos de seleção: via Roleta e via Torneio, que são escolhidos via parâmetro da linha de comando (explicado na seção Via Linha de Comando).

A **seleção via Roleta** é realizada pelo método Selecao.roleta() que é privado e é chamado quando o método Selecao.seleciona_candidatos() recebe como parâmetro a variável operador_selecao=0. O método da roleta recebe como uma parâmetro adicional, além da matriz contendo os cromossomos e a matriz de fitness, a quantidade de vezes que a "roleta será girada", ou seja, quantos cromossomos serão selecionados para compor a nova população de candidatos (sem contar com os cromossomos da subpopulação).

Primeiramente, o algoritmo usa os métodos presentes na classe Fitness para obter uma matriz com os *fitness* ordenados e com os índices do cromossomo que cada um dos *fitness* ordenados pertence. A seguir, novamente a partir dos métodos da classe Fitness, obtém uma matriz com os *fitness* acumulados. Depois de conseguir essa matriz, a ideia é gerar um número aleatório ("girar a roleta"), entre 0 e o número máximo na matriz de *fitness* acumulados, e identificar qual é o índice do intervalo em que esse número está situado na matriz de *fitness* acumulados. Assim o cromossomo escolhido para compor a população de candidatos tem o índice que está armazenado na matriz de índices de cromossomos ordenados na posição de mesmo índice do intervalo do número da roleta.

Por exemplo: se um número r gerado pela roleta é menor que o número na posição 3 da matriz e maior que o número da posição 4, o intervalo de r tem índice 3. Supondo que na posição 3 da matriz de índices ordenados (retornada pelo método Fitness.ordena_fitness()) tenha o número 9, isso quer dizer que o cromossomo escolhido é o da linha de índice 9 da matriz da população com todos os cromossomos que estão participando da seleção. Esse processo é repetido várias vezes, ou seja, a roleta é girada N vezes, sendo N igual ao inteiro passado como parâmetro na variável quantidade_roleta_gira.

A **seleção via Torneio** é mais simples de implementar do que a seleção via roleta pelo fato de não precisar fazer operações adicionais com a matriz de *fitness*. Esse algoritmo se baseia na ideia da disputa entre dois indivíduos em uma competição onde o melhor indivíduo sai vencedor. Mas nem sempre.

Levando em consideração o fato que nem sempre o melhor vence, o algoritmo no método Selecao.torneio() começa definindo uma probabilidade *k* para o melhor indivíduo sair vencedor. Modelamos de tal forma que essa probabilidade é definida dentro do método via código, nas primeiras linhas, ao invés de ser um parâmetro da função. Depois disso são

escolhidos aleatoriamente dois indivíduos da população que está sendo selecionada e, então, é gerado um número *r* aleatório para esse par de cromossomo.

Se r for menor que k (a probabilidade definida para a seleção) o cromossomo com maior *fitness* fará parte da população de candidatos a crossover, caso for maior, o cromossomo com menor *fitness* será escolhido. O número k foi escolhido de acordo com os melhores resultados analisados na seção Análises.

Uma seleção feita pura e simplesmente pelos métodos a cima podem eventualmente eliminar os cromossomos de melhor *fitness* da população. Por isso, há a opção de manter uma subpopulação com um número controlado de indivíduos com melhor *fitness* a cada seleção feita em cada geração passada. O tamanho dessa população é passada como parâmetro (explicado melhor na seção Parâmetros Via Código) e ela é adicionada a população que e retornada pelo método Selecao.seleciona_candidatos(), conjunto agora que vira a nova população corrente que será submetida ao crossover, explicado a seguir na seção Crossover.

d. Crossover

Foram implementados três tipos de crossovers dentro da classe Crossover. O método estático aplica_crossover() retorna uma matriz (do tipo Matrix) e controla o tipo de crossover que será executado. Para que este procedimento possa ser realizado com êxito, os seguintes parâmetros devem ser passados:

- Matrix populacao: Matrix que representa a população na qual deseja-se aplicar a operação. Tal população é gerada depois da realização da seleção, explicada anteriormente na seção Seleção.
- double taxa_crossover: *double* que representa o tamanho do corte do cromossomo que irá definir os genes para o crossover (detalhes em Parâmetro Via Linha de Comando);
- int tipo_crossover: um número inteiro que identifica qual dos três tipos de crossovers será executado, cada um deles será identificado e explicado no decorrer deste tópico;
- boolean pais_sobrevivem: determina a sobrevivência dos pais (cromossomos iniciais) após aplicação.

O método crossover_ordem() gera uma nova população a partir do **crossover baseado em ordem**, é executada quando a variável tipo_crossover recebe o valor “2” e funciona da seguinte forma: Selecionamos um conjunto de posições de forma randômica em um cromossomo que pode ser denotado como pai 2 (P2). Estes genes (posições) selecionados

anteriormente são localizados em um outro pai 1 (P1). Em seguida, os genes são reordenados para que estejam na mesma ordem presente no segundo pai, porém, mantendo as posições de P1. Posteriormente, os genes já reordenados são copiados para um filho 1 (F1) nas mesmas posições de P1. Para a obtenção de um outro filho, o mesmo processo é feito, trocando apenas o pai 1 com o pai 2, gerando um filho F2 diferenciado.

Já o método crossover_posicao() gera uma nova população a partir do **crossover baseado em posição**, este método é executado quando a variável tipo_crossover recebe o valor "1". Começa pela seleção de um conjunto de posições randômicas de um dos pais em questão, os genes destas posições são copiadas em um filho exatamente na mesma posição em que se encontram no pai de origem.

Após a cópia, os genes replicados são "removidos" do segundo pai, extraindo-se os que faltam no filho criado, os genes faltantes são copiados na ordem em que se encontram da esquerda para a direita nas posições sem valores do filho.

Atribuindo o valor "0" para a variável tipo_crossover, o método crossover_OX é executado. Tal método executa o **crossover dito como cruzado (ou de ordem)**, funcionando da seguinte maneira: Deve-se selecionar dois pontos para corte, definido um subconjunto do cromossomo selecionado, esse subconjunto é copiado para um filho, portanto, dado um pai (P1), seu subconjunto selecionado é copiado para um filho (F1) na mesma posição que estão os genes do pai. Com um pai (P2), preenchemos as posições que restaram em F1 copiando de P2 o primeiro gene que está posicionado depois do segundo ponto de corte e que ainda não está presente no filho para a primeira posição que não está preenchida no filho depois do segundo ponto de corte. O processo de cópia é feito com os próximos genes de P2 até que os genes de F1 estejam completos (com todos os valores atribuídos).

e. Mutação

A cada geração, os cromossomos da população corrente são submetidos ao processo de mutação. Depois do crossover, esta é a última "transformação" da população antes que o melhor *fitness* seja extraído. Implementamos três tipos distintos de mutação: mutação simples, mutação inversiva e a mutação que chamamos de "alternativa", que podem ser escolhidas mudando os parâmetros via código (como é melhor explicado na seção Parâmetros Via Código).

Cada um dos métodos de mutação retornam uma nova população que tem o mesmo tamanho da população que foi recebida como parâmetro. Nessa nova população estão todos os cromossomos mutantes que passaram pelo processo de mutação, ou seja, todos eles retornam uma matriz com a população mutante.

A **mutação alternativa** é aplicada pelo método Mutacao.mutacao_alternativa() que tem acesso privado. É chamado pelo método Mutacao.aplica_mutacao() quando o parâmetro do tipo da mutacao é igual a 1 (tipo_mutacao=1).

O algoritmo desse tipo de mutação faz com que todos os cromossomos da a população que foi passada para o método se tornem cromossomos mutantes. A cada cromossomo, define-se uma "zona de mutação", delimitada por um gene inicial e um gene final (semelhante a "zona de corte" do crossover. O primeiro gene da zona é gerado aleatoriamente e o gene final é definido pela taxa de mutação definida para o problema. Por exemplo: caso a taxa de mutação tenha sido definida como 20% e o tamanho do cromossomo é 100, o gene final está localizado (0,2)*(100)-1=20-1=19 genes a frente do cromossomo inicial, o que significa que 20 genes estão na zona de mutação do cromossomo.

Após definida a "zona de mutação", os genes são invertidos de forma que o primeiro gene da zona fique no lugar do último e vice-versa, o segundo gene da zona fique no lugar do penúltimo da zona e vice-versa, e assim por diante, de tal forma que o gene localizado no meio da zona permaneça na mesma posição. Esse processo é aplicado em todos os cromossomos da população original.

Por exemplo: supondo que o cromossomo seja 10->2->4->7->1->6->8->3 e a zona de corte seja do gene da posição 2 até o gene da posição 4, ou seja, a zona de mutação é 2->4->7. Aplicada a mutação alternativa, a zona se inverteria 7->4->2 e o cromossomo mutante resultante é: 10->7->4->2->1->6->8->3. Assim o cromossomo mutante será o mais novo indivíduo da nova população "mutante".

A **mutação simples** tem uma ideia bem mais simples que a mutação alternativa. Ela é aplicada pelo método Mutacao.mutacao_simples() que tem acesso privado e é chamado pelo método Mutacao.aplica_mutacao() quando o parâmetro do tipo da mutação é igual a 0 (tipo_mutacao=0).

Para cada um dos genes do cromossomo, o algoritmo gera um número randômico e se esse número for maior que a taxa de mutação, esse gene troca de posição com o próximo gene na sequência do cromossomo. Se a condição de troca for satisfeita para o último gene do

cromossomo, este troca de lugar com o primeiro gene. Feito esse processo para todos os genes, esse cromossomo integrará a nova população mutante e o algoritmo segue para aplicar esse processo no próximo cromossomo da população, até que todos os cromossomos tenham sido iterados.

Exemplificando: dado o cromossomo 10->3->2->4->7. Supondo que o gene da posição 5 tenha gerado um número que satisfez a condição de troca, então o novo cromossomo mutante será: 7->3->2->4->10.

A **mutação inversiva alterada** é aplicada pelo método Mutacao.mutacao_inversiva_alterada() que tem acesso privado e é chamado pelo método Mutacao.aplica_mutacao() quando o parâmetro do tipo da mutação é igual a 2 (tipo_mutacao=2).

O algoritmo dessa mutação segue quase a mesma ideia da mutação simples, mas ao invés do gene trocar de lugar com o próximo gene na sequência do cromossomo quando a condição de troca for satisfeita, o algoritmo gera aleatoriamente uma nova posição para esse gene fazendo com que os outros genes se desloquem para a a esquerda (*shift*) de tal forma a "abrir espaço" para que o gene iterado seja posicionado em seu novo lugar.

Como por exemplo: dado o cromossomo 8->5->6->1->3. Supondo que o gene da posição 2 tenha gerado um número que satisfez a condição de troca e a nova posição gerada foi a posição 4, por tanto, o novo cromossomo mutante será: 8->6->1->5->3.

Assim como no processo de seleção, há a possibilidade do cromossomo com melhor *fitness* seja eliminado no final do processo de mutação. Por tanto, foi implementado a opção de escolher via parâmetro quais são os N cromossomos que não sofrerão mutação. Assim, antes da mutação ser realizada, os N cromossomos com maior *fitness* são separados para, depois de ser gerado a população mutante, serem adicionados a nova população corrente, evitando dessa forma que os N melhores indivíduos não "morram" durante as gerações (detalhes na seção Parâmetros Via Código.

f. Threads

Para otimizar o tempo de processamento das etapas do crossover e de mutação de cada uma das gerações foi implementado o recurso de *multi-threading*. Nessas etapas, o algoritmo implementado na classe br.com.ia.ga.threads.AlgoritmoGeneticoThread, mais especificamente no método get_melhor_caminho(), separa a população em *n* partes iguais

para serem submetidas ao processo corrente (mutação ou crossover) em paralelo, sendo *n* o número de threads define via parâmetro no código (detalhes do parâmetro na seção Parâmetros Via Código).

As duas classes que implementam a classe *thread* são CrossoverThread e MutacaoThread (como explicado na introdução da seção Sobre implementação) recebem em seus construtores os parâmetros necessários para aplicar o crossover e a mutação, respectivamente, e também a instância do grupo de threads a qual vai participar. Cada instância dessas classes recebe uma das partes da população, conduzindo esta para ser submetidas às etapas de crossover e mutação, respectivamente.

O algoritmo separa uma parcela da população, atribui esta para uma *thread* e inicia o processo em *background*, repetindo esses passos para as *n threads* guardando todas elas em um vetor. Todas elas são controladas a partir do grupo de threads (classe ThreadGroup) pela *thread main* que, após todas as *threads* serem iniciadas, fica aguardando todo o grupo ficar desativado, o que indica que a etapa de crossover ou de mutação para aquela geração foi finalizada. Depois que todas foram finalizadas, as populações resultantes de cada uma das *threads* no vetor são "compiladas" em uma única população, se tornando essa a população corrente para a geração.

g. Saída

Para entender melhor as saídas geradas pelo algoritmo produzido, precisamos inicialmente detalhar o funcionamento do método get_melhot_caminho(), localizado na classe AlgoritmoGenetico.java e executado na classe principal Caixeiro.java.

O método recebe diversos parâmetros que são definidos no próximo item (Parâmetros), além da matriz de cidades para ser executado. No início do algoritmo é inicializado uma população aleatória a partir do método gera_populacao_aleatoria (encontrado também na classe AlgoritmoGenetico), tendo o *fitness* de cada cromossomo da população calculado para que se posso obter a diversidade inicial. Em seguida, enquanto o número de gerações rodadas for menor que o número máximo de gerações (dado como parâmetro) e a diversidade da população for maior que a diversidade mínima (também dada como parâmetro) as etapas para encontrar o melhor caminho no problema do caixeiro viajante são realizadas. As etapas consistem em:

- Calcular o *fitness* da nova geração, calculado com o auxílio de uma matriz que possui as distancias entre as cidades;
- Gerar uma nova população com os candidatos a crossover, primeiro uma subpopulação com os melhores indivíduos é criado, depois é executado a seleção dentro da população;
- Aplicação de crossover na nova população selecionada;
- Aplicação de mutação na nova população gerada pelo operador de crossover;
- Definição da nova população depois do operador de mutação como a população que podemos chamar de definitiva;
- O melhor *fitness*, a média dos *fitness* e a diversidade dos indivíduos são obtidos ao final da geração.

A saída do método é o retorno de uma matriz com todos os melhores cromossomos em cada geração executada. A partir do método descrito, as outras saídas explicadas a seguir podem ser visualizadas (via terminal, arquivo e gráfico).

As saídas contidas no terminal durante a execução do programa auxiliam no acompanhamento das gerações em busca do melhor resultado. Primeiro são mostrados os parâmetros inicias e seus respectivos valores. Após tais apresentações, as etapas do algoritmo genético (cálculo do *fitness*, seleção, crossover e mutação) são executadas e seus progressos são informados (se uma determinada etapa foi concluída, mensagens são impressas no terminal) a cada geração rodada, juntamente com o melhor *fitness*, a média da população e sua diversidade. Após o término da última geração, são impressos os parâmetros do algoritmo rodado e o tempo de execução (duração). Por último, o melhor caminho e seu respectivo fitness é gerado.

A saída em arquivo após o término das gerações determinadas nos parâmetros é escrita em formato ".txt" e é incluída no diretório "resultados", sendo possível a sua identificação pela nomeação. O arquivo gerado primeiramente lista todos os parâmetros e seus respectivos valores atribuídos para a execução do programa, incluindo também o tempo de execução para os determinados parâmetros atribuídos. Posteriormente, o melhor caminho obtido nesta execução em específico é mostrado com a ordem dos genes (cidades) que formam o melhor cromossomo e seu respectivo *fitness*. Ao final, são mostrados os melhores caminhos obtidos a cada geração (melhor cromossomo).

Gráficos são gerados para o auxílio das análises, sintonização de parâmetros e verificação de convergência pois o número de gerações pode ser considerado elevado e muito sensível com parâmetros ruins ou problemas na codificação. A cada nova geração, um melhor *fitness* é achado e a média (do *fitness*) da população é calculada para serem plotadas num gráfico em tempo real. Para a geração dos gráficos foi utilizado a biblioteca jchart2d, utilizado na classe Grafico_Dinamico.java (localizado em br.com.ia.ga).

h. Parâmetros

A execução do Algoritmo Genético presente neste exerício-programa depende de 4 parâmetros (incluindo o arquivo de entrada) que são passados via linha de comando e 12 parâmetros que são passados via código. Todos são apresentados e explicados a seguir.

i. Via Linha de Comando

- Taxa de crossover: Um *double* que é guardado em taxa_crossover e define o tamanho do corte (parcela do cromossomo) que fará parte do crossover, ou seja, se a taxa for de 80% (por exemplo) significa que 80% dos genes de um cromossomo irão para um dos filhos e os outros 20% virão do outro pai selecionado para a operação. Por se tratar de uma taxa, esta variável deve receber valores entre 0 e 1, ou seja, 80% é representado por 0,8.
- Taxa de mutação: Trata-se também de um *double* (que é guardado em taxa_mutacao) e pode assumir diferentes funções dependendo do tipo de mutação que será empregado. Na mutação alternativa, a taxa define o trecho de um cromossomo que sofrerá a operação, diferentemente das outras duas mutações (inversiva e simples) onde a taxa define a probabilidade de um gene sofrer a mutação em questão, funcionando como um limiar.
- Operador de seleção: Este pode receber "0" (se o tipo de seleção escolhido for a seleção via roleta) ou "1" (caso seja a escolhido a seleção via torneio) como valores, sendo guardada na variável inteira operador_selecao.
- Arquivo de entrada: O arquivo de entrada é passado via linha de comando, seu nome é armazenado na variável nome_arquivo (*String*) e o arquivo é lido pela classe Manipulador_Arquivo_Entrada.java. Deve-se ressaltar que o arquivo "ncit100.txt" não foi o único arquivo utilizado durante a construção do Algoritmo Genético para o

problema do Caixeiro viajante, portanto, arquivos com números diferentes de cidades podem ser passados para o programa.

ii. Via Código

- Número de *threads*: Este parâmetro é guardado na variável inteira numero_threads e é utilizada na implementação do Algoritmo_GeneticoThread.java. Este algoritmo separa a população tratada em *n* partes iguais para realizar a operação de crossover ou mutação, onde *n* é o número de *threads* passado como parâmetro. Portanto, se o número de *threads* é igual a zero, o melhor caminho para a resolução do problema do caixeiro será obtido apenas pelo main realizando operador (crossover e mutação) sobre a população inteira. Se o número de *threads* é *i,* a classe Caixeiro.java irá utilizar Algoritmo_GeneticoThread.java para a obtenção do melhor caminho, usando *i threads* a mais. Com isso, a população é dividida em *i* partes e cada *thread* opera (aplica a crossover e mutação) em uma das parcelas obtidas. Enquanto isso, a *thread* principal main aguarda as outras terminarem.
- Tamanho da população inicial: Um número inteiro guardado em tamanho_populacao_inicial e define a quantidade de indivíduos da população inicial do algoritmo genético. A população pode ser definida inicialmente, por exemplo, com 200 indivíduos.
- Diversidade mínima: Trata-se de um *double* que guarda em diversidade_minima o valor para a diversidade da população em que se deve parar a operação de crossover.
- Número de gerações máximo: Um inteiro atribuído em numero_geracao_maximo que determina o número máximo de gerações que devem ser rodados no algoritmo. O parâmetro deve receber valores altos em busca do melhor resultado.
- Número de candidatos para crossover: Uma variável do tipo inteiro atribuído em numero_candidatos_crossover. Determina a quantidade de indivíduos (no máximo) que terá a população de candidatos a crossover.
- Quantidade de melhores indivíduos para a subpopulação: Trata-se do atributo inteiro quantidade_subpopulação e determina qual será a quantidade de melhores indivíduos que irá compor a subpopulação de candidatos, ou seja, permanecerão como candidatos.

- Tipo de crossover: Variável inteira (tipo_crossover) que tem como função determinar qual das três implementações do operador de crossover será empregado no algoritmo genético. O parâmetro deve assumir apenas três valores (0, 1 ou 2), onde cada valor indica um tipo de crossover (OX, baseado em posição e baseado em ordem, respectivamente). Essa verificação é feita na classe Crossover.java.
- Pais sobrevivem no crossover: Variável booleana onde a atribuição de "*true*" implica que os pais irão continuar na população pós-crossover (depois da aplicação deste operador), caso seja "*false*", os pais não continuam na população após o crossover.
- Tipo de mutação: Responsável por indicar qual tipo de mutação será utilizado (mesmo caso da variável tipo_crossover). Se for atribuído o valor 0, será executado a mutação simples, caso seja 1, a mutação alternativa e se for escolhido 3, a variação da mutação inversível será rodada.
- Quantidade de não mutantes: Número inteiro atribuído em quantidade_cromossomo_nao_mutantes que determina a quantidade de indivíduos (cromossomos) que não sofrerão mutação.
- Tamanho da população aleatória depois de uma geração: Variável guardada em tamanho_populacao_aleatoria e limita o tamanho que deverá possuir a população aleatório ao final de cada geração.
- Intervalo de tempo para acrescentar população aleatória: Variável inteira atribuída à geracao_populacao_aleatoria. Determina o intervalo de gerações na qual será inserido novos indivíduos na população de forma aleatória, portanto, atribuindo o valor 10 temos a cada dez gerações, novos indivíduos inseridos.

2. Análises

As análises a seguir são baseadas em um plano de teste que visou alterar os parâmetros de tal forma que no final se conseguisse um solução ótima. Portanto, não foram feito testes com algumas combinações de parâmetros por não haver necessidade, sabendo que tais combinações não agregariam valor à busca do melhor resultado.

Cada uma das combinações de parâmetros foram testadas entre 5 à 10 vezes, já que o fator de aleatoriedade está presente em várias partes do algoritmo genético aqui implementado. Juntamente à este documento, acompanha três evidências dos melhores resultados de cada combinação:

- a primeira evidência é um arquivo texto com os parâmetros da combinação, o melhor cromossomo conseguido e o seu *fitness*;
- a segunda é uma imagem com alguns valores de melhor *fitness*, média dos *fitness* e a diversidade das últimas gerações executadas; e
- a terceira evidência é uma imagem com o gráfico gerado dinamicamente durante a execução, onde o eixo x representa as épocas executadas e o eixo y *fitness* em micro-unidades (obs.: infelizmente, o eixo y não tem a indicação do *range* dos valores e por conta de tempo e viabilidade, não foi possível modificar o pacote *Chart2D*, usado para a geração do gráfico dinâmico). A função em vermelho representa a evolução do melhor *fitness* e a em azul, representa a média do *fitness* da população.

Dentro do diretório "/resultados", as evidências estão organizadas em diretórios que foram nomeados pela intenção do que se queria testar no momento e com um número que representa a ordem no plano de testes. A melhor combinação de parâmetros para curto prazo (Dez mil gerações) e para longo prazo (Cem mil gerações) estão separadas em diretórios sem número de identificação e são cópias de evidências de algum dos testes realizados. No total, o plano de teste tentou "envolver" 11 (onze) parâmetros:

- Tipo de crossover;
- Tipo de mutação;
- Tipo de seleção;
- Tamanho da população inicial (mudando assim o número de candidatos a crossover);
- Tamanho da subpopulação que não sofre seleção;
- Número de cromossomos que não sofrem mutação (a segunda população que não é submetida a etapa de mutação);
- Os cromossomos pais sobrevivem ou não á etapa de crossover;
- Número de candidatos (selecionados) para sofrerem crossover;
- Com ou sem as subpopulações;
- Taxa de crossover; e
- Taxa de mutação.

Mais parâmetros poderiam ser explorados, como o acréscimo de uma população aleatória durante as épocas e o intervalo com que ela seria incluída, ou até mesmo a mudança de cada um desses parâmetros durante as épocas e a implementação de um meta-GA (que não foi implementado, e que seria uma algoritmo genético para gerar a melhor combinação de

parâmetros), mas por conta de tempo, essas combinações de parâmetros não puderam ser feitas.

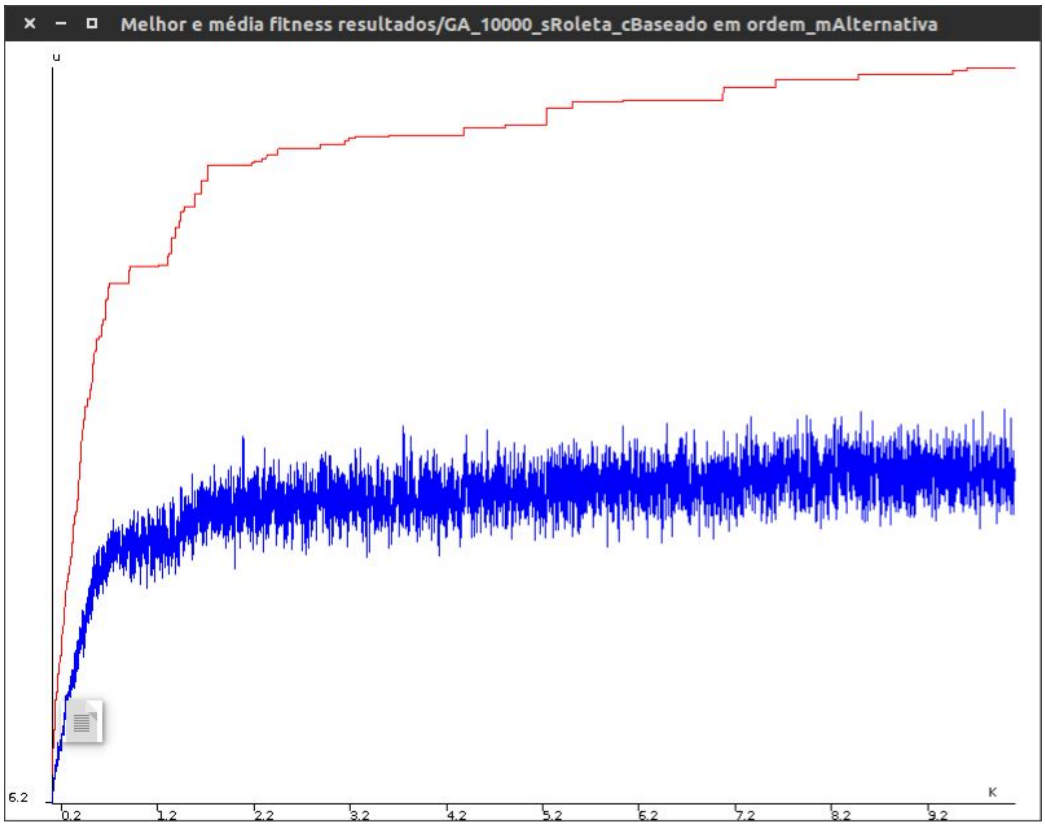
a. Em Busca do Melhor Fitness

i. Tipos de Crossover

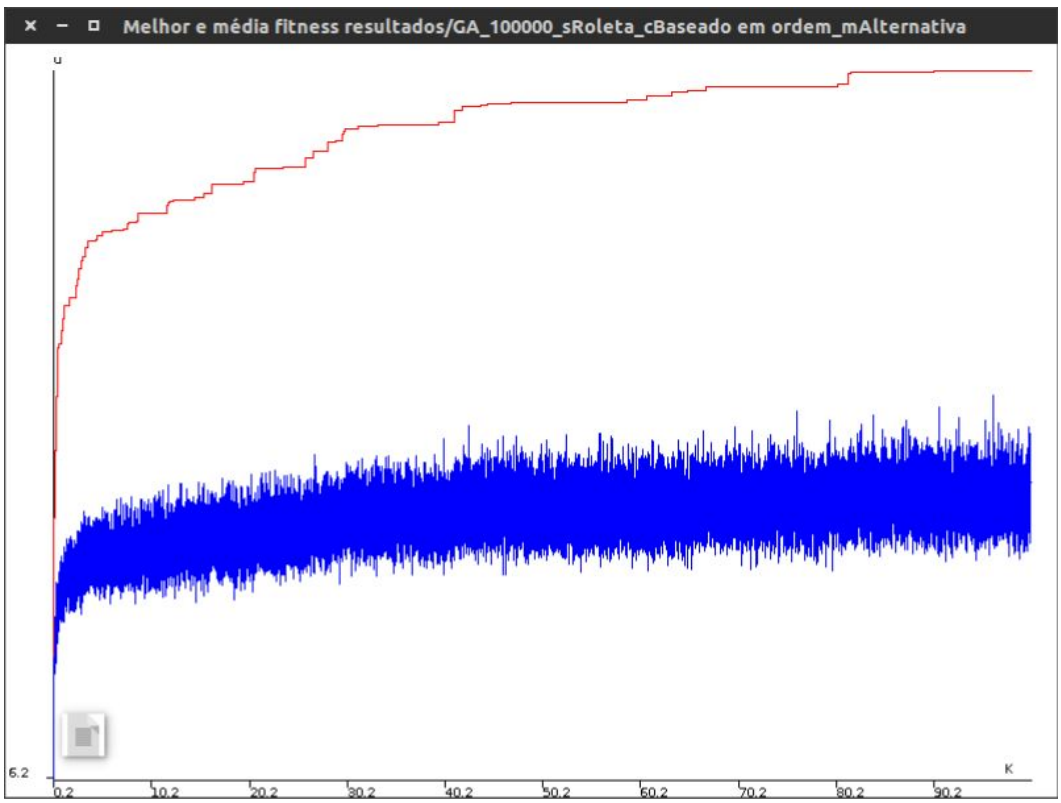
Os primeiros testes foram realizados em função do tipo de crossover que seria utilizado. Utilizamos uma taxa de crossover igual a 80%, uma taxa de mutação 20%, usamos a seleção via Roleta e a mutação Alternativa. A população ficou dividida em: 100 cromossomos para a população inicial, 100 candidatos a crossover, 10 na subpopulação, e 5 cromossomos não mutantes (segunda subpopulação). Para cada um dos testes de tipo de crossover, usamos 3 *threads* e testamos tanto para 10000 (dez mil) e 100000 (cem mil) gerações. As evidências desses testes estão em /resultados/"1 Desempenho_Crossovers_g10000_mAlternativa" e /resultados/"1 Desempenho_Crossovers _g100000_mAlternativa".

O melhor *fitness* adquirido com crossover baseado em ordem em 10,000 gerações foi aproximadamente 3.1881E-5 e em 100,000 gerações o melhor resultado foi 2.741E-5. Com crossover baseado em posição em 10,000 gerações, conseguiu-se aproximadamente 3.2553E-5 e em 100,000 gerações o melhor resultado foi 3.9132E-5. Dentre os três tipos, o crossover OX teve um resultado melhor, tanto em 10,000 gerações quanto para 100,000 gerações: 4,1796E-5 e 4,1405E-5, respectivamente.

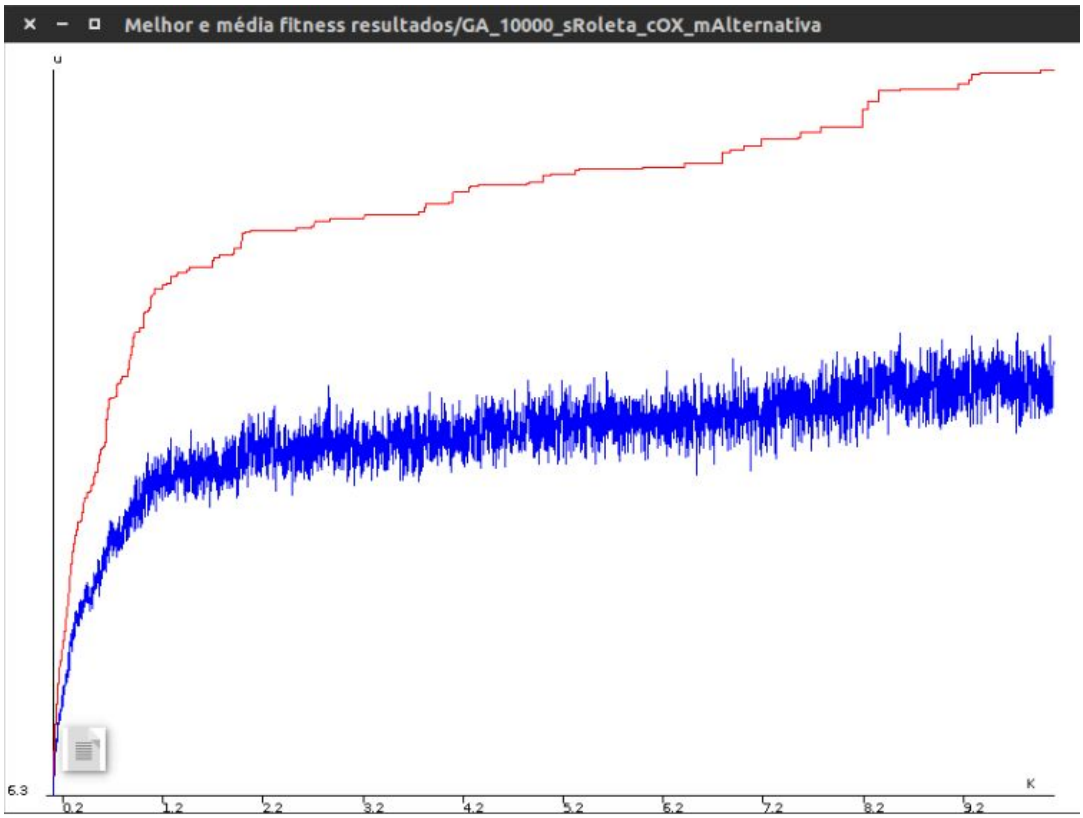
Os resultados mostram que uma população inicial boa pode fazer toda a diferença, como observamos nos testes de 10,000 gerações do crossover baseado em ordem e OX. Ambos tiveram resultados melhores no teste de "curto prazo". Nas figuras 2.a.i.1 e 2.a.i.2 (para baseado em ordem), 2.a.i.3 e 2.a.i.4 (para OX) mostram mesmo com uma boa evolução nas primeiras 10 mil gerações no teste a longo prazo, não se obteve um melhor *fitness*.



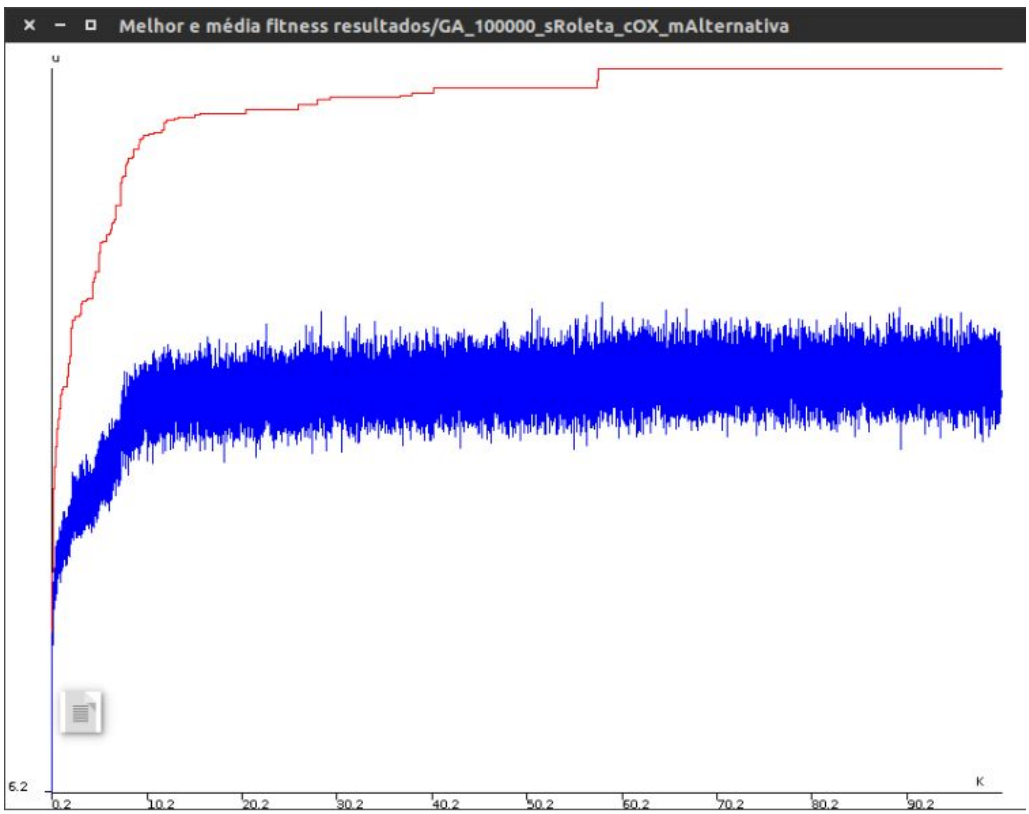
2.a.i.1 - Gráfico do Crossover Baseado em Ordem com 10,000 gerações



2.a.i.2 - Gráfico do Crossover Baseado em Ordem com 100,000 gerações



2.a.i.3 - Gráfico do Crossover OX com 10,000 gerações



2.a.i.4 - Gráfico do Crossover X com 100,000 gerações

Entre o crossover baseado em ordem e o crossover OX, vemos que diversidade da população foi maior no crossover que mostrou os melhores resultados, variando entre 1,45E-5 e 1,64E-5 nas últimas gerações, enquanto o crossover baseado em ordem variou de 1,20E-5 à 1,28E-5. As médias, porém, foram bem mais elevadas no OX: 2,72E-5 e 2,47E-5 contra 1,45E-5 e 1,52E-5 do baseado em ordem.

Dentre os três tipos de crossover, o que mais demorou para executar o teste de 100 mil gerações foi o crossover baseado em ordem (2165 seg.), seguido do crossover OX (2144 seg.) e depois pelo crossover baseado em posição (2127 seg.).

Portanto, elegendo o crossover OX como o melhor tipo de crossover devemos estar ciente de que, apesar de obter o caminho com melhor resultado, ao final da execução teremos uma maior diversidade, uma média variando muito durante as gerações e não teremos o algoritmos mais rápido dentre os três crossover analisados aqui.

ii. Tipos de Mutação

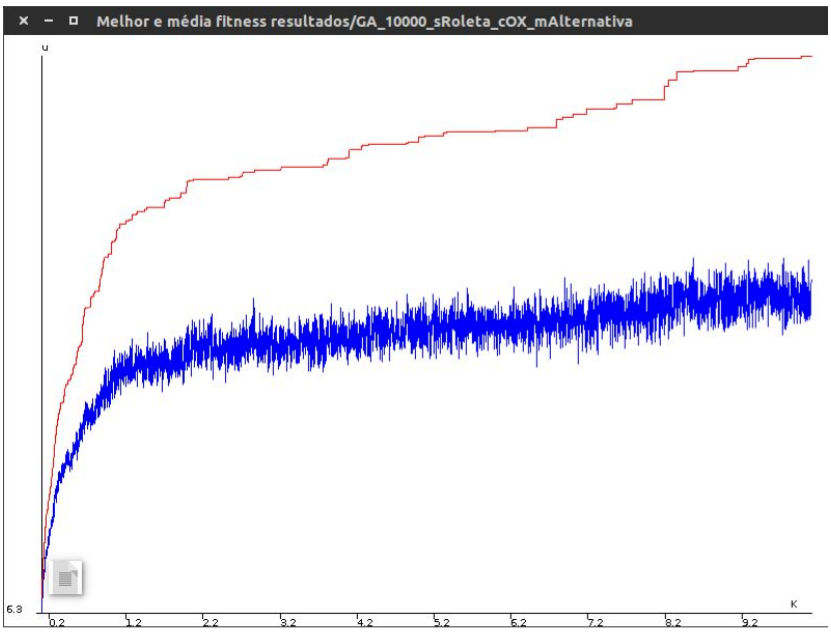
Após definido qual tipo de crossover utilizar, testamos os tipos de mutação. Mantendo as mesmas taxas de crossover e mutação, 80% e 20%, usamos a seleção via Roleta agora fixando o crossover OX. A população da mesma maneira também: 100 cromossomos para a população inicial, 100 candidatos a crossover, 10 na subpopulação, e 5 cromossomos não mutantes (segunda subpopulação). Para cada um dos testes de tipo de mutação, usamos 3 *threads*, fazendo testes de curto prazo, 10000 (dez mil), e longo prazo, 100000 (cem mil) gerações. As evidências desses testes estão em /resultados/"1 Desempenho_Mutacao_g10000 _cOX" e /resultados/"1 Desempenho_Mutacao _g100000_cOX".

O melhor *fitness* adquirido com a mutação alternativa em 10,000 gerações foi 4,1796E-5 e em 100,000 gerações o melhor resultado foi 4,1405E-5, como observado nos testes variando os tipos de crossover. Com a mutação inversível, rodando 10,000 gerações, conseguiu-se aproximadamente 1,9601E-5 e em 100,000 gerações o melhor resultado foi 2,393E-5. A mutação simples atingiu o pior resultado a curto prazo, 1,9280E-6, mas superou o melhor *fitness* da mutação inversível, a longo prazo: 2,5608E-6.

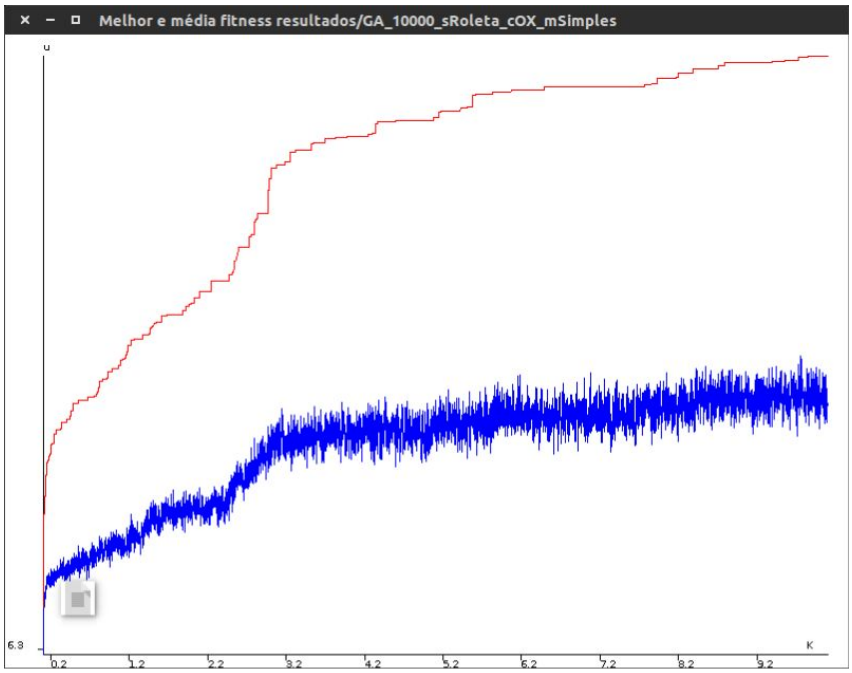
Além dos melhores *fitness*, podemos observar como a média e a diversidade evoluíram diferente em cada uma das mutações (figuras 2.a.ii.1, 2.a.ii.2 e 2.a.ii.3). Na mutação inversível, a média custa a acompanhar o mesmo crescimento do melhor *fitness*, ao contrário do que acontece nas outras duas mutações, onde a média tem uma aceleração bem

mais próxima da aceleração observada no melhor *fitness*. Essa diferença chama atenção principalmente nas primeiras 1000 gerações, onde cromossomos com melhores *fitness* são mais fáceis para conseguir.

Como a mutação inversível e a mutação simples atingiram um melhor *fitness* bem similar um a outro para o teste a curto prazo, podemos nos atentar a diferença da média e da diversidade entre os dois nas últimas gerações. Enquanto a média na mutação inversível variou entre 0,794E-5 e 0,886E-5, na mutação simples, a média flutuou de 1,153E-5 à 1,944E-5.

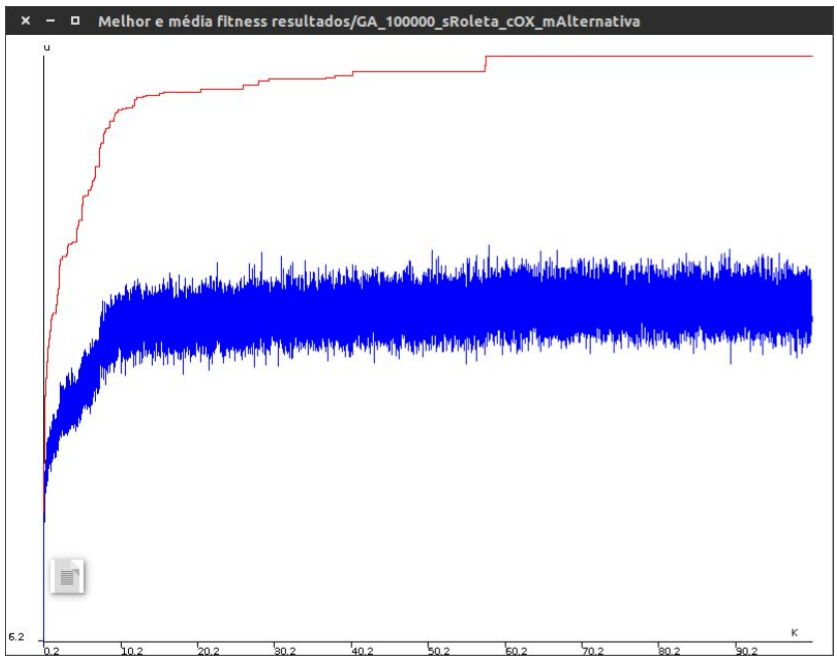


2.a.ii.1 - Gráfico da Mutação Alternativa em 10 mil gerações

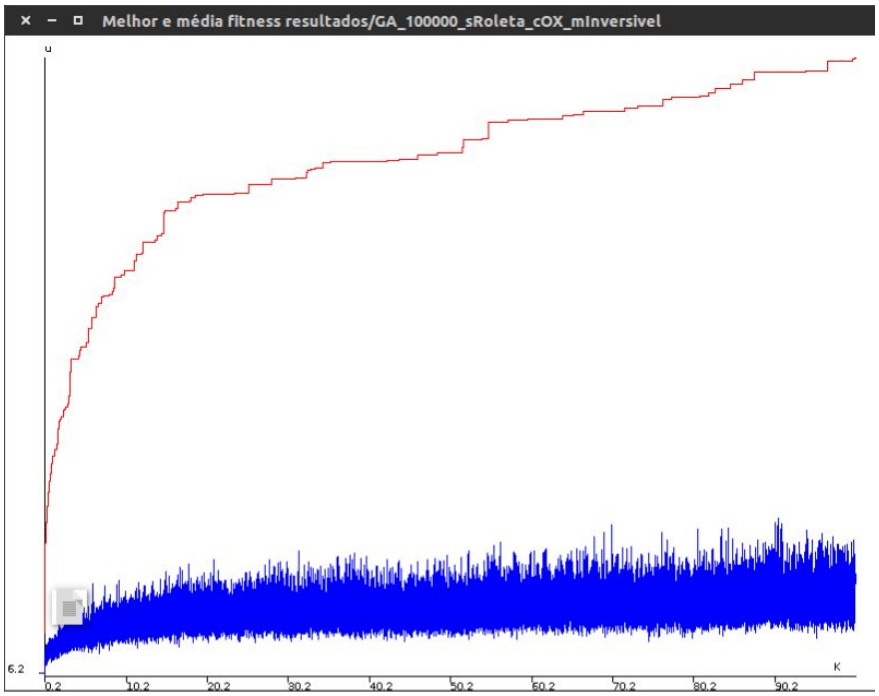


2.a.ii.3 - Gráfico da Mutação Simples em 10 mil gerações

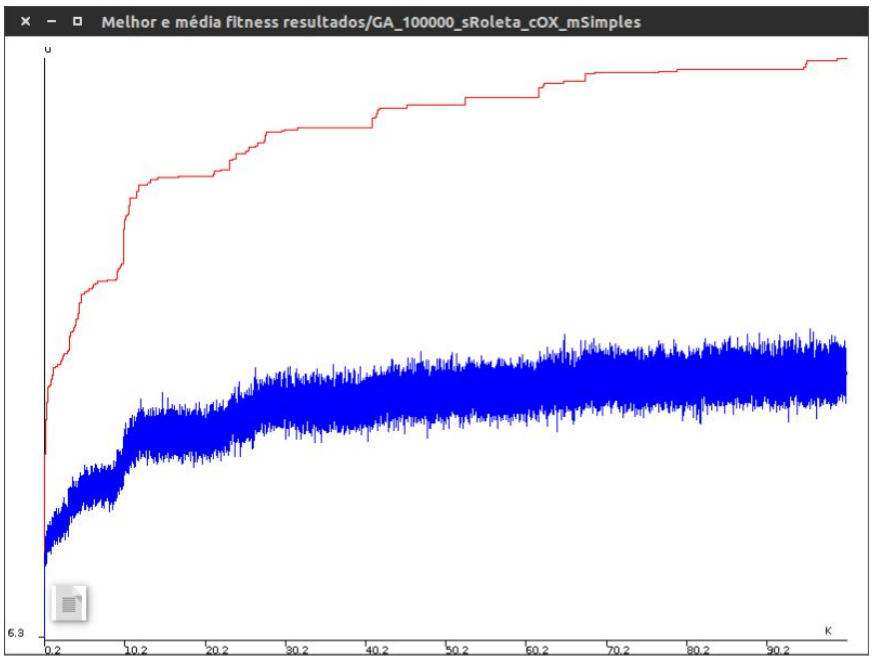
Assim como aconteceu no teste de desempenho dos tipos de crossover, a melhor mutação teve uma boa evolução nas primeiras gerações, mas melhorou pouquíssimas vezes após as 30,000ª geração, ao contrário dos outros tipos de mutação. Enquanto a mutação alternativa não teve nenhuma melhora nas últimas 30 mil gerações, a mutação inversível e a mutação continuaram evoluindo, sendo que essa última, mais timidamente (figuras 2.a.ii.4, 2.a.ii.5 e 2.a.ii.6)



2.a.ii.4 - Gráfico da Mutação Alternativa em 100 mil gerações



2.a.ii.5 - Gráfico da Mutação Inversível em 100 mil gerações



2.a.ii.6 - Gráfico da Mutação Simples em 100 mil gerações

Além de chegar no melhor resultado, a mutação alternativa teve o melhor tempo na execução (125 seg.). A mutação inversiva teve o pior tempo: 148 segundos, já a mutação simples atingiu 129 segundos.

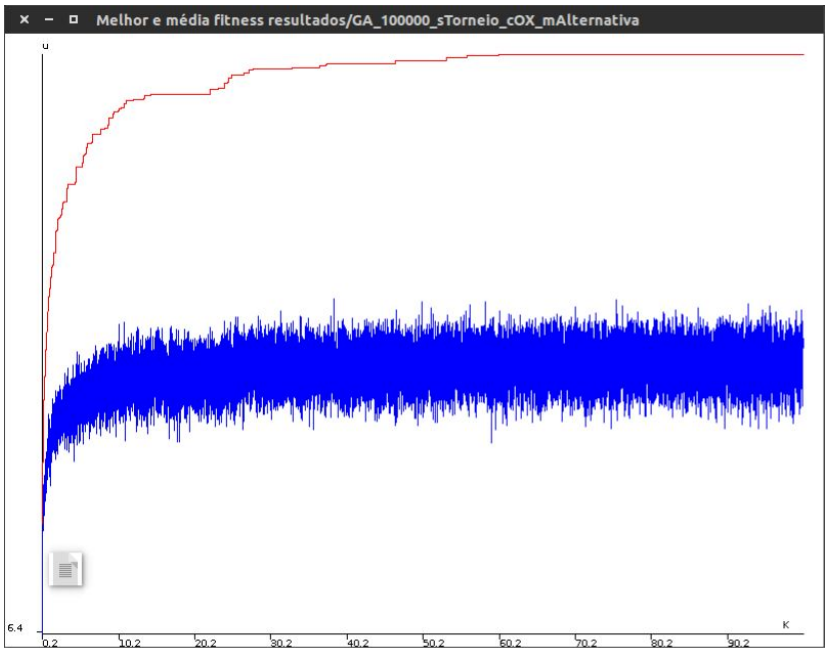
Por atingir o melhor *fitness*, a mutação alternativa foi escolhida como o parâmetro a ser mantido nas próximas combinações, assim como o crossover OX.

iii. Tipos de Seleção

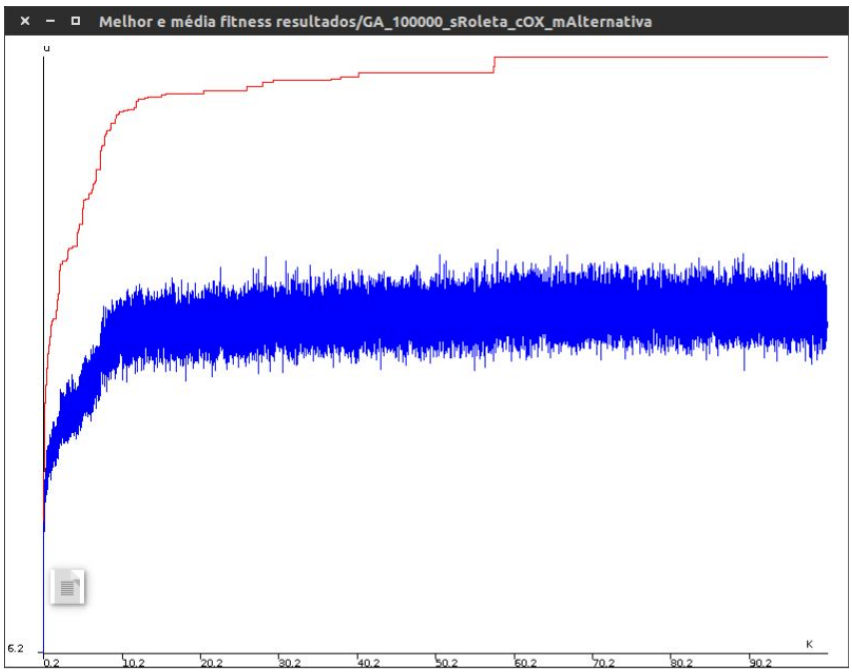
A combinação crossover OX e mutação alternativa foi testada usando os dois diferentes tipos de seleção: roleta e torneio. Mantendo as mesmas taxas de crossover e mutação, 80% e 20% respectivamente, e também a população: 100 cromossomos para a população inicial, 100 candidatos a crossover, 10 na subpopulação, e 5 cromossomos não mutantes (segunda subpopulação). Para cada um dos testes de tipo de seleção, usamos 3 *threads*, fazendo testes de curto prazo, 10000 (dez mil), e longo prazo, 100000 (cem mil) gerações. As evidências desses testes estão em /resultados/"1 Desempenho_Selecao_g10000_cOX_mAlternativa" e /resultados/"1 Desempenho_Selecao_g100000_cOX_mAlternativa".

A Seleção via Roleta atingiu, como já vimos nas analises de crossover e mutação, em 10,000 gerações foi 4,1796E-5 e em 100,000 gerações o melhor resultado foi 4,1405E-5. Com a seleção via Torneio não chegamos aos mesmo resultados: a curto prazo atingimos 3,8683E-5 e a longo prazo 3,73409E-6. Ambos os testes de a longo prazo mostraram problemas com a população gerada inicialmente, e por isso, ficaram a baixo aos testes de curto prazo.

Em questão de evolução nas últimas gerações, as duas seleções não se diferem muito, já que ambas não conseguiram melhores *fitness* nas últimas 30 mil gerações. O fato interessante está nos números da diversidade, nessas últimas gerações: embora, a seleção via roleta tenha conseguido o melhor resultado, a seleção via torneio tem uma diversidade maior. Ou seja, ao contrário do que aconteceu nas análises de crossover e mutação, o tipo que obteve o melhor resultado conseguiu manter uma diversidade menor comparado aos outros tipos.



2.a.iii.1 - Gráfico da Seleção via Torneio em 100 mil gerações



2.a.iii.2 - Gráfico da Seleção via Roleta em 100 mil gerações

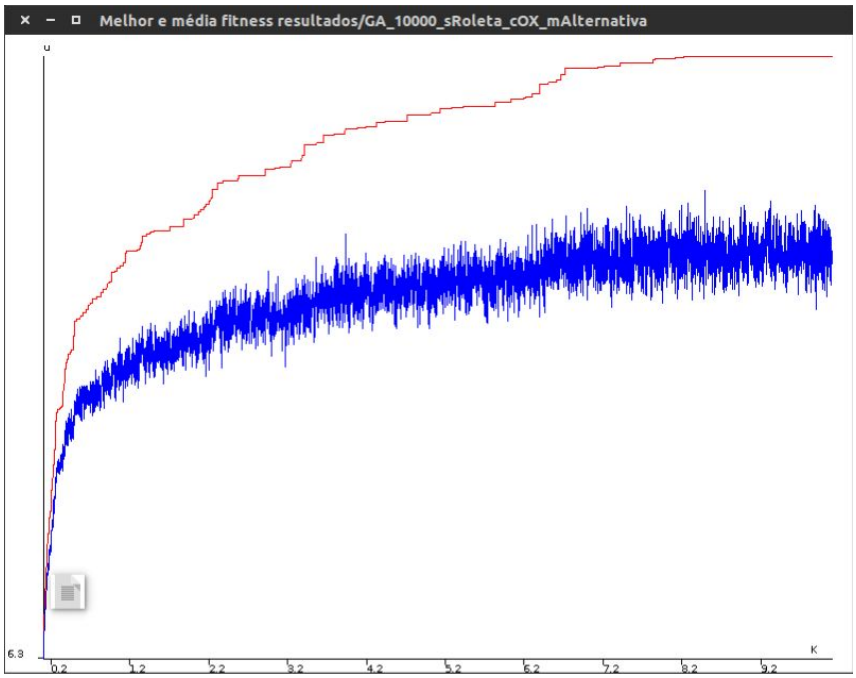
iv. Tamanho da População Inicial

Cada uma das operadores para cada etapa do processo de gerar uma nova geração já foi definido para se atingir a melhor solução: crossover OX, mutação alternativa e seleção via Roleta. Agora, os aspectos da população devem ser definidos. Até agora estávamos usando

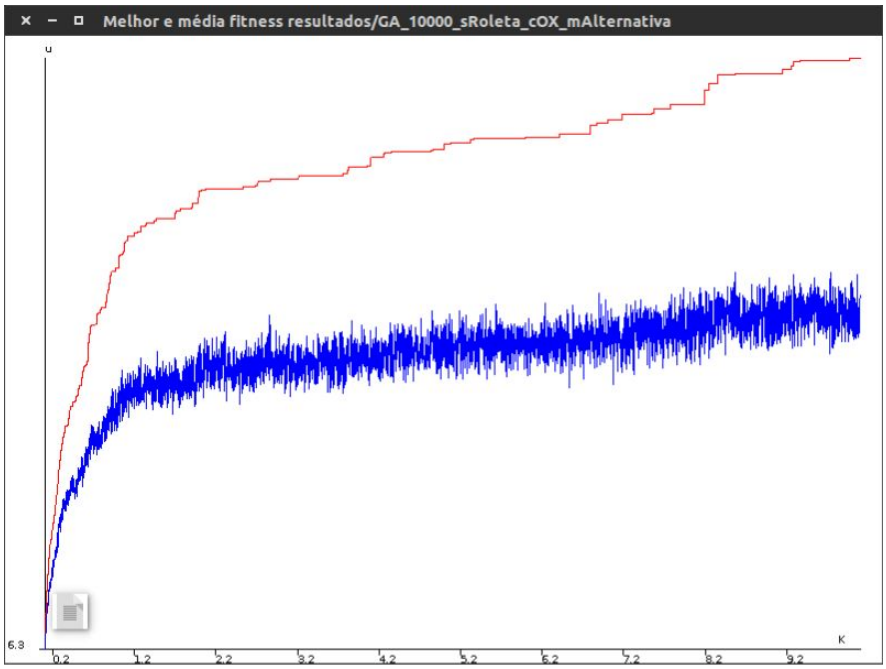
como população: 100 cromossomos para a população inicial, 100 candidatos a crossover, 10 na subpopulação, e 5 cromossomos não mutantes (segunda subpopulação).

Primeiro alteramos a quantidade de cromossomos que integram a população inicial (mudando o número de candidatos para ser igual a população inicial também). Fizemos testes com a população inicial valendo 50, 100, 150 e 200. Para cada um desses valores de população inicial usamos 3 *threads* e fazendo testes de curto prazo, 10000 (dez mil), e longo prazo, 100000 (cem mil) gerações. As evidências desses testes estão nos diretórios de 4 à 9.

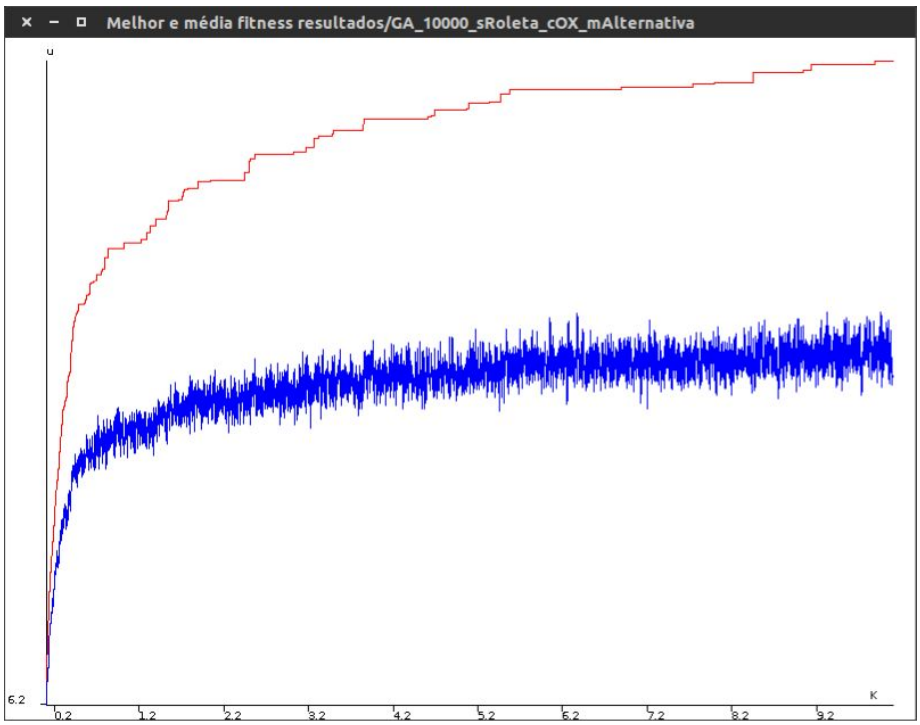
No curto prazo, nenhuma dessas mudanças resultaram em melhores a valores de *fitness*, mas observou-se que quanto maior a população inicial e candidatos a crossover, maior é a diversidade. Quando mantemos a população inicial como 50 a diversidade nas últimas gerações varia entre 0,975E-5 e 1,216E-5, mas quando aumentamos para 200, a diversidade varia de 1,639E-5 à 1,7933E-5. Evidentemente, o tempo de execução aumenta também, conforme aumentamos o tamanho dessa população.



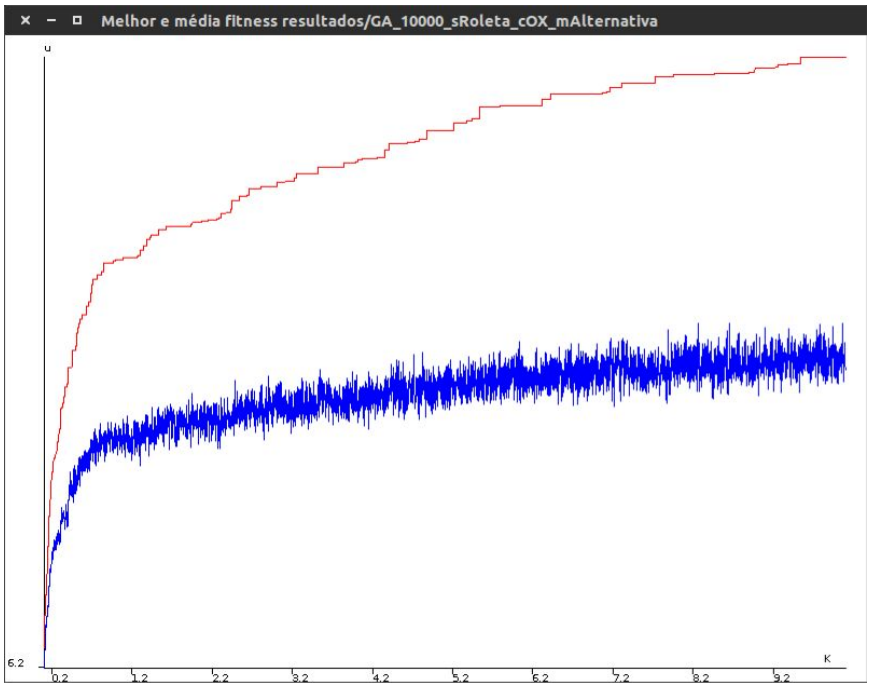
2.a.iv.1 - Gráfico População Inicial 50 em 10 mil gerações



2.a.iv.2 - Gráfico População Inicial 100 em 10 mil gerações

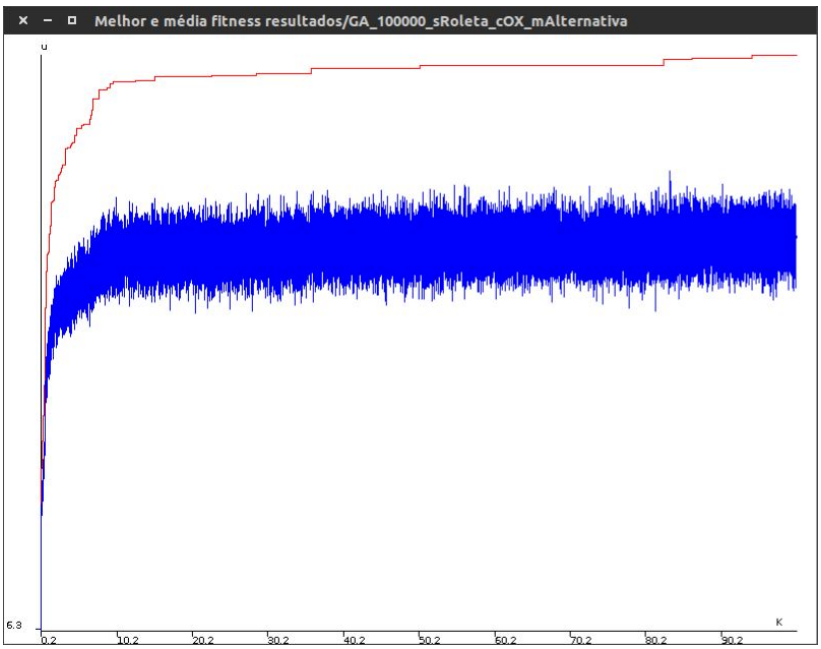


2.a.iv.3 - Gráfico População Inicial 150 em 10 mil gerações

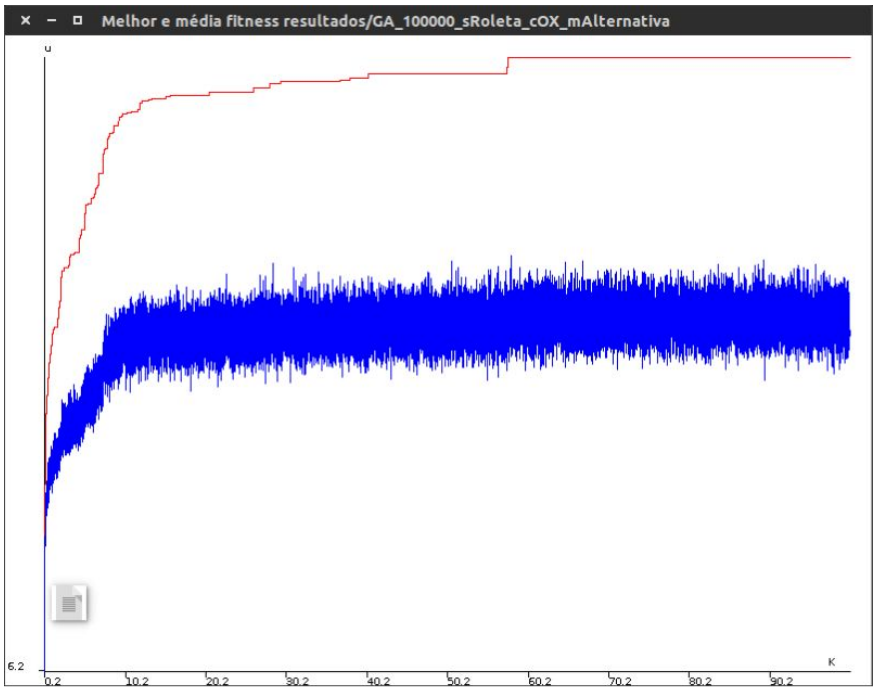


2.a.iv.4 - Gráfico População Inicial 200 em 10 mil gerações

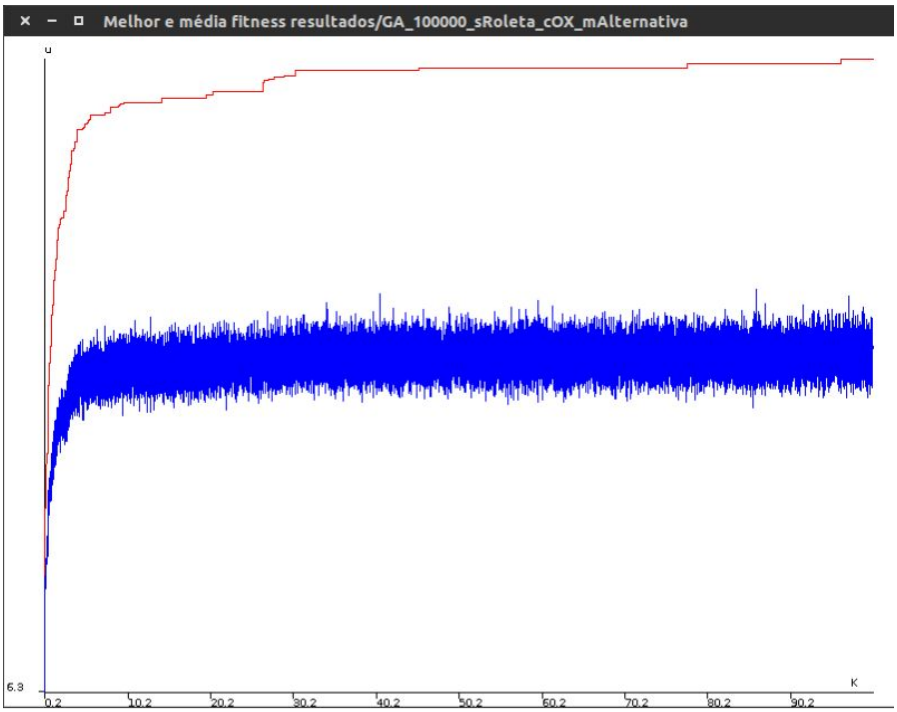
Já a longo prazo, com a população inicial de tamanho 150 conseguimos atingir um *fitness* de 4,2164E-5 e pudemos observar que com o aumento da população inicial, melhores *fitness* deixam de ser gerados mais cedo. Para uma população inicial de 50 cromossomos, nem nas últimas 10 mil gerações melhores *fitness* deixaram de ser atingidos, mas para uma população de 200 cromossomos iniciais, a partir da 30,000ª geração, melhores caminhos não foram mais gerados.



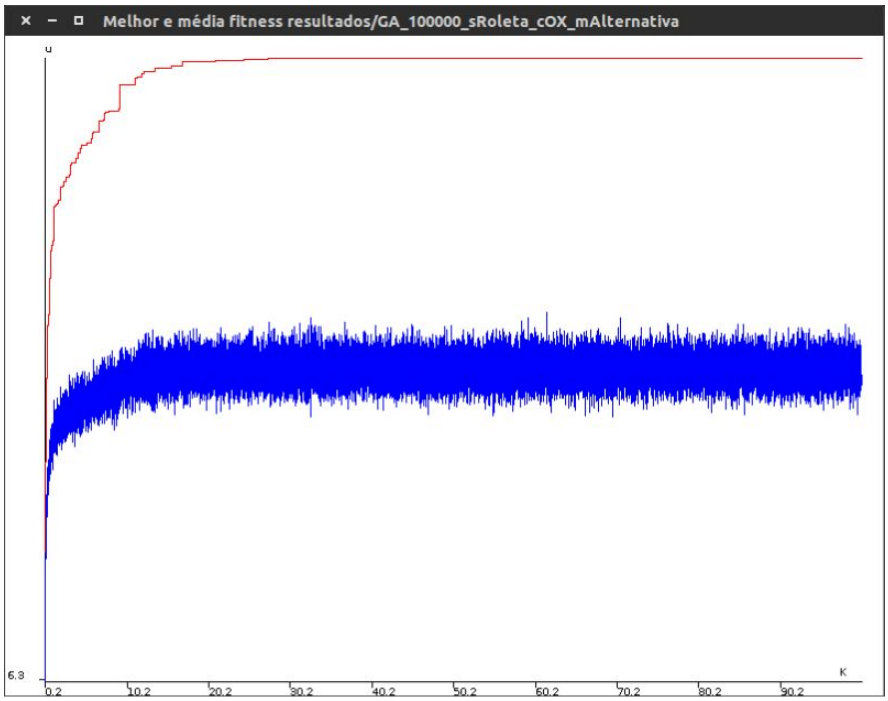
2.a.iv.5 - Gráfico População Inicial 50 em 100 mil gerações



2.a.iv.6 - Gráfico População Inicial 100 em 100 mil gerações



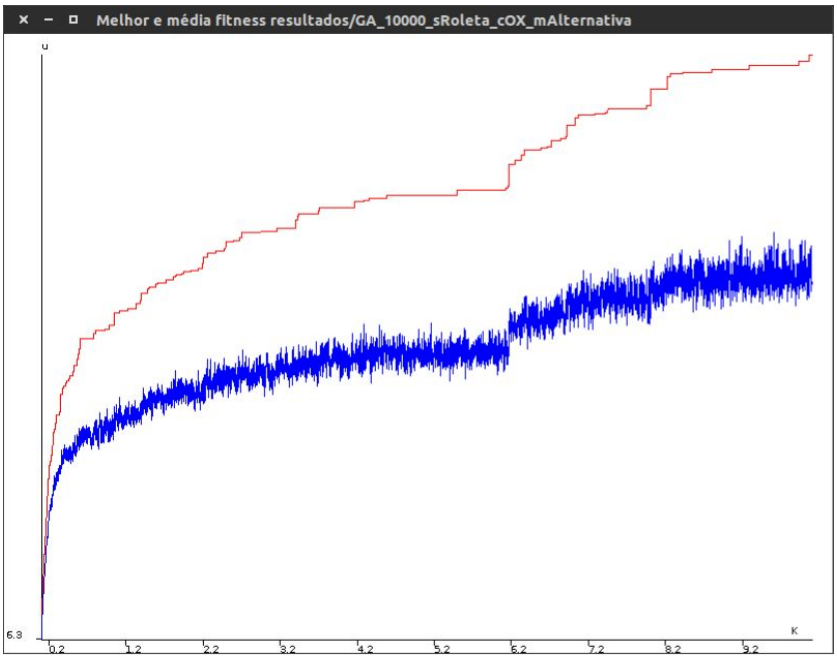
2.a.iv.7 - Gráfico População Inicial 150 em 100 mil gerações



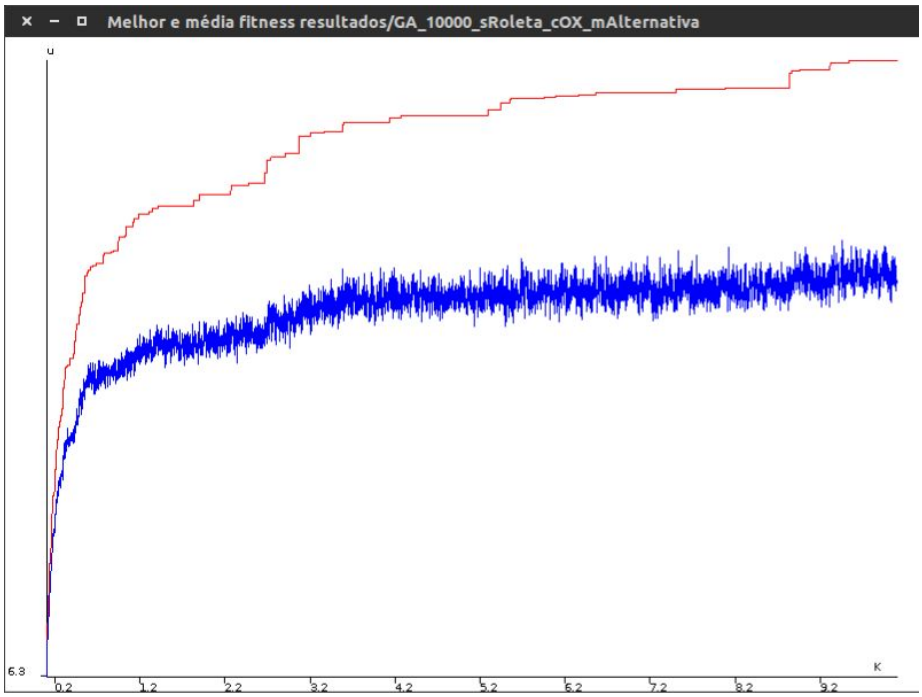
2.a.iv.8 - Gráfico População Inicial 200 em 100 mil gerações

Ao invés de mudar a quantidade da população inicial, mudamos a quantidade da subpopulação de 10 cromossomos para: 1, 10, 25, 40 e 50. Nos testes a curto prazo, conseguimos um melhor *fitness* ao manter 100 como população inicial e trocando para 25 o número de cromossomos que farão parte da subpopulação: 4,22753E-5, melhor até que o resultado adquirido a longo prazo no teste com 150 indivíduos na população inicial e 10 não mutantes.

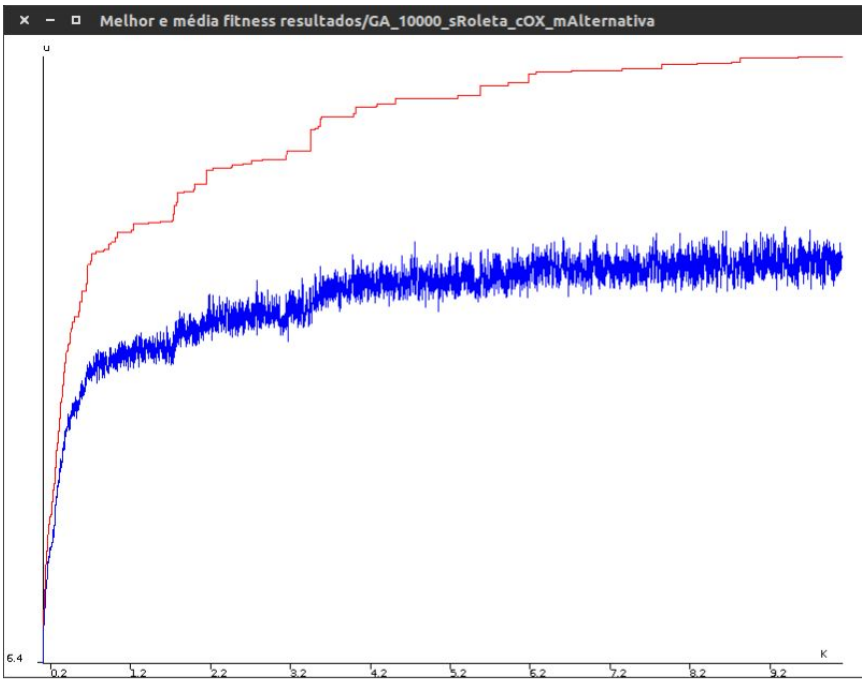
Aumentando essa subpopulação, observa-se que a diversidade vai diminuindo, já que cada vez mais melhores cromossomos são “salvos” da seleção e são garantidos para serem submetidos ao crossover.



2.a.iv.9 - Gráfico da Subpopulação de 25 em 10 mil gerações

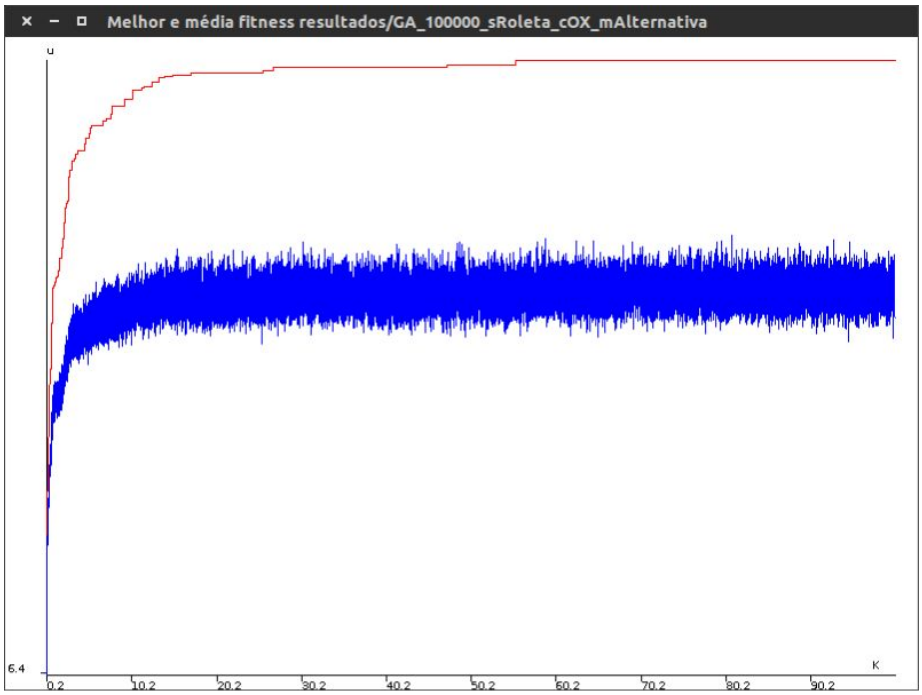


2.a.iv.10 - Gráfico da Subpopulação de 40 em 10 mil gerações



2.a.iv.11 - Gráfico da Subpopulação de 50 em 10 mil gerações

Essa combinação de população inicial e de subpopulação à longo prazo, conseguimos um resultado ainda melhor: 4,42796E-5 como melhor *fitness*. Apesar do teste ter sido feito para 100 mil gerações, esse valor de *fitness* foi atingido antes da 60000ª geração. Esse resultado foi atingido 8 minutos mais rápido do que o teste de longo prazo que tinha atingido o melhor *fitness*.

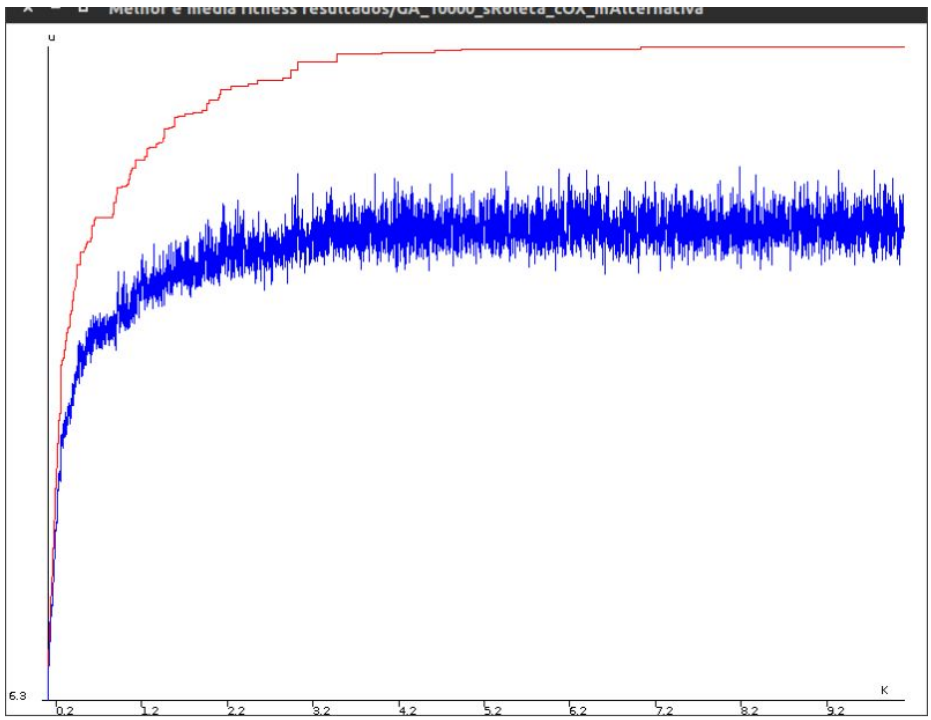


2.a.iv.12 - Gráfico da Subpopulação de 25 em 100 mil gerações

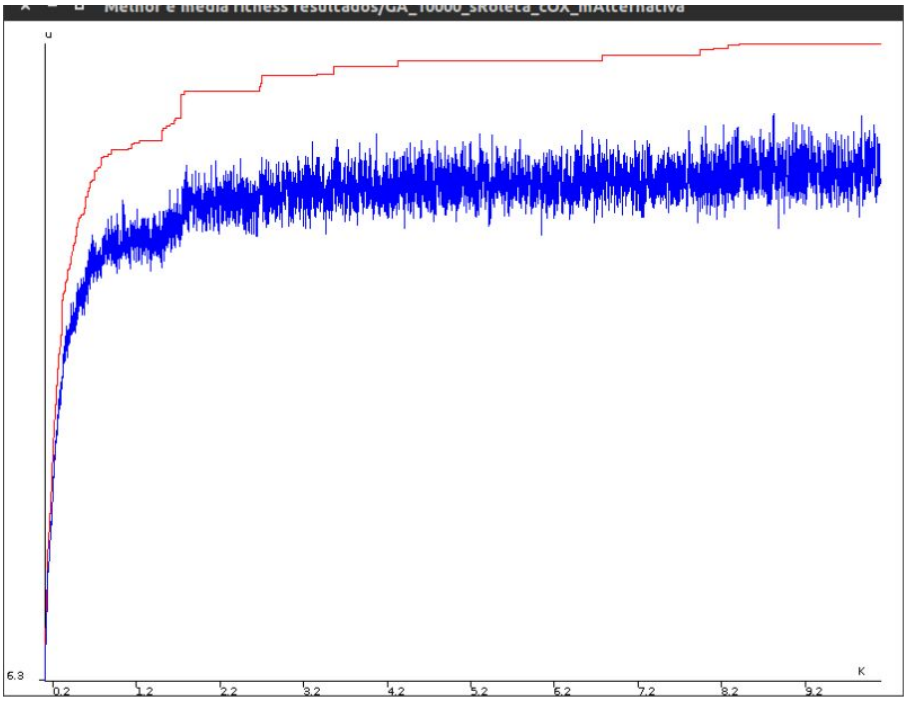
Depois de alterar a subpopulação, mudamos os valores da segunda subpopulação que não passa pela etapa de mutação, salvando assim, mais uma vez, os melhores cromossomos. Essa população de não mutantes era de 5 cromossomos nas combinações que chegamos ao melhores valores à curto e a longo prazo. Alteramos esse número para 1, 5, 10, 15, 20, 25 e 50. Quando alteramos de 5 para 20, alcançamos o melhor resultado do *fitness* a curto prazo: 4,341E-5.

Assim como quando alteramos o tamanho da primeira população, observamos que aumentando o número de cromossomos não mutantes, a diversidade diminui. Quando alteramos para 1 cromossomos essa população, a diversidade varia de 1.24E-5 à 1.41E-5, já quando aumentamos para 50 cromossomos a população varia entre 0,552E-5 e 0,78E-6.

Além da diversidade, podemos notar mesmo a curto prazo que quanto mais aumentamos a população não mutante é cada vez mais difícil gerar cromossomos com melhores *fitness* com o avançar das gerações. Quando essa população tem tamanho 15, já observamos que os "saltos" da função do melhor *fitness* são bem menores e frequentes nas últimas 3 mil gerações do que quando essa população tem tamanho 1.

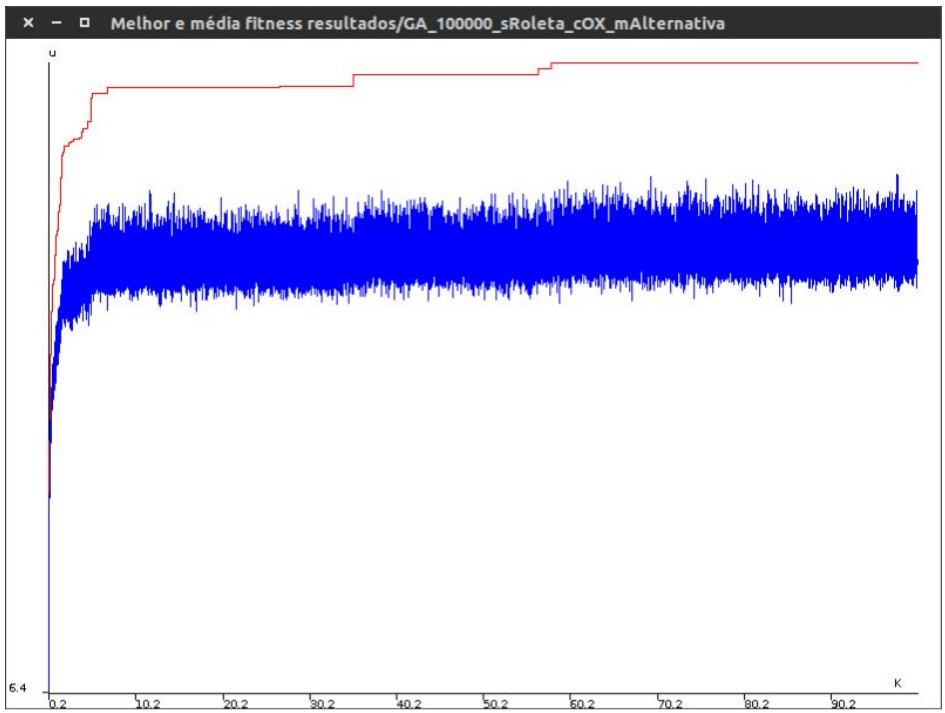


2.a.iv.13 - Gráfico de Não Mutantes 20 em 10 mil gerações



2.a.iv.14 - Gráfico de Não Mutantes 50 em 10 mil gerações

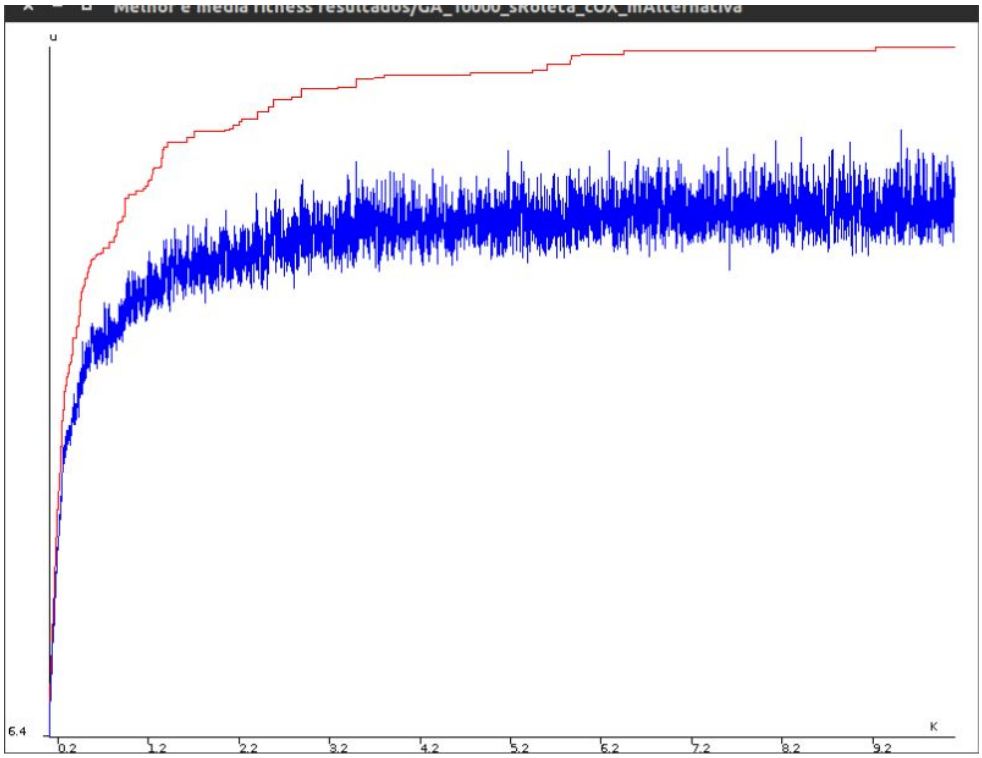
A longo prazo, a combinação dos melhores parâmetros até o momento da análise com 25 cromossomos na população não mutante gerou uma caminho com *fitness* 4,6616E-5, o melhor *fitness* até o momento, com média variando de 3,37E-5 à 3,80E-6 nas últimas gerações. Assim como no teste a longo prazo anterior, esse teste atingiu o melhor *fitness* antes de precisar gerar as 30 mil últimas gerações.



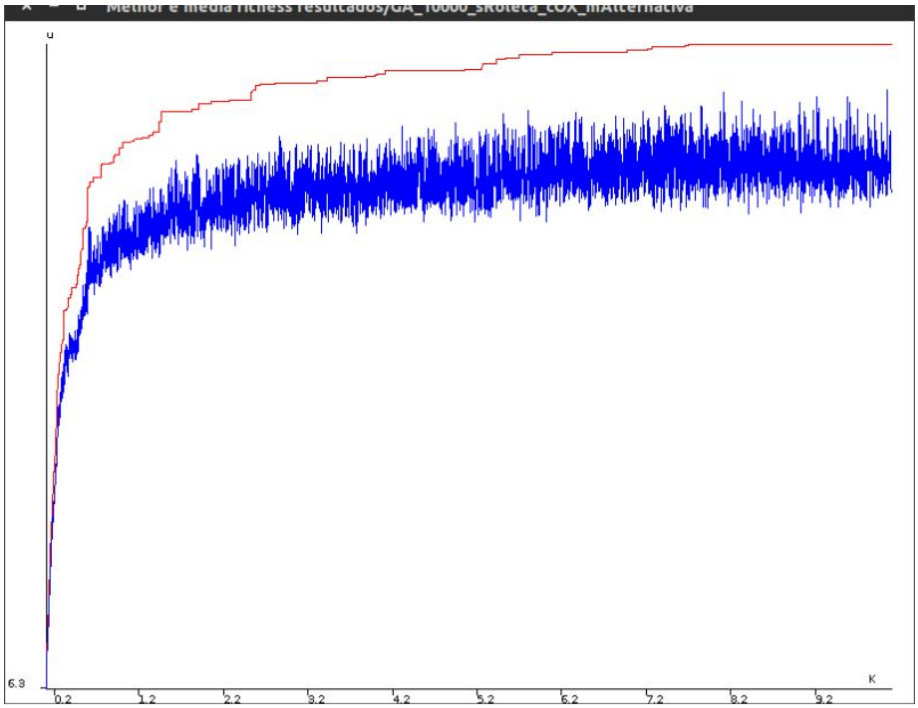
2.a.iv.15 - Gráfico de Não Mutantes 20 em 100 mil gerações

Agora, mudando a quantidade de cromossomos selecionados pela roleta (candidatos a crossover), conseguimos atingir o melhor resultado a curto prazo: 4,443E-5. O caminho com esse *fitness* foi alcançado mudando o número candidatos de 100 para 50, ou seja, metade da população inicial. Diminuindo 50 cromossomos da população corrente, diminuímos quase pela metade o tempo de execução: antes, conseguimos 4,341E-5 com 132 segundos, agora alcançamos 4,443E-5 com 75 segundos.

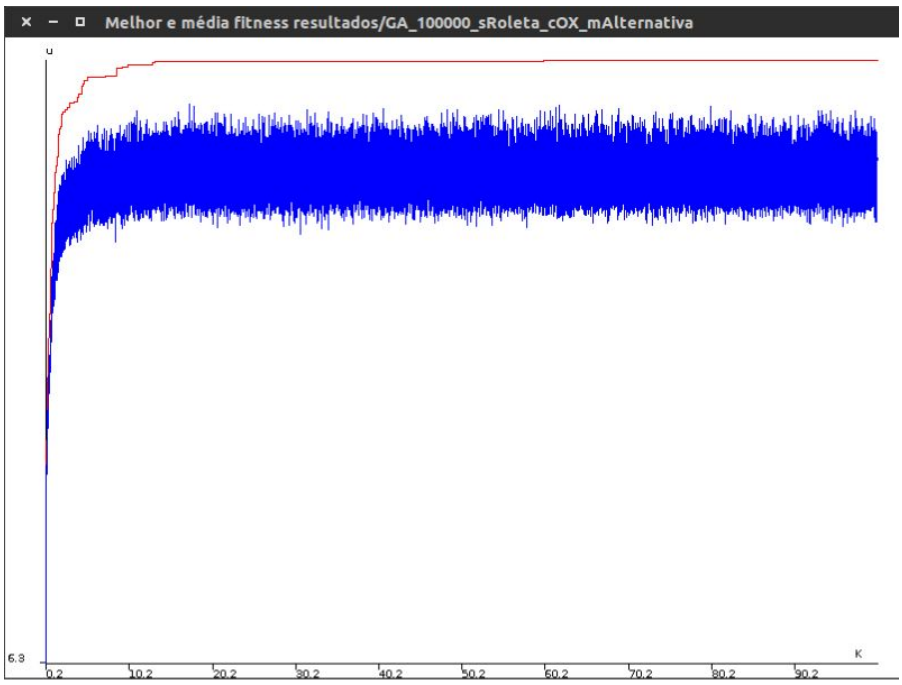
Podemos notar que conforme diminuímos a quantidade de candidatos, menor é a diversidade, que pelo gráfico dos testes podemos observar a distância entre a função do melhor *fitness* e a média da população.



2.a.iv.16 - Gráfico de Candidatos 75 em 10 mil gerações

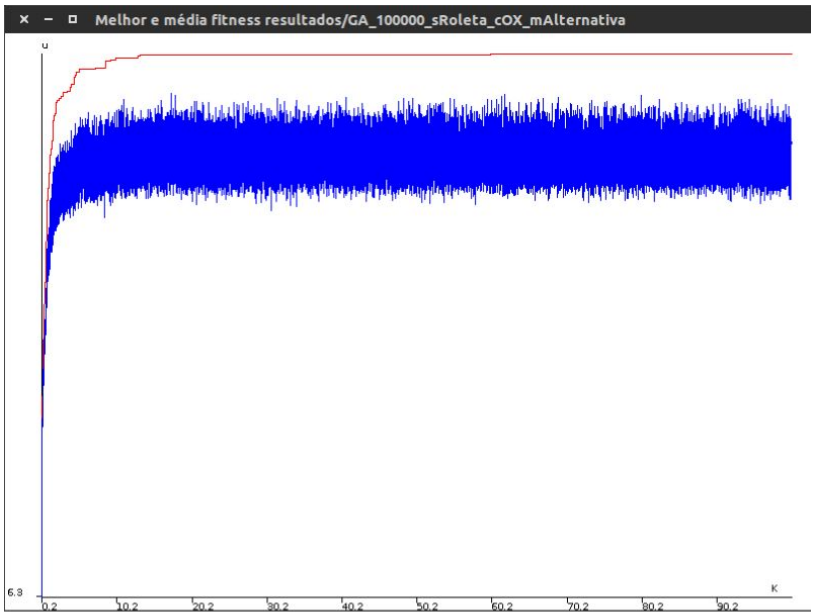


2.a.iv.17 - Gráfico de Candidatos 50 em 10 mil gerações



2.a.iv.19 - Gráfico de Candidatos 50 em 100 mil gerações

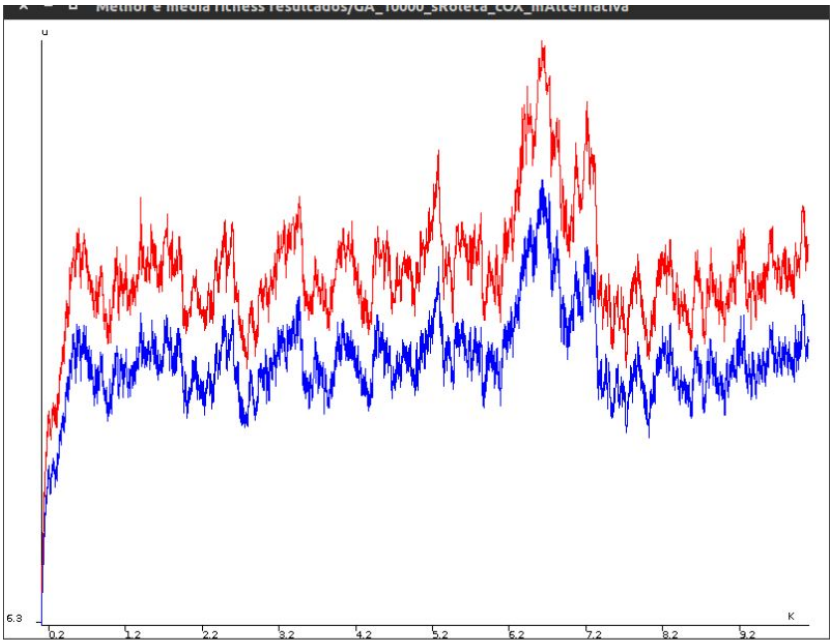
Apesar dessa combinação atingir a curto prazo o melhor *fitness* até agora, a longo prazo, essa combinação atingiu no máximo 4,480E-5 ficando a baixo do melhor *fitness* a longo prazo atingido até agora, embora com uma diversidade menor e com um tempo menor de execução.



2.a.iv.19 - Gráfico de Candidatos 50 em 100 mil gerações

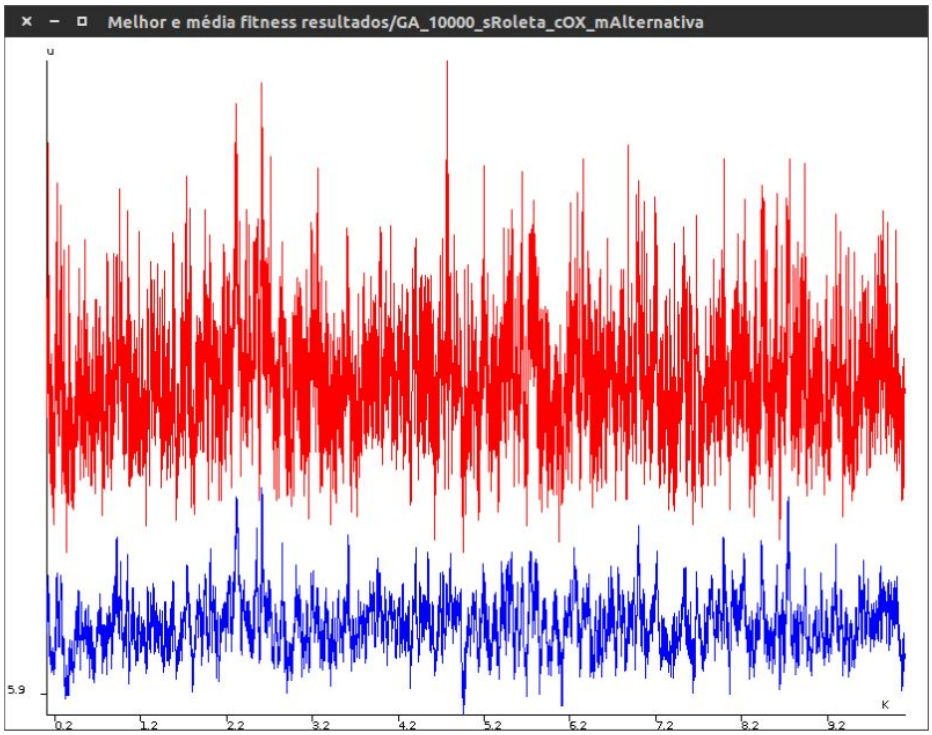
Mais dois aspectos da população foram alterados, mas não obtivemos melhores *fitness* e essas combinações serviram apenas para fim de análise. Primeiro, observamos o comportamento da população quando colocamos que os pais não sobrevivem ao crossover.

Isso resultou numa variação bem maior tanto do melhor *fitness* e da média do *fitness* da população, e a melhor solução conseguida na última geração não foi a melhor obtida dentre todas as gerações e mesmo com os melhores parâmetros até agora, quando os pais não sobrevivem, o melhor *fitness* alcançado foi 1,924E-5. Por outro lado, a diversidade se manteve baixa durante todo o processo, e nas últimas gerações ela variou de 2,739E-6 à 3,904E-6.



2.a.iv.20 - Gráfico de Pais Não Sobrevivem em 10 mil gerações

Outro mudança interessante nos parâmetros na combinação onde os pais não sobrevivem, foi tirar as duas subpopulações durante as gerações. Podemos ver que a variação das duas funções aumenta ainda mais e até mesmo a diversidade que se mantinha baixa antes, aumentou e varia drasticamente. O melhor *fitness* conseguido aqui foi 6,865E-6.



2.a.iv.21 - Gráfico de Pais Não Sobrevivem e Sem Subpopulações em 10 mil gerações

Portanto, a combinação de definida até agora para curto prazo é:

- Crossover OX, Mutação Alternativa, População Inicial=100, Subpopulação=20, Candidatos a Crossover=50, Não mutantes=25 , taxa de Crossover=0,8 e taxa de Mutação=0,2 (esses dois últimos parâmetros não foram analisados ainda).

  A longo prazo, a melhor combinação até agora é:
- Crossover OX, Mutação Alternativa, População Inicial=100, Subpopulação=20, Candidatos a Crossover=100, Não mutantes=25 , taxa de Crossover=0,8 e taxa de Mutação=0,2 (esses dois últimos parâmetros não foram analisados ainda).

v. Taxa de Crossover

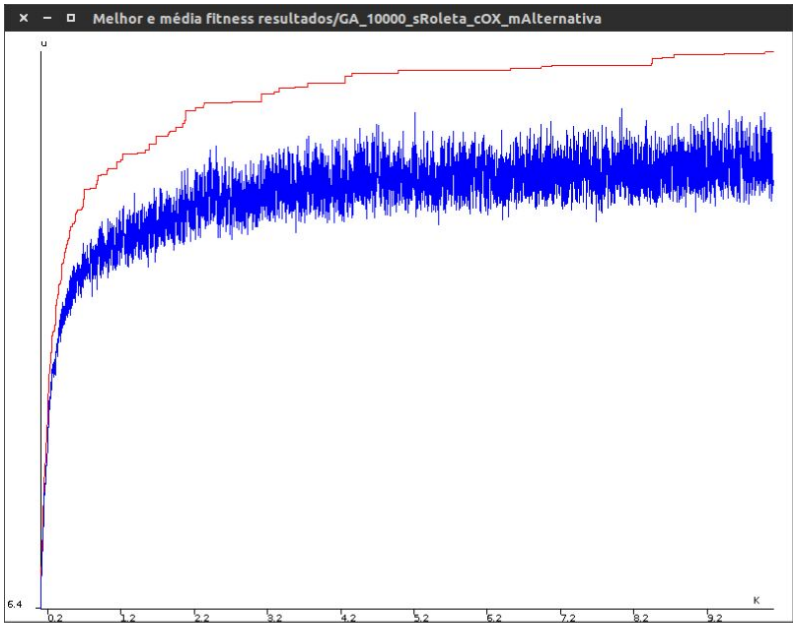
Até o momento, mantivemos a taxa de crossover como 80% em todas as análises. As evidências presentes do diretório 10 ("10 Desempenho_TaxaCrossover_GA_10000_sRoleta_cOX_mAlternativa") envolvem testes em que modificamos a taxa de crossover de 80% para 70%, 60% e 50%.

Com a taxa de 80%, alcançamos o *fitness* 4,4431 como vimos antes. A partir da 8000ª geração não tivemos evolução e tivemos uma baixa diversidade (variando de 6,83E-6 à 8,72E-6 nas últimas gerações). Alterando para 70%, o melhor *fitness* adquirido foi levemente mais baixo: 4,4258E-5 e a diversidade variando entre 7,16E-6 e 9,11E-6, ou seja, com uma variação da diversidade maior. Apesar de um *fitness* menor, a taxa de crossover 70% permitiu que a evolução do melhor *fitness* perdure por mais gerações de tal forma que nem nas últimas 500 gerações o algoritmo deixou de encontrar um melhor cromossomo.

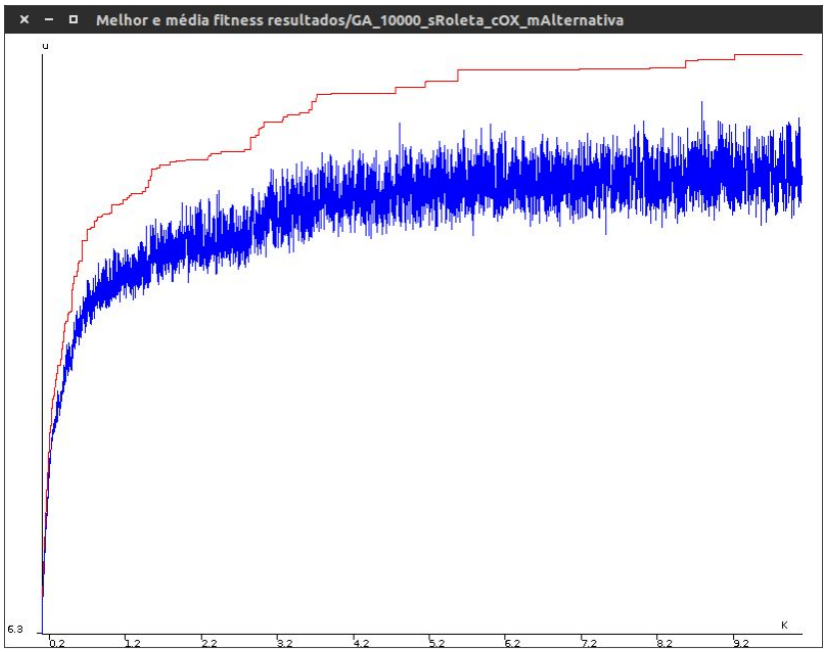
Quando alteramos a taxa para 60% adquirimos o melhor *fitness* entre todas as combinações já realizadas a curto prazo: 4,567E-5. O tempo de execução aumentou 3 segundos em relação quando tínhamos a taxa de 80% e adquirimos uma diversidade menor que as duas outras taxas testadas: de 7,71E-6 à 9,55E-6 nas últimas gerações. Com essa taxa, a evolução do melhor *fitness* perdura mais e tem "saltos" maiores nas últimas gerações comparado com a taxa de 80%.

Um *fitness* ainda melhor foi adquirido a curto prazo com a taxa de crossover de 50%: 4,6098E-5. Ganhamos no valor em melhor *fitness* e um segundo na execução em troca de diversidade, que agora está variando de 8,77E-6 e 1,18E-5. A evolução do *fitness* nas últimas gerações também foi "sacrificada", e o melhor *fitness* foi alcançado antes de terminar a execução das primeiras 7 mil gerações.

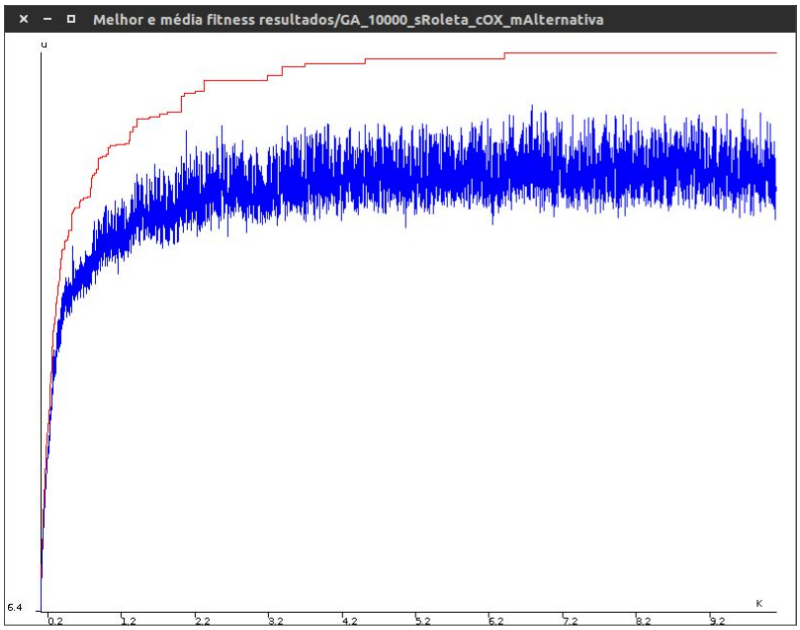
Quando diminuímos ainda mais a taxa de crossover, para 40%, o melhor *fitness* adquirido é bem mais baixo do que conseguido na taxa 50%: 4,1965, mas a evolução nas últimas épocas é bem melhor, justamente por não ter um melhor *fitness* tão alto quanto.



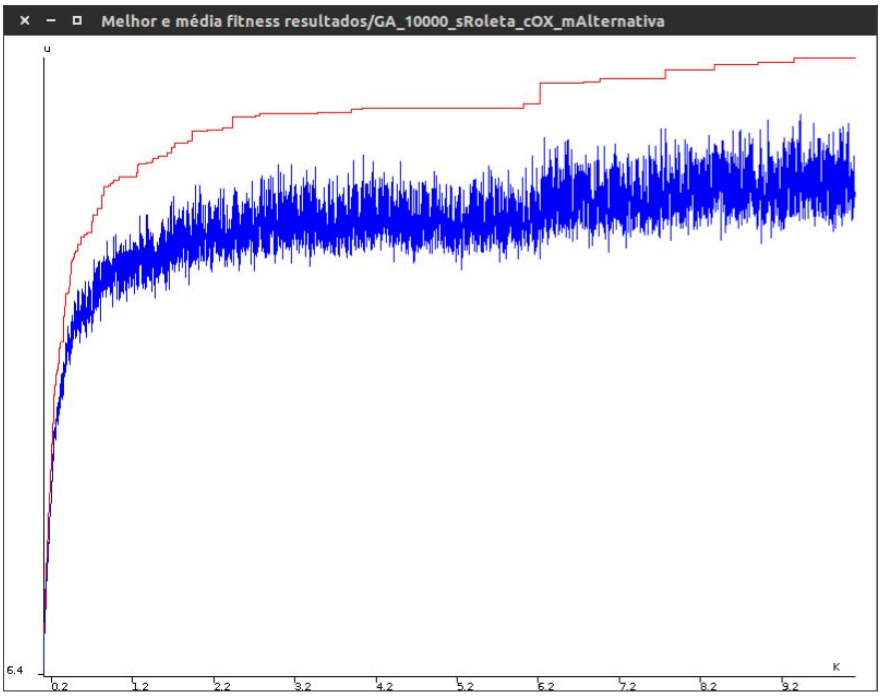
2.a.v.2 - Gráfico da Taxa de Crossover 70% em 10 mil gerações



2.a.v.3 - Gráfico da Taxa de Crossover 60% em 10 mil gerações

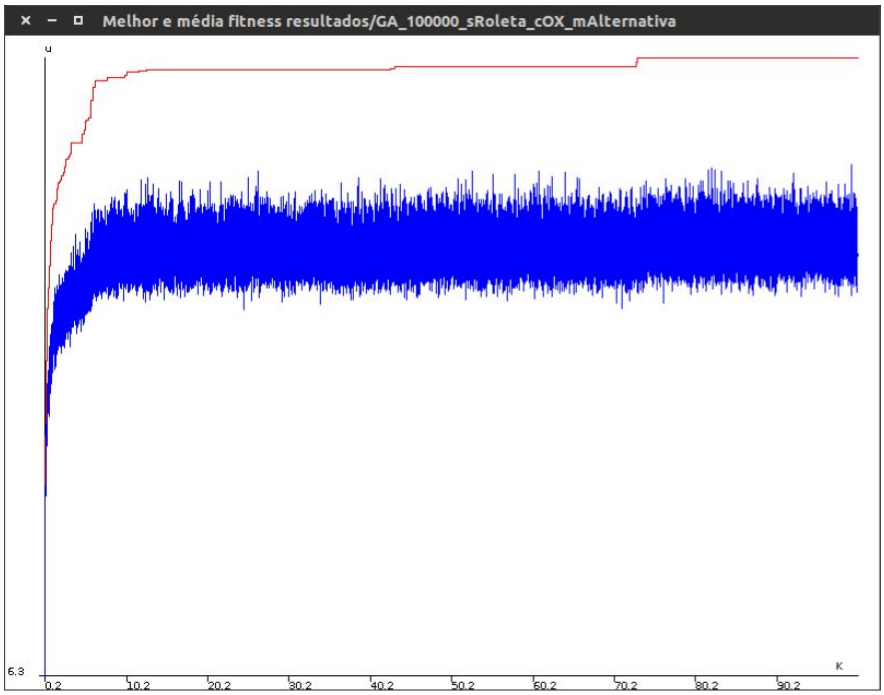


2.a.v.4 - Gráfico da Taxa de Crossover 50% em 10 mil gerações



2.a.v.5 - Gráfico da Taxa de Crossover 40% em 10 mil gerações

Ao testar a taxa de crossover de 50% a longo prazo (100 mil gerações) alcançamos o melhor valor de *fitness* até o momento: 4,6980464E-5 após 35 minutos. A diversidade aumentou mas a variação dela teve uma leve mudança, quase que insignificante.



2.a.v.6 - Gráfico da Taxa de Crossover 50% em 100 mil gerações

Portanto, as melhores combinações de parâmetros para curto prazo é:

- Crossover OX, Mutação Alternativa, População Inicial=100, Subpopulação=20, Candidatos a Crossover=50, Não mutantes=25 , taxa de Crossover=0,5 e taxa de Mutação=0,2 (esse último parâmetro não foi analisado ainda). *Fitness*: 4,6098E-5.

A longo prazo, a melhor combinação até agora é:

- Crossover OX, Mutação Alternativa, População Inicial=100, Subpopulação=20, Candidatos a Crossover=100, Não mutantes=25 , taxa de Crossover=0,5 e taxa de Mutação=0,2 (esse último parâmetro não foi analisado ainda). *Fitness*: 4,6980464E-5.

vi. Taxa de Mutação

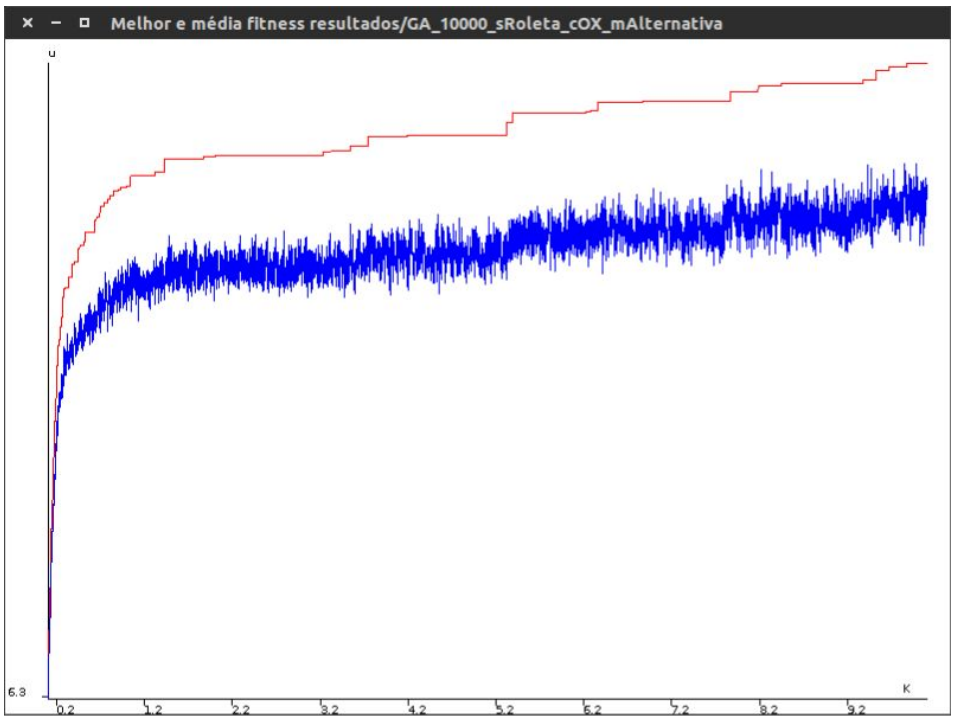
O último parâmetro testado é a taxa de mutação. As evidências presentes do diretório 11 (“11 Desempenho_TaxaMutacao_GA_10000_sRoleta_cOX_mAlternativa”) envolvem testes em que modificamos a taxa de mutação de 20% para 10%, 25%, 30% e 40%.

A taxa de 10% resultou num *fitness* máximo de 3,9549E-5, bem a baixo do ótimo conseguido a curto prazo e a longo prazo. Por outro lado, a diversidade se manteve mais baixa do que os testes que conseguiram os resultados ótimos (entre 6,357E-6 e 8,026E-6). Com taxa 25%, o melhor *fitness* aproximou-se da solução ótima conseguida até agora:

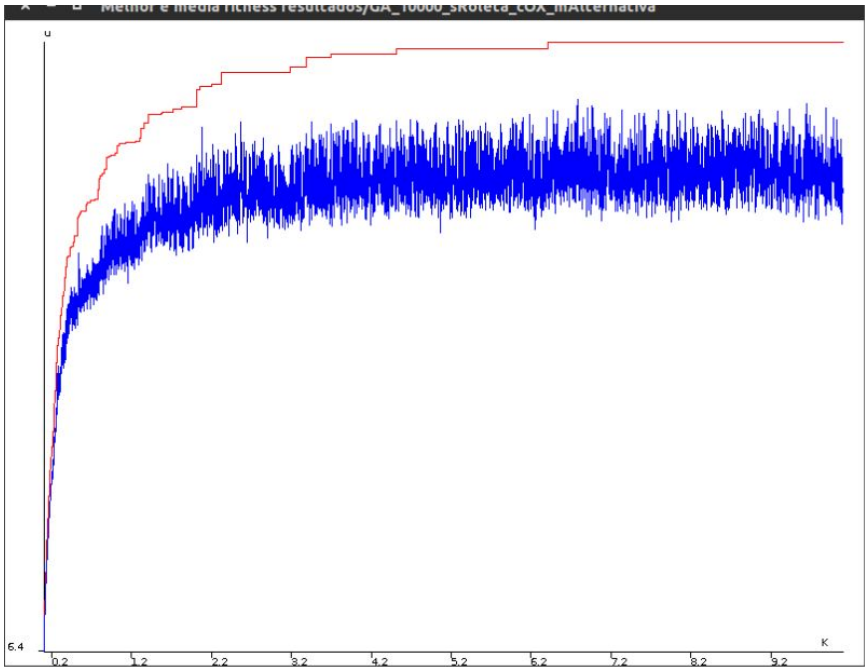
4,51738E-5. O aumento da taxa resultou num aumento da diversidade (nas últimas gerações, varia de 1,119E-6 à 1,29E-6).

Com taxa 30%, o melhor *fitness* obtido foi levemente mais alto do que a taxa 25%, mas ainda mais baixo do que o ótimo alcançado: 4,5478E-5. A diversidade superou tanto a combinação ótima com taxa 20% quanto as taxas de 10% e 25%: variando de 1,26E-5 à 1,33E-5.

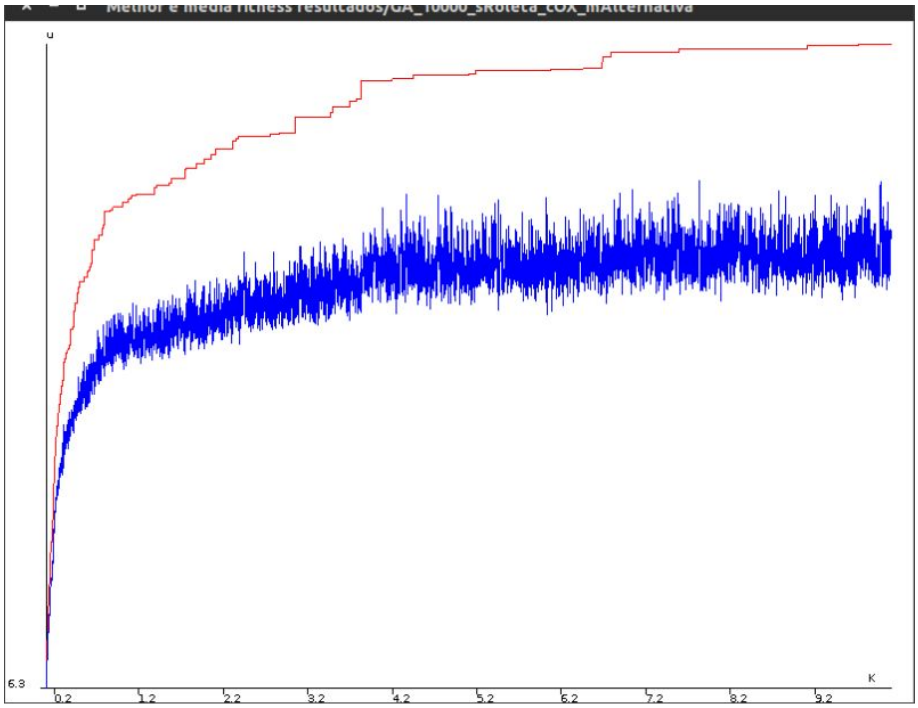
Quando aumentamos a taxa ainda mais, para 40%, o melhor *fitness* se distancia da melhor solução conseguida até agora, mas dessa vez a diversidade não mudou muito, ficando levemente mais baixa: nas últimas gerações, a diversidade variou de 6,7E-6 à 11,33E-6. Por outro lado, o tempo de execução mudou drasticamente: de 77 segundos da solução ótima passou para 131 segundos. Como não tivemos melhores resultados, não fizemos testes à longo prazo para nenhuma dessas novas combinações.



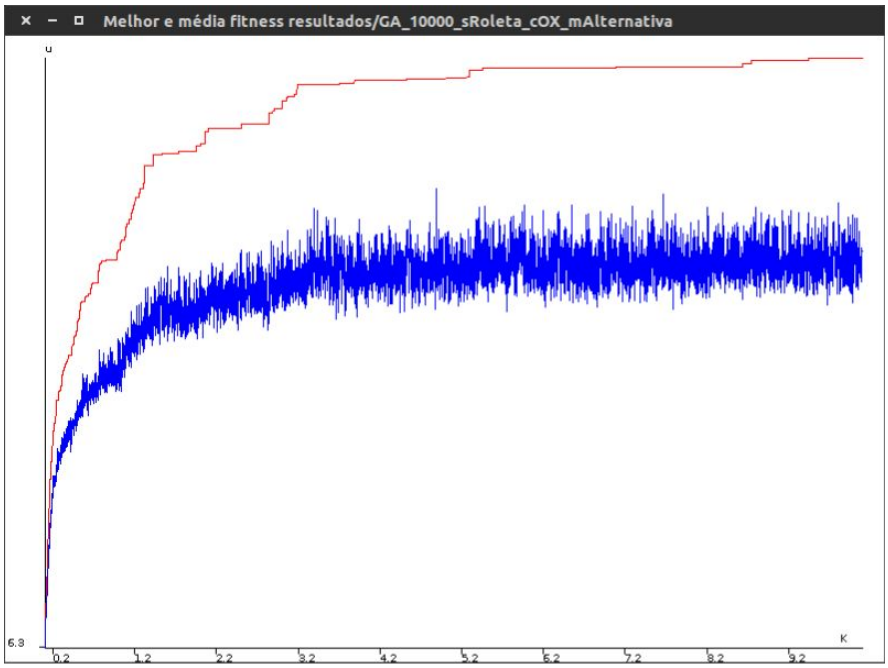
2.a.vi.1 - Gráfico da Taxa de Mutação 10% em 10 mil gerações



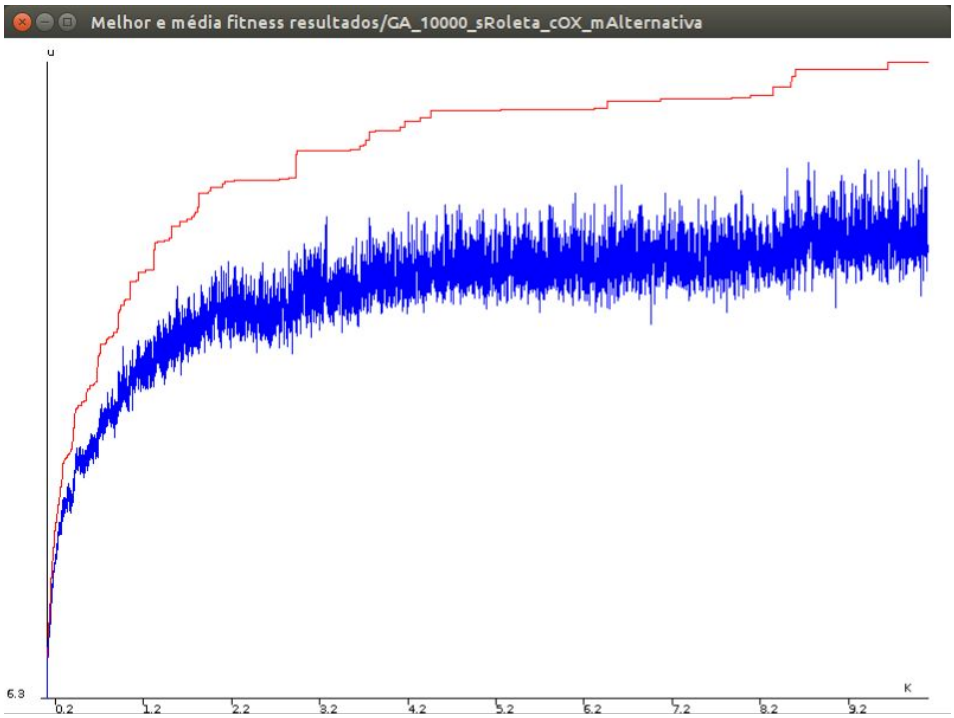
2.a.vi.2 - Gráfico da Taxa de Mutação 20% em 10 mil gerações



2.a.vi.3 - Gráfico da Taxa de Mutação 25% em 10 mil gerações



2.a.vi.4 - Gráfico da Taxa de Mutação 30% em 10 mil gerações



2.a.vi.5 - Gráfico da Taxa de Mutação 40% em 10 mil gerações

3. Bibliotecas Utilizadas

Links para download das bibliotecas e suas documentações.

JAMA

- Site para download:

    - http://math.nist.gov/javanumerics/jama/

- Documentação:
    - http://math.nist.gov/javanumerics/jama/doc/

Jama Utils

- Site para download:
    - http://www.seas.upenn.edu/~eeaton/software.html

- Documentação:
    - http://www.seas.upenn.edu/~eeaton/software/Utils/javadoc/edu/umbc/cs/maple/utils/JamaUtils.html

JChart2D

- Site para download:
    - http://www.java2s.com/Code/Java/Chart/JChart2DDynamicChart.htm