Universidade Federal de Santa Catarina
Departamento de Informática e Estatística

Genetic Scheduler: Um Algoritmo Genético para Escalonamento de Tarefas com Restrição Temporal em Sistemas Distribuídos

Ruann Magalhães Homem

Florianópolis, Junho de 2017

Universidade Federal de Santa Catarina

Departamento de Informática e Estatística

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE SISTEMAS DE INFORMAÇÃO

Genetic Scheduler: Um Algoritmo Genético para Escalonamento de Tarefas com Restrição Temporal em Sistemas Distribuídos

Ruann Magalhães Homem

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação

Florianópolis, Junho de 2017

Genetic Scheduler: Um Algoritmo Genético para Escalonamento de Tarefas com Restrição Temporal em Sistemas Distribuídos

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação

_____

Prof.ª Dr.ª, Luciana de Oliveira Rech

Orientadora

Banca Examinadora:

_____

Prof. Dr., Cristian Koliver

_____

Bel. Antonio José Resende de Campos

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Reduções

AG	Algoritmos Genéticos
STR	Sistema de Tempo Real
OX3	*Order Crossover* 3
RM	Taxa Monotônica
EDF	*Earliest Deadline First*
DM	*Deadline* Monotônico
AWA	Pesos adaptativos
SA	*Simulated Annealing*

Resumo

Sistemas de Tempo Real (STR) apresentam a necessidade de executar suas tarefas dentro de um limite de tempo conhecido como *deadline*, caracterizando assim um limite temporal em sua execução. Devido a essa restrição, a ordenação na execução de suas tarefas é um fator crítico para este tipo de sistema. A essa ordenação é dado o nome de escalonamento das tarefas. Encontrar uma lógica capaz de gerar escalonamentos válidos, ou seja, escalonamentos que possam garantir as limitações temporais na execução das tarefas, é uma das principais atividades que devem ser executadas no desenvolvimento de um sistema de tempo real. Como exemplos desse tipo de sistema pode ser citado: sistemas de controle de tráfego aéreo e sistemas de streaming, entre outros.

Algoritmos Genéticos (AG) são uma alternativa interessante para a busca de soluções ótimas ou quase-ótimas para problemas de geração de escalonamentos para tarefas de STR em ambientes distribuídos.

O objetivo deste trabalho é apresentar uma solução para o problema de geração de escalonamentos de tarefas que possuem limitação no tempo de sua execução, podendo a falha em executar dentro do tempo delimitado gerar problemas de performance ou até levar a sérios erros no sistema.

Neste trabalho é apresentado um algoritmo genético capaz de encontrar um escalonamento válido para um conjunto de tarefas que precisam ser executadas em uma determinada configuração de um sistema. É descrito ao longo do trabalho todos os componentes deste algoritmo: modelo do cromossomo, operadores genéticos e critérios de avaliação da qualidade para determinar a validade dos escalonamentos gerados. Por fim, são demonstrados os testes realizados utilizando o algoritmo, detalhando os parâmetros para o algoritmo e métricas obtidas em sua execução.

Palavras Chaves: Algoritmos Genéticos. Sistemas Distribuídos. Sistemas de Tempo Real. *Deadline*, Restrição Temporal

1. Introdução

STR são aqueles onde cada tarefa tem como requisito uma restrição temporal, ou seja, elas possuem um limite de tempo para sua execução conhecido como *deadline*. Frequentemente essas tarefas possuem restrições na ordem ao qual elas devem executadas. Quando há uma restrição deste tipo as tarefas terão uma lista contendo tarefas precedentes que precisam ser executadas antes que essas possam começar a sua execução. Este tipo de sistema tem sido muito utilizado em áreas como automatização de fábrica, sistemas embarcados e sistemas multimídia entre outros (KONAR, 2017).

STR podem ser executados nos mais diversos ambientes computacionais. Pode-se citar como exemplo: sistemas distribuídos, sistemas paralelos, sistemas multiprocessadores. Cada um desses ambientes oferece vantagens e desvantagens para um STR. Este trabalho é voltado para um STR sendo executado em um sistema de clusters de processamento.

Uma das abordagens que podem ser utilizadas na resolução do problema de escalonamento de tarefas com restrições temporais é a técnica AG. Um AG simula um ambiente de competição entre as soluções já conhecidas, chamadas de cromossomos que formam uma população, para obter soluções melhores através da utilização de operadores genéticos nos melhores cromossomos conhecidos. A utilização de um AG pode ser realizada em conjunto com uma ferramenta de simulação para avaliar as possíveis soluções que o algoritmo gerou (SEBESTYEN e HANGAN, 2012). Com isso sendo uma alternativa para validar os escalonamentos além das técnicas de análise de escalabilidade.

AG é uma abordagem interessante na solução do problema de encontrar escalonamentos factíveis para tarefas em STR em ambientes distribuídos (AZKETA, 2011) (SEBESTYEN e HANGAN, 2012).

1.1. Motivação

Com os trabalhos relacionados analisados, constatou-se que o problema de geração de escalonamentos que respeitem as restrições temporais pode ser abordado de diversas formas. Trabalhos como (SEBESTYEN e HANGAN, 2012), (KONAR, 2017), (GHARSELLAOUI, 2015) alteraram as características temporais das tarefas utilizando em conjunto com técnicas de escalonamento conhecidas como EDF (*Earliest Deadline First*) para gerar escalonamentos válidos. Poucos trabalhos como (AZKETA, 2011), geraram escalonamentos utilizando apenas AGs analisando as características já definidas para as tarefas, ou seja, sem alterar o *deadline* ou o tempo de execução.

(AZKETA, 2011) e (GHARSELLAOUI, 2015) utilizaram técnicas de análise de escalabilidade para verificar seus resultados obtidos ou não especificaram como foram obtidos os dados utilizados no *fitness*. (SEBESTYEN e HANGAN, 2012) testaram seus resultados com ferramentas de simulação tentando executar os escalonamentos gerados em um sistema controlado e mais próximo de um cenário real. Com isso, a motivação desse trabalho foi aplicar as técnicas de AG

para criar um algoritmo que possa gerar escalonamentos válidos para um conjunto de tarefas, validando-os através de simulações.

1.2. Objetivos

Desenvolver um AG capaz de escalonar um conjunto de tarefas que devem ser executadas em um sistema distribuído. As tarefas a serem escalonadas possuem restrições temporais. O objetivo do AG é encontrar um escalonamento válido que apresente os melhores tempos de folga na execução destas tarefas, ou seja, uma maior diferença entre o *deadline* e o tempo de execução.

1.2.1. Objetivos Específicos

- Pesquisar o estado da arte na área de AG e escalonamento em STR;
- Investigar e determinar as ferramentas a serem utilizadas durante o desenvolvimento do projeto;
- Desenvolver e implementar o AG;
- Realizar uma análise da performance do AG e dos escalonamentos por ele gerados;

1.3. Metodologia

Este trabalho foi desenvolvido baseado em uma pesquisa exploratória, que resultou no desenvolvimento de um software utilizando os conceitos e técnicas encontrados nos trabalhos relacionados. Este trabalho foi realizado nas seguintes etapas:

- Pesquisa e análise da literatura de AG para resolução do problema de geração de escalonamentos para STR.
- Definição da arquitetura e ferramentas que foram utilizadas no desenvolvimento do software.
- Definição dos operadores genéticos a serem utilizados no AG.
- Desenvolvimento da primeira versão do algoritmo.
- Teste inicial para validação dos resultados.
- Ajustes e correções no software desenvolvido.
- Testes finais e avaliação dos resultados obtidos.

1.4. Limitações

Além da restrição temporal, um STR possui diversos problemas que precisam ser tratados afim de garantir a sua correta execução. Por exemplo, o envio de mensagens entre os processadores através de uma rede de comunicação e o tratamento que pode ser realizado quando uma tarefa falha em sua execução. Este trabalho trata da questão do escalonamento

destas tarefas, enfatizando sua restrição temporal e não aborda a questão de comunicação e garantia de entrega das mensagens.

Para os tipos de tarefas possíveis para STR, este trabalho foi desenvolvido considerando que as tarefas são do tipo Hard, ou seja, todas precisam executar dentro dos seus deadlines. Porém, com a revisão da função fitness do *Genetic Scheduler* seria possível aceitar as tarefas do tipo soft no escalonamento.

1.5. Organização do texto

Este trabalho está organizado em seis capítulos. O capítulo atual é responsável por apresentar uma visão geral e resumida do trabalho, demonstrando os resultados desejados e a forma como esse trabalho foi desenvolvido considerando suas limitações.

O segundo capítulo aprofunda os conhecimentos das técnicas de AG e os conceitos básicos para STR. Apresentando toda a base teórica necessária para a compreensão dos capítulos seguintes.

O terceiro capítulo traz de forma resumida dados e conceitos de trabalhos relacionados sobre o tema da utilização de AG para escalonamentos de tarefas para STR.

O quarto capítulo descreve o algoritmo desenvolvido, detalhando as ferramentas utilizadas e os operadores genéticos utilizados. No final do capítulo é apresentado o processo de desenvolvimento de um STR que utilizaria este AG.

O quinto capítulo apresenta os resultados obtidos através dos testes realizados, assim como a análise dos dados gerados durante os testes.

Finalmente o sexto capítulo discute as conclusões e considerações sobre o algoritmo desenvolvido, seguido com sugestões para possíveis trabalhos futuros.

2. Fundamentação Teórica

2.1. Introdução

Esta seção tem como objetivo introduzir os conceitos básicos para a área de AG e STR. Esses conceitos são a base para o algoritmo desenvolvido neste trabalho.

Para a área de AG é apresentada uma definição para esse tipo de algoritmo, seguida de um breve histórico do mesmo. Nas seções seguintes é apresentada a estrutura básica para um AG seguida de definições e exemplos de cada parte que pode compor o mesmo.

Para STR é apresentada uma definição desse tipo de sistema. Logo após, é discutido brevemente o que são as restrições temporais e as relações de precedência e exclusão. Para finalizar o capítulo é apresentada a definição de algoritmos de escalonamento, seguida de exemplos como *Earliest Deadline First* (EDF), Taxa Monotônica (RM) e *Deadline* Monotônico (DM).

2.2. Algoritmos Genéticos

AG são algoritmos que utilizam modelos computacionais inspirados nos processos naturais de evolução das espécies para resolver problemas computacionalmente complexos (LINDEN, 2012). Esses algoritmos são técnicas heurísticas de otimização global que tentam maximizar uma função de avaliação (*Fitness*) para encontrar a solução para um problema. Para isso o AG cria uma população contendo diversas soluções possíveis para esse problema. Cada membro dessa população é denominado indivíduo. A cada interação do algoritmo o mesmo seleciona um conjunto de indivíduos baseado na função de avaliação, que mede a qualidade da solução, para aplicar operadores genéticos (de *crossover* e mutação). Os novos indivíduos resultantes compõem a população da próxima geração, que tendem a conter soluções "evoluídas", ou seja, soluções que resolvem melhor o problema. Essas soluções evoluídas irão se aproximar de soluções ótimas a cada passo do algoritmo.

2.2.1. Histórico dos Algoritmos Genéticos

Os primeiros estudos na área de AG foram realizados no início da década de 40. Nesses primeiros estudos os pesquisadores estavam mais interessados no processo cognitivo e de aprendizado do que o processo evolutivo. No início de década de 50 foram realizadas pesquisa que procuravam sistemas genéricos inspirados em modelos naturais que pudessem encontrar soluções candidatas para problemas que eram computacionalmente complexos de serem resolvidos.

Em 1957 Box apresentou um esquema de operações evolucionárias. Essas operações realizavam mudanças sistêmicas em um conjunto pequeno de variáveis. No início da década de 1960 Bledsoe e Bremmerman introduziram o conceito de genes em seus trabalhos e desenvolveram os precursores dos operadores de recombinação (LINDEN, 2012).

Em 1975 Jon Holland publicou seu livro, "Adaptation in natural and Artifical System" onde apresentou um estudo formal da evolução das espécies, nesse trabalho aonde ele propôs um modelo heurístico computacional inspirado na teoria da evolução. No trabalho ele apresenta este modelo como uma metáfora dos sistemas de evoluções naturais sendo simulada dentro de computadores. Tal modelo conseguia apresentar boas soluções para problemas insolúveis computacionalmente.

2.2.2. Conceitos de Algoritmos Genéticos

Os AG são um ramo dos algoritmos evolucionários, sendo esses um ramo na área de inteligência artificial. Os algoritmos evolucionários usam modelos computacionais dos processos evolucionários como ferramentas para resolver os problemas (LINDEN, 2012).

O AG utiliza estruturas inspiradas na teoria da evolução das espécies em um processo similar a similar ao desta teoria, para obter melhores soluções para um problema. Para qualificar as soluções encontradas é utilizado uma função de avaliação denominada de *fitness*.

Os operadores genéticos são processos computacionais que simulam os fenômenos naturais de evolução que podem ser observados na natureza. Esses fenômenos são o que guia a evolução na natureza. Como paralelo à natureza esses processos evoluem as soluções já encontradas nas populações em direção à solução ótima. Como exemplo desses processos podemos mencionar a reprodução sexuada e a mutação genética.

Os AGs são algoritmos de otimização global que se opõem a métodos como o hill climbing que seguem uma derivada de uma função de forma a encontrar o máximo dessa função, pois o AG procura em múltiplos pontos no espaço de busca evitando assim cair em máximos locais (LINDEN, 2012).

2.2.3. Estrutura básicas para Algoritmo Genético

Para os AGs a estrutura mais básica é o gene (LINDEN, 2012). Os genes controlam uma ou mais características de um indivíduo. Todo gene possui uma posição definida dentro do indivíduo chamada locus. O mesmo recebe um valor de acordo com o alfabeto decidido durante a codificação, que indica como o gene altera o conjunto de características ligadas a ele.

Ao conjunto de genes é dado o nome de cromossomo (LINDEN, 2012). Diferente da teoria da evolução natural, geralmente um cromossomo representa todas as características de um indivíduo dentro de uma população enquanto na natureza um indivíduo é composto por mais de um cromossomo.

2.2.4. Objetivos dentro do Algoritmo Genético

O principal objetivo de um AG é procurar no espaço onde existem todas as soluções possíveis para o problema a ser resolvido, chamado de espaço busca, as melhores soluções que se aproximem mais de uma solução ótima para resolução de um objetivo a ser maximizado.

Para alcançar esse objetivo o AG pode ser separado em dois componentes: exploration e explotation (LINDEN, 2012). O componente de exploration se encarrega em caminhar pelo espaço de busca procurando por indivíduos com características diferentes das que já existem nas soluções já encontradas na população atual. Em contrapartida, o componente de explotation se preocupa em obter e manter as melhores soluções encontradas pela aplicação dos operadores genéticos na atual população.

2.2.5. Processo básico do Algoritmo Genético

O AG atua em uma população de possíveis soluções. O algoritmo seleciona os melhores indivíduos dentro população para aplicar os operadores genéticos e gerar novas soluções. Geralmente os operadores aplicados são os de *crossover* e mutação, embora outros também possam ser utilizados. Esses passos são executados diversas vezes sendo uma única execução desse processo denominado de geração. Gerações são executadas até ser alcançado um determinado critério de parada, sendo essa a finalização do processo de evolução (LINDEN, 2012).

2.2.5.1. Codificação

A codificação se refere a encontrar uma representação computacional para as soluções dos espaços de busca. Essa representação será utilizada no AG para que o mesmo possa manipular as soluções com os operadores genéticos a fim de gerar novas soluções. A qualidade da codificação irá determinar a facilidade com qual o AG irá manipular essas soluções através dos operadores genéticos. Uma forma clássica de codificação é a codificação de binários, sendo o cromossomo formado de uma sequência de 0´s e 1´s.

Para definir essa representação é preciso definir um alfabeto a ser utilizado. Esse alfabeto é um conjunto de símbolos que pode ser utilizado para criar um cromossomo. No caso clássico da representação binaria o alfabeto é A = {0,1}, sendo esses os únicos símbolos que podem ser utilizados para gerar um indivíduo na população.

Em alguns casos não é somente o alfabeto utilizado que tem significado para representação. Cromossomos onde a ordem em que as informações aparecem podem representar isso na ordenação dos seus genes. Neste caso, a primeira posição no gene poderia representar uma atividade a ser executada primeiro, por exemplo. Essa representação é denominada de representação permutacional.

A codificação das informações em genes e, consequentemente em cromossomos, é um ponto crucial nos AGs. Pois é essa codificação, em conjunto com a função de avaliação que liga um AG ao problema a ser resolvido (LINDEN, 2012). A correta codificação poderá contribuir muito para o sucesso do AG.

2.2.5.2. População no Algoritmo genético

De acordo com (LINDEN, 2012) AGs mantêm um conjunto de soluções codificadas, denominado de população, para ser utilizado pelos operadores genéticos. Dessa forma, essas

soluções podem ser modificadas pelos operadores, gerando novas soluções com novas características e, assim, percorrer o espaço de busca procurando por soluções que possam melhor resolver o problema.

Para a inicialização da população inicial geralmente é utilizado o método de escolha aleatória sem observar quaisquer características deste indivíduo (LINDEN, 2012). Geralmente esse método apresenta bons resultados pois as leis da probabilidade indicam que iremos cobrir o espaço de busca de forma uniforme, o que contribuirá de forma positiva no componente de exploração do AG.

Existem estudos que utilizam o conhecimento do domínio do problema, ou soluções prévias de uma base de dados, para inicializar a população inicial. Isto pode ajudar no algoritmo pois esses indivíduos provavelmente já possuem boas características que podem ser passadas adiante pelos operadores genéticos

Um conceito muito importante ligado a população é o conceito de diversidade na população. Este conceito se refere ao quanto os indivíduos dentro da população são diferentes entre si, aumentando o número de características distintas presentes na população (LINDEN, 2012). Isso é muito importante, pois o AG precisa de uma alta diversidade genética dentro da população para obter indivíduos com novas características.

Todo AG sofre com o tempo com a perda da diversidade através da convergência genética. Isso acontece, porque o processo evolutivo tende a privilegiar os indivíduos melhor adaptados (indivíduos com melhores avaliações) para a reprodução. Com isso, ao longo das gerações, os filhos gerados no processo de *crossover* ficam mais parecidos com seus pais a cada iteração, recebendo características muito parecidas de ambos.

- **População usando Elitismo**

Uma estratégia muito importante para a geração de uma nova população após aplicar os operadores genéticos na população atual é o elitismo. No elitismo, a cada geração N indivíduos da população anterior são mantidos na nova população. Os N indivíduos mantidos são os N cromossomos com maior *fitness* (LINDEN, 2012). Esse N é um número que precisa ser decidido no desenvolvimento do AG. Essa estratégia é utilizada para garantir que não serão perdidas as melhores soluções já encontradas. Isso acontece porque os operadores genéticos não podem garantir que os cromossomos gerados serão superiores aos cromossomos origem. Essa estratégia sempre garante que os melhores indivíduos que já encontramos não serão perdidos.

- ***Steady state***

O *steady state* é uma estratégia aplicada na população para reproduzir a conceito de população do “mundo real” (LINDEN, 2012). No mundo real, os indivíduos nascem e morrem aos poucos ao invés de uma geração substituir a outra completamente. Seguindo essa ideia, uma quantidade N dos filhos substituem a mesma quantidade N dos pais na próxima população.

Em algumas implementações isso é feito de forma aleatória enquanto em outras os piores pais são substituídos pelos melhores filhos.

2.2.5.3. Função de avaliação

A função de avaliação do indivíduo é utilizada pelo AG para determinar a qualidade da solução codificada no cromossomo. Isso significar mensurar o quão bem essa solução resolve o problema. Em conjunto com a codificação, a função de avaliação é o único elo que o AG tem com o problema a ser resolvido (LINDEN, 2012). Isso acontece porque essa função avalia o indivíduo somente através das características que o mesmo apresenta para a resolução do problema. Por isso, ela ignora a forma com que os indivíduos foram modificados pelos operadores ou mesmo a forma como eles estão codificados. Ou seja, ela não considera aspectos ligados à implementação do AG e somente aspectos ligados ao problema.

A função de avaliação deve ser uma função de maximização (Linden, 2012), ou seja, se um indivíduo A1 representa uma solução melhor que o indivíduo A2, então o A1 deve gerar um valor de *fitness* f(A1) superior ao gerado por f(A2).

A função de avaliação também deve embutir todo conhecimento possível sobre o problema a ser resolvido (LINDEN, 2012). Assim, ela deve contemplar todos os objetivos a serem maximizados e todas as restrições que as soluções devem atender para serem válidas.

Geralmente a função de avaliação é dada por um valor numérico. Quanto mais a solução atende ao problema, maior o valor numérico da função. Penalidades por não atender as restrições são feitas subtraindo valores, geralmente fixos, da função de avaliação. Geralmente essa função de avaliação é chamada de *fitness* no AG.

2.2.5.4. Seleção

O processo de seleção dos indivíduos atua no AG de forma análoga a seleção natural que ocorre na natureza. Ou seja, ele deve selecionar os indivíduos mais adaptados para que esses possam passar suas características para os novos cromossomos a serem gerados pelos operadores genéticos (LINDEN, 2012). No caso do AG, os indivíduos mais adaptados são aqueles que possuem maior *fitness*.

Embora o processo de seleção deva privilegiar os indivíduos com maior *fitness,* não devem ser ignorados os indivíduos com *fitness* baixos. Existem dois motivos para selecionar indivíduos com *fitness* baixo. O primeiro é que esses indivíduos podem possuir características que não estão presentes nos indivíduos com maior *fitness* mas que podem levar a descoberta melhores indivíduos. Até mesmo o pior indivíduo dentro da população pode possuir características positivas únicas para obtenção do melhor indivíduo (LINDEN, 2012). O segundo motivo para o uso de indivíduos com *fitness* mais baixo é evitar que a população seja dominada por indivíduos que possuem as mesmas características, pois os operadores genéticos terão poucas oportunidades de gerar novos indivíduos diferentes dos já existente na população. Esta situação pode levar a uma convergência genética prematura no AG impedindo que ele percorra uma área maior no espaço de busca e se limite a uma área menor conhecida como máximo local.

Os máximos locais possivelmente possuem soluções com boas avaliações, mas geralmente ainda estão longe do espaço que contém a solução ótima para o problema.

Portanto, o método de seleção não deve ser muito restritivo dando oportunidade de cromossomos com baixa avaliação participarem do processo evolutivo. Porém, ele deverá sempre privilegiar os melhores indivíduos para que os mesmos possam passar suas características com mais frequência que os indivíduos menos adaptados.

- **Roleta Viciada**

Um dos métodos mais utilizados para a seleção de indivíduo é o método da roleta viciada. Este método propõe criar uma roleta virtual onde cada cromossomo receberá uma porção da roleta de acordo com o seu *fitness*. Deste modo, os melhores indivíduos ocuparão as maiores partes da roleta devido ao seu alto *fitness* (LINDEN, 2012). Depois de completar a roleta com todos os indivíduos, é sorteada uma posição dessa roleta N vezes para sortear N indivíduos. Com isso, os melhores indivíduos terão maiores chances de serem selecionados, embora indivíduos com menores chances ainda possam ser selecionados.

- **Torneio**

No método de seleção por torneio, é selecionado de forma aleatória um conjunto de k indivíduos, sendo o valor mínimo de k = 2. Estes cromossomos irão competir entre si com as suas avaliações. Aquele indivíduo que estiver participando no torneio e tiver o maior *fitness* será o cromossomo selecionado.

No conjunto de indivíduos que são selecionados pode se permitir ou não a repetição de indivíduos. Caso seja permitido que um indivíduo seja repetido no conjunto que irá participar do torneio, poderá haver casos onde todos os participantes do torneio sejam, na verdade, o mesmo indivíduo.

Uma diferença importante deste processo de seleção comparado com o método da roleta viciada é que os melhores indivíduos somente terão, como vantagem, o seu alto valor de *fitness*. Isto significa que os mesmos não serão favorecidos na escolha de quais indivíduos irão participar do torneio, fazendo com que todos os cromossomos tenham chances iguais de participar (LINDEN, 2012). Sua única vantagem real é que se escolhidos, os melhores indivíduos serão selecionados, por causa das suas avaliações.

Uma desvantagem deste processo é que a única chance do pior indivíduo de ganhar um torneiro será participar de um torneio onde somente ele esteja competindo. Este tipo de torneio só pode ocorrer quando o algoritmo permite a repetição de indivíduos no conjunto do torneio. No final, as chances do pior indivíduo são muitas baixas em torneios que permitem repetição e inexistente em um torneio sem repetição.

- **Amostragem Estocástica Uniforme**

Neste método de seleção, todos os indivíduos são dispostos em segmentos contíguos de uma linha. Cada indivíduo recebe um segmento proporcional ao seu *fitness*. Depois de criar

todos os segmentos, um número i entre 0 e 1/N será sorteado. Usando esse número i serão criados N posições igualmente espaçadas em uma reta. Estas N posições irão apontar para um segmento. Com isso, serão selecionados N indivíduos donos dos segmentos que as N posições apontam (LINDEN, 2012).

De forma análoga ao método da roleta viciada, os melhores indivíduos receberão segmentos maiores e, portanto, irão cobrir a maior parte da linha sendo selecionados com maior frequência.

- ***Ranking***

A seleção por *Ranking* é um método que tem como objetivos evitar a convergência genética prematura e a dominância do super indivíduo na população. Neste método todos os cromossomos são ordenados de forma decrescente de acordo com o seu *fitness*. Todos os indivíduos usam o seu *ranking*, ou seja, a posição que ficaram quando foram ordenados, no lugar de seu *fitness*. Após definir todos os *ranking*s para todos os indivíduos na população, é utilizado outro método de seleção para selecionar os pais (LINDEN, 2012).

2.2.5.5. Operador de cruzamento (*Crossover*)

O operador de *crossover* é o operador que simula a reprodução sexuada que acontece na natureza. Através desse operador indivíduos trocam material genético, ou seja, seus genes (LINDEN, 2012). Com essa troca de genes podemos gerar novos indivíduos com as características combinadas de seus pais. No AG esse operador é aplicado a 2 ou mais cromossomos previamente selecionados pelo processo de seleção e acontece com uma probabilidade parametrizada.

No operador de *crossover* pode-se verificar a necessidade do processo de seleção não ser muito restritivo: se esse operador for aplicado a indivíduos muito parecidos, então os filhos gerados também serão muito parecidos. Isso acontece por não haver muitos genes diferentes entre os pais que possam levar a criação de um indivíduo muito diferente. O principal objetivo desse operador é atuar sobre o componente de explotation, ou seja, obter melhores indivíduos.

- ***Crossover* de K pontos**

O *crossover* de k pontos sorteia de forma aleatória k pontos, 2 sendo um valor comum para k, nos cromossomos pais e os segmentam em k + 1 segmentos. Com esses segmentos serão gerados dois filhos. O primeiro filho recebe o primeiro segmento do pai seguido do segundo segmento do segundo pai seguindo assim sucessivamente, alternando os segmentos de cada pai até preencher todo o cromossomo do primeiro filho. O segundo filho recebe todos os segmentos não utilizados no primeiro filho, começando pelo primeiro segmento do segundo pai e alternando os segmentos de cada pai (LINDEN, 2012).

- ***Crossover* uniforme**

Diferente do *crossover* de k pontos, o *crossover* uniforme decide gene a gene para cada posição qual dos pais irá passar seu gene adiante. Para cada gene dos filhos, será sorteado um

número entre 0 e 1. Se o número sorteado foi 0, o primeiro filho irá receber o gene correspondente do primeiro pai e o segundo filho irá receber o gene correspondente do segundo pai. Caso contrário o número sorteado seja 1, o gene do primeiro pai irá para o segundo filho e o gene do segundo pai irá para o primeiro filho. Este processo é repetido até preencher todos os genes de todos filhos (LINDEN, 2012).

- ***Crossover* baseado em ordem – *Order Crossover* 3**

O *Crossover* baseado em ordem ou *Order Crossover* 3 (OX3) é uma versão especial do *crossover* uniforme para representações permutacionais. Este operador é utilizado para preservar as posições relativas dos genes. Neste operador, 2 pontos de corte são selecionados e o primeiro pai passa todos os seus genes entres os pontos para o primeiro filho. Então é feita uma lista de todos os genes do primeiro pai fora dos pontos de corte. Esta lista é permutada de forma que os elementos estejam na mesma ordem que aparecem no segundo pai. Para finalizar todos esses genes são passados para o primeiro filho. O segundo filho é gerado de forma análoga substituindo o pai 1 pelo pai 2 no processo.

2.2.5.6. Mutação

O operador de mutação é o operador que aumenta a variabilidade genética na população alterado os valores dos genes de forma aleatória respeitando o alfabeto definido pela codificação (LINDEN, 2012). Com a aplicação desse operador podemos evitar uma convergência genética prematura e aumentar a varredura no espaço de busca, aumentando a variabilidade genética na população.

Normalmente é conferida uma probabilidade baixa para esse operador para evitar que ele acabe alterando indivíduos bons trocando suas boas características por características piores.

O principal objetivo desse operador é atuar no componente de exploração, aumentado a quantidades de novos indivíduos com novas características melhorando, assim, a nossa busca no espaço de soluções possíveis para o problema.

2.3. Sistemas de Tempo Real

STR são aplicações que apresentam requisitos de tempo real na execução de suas tarefas, principalmente restrições temporais. Portanto, um STR é um sistema computacional que deve reagir a estímulos oriundos do seu ambiente em prazos específicos (FARINES et al., 2000). Com isso, um STR precisa apresentar dois comportamentos importantes: o primeiro comportamento é a execução lógica. Ou seja, o sistema precisa apresentar saídas que estejam logicamente corretas para as entradas que recebeu. O segundo comportamento esperado é a previsibilidade temporal. Toda a ação dentro do sistema precisa ser executada dentro de um tempo controlado. Uma falha em executar dentro deste tempo é considerada uma falha temporal. Existem sistemas como controle de tráfego aéreo e sistemas militares de defesa que possuem restrições temporais mais rígidas do que sistemas como teleconferências através da Internet e as aplicações de multimídia em geral, que possuem restrições temporais mais brandas.

2.3.1. Restrições Temporais (STR)

A principal característica temporal de uma tarefa é que ela deve cumprir um prazo de execução chamado de "*deadline*" (FARINES et al., 2000). Normalmente uma tarefa deve terminar sua execução antes de seu *deadline*. A consequência de uma tarefa não alcançar seu *deadline* classifica as tarefas em dois grandes grupos:

- **Tarefas Críticas (tarefas "hard")**: quando a falha de alcançar o *deadline* trará consequências catastróficas para o sistema e para o ambiente no qual ele opera.

- **Tarefas Brandas ou Não Críticas (tarefas "soft")**: quando a falha em alcançar o *deadline* não é muito severa ao sistema e ao ambiente, geralmente implicando em perda de desempenho ao sistema.

Outra característica temporal das tarefas está relacionada à regularidade na sua ativação no sistema. De acordo com essa característica, as tarefas podem ser classificadas:

- **Tarefas Periódicas**: a ativação da tarefa se repete após um intervalo de tempo chamado de período em uma sequência infinita.
- **Tarefas Aperiódicas ou Tarefas Assíncronas**: a ativação da tarefa é aleatória e não necessariamente em uma sequência infinita, podendo estar respondendo a um estímulo externo ao sistema.

Ainda dentro de restrições temporárias, as seguintes características são de importância para o escalonamento das tarefas:

- **Tempo de computação ("*Computation Time*")**: tempo total de execução da tarefa.
- **Tempo de início ("*Start Time*")**: tempo corresponde ao instante de início do processamento da tarefa em uma ativação.

- **Tempo de término ("*Completion Time*")**: instante de tempo em que se completa a execução da tarefa.
- **Tempo de chegada ("*Arrival Time*")**: instante em que o escalonador toma conhecimento de uma ativação dessa tarefa. Em tarefas periódicas, o tempo de chegada coincide sempre com o início do período da ativação. As tarefas aperiódicas apresentam o tempo de chegada coincidindo com o tempo da requisição do processamento aperiódico.
- **Tempo de liberação ("*Release Time*")**: o instante de sua inclusão na fila de Pronto (fila de tarefas prontas).

Todos esses tipos de tarefa possuem dois deadlines. O primeiro *deadline* é o absoluto que é um instante no tempo onde a tarefa deve completar a sua execução. O segundo *deadline* é o *deadline* relativo. O *deadline* relativo é a quantidade de tempo após o seu tempo de chegada no qual a tarefa pode ser executar. Ou seja, tempo de chegada somado ao *deadline* relativo deve ser igual ao *deadline* absoluto da tarefa.

Para o conjunto de tarefas existe o conceito de hiperperíodo. Este é um espaço de tempo calculado a partir do mínimo múltiplo comum dos períodos das tarefas periódicas do conjunto a ser escalonado (FARINES et al., 2000). Dentro do hiperperíodo as tarefas apresentarão um comportamento cíclico de execução, pois neste período é garantido que todas as tarefas tenham pelo menos uma ativação. Com isso é possível avaliar se um escalonamento é avaliado somente analisando esse o hiperperíodo de um STR

2.3.2. Relações de Precedência e de Exclusão

Uma relação de precedência entre duas ou mais tarefas pode ser definida como a necessidade de uma tarefa x só poder começar seu processamento após o término do processamento de uma tarefa y (FARINES et al., 2000). Essa necessidade pode ser uma necessidade lógica, onde a ação da tarefa y deve ser executada antes da x, ou pode ser uma necessidade de informação, onde y irá gerar dados que x precisará para sua execução. Esse tipo de relação geralmente é ilustrado através de um grafo acíclico orientado, onde os nós são as tarefas e os arcos descrevem que uma tarefa deve ser precedida pela outra tarefa.

Outra relação que duas ou mais tarefas podem compartilhar é a relação de exclusão. Ela ocorre quando uma tarefa x está utilizando um recurso necessário para a execução de uma seção crítica de uma tarefa y. Desta forma, impede que a tarefa y comece ou continue a sua execução. Ambas as relações impõem restrições aos possíveis escalonamentos que podem ser criados para determinado conjunto de tarefas.

2.4. Algoritmos de Escalonamento

Escalonamentos são definidos como a ordenação das tarefas para sua execução na fila de pronto (FARINES et al. 2000). A fila de Pronto é a fila de tarefas que estão prontas para serem executadas assim que um novo processador estiver disponível. Esse processo é repetido até que todos as tarefas tenham sido executadas.

Os escalonamentos possuem algumas classificações de acordo com algumas características do algoritmo:

- Preemptivos ou não preemptivos: a execução de uma tarefa pode ser interrompida para executar uma tarefa mais prioritária.
- Escalonamento Estático: as tarefas recebem sua prioridade somente uma vez, e essa prioridade não muda durante toda a execução
- Escalonamento Dinâmico: atribui uma nova prioridade para toda ativação de uma instância de uma tarefa. Com isso a tarefa somente receberá sua prioridade quando a mesma entrar na fila de prontas para executar.
- Escalonamento Off-line: escalonamento gerado em tempo de projeto.
- Escalonamento Online: é gerado em tempo de execução.

Para os escalonamentos é possível medir a taxa de utilização do conjunto de tarefas. Para calcular isso é necessário calcular a utilização de cada tarefa que serve como uma medida da ocupação do processador pela mesma, conforme a equação:

$$U_i = \frac{C_i}{P_i} \ se\ a\ tarefa\ \text{é}\ periodica$$

$$U_i = \frac{C_i}{Min_i} \ se\ a\ tarefa\ \text{é}\ aperiodica$$

Onde U_i,C_i, P_ie Min_i são respectivamente a taxa de utilização, o tempo máximo de computação, o período e o intervalo mínimo entre requisições da tarefa T_i (FARINES et al., 2000). A taxa de utilização total para o escalonamento será a soma da utilização das tarefas dividida pelo número total de processadores disponíveis.

2.4.1. Escalonamento de Sistemas de Tempo Real

Segundo (FARINES et al., 2000) o principal problema de um STR consiste em garantir a previsibilidade da execução das tarefas do sistema dado um conjunto de recursos limitados. Dada esta definição, a definição da ordem de execução das tarefas (escalonamento das tarefas) é a principal questão para o sucesso do STR. No escalonamento para STR a principal preocupação é definir a ordem de execução das tarefas segundo determinada lógica que garanta as restrições temporais.

Existem algoritmos considerados ótimos para o escalonamento de tarefas com as seguintes características:

- Todas as tarefas são periódicas;
- É conhecido a priori os tempos de chagada para todas as tarefas;
- É conhecida a carga computacional para execução das tarefas.

Para todos esses algoritmos é utilizado um esquema dirigido a prioridade. A prioridade indica a ordem na qual as tarefas serão executadas pelos processadores. As prioridades são atribuídas pelos aspectos temporais das tarefas e não pela a importância ou o pelo grau de confiabilidade da tarefa.

2.4.2. Escalonamento Taxa Monotônica - RM

O escalonamento RM é um escalonamento dito ótimo entre os escalonadores de prioridade fixa para o seu problema (FARINES et al., 2000). O cenário no qual o RM é aplicado possui as seguintes características:

- As tarefas são periódicas e independentes;
- O "*deadline*" de cada tarefa coincide com o seu período;
- O tempo de computação de cada tarefa é conhecido e constante ("Worst Case Computation Time");
- O tempo de chaveamento entre tarefas é assumido como nulo.

O RM atribui a prioridade de cada tarefa de acordo com o seu período. Todas as tarefas são ordenadas por seus períodos. Quanto maior o período da tarefa menor é a sua prioridade no escalonamento. Como os períodos das tarefas não mudam durante a execução, o RM é considerado um escalonamento estático (FARINES et al., 2000).

2.4.3. Escalonamento *Earliest Deadline First* - EDF

Assim como o escalonamento RM, o escalonamento EDF também é considerado ótimo para a classe de problema que ele resolve. As características do cenário que o EDF atende são idênticas aquelas do cenário do RM.

No EDF as tarefas recebem sua prioridade em relação ao seu *deadline* absoluto. Quanto menor o seu *deadline* maior será a prioridade atribuída aquela tarefa. Diferente do RM que é um escalonamento estático o EDF é um escalonamento dinâmico. Isso é necessário no EDF pois o *deadline* absoluto somente pode ser calculado na ativação de uma nova instancia desta (FARINES et al., 2000).

2.4.4. Escalonamento *Deadline* Monotônico - DM

O escalonamento DM é uma extensão do RM. No RM os valores de *deadlines* relativos são limitados pelo período da tarefa, o que para muitas aplicações pode ser considerado muito restritivo (FARINES et al., 2000). As características do cenário em que o DM é utilizado são idênticas ao cenário do RM, com exceção da restrição do *deadline* ser igual ao período da tarefa. No DM o *deadline* poder ser igual ou menor do que o período da tarefa. No DM, ao contrário do RM, as tarefas recebem sua prioridade de acordo com o valor do seu *deadline* relativo. Com isso, quanto menor o seu *deadline* relativo maior será a prioridade dada a essa tarefa.

2.5. Considerações Finais

Neste capítulo foi apresentado todo o embasamento teórico utilizado no desenvolvimento deste trabalho. Na primeira parte foi definido o que é um AG e detalhadas todas as partes que o compõem. Isso é necessário para a compreensão da técnica utilizada neste trabalho. Todos esses conceitos, assim como a suas utilizações no algoritmo, são demonstrados nas seções 4.6 até 4.14.2

Na segunda e terceira parte, foi definido o que caracteriza um STR, e foram apresentados os algoritmos clássicos para escalonamento de um conjunto de tarefas periódicas. Todos esses são utilizados nas seções 4.2, 4.4, 4.5 e 4.15, e também durante todo o capítulo 5, para definir o problema que este trabalho resolve e como os cenários de testes foram criados para validar o AG.

3. Trabalhos Relacionados

3.1. Introdução

Neste capítulo são apresentados de forma resumida os trabalhos relacionados ao presente trabalho. O principal objetivo deste capítulo é discutir o estado da arte e analisar a forma como ele abordaram o problema em busca da solução. Muitas das técnicas e ideias aqui apresentadas foram utilizadas no AG desenvolvido com o intuito de reproduzir os efeitos que contribuem na resolução do problema.

3.2. Scheduling algorithm for real-time tasks using multi objective hybrid genetic algorithm in heterogeneous multiprocessors system

Em (YOO e MITSUO, 2007) foi desenvolvido um AG para a geração de escalonamentos de tarefas para um STR multiprocessador. Em conjunto com o AG foi utilizado a técnica de *Simulated Annealing* (SA) e a abordagem de Pesos adaptativos (AWA). O objetivo do AG desenvolvido é escalonar um conjunto de tarefas soft em processadores com diferentes capacidades de processamento para diminuir o atraso total das tarefas e o tempo total na execução do escalonamento.

Para o cenário deste trabalho, a relação de precedência é feita por um grafo acíclico das tarefas. O conjunto de tarefas possui 3 características importantes: uma tarefa não pode ser substituída por outra durante a sua execução em um processador; um processador executa somente uma tarefa; e uma tarefa é executada somente por um processador. Para toda tarefa é conhecido o conjunto de tarefas predecessoras e sucessoras, *deadline* e tempo de execução em tempo de projeto. Toda tarefa possui o menor instante possível no qual ela pode começar a executar. Todas essas tarefas são escalonadas em um conjunto de processadores heterogêneos.

Para a representação do cromossomo no algoritmo de (YOO e MITSUO 2007) foi utilizado um *array* de genes em cromossomo permutacional. O gene é codificado como uma tupla de chave e valor. O valor da chave é um valor numérico que representa unicamente uma tarefa dentro do conjunto de tarefas a ser escalonado. O valor da tupla também é um valor numérico que define em qual processador a tarefa definida na chave será alocada. A tupla no gene define como as tarefas são alocados nos processadores. Para a definição da ordem de execução das tarefas é utilizada a ordem a qual as mesmas aparecem no cromossomo.

Para avaliação da qualidade dos indivíduos são utilizadas duas funções para cada um dos objetivos do escalonamento. A primeira função calcula o tempo total de execução dos escalonamentos. A segunda função calcula a soma do tempo que uma tarefa leva para terminar a sua execução depois de ter ultrapassado o seu *deadline*. Para isso é calculado a diferença entre o tempo de execução de cada tarefa e seu *deadline*. Caso o valor da diferença seja menor que 0, o valor é substituído por 0 na soma da função.

Para consolidar as duas funções no *fitness* foi utilizado o AWA. Para ambas as avaliações são encontrados os menores e maiores valores dentro da população. Então cada avaliação é multiplicada por um peso, que é a razão inversa da diferença do maior valor com o menor valor para aquela avaliação dentro da população. Isso é feito para ter diferentes pesos para cada avaliação à cada geração

Para a geração da população inicial foi utilizado um algoritmo que respeita a relação de precedência entre as tarefas. Esse algoritmo começa alocando em processadores aleatórios todas as tarefas que não possuem tarefas antecessoras. Essas tarefas são ordenadas de forma aleatória no cromossomo. O algoritmo então repete este processo para todas as tarefas sucessoras das tarefas já escalonadas até escalonar todas as tarefas.

Para seleção dos cromossomos foi utilizado o método da roleta viciada. Como operador de *crossover* foi utilizado o operador de um ponto de corte customizado. Durante a operação de *crossover* a parte das chaves são separadas dos valores de todos os genes nos cromossomos dos pais. O *array* dos valores que foi gerado por esse processo é submetido ao operador clássico de *crossover* de um ponto de corte. No final as duas partes são combinadas novamente. Para o operador de mutação foi utilizado o operador clássico de mutação, onde cade gene tem uma chance de ser modificado aleatoriamente.

Para a substituição da população entres as gerações no AG foi utilizado o algoritmo SA como algoritmo de decisão para substituir os pais pelos filhos. O SA é um algoritmo de otimização local inspirado nas leis da termodinâmica para o cozimento de sólidos. No SA a função *fitness* é utilizada em conjunto com Fator de Boltzmann (função clássica da termodinâmica utilizada nas SAs) e uma constante chamada temperatura para decidir se os filhos devem substituir seus pais na próxima população. A cada geração, a temperatura é reduzida de acordo com a taxa de resfriamento. Isto faz com que nas primeiras gerações, exista uma maior aleatoriedade na substituição dos pais pelos filhos e, nas gerações finais, o SA tende a escolher o cromossomo com melhor *fitness*. O SA foi utilizado para aumentar a convergência no AG.

Este AG não foi validado em nenhum cenário real. Todos os seus testes foram realizados comparando o AG criado neste trabalho com três outras soluções. Os testes deste trabalho tinham como critério de parada um número máximo de gerações. Foi utilizado uma probabilidade de *crossover* de 70% e uma probabilidade de mutação de 30%. O tamanho da população é de 30 indivíduos. Foram gerados 3 cenários de teste com um número de 10,50 e 100 de tarefas a serem escalonadas. Foi utilizado o algoritmo *P Method* que se baseia em uma distribuição normal de uma matriz adjacente criada a partir do grafo de escalonamento. O tempo de computação e o *deadline* foram gerados de forma aleatória utilizando uma distribuição normal e outra exponencial. Para todos os 3 cenários foi verificado que AG pode obter um resultado um pouco melhor do que as outras 3 soluções utilizadas nos testes.

3.3. Real-Time Reconfigurable Scheduling of Multiprocessor Embedded Systems Using Hybrid Genetic Based Approach

Em (GHARSELLAOUI, 2015) foi desenvolvido um AG com objetivo resolver o problema do agendamento de tarefas soft em um sistema multiprocessador de tempo real utilizando uma abordagem híbrida de AG e pesquisa tabu em um sistema que pode ser reconfigurado. Essa reconfiguração adiciona ou altera as tarefas no sistema sendo necessário gerar um novo escalonamento válido.

O cenário, no qual (GHARSELLAOUI, 2015) aplicou, considera um STR que possui um conjunto N de tarefas periódicas, aperiódicas e esporádicas. A este conjunto de N tarefas é adicionado um novo conjunto de M de tarefas. O novo conjunto formado dessa soma sofre as reconfigurações necessárias para se adaptar a o cenário onde o mesmo será executado.

Para avaliar que se o novo conjunto de tarefa é escalável para sistema após a reconfiguração é utilizado uma função que calcula a soma das diferenças entre o *deadline* e o término de execução para cada tarefa. Para que o escalonamento seja considerado válido o valor dessa função precisa ser maior ou igual a zero. Essa avaliação é feita dentro de um intervalo de tempo chamado hiperperíodo. É considerado que as tarefas aperiódicas e esporádicas terem somente uma ativação dentro desse período neste trabalho.

A representação para o cromossomo deste AG é uma lista encadeada de genes. Todos os genes contêm um par constituído de uma String e um valor inteiro. O valor da String identifica a tarefa a ser escalonada. O valor inteiro representa o tempo de chegada de uma instância da tarefa a ser processada. O tamanho do cromossomo é definido pela quantidade de ativações de todas as tarefas dentro do hiperperíodo.

Todo gene no cromossomo precisa respeitar 2 restrições para que o mesmo seja válido. A primeira é referente a um tempo mínimo entre a chegada de 2 ativações de uma mesma tarefa. Esse valor mínimo entre cada instância é definido para cada tarefa em tempo de projeto. A segunda é referente a um tempo de chegada máximo para uma instância. Para facilitar a manipulação do cromossomo, todos os genes com ativações da mesma tarefa são ordenados de forma crescente de acordo com o seu *deadline* em uma mesma parte da lista.

A função *fitness* é a soma da diferença do *deadline* e do término de execução de cada tarefa. Quando o *fitness* apresenta valores maiores que zero há a garantia de que todas as tarefas executaram dentro de seus *deadline*s e algumas tarefas tiveram folgas no término de suas execuções. Então os indivíduos considerados ótimos são os que apresentam valores iguais ou maior que zero para função *fitness*.

A inicialização da população é feita de forma aleatória. Todas as ativações de cada tarefa dentro do hiperperíodo são adicionadas com valores aleatórias de tempo de chegada. Para garantir a segunda restrição o hiperperíodo é calculado para o conjunto a ser escalonado antes de atribuir os valores aleatórios. Quando todos os genes recebem seus valores de chegada,

os genes das mesmas tarefas são agrupados e ordenados de forma crescente de acordo com os seus *deadline*s e são adicionados a uma lista encadeada.

O tamanho da população utilizado neste trabalho é de 20 indivíduos. O modulo da população utilizado neste trabalho é o *steady state*. Esse algoritmo executa por 50 gerações ou até encontrar a primeira solução com um *fitness* igual ou maior que zero.

Para a seleção dos cromossomos a serem utilizados pelos operadores genéticos foi escolhido o método da roleta viciada. Como operador de *crossover* foi selecionado o *crossover* de 2 pontos de corte. Após o *crossover,* todos genes em cada filho são agrupados e ordenados assim como foi feito na inicialização da população.

Neste trabalho o operador de mutação clássico foi substituído pelo algoritmo de pesquisa tabu. Ele é um algoritmo de busca local onde, partindo-se de uma solução conhecida para um problema, é avaliado se os vizinhos locais representam uma solução melhor para o problema. Utilizando os genes anteriores e sucessores, é calculado um novo valor para o gene utilizando a busca Tabu.

Para testar o algoritmo desenvolvido neste trabalho foram executados 1000 testes, onde ele é comparado com o algoritmo desenvolvido no trabalho a qual este se baseia. A Tabela 1 é mostra os resultados obtidos nas 1000 execuções. Não é mencionado se o algoritmo foi testado em algum cenário real.

Tabela 1 – Resultado da simulação (GHARSELLAOUI, 2015)

Abordagem	Pior Resultado	Resultado Médio	Melhor Resultado
Algoritmo hibrido	$1.020*10^{-8}$	$1.017*10^{-8}$	$1.010*10^{-8}$
Algoritmo clássico	$1.028*10^{-8}$	$1.090*10^{-8}$	$1.020*10^{-8}$

3.4. Permutational genetic algorithm for the optimized assignment of priorities to tasks and messages in distributed real-time systems

(AZKETA, 2011) apresenta um AG que tem como objetivo a definição de prioridades para tarefas e mensagens em um sistema distribuído de tempo real. Esse escalonamento é feito com prioridades fixas definidas pelo AG.

O sistema distribuído onde as tarefas e mensagens são escalonados neste trabalho é composto de P processadores, O escalonador usa prioridades fixa. Os processadores estão conectados por N redes de comunicação. Este sistema executa A transações que possuem um período de ativação e um *deadline*. Estas A transações possuem T tarefas que executam com o pior tempo de execução. Estas T tarefas possuem o mesmo período de ativação da transação a qual pertencem e um *deadline* específico. As tarefas podem ter relações de precedências entre si. Essas tarefas trocam M mensagens. Se essas tarefas estão executando no mesmo

processador, as trocas destas mensagens ocorrem por um mecanismo de memória compartilhada. Caso executem em processadores diferentes as mensagens são enviadas por uma das N redes de comunicação.

O cromossomo usado neste por (AZKETA, 2011) é um *array* de genes. Esses genes possuem um campo Code que identifica unicamente uma tarefa ou mensagem. O outro campo é o Value que mapeia a tarefa do gene em um processador ou rede de comunicação. Dessa forma o gene representa a alocação de uma tarefa ou mensagem em um processador ou rede. O cromossomo neste trabalho é um cromossomo permutacional onde dos genes define a sua prioridade no escalonamento.

Para avaliação dos indivíduos gerados pelo AG é utilizado uma função chamada "scheduling index factor" que verifica quantas transações alcançaram seus *deadline*s e o quão bem o fizeram. Essa função é a soma da diferença entre o *deadline* e o pior tempo de resposta de todas as transações. Se essa função possui um valor menor que zero isso, então algumas transações não alcançaram os seus *deadline*s. Um valor igual a zero indica que todas as transações executaram exatamente dentro dos seus *deadline*s. Quando o valor é maior do que zero, as transações executaram antes do seu *deadline*. Para o cálculo do pior tempo de resposta é utilizado o MAST, é uma ferramenta de análise holística de escalabilidade. Para esse trabalho foi utilizado a análise offset based.

Para a inicialização da população é utilizada um algoritmo para atribuição otimizada de prioridade chamado o HOPA. Para utilizar essa heurística na geração da população inicial o sistema define de forma aleatória duas constantes. A primeira constante Ka indica o quanto os recursos (processadores e redes de comunicação) afetam a atribuição dos *deadline*s das tarefas. De forma análoga a segunda constante Kr mede o quanto as próprias tarefas e mensagens afetam os *deadline*s das tarefas.

Como método de seleção foi utilizado o torneio com 2 participantes. Como operador de *crossover* foi utilizado o operador OX3. Como operador de mutação foi utilizado o *Insertion* Mutation (ISM). No operado ISM quando o gene é escolhido para sofrer mutação o mesmo tem a sua posição dentro do cromossomo alterado de forma aleatória. O operador de mutação neste trabalho é aplicado sobre cada gene.

O último operador a ser aplicado é o *Clustering*. Cujo objetivo é agrupar todos os genes referente a um mesmo processador ou rede de comunicação. Isso cria bloco contínuos referentes a um mesmo recurso do sistema que durante a execução do operador de *crossover* aumenta a probabilidade que as características sejam passadas juntas para os filhos gerados.

Neste AG foi utilizado elitismo de um indivíduo. O AG tem como critério de parada o tempo de execução de 1800 segundos. Na Tabela 2 é mostrada a configuração utilizada para execução dos testes para este AG.

Tabela 2 – Configuração do Algoritmo (AZKETA, 2011)

Tamanho da população	50 Indivíduos
K_a	[3,5]
K_r	(0,5]
Probabilidade de Crossover	80%
Probabilidade de mutação	0.1%
Elitismo	1Indivíduo
Tempo máximo de execução	1800 segundos

O algoritmo proposto neste trabalho não foi aplicado em um cenário real. O ambiente foi simulado com 8 processadores e 3 redes de comunicação. Foram utilizados um modelo logico com 7 transações contendo 50 tarefas com 43 mensagens no total. A cada experimento, a carga do sistema era aumentada e os resultados obtidos eram comparados com os resultados do HOPA.

3.5. Genetic Approach for Real-Time Scheduling on Multiprocessor Systems

(SEBESTYEN e HANGAN, 2012) propõem o uso de AG para gerar escalonamentos para sistemas multiprocessadores de tempo real. O objetivo é encontrar um escalonamento que respeite todas as restrições temporais impostas para um conjunto de tarefas e mensagens. As tarefas neste trabalho são do tipo hard real time. Para alcançar esse objetivo, ele trabalha com dois aspectos de um escalonamento para STR: a alocação das tarefas nos nós de processamento e os *deadline*s intermediários das tarefas.

Todas as tarefas e mensagens são agrupadas em uma lista encadeada que representam transações que devem ser processadas de forma periódica. Toda tarefa possui um tempo de execução C que deve ser executado dentro de um *deadline* intermediário D. Cada tarefa também possui uma lista de processadores preferenciais a qual a mesma pode executar. Neste trabalho as mensagens são consideradas um tipo especial de tarefa que são executadas pelas redes de comunicação.

As tarefas são alocadas em um conjunto P de processadores que são interconectados por um conjunto N de redes de comunicação por onde as mensagens são enviadas. Todos os P processadores possuem escalonadores que implementa o algoritmo EDF.

O cromossomo é representado por um *array* de genes. Cada gene representa uma das duas características de uma tarefa: *deadline* ou alocação em um processador/rede. Para a alocação é usado um valor literal que identifica um processador ou rede. Para o *deadline* é usado um valor inteiro que representa a unidades de tempo, sendo esse valor o *deadline* intermediário da tarefa. O cromossomo possui um par de genes para cada tarefa ou mensagem. O primeiro

valor é referente a alocação e o segundo ao *deadline*. A ordem a qual as tarefas aparecem no cromossomo é a ordem das transações e dentro das mesmas a ordem das tarefas e mensagens.

Para avaliação dos indivíduos gerados é utilizado uma função composta de 4 avaliações. A primeira avaliação é a soma da diferença entre o tempo de resposta e o *deadline* de todas as transações que não executaram dentro do seu *deadline*. A segunda avaliação é idêntica à primeira com a diferença que todas as transações são consideradas no cálculo. A terceira avaliação é a soma da diferença entre o tempo de resposta e o *deadline* intermediários de todas as mensagens e tarefas no escalonamento. A quarta e última avaliação é a uniformidade na utilização dos processadores. Para esta avaliação é somada as diferenças entre o fator de utilização dos processadores e a fator de utilização médio do sistema. Todas estas avaliações são somadas com fatores de peso. A primeira avaliação tem o peso de 100. A quarta avaliação tem o peso de 100 e a soma da segunda e terceira avaliação tem o peso de 1 na soma total do *fitness*. Quanto menor o valor nestas quatro avaliações melhor o indivíduo.

Para geração da população inicial foi utilizado um algoritmo de 2 passos. No primeiro passo é utilizado uma heurística de alocação como First fit ou Round Robin para alocar as tarefas nos processadores. Ainda neste passo é atribuído o menor valor possível para os *deadline*s intermediários das tarefas e mensagens. No segundo passo um indivíduo é selecionado e copiado. Esta cópia sofre mutações sucessivas e depois é adicionado à população. O tamanho da população é de 60 indivíduos. Foi utilizado o elitismo na população passando os melhores indivíduos para a próxima população.

Para a seleção de indivíduos foi utilizado o torneio de 2 indivíduos sem repetição. O operador *crossover* de 2 pontos de cortes foi utilizado para. Caso os pontos de cortes coincidam ou um deles seja o extremo do cromossomo novos pontos de corte são gerados. Este operador foi utilizado com uma probabilidade de 70% neste trabalho. Também foi utulizado utilizado a técnica de elitismo para passar os melhores indivíduos para a aproxima geração.

Como operador de mutação foi utilizado uma versão customizada do operador clássico. Na versão customizada, o valor utilizado para alterar o gene vem de um subconjunto definido a partir de um intervalo gerado a partir do valor atual. Neste trabalho o intervalo é gerado com os valores de 25% e 50%. O operador de mutação é aplicado com 1% para cromossomos pequenos com no máximo 30 genes e 0.3% para cromossomos para cromossomos maiores. O operador de mutação pode ser aplicado uma segunda vez na população caso seja identificado que a diversidade genética é menor que um valor limite não especificado.

(SEBESTYEN e HANGAN, 2012) não validarão o algoritmo em um cenário real. Para validar o AG criado foram desenvolvidas duas ferramentas. A primeira ferreamente é o RTMultSim, um simulador para o escalonamento e execução de STR para multiprocessadores. A ferramenta executa uma simulação do escalonamento pelo dobro do tempo do hiperperíodo. A segunda ferramenta é um gerador de conjuntos de tarefas. Este conjunto é gerado com um fator de utilização médio como parâmetro. A ferramenta gera este conjunto de tarefas com um fator aleatório de utilização para cada tarefa.

Foram criados dois cenários de teste para validação do algoritmo. O primeiro cenário possui somente tarefas independentes, sem relação de precedência. Neste cenário, 20 tarefas eram escalonadas com 4,6 e 8 processadores com um total de utilização de 90%. O algoritmo rodava por cerca de 90 gerações. Foram encontrados escalonamentos que quase respeitavam suas restrições temporais. O segundo cenário foi um conjunto de 12 transações a serem escalonadas em 8 processadores distribuídos. Neste cenário o AG foi executado por cerca de 300 gerações. Novamente o AG encontrou escalonamentos que são muito próximo de um escalonamento factível.

3.6. An improved Hybrid Quantum-Inspired Genetic Algorithm (HQIGA) for scheduling of real-time task in multiprocessor system

O trabalho de (KONAR, 2017) teve como objetivo o desenvolvimento de um AG capaz de criar um escalonamento de tarefas com restrições temporais em um sistema multiprocessador. No contexto do trabalho, um conjunto m de tarefas precisa ser executada por um conjunto n de processadores. Toda a tarefa possui um tempo de chegada onde a mesma é incluída em uma fila para ser executada, um pior tempo de execução, um limite de tempo para ser executada (*deadline*), um instante onde ela começa a ser executada e outro onde ela termina sua execução. O algoritmo deve selecionar entre as tarefas que estão na fila para serem executadas e decidir em qual processador elas serão executadas de forma que todas respeitem seus respectivos *deadline*s.

Todas as tarefas da fila de pronto para execução são ordenadas utilizando o algoritmo EDF que coloca as tarefas como menor *deadline* no início da fila. Caso duas ou mais tarefas possuam o mesmo *deadline,* é utilizado o algoritmo shortest computation first (SCTF) que dá preferência à tarefa com o menor tempo computacional. Com fila inicial o algoritmo seleciona n tarefas, onde n é o número de processadores disponíveis ou o número de tarefas restantes (caso esse seja menor do que o número de processadores), para escalonar. Depois que ele escalona essas n tarefas, ele seleciona as próximas n tarefas e repete o processo. Esse algoritmo é repetido até que todas as tarefas sejam escalonadas nos processadores disponíveis.

A alocação das tarefas selecionadas da fila de pronta é feita no cromossomo. O cromossomo é um *array* de n inteiros onde n é quantidade de tarefas selecionadas. O valor do *array* indica a tarefa a ser escalonada e o índice é o processador onde aquela tarefa será escalonada. É importante notar que o tamanho do cromossomo é variado, pois pode acontecer que haja uma quantidade menor de tarefas na fila se comparado com a quantidade de processadores. O cromossomo foi desenvolvido utilizando qubits, a unidade básica de informação no campo da computação quântica.

Este AG não utiliza nenhum dos operadores genéticos clássicos (Mutação e *Crossover)*. Para gerar uma nova população, é utilizado a técnica *rotation gate* da computação quântica no melhor indivíduo da atual população. Foi escolhida essa técnica porque ela apresenta uma melhora na performance da execução do AG. Ela também gera os novos indivíduos somente utilizando o atual melhor indivíduo, assim aumentado a convergência do algoritmo.

Para o *fitness* foram utilizadas duas funções de avaliações. A primeira função avalia se no escalonamento gerado pelo cromossomo todas as tarefas irão executar dentro dos seus *deadline*s. Caso alguma tarefa falhe nesse quesito, todo escalonamento é considerado inválido e o *fitness* desse indivíduo será igual a zero. A segunda função somente é avaliada se a primeira não recebeu o valor zero. A segunda função avalia se a taxa de utilização de cada processador possui valores similares. O objetivo é manter todos os processadores ocupados pela maior quantidade de tempo possível. Isso também evita que um processador tenha muitas tarefas para executar enquanto um segundo processador está muito ocioso. O AG tem como critério de parada o número de 100 gerações com 30 indivíduos em cada geração.

Esse AG não foi válidado em um cenário real. Foram criados 10 cenários de testes com uma quantidade variada de tarefas onde esse algoritmo foi comparado com outros 2 AG`s que não utilizam a computação quântica. O algoritmo criado teve um resultado bastante similar quando comparados aos outros dois algoritmos.

3.7. Análise dos trabalhos relacionados

Em todos os trabalhos relacionados é verifica-se que a representação mais comum para o cromossomo para este problema é um *array* onde um dos valores representa o processador e outro representa a tarefa ou uma característica temporal. Geralmente o cromossomo também é permutacional, onde a ordem dos genes é utilizada para indicar a prioridade das tarefas no escalonamento. Seguindo esse padrão, o presente trabalho também utiliza um cromossomo permutacional onde os genes representam as tarefas e suas posições representam no cromossomo a prioridade que elas possuem no escalonamento. A diferença é que para este cromossomo uma tarefa não é alocada para um processador e sim para um nó, que possui N processadores.

Para a avaliação do cromossomo o fato de uma tarefa executar ou não dentro do escalonamento é tratado como uma penalidade no *fitness*. Algumas características, como a taxa de utilização, também foram utilizadas na avaliação. Neste trabalho é usada uma abordagem similar à (AZKETA, 2011), onde uma função contabiliza quantas tarefas não alcançaram seus *deadline*s e o tempo de folga total do escalonamento. Como diferencial, foi preciso calcular a mesma função para todos os nós que temos no sistema. Caso exista um nó que possua pelo menos uma tarefa que não executou dentro do seu *deadline,* todo o escalonamento é considerado inválido.

Trabalhos como (SEBESTYEN e HANGAN, 2012), (KONAR, 2017) e (GHARSELLAOUI, 2015) utilizaram o algoritmo de escalonamento EDF, enquanto (AZKETA, 2011), (YOO e MITSUO, 2007) utilizaram um algoritmo de prioridade fixa. Este trabalho é similar ao último grupo, utilizando o algoritmo de prioridade fixa.

Para os operadores genéticos a maioria dos trabalhos utilizaram operadores básicos e bastante conhecidos como: *crossover* de dois pontos e mutação clássica. Alguns trabalhos se destacam por utilizar outras técnicas em conjunto com os seus algoritmos. Em (GHARSELLAOUI, 2015) a busca tabu foi utilizada no lugar do operador de mutação. Em (KONAR, 2017) a

modelagem do cromossomo e os operadores foram feitos com técnicas da computação quântica. (YOO e MITSUO, 2007) utilizou a técnica de SA em conjunto com o seu AG para melhorar a convergência do algoritmo. Para este trabalho foram utilizados operadores voltados para cromossomos permutacionais: OX3 assim como em (AZKETA, 2011) para *crossover* e *Twors* para a mutação.

Para a seleção de indivíduos foi utilizado o mesmo operador dos trabalhos de (Yoo e Mitsu, 2007) e (GHARSELLAOUI, 2015) a roleta viciada. Porém, ela foi utilizada em conjunto a técnica de *Ranking*.

Todos os trabalhos relacionados solucionavam o problema definindo o escalonamento das tarefas nos processadores disponíveis e a prioridade das tarefas no mesmo. Alguns trabalhos como (SEBESTYEN e HANGAN, 2012) não alteravam diretamente a prioridade, e sim alguma característica que fora utilizada para a definição da mesma. (SEBESTYEN e HANGAN, 2012) alteravam os *deadline*s das tarefas para alterar as prioridades utilizando o algoritmo EDF (que privilegia um *deadline* menor). Neste algoritmo, foi trabalhado com a prioridade da tarefa e a sua alocação em um nó específico.

Para os testes das soluções obtidas, dois trabalhos se destacam. (SEBESTYEN E HANGAN, 2012) utilizaram a ferramenta RTMultSim para realizar simulações com os escalonamentos gerados, testando as suas soluções em um cenário mais próximo do cenário real. (Azketa, 2011) utilizou um modelo matemático para verificar se as tarefas alcançavam os seus *deadline*s e as folgas que elas apresentavam em suas execuções. No presente trabalho foi utilizado uma ferramenta de simulação como foi feito em (SEBESTYEN E HANGAN, 2012) para gerar os insumos necessários para o cálculo do *fitness*. Para o cálculo do *fitness* foi utilizado a função "scheduling index factor" apresentada em (Azketa, 2011). A Tabela 3 mostra a comparação dos trabalhos relacionados para com o *Genetic Scheduler*.

Tabela 3 – Comparaçõa dos trabalhos relacionados com o Genetic Scheduler

Trabalho	Cromossomo	Objetivo	Escalonador	Seleção	Crossover	Mutação	Simulação no Fitness
Permutational genetic algorithm for the optimized assignment of priorities to tasks and messages in distributed real-time systems	Array onde o índice representa a tarefa e o valor representa o processador e a posição do generepresenta sua prioridade. Cromossomo permutacional	Definir a prioridades das tarefas para que elas executem dentro dos seus deadlines.	Prioridade Fixa	Torneio de 2 indivíduos	Order Crossover versão 3 (OX3).	ISM	Sim
Genetic Approach for Real-Time Scheduling on Multiprocessor Systems	Array onde o índice representa a tarefa e o valor pode representar processador ou o deadline da tarefa.	Encontrar um escalonamento que respeite todas as restrições temporais do conjunto de tarefas a ser executado	EDF	Torneio de 2 indivíduos	Crossover de 2 pontos de corte	Mutação Clássica	Sim

Real-Time Reconfigurable Scheduling of Multiprocessor Embedded Systems Using Hybrid Genetic Based Approach	Array onde o índice representa a tarefa e o valor representa o tempo de chegada no processador.	Encontrar um escalonamento que respeite todas as restrições temporais do conjunto de tarefas a ser executado depois de uma reconfiguração nos sistemas	EDF	Roleta Viciada	Crossover de 2 pontos de corte	Tabu Search	Não
An improved Hybrid Quantum-Inspired Genetic Algorithm (HQIGA) for scheduling of real-time task in multiprocessor system	Array onde o índice representa a processador e o valor representa a tarefa.	Encontrar um escalonamento que respeite todas as restrições temporais do conjunto de tarefas a ser executado	EDf	Não foi utilizado	Rotation Gate	Não foi utilizado	Não
Scheduling algorithm for real-time tasks using multiobjectivehybrid genetic algorithm in heterogeneous multiprocessors system	Array onde o índice representa a tarefa e o valor representa o processador e a posição do gene representa a ordem que a tarefa será executada	Alocar todas as tarefas para dimuir o atraso de todas as tarefas e o tempo de processamento do escalonamento	Prioridade Fixa	Roleta Viciada	Crossover de 2 pontos de corte	Mutação Clássica	Não
Genetic Scheduler	Array onde o índice representa a tarefa e o valor representa um nó de processamento de um cluster e a posição do gene representa sua prioridade. Cromossomo permutacional	Encontrar um escalonamento válido para um conjunto de tarefas. Este conjunto é executado em um cluster. O Genetic Scheduler deve definir a alocação nos nós de processamento e a priodaridades das tarefas.	Prioridade Fixa	Roleta Viciada e Ranking	Order Crossover versão 3 (OX3).	Twors	Sim

4. Genetic Scheduler

Este trabalho teve como objetivo o desenvolvimento de um AG capaz de gerar escalonamentos válidos para um sistema distribuído de tempo real que fossem avaliados através de simulações. Foi simulado um cluster composto por 2 nós de processamento, cada um contendo 2 de processadores, para executar as tarefas com restrição temporais.

O AG deve escalonar este conjunto de tarefas definindo suas alocações nos processadores e a prioridade que as mesmas executam neles. Para avaliar as soluções produzidas durante o processamento do AG foi utilizada a simulação do processamento dos escalonamentos desenvolvidos.

O principal objetivo do AG é encontrar uma solução que represente um escalonamento válido, ou seja, onde todas as tarefas executem dentro dos seus *deadline*s. O objetivo secundário do AG é fazer com que as tarefas dentro do escalonamento possuam os melhores tempos de folga possíveis em suas execuções. O tempo de folga neste trabalho é definido como a diferença entre o *deadline* e o tempo de execução das tarefas. Um maior o tempo de folga indica que as características temporais dessa tarefa podem ser alteradas sem necessariamente ter que gerar um novo escalonamento. O escalonamento gerado pelo AG define a alocação nos nós e a prioridade de execução de cada tarefa a ser escalonada.

As quantidades de nós de processamento e de processadores são conhecidas a priori. É considerado que o sistema, com o seu conjunto de tarefas, está na etapa de desenvolvimento, ou seja, está sendo definido e validado. Ele é classificado como preemptivo, off-line e estático. Com isso, o escalonamento gerado pelo AG também é estático e será utilizado por um escalonador central no sistema para decidir qual processador executará qual tarefa com uma determinada prioridade. Sendo assim, é suposto que esse sistema sempre receberá o mesmo conjunto de tarefas a cada ciclo de execução.

Nas próximas seções são descritos o fluxo do algoritmo criado e as premissas do algoritmo. Nas seções seguintes é detalhado o conjunto de ferramentas utilizadas. Dando continuidade, é demonstrado com detalhe cada passo da execução do AG. A seção seguinte descreve como é a função *fitness* e como a simulação é utilizada para gerar os dados necessários para o cálculo. A última seção discute como seria o desenvolvimento de um STR utilizando o *Genetic Scheduler*.

4.1. Fluxo do AG

O AG inicializa recebendo um arquivo XML que contém o conjunto de tarefas com suas características temporais. A quantidade de nós de processamento e o número de processadores utilizados são variáveis configuradas no AG. No primeiro passo o algoritmo define quais métodos de seleção, *crossover*, mutação e modulação da população serão utilizados durante a sua execução. Nesta etapa também é definido o modelo de cromossomo e a função *fitness* do AG.

Por último são definidas as probabilidades de *crossover* e mutação e o critério de parada para o AG.

No passo seguinte o AG gera a população inicial utilizando o arquivo XML recebido por parâmetro na inicialização. Essa população tem seu *fitness* avaliado pela função de avaliação. Após todos os indivíduos terem sido avaliados, passam pelo processo de seleção, *crossover*, mutação e então será gerada uma nova população. Este processo é repetido até que o critério de parada tenha sido alcançado. O AG retorna um arquivo de texto contendo a prioridade e alocação nos processadores para cada tarefa no conjunto a ser escalonado. O fluxo completo do algoritmo é demonstrado na Figura 1 seguido de um exemplo do arquivo resultado XML na Figura 2.

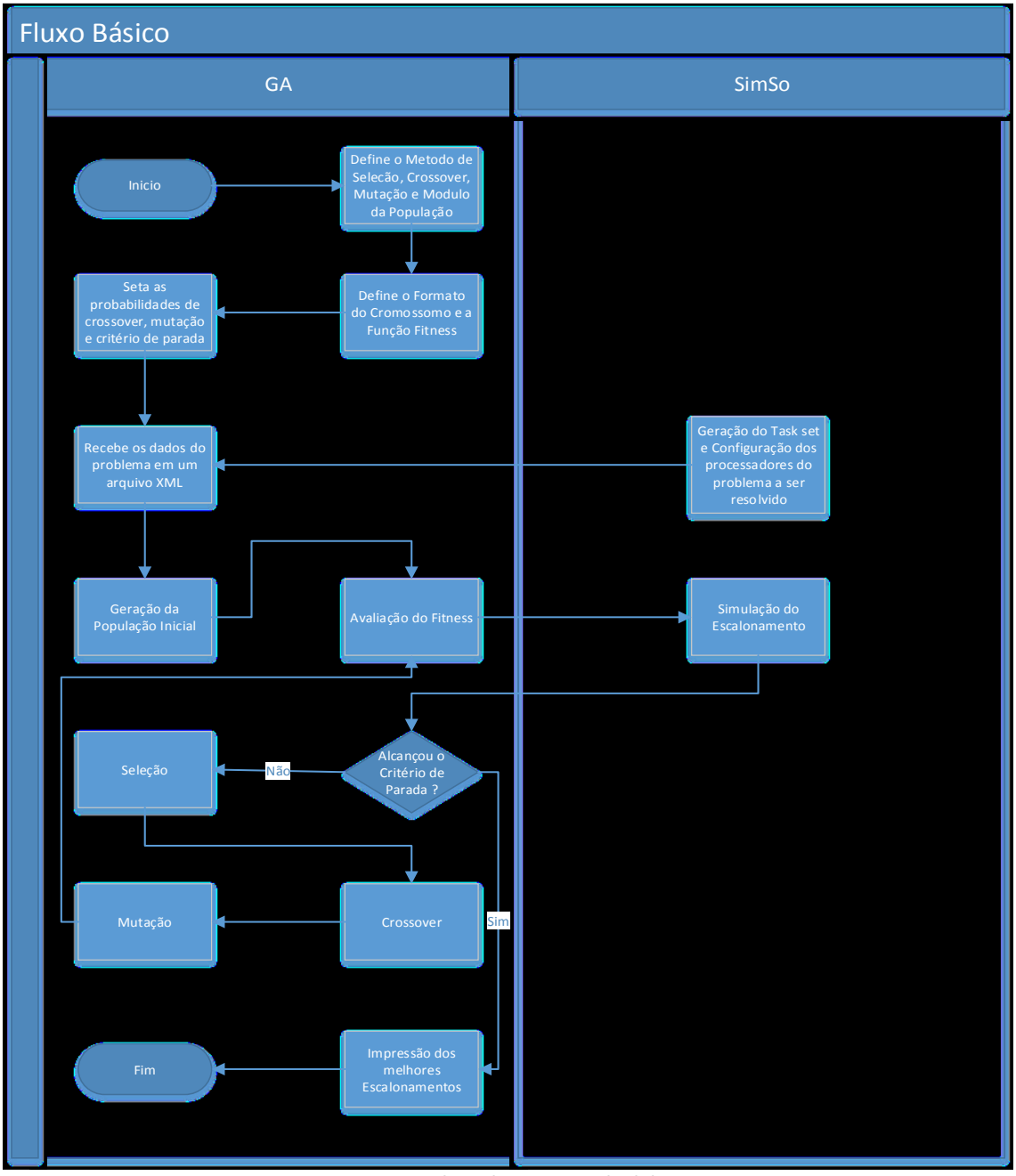


Figura 1 – Fluxo do *Genetic Scheduler*

```
<?xml version="1.0" encoding="UTF-8"?>
<schedule>
  <cluster id="1">
    <processor id="1" />
    <processor id="2" />
    <task id="2" priority="0" />
    <task id="3" priority="3" />
    <task id="6" priority="5" />
  </cluster>
  <cluster id="2">
    <processor id="1" />
    <processor id="2" />
    <task id="1" priority="1" />
    <task id="4" priority="2" />
    <task id="5" priority="4" />
  </cluster>
</schedule>
```

Figura 2 - Exemplo do arquivo de resultado do *Genetic Scheduler*

4.2. Premissas do algoritmo

Para este algoritmo foram assumidas as seguintes premissas:

- As tarefas são periódicas e independentes.
- O "*deadline*" de cada tarefa coincide com o seu período.
- O tempo de computação de cada tarefa é conhecido e constante (“Worst Case Computation Time”).
- O tempo de chaveamento entre tarefas é assumido como nulo.

4.3. Framework para AG Genetic Sharp

Para o desenvolvimento deste trabalho foi utilizada a biblioteca GeneticSharp (GeneticSharp, 2017). Ela é uma biblioteca desenvolvida em C#[1] para facilitar o desenvolvimento de AGs. Nesta biblioteca todo o fluxo básico que um AG precisa já está implementado.

A biblioteca espera receber duas classes básicas para o seu funcionamento. A primeira, é uma classe que herda da classe ChromosomeBase ou implementa a interface IChromosome. Esta classe deve definir como criar o cromossomo para o problema que se deseja resolver recebendo como parâmetro o tamanho desejado para o cromossomo.

A segunda classe que a biblioteca espera é uma classe que implementa a interface IFitness. Esta classe deve definir o mecanismo para o cálculo do *fitness* para o objeto IChromosome, recebido como parâmetro da função Evaluate. O diagrama de classes é mostrado na Figura 3.

[1] C# é uma linguagem de programação orientada a objetos que faz parte do framework .NET, ambos desenvolvido pela Microsoft.

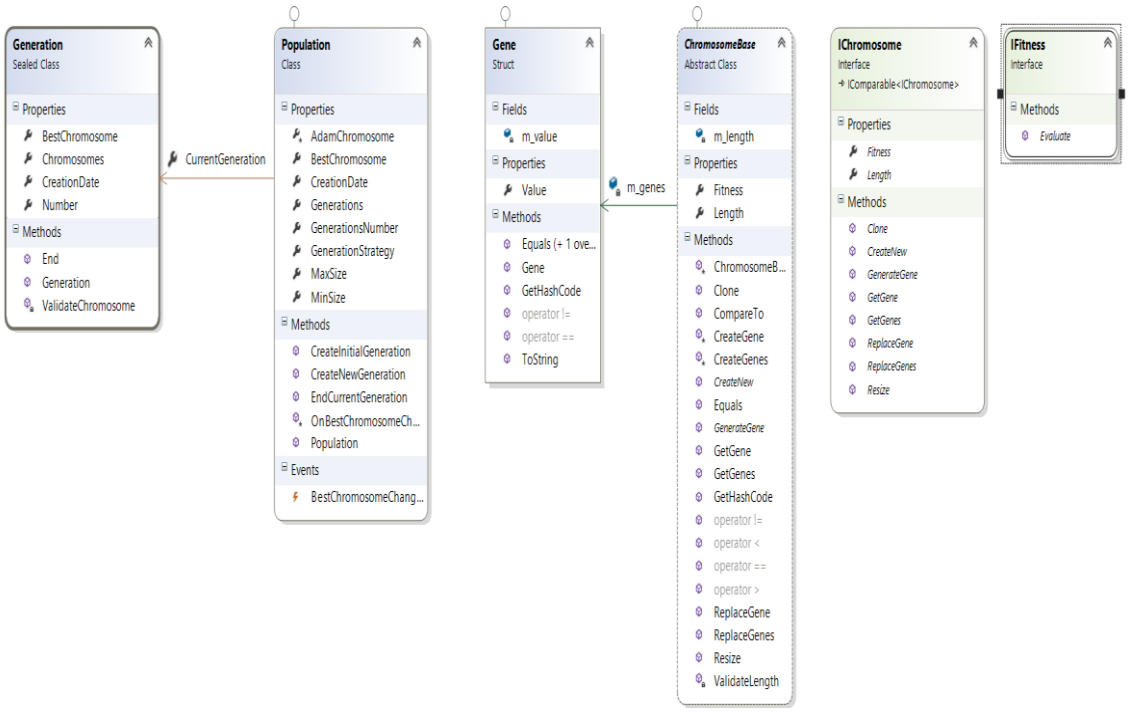


Figura 3 – Diagrama de classe UML das classes bases do Genetic Sharp

Com ambas as classes que representa o cromossomo e o *fitness* criadas, o próximo passo é definir os operadores de seleção, *crossover*, mutação e reinserção no começo da execução. Para todos esses operadores a biblioteca já possui muitos métodos clássicos implementados. Caso seja necessário, é possível implementar novos métodos para todos esses operadores através de interfaces especificas para cada um. A Figura 4 mostra as classes dos operadores genéticos por meio de um diagrama de classes.

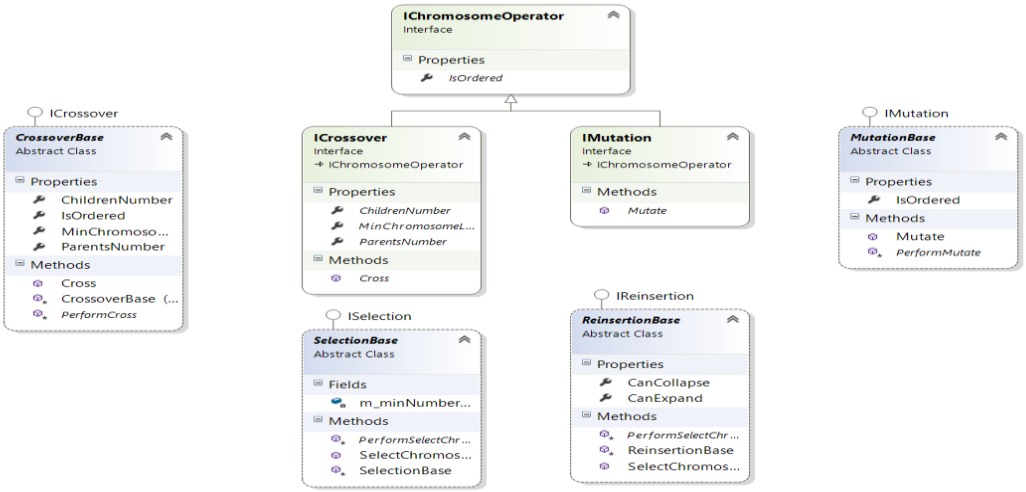


Figura 4 – Diagrama de classe UML das classes dos operadores genéticos

Por último, é necessário definir o critério de parada do AG. A biblioteca já disponibiliza vários métodos clássicos, como número máximo de gerações. Caso seja necessário, podem ser

implementados novos métodos criando uma classe que herde de TerminationBase ou implemente a interface ITermination. É possível criar uma composição de critérios de paradas AndTermination (classe utilizada para quando ambos os critérios devem ser atendidos) e OrTermination (classe utilizada para quando é necessário somente um critério ser atendido) para critérios de paradas mais complexos. Na Figura 5 temos o último diagrama de classes para as classes do critério de parada.

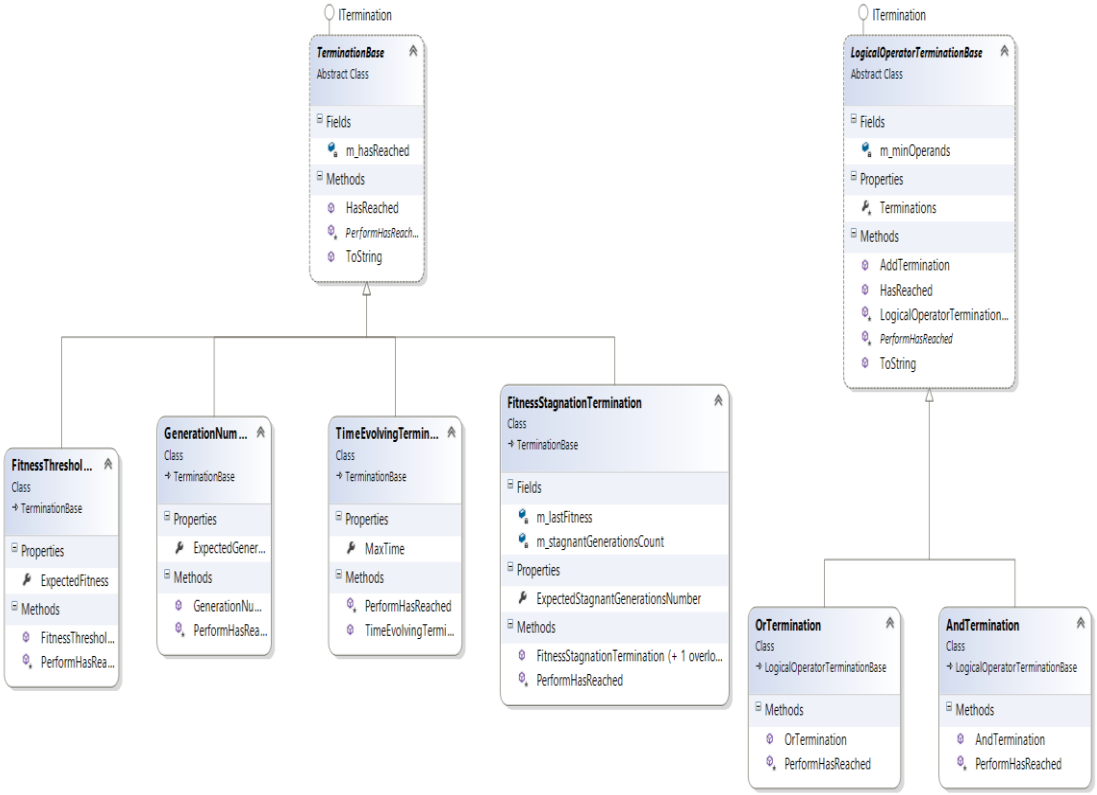


Figura 5 – Diagrama de classe UML das classes do critério de parada

4.4. Geração de Dados para o *Fitness* utilizando SimSo

Pela necessidade do AG avaliar os escalonamentos gerados pelo processo evolutivo através da função *fitness,* foi utilizado a ferramenta SimSo (SimSo, 2017). Nesta ferramenta são executadas simulações de escalonamentos para STR em um ambiente distribuído. Estas simulações são feitas com o objetivo de gerar dados a serem utilizados no cálculo do *fitness*.

O modelo que é submetido para a simulação no SimSo considera um conjunto de tarefas N que precisam ser alocadas em M nós de processamento. Todas as tarefas que pertencem ao conjunto N contêm as seguintes propriedades:

- **Identifier:** um identificador numérico que é único entre todas as tarefas que serão executadas dentro de uma simulação especifica. Essa propriedade identifica a tarefa de forma única dentro de uma simulação.

- ***Task Type*:** todas as tarefas possuem um tipo específico. Existem 3 tipos de tarefas no SimSo: periódicas, aperiódicas e esporádicas. Todas as tarefas são do tipo periódica, ou seja, possuem uma nova ativação depois de um período específico.
- ***Period*:** um tempo específico definido em ms onde uma nova ativação da tarefa acontece. Essa é uma propriedade especifica das tarefas do tipo periódica. Essa propriedade é gerada pelo SimSo na geração do *task set*.
- ***Deadline*:** um tempo definido em ms que determina o limite de tempo que uma tarefa pode executar. Caso a tarefa não consiga terminar a sua execução dentro do *deadline* a mesma é abortada e considerada como falta na simulação. Essa propriedade é gerada pelo SimSo na geração do *task set*.
- ***Worst Execution Time (WCET)*:** um tempo definido também em ms que representa o pior tempo de execução da tarefa em um processador. Essa propriedade é gerada pelo SimSo na geração do conjunto.
- ***Priority*:** um número único dentro da simulação que representa a prioridade dessa tarefa. Quanto menor for esse valor, maior é a prioridade para o escalonador. Essa propriedade é definida no AG.

Cada um dos M nós são compostos por P processadores para executar o conjunto de tarefas alocados para o nó. O número P de processadores é igual para todos os nós, ou seja, se o primeiro nó possuir 2 processadores, todos os demais nós também terão 2 processadores. Todos os processadores possuem a mesma configuração, portanto todo processador contém capacidade de processamento igual. Todos os M nós possuem um escalonador de prioridade fixa que decide como cada tarefa será alocada no processador para a execução seguindo uma prioridade fixa.

4.5. Critério de qualidade do escalonamento

Para avaliar os escalonamentos gerados foram utilizados dois critérios de avaliação. O primeiro critério verifica se todas as tarefas foram executadas dentro dos seus *deadline*s. O segundo critério avalia a folga entre o *deadline* e a execução da tarefa.

No cenário proposto para este trabalho, todas as tarefas a serem escalonadas precisam necessariamente alcançar seus *deadline*s. Caso isso não aconteça todo o escalonamento é considerado inválido. Ainda nesse critério é verificado se, para as tarefas que falharam, existe pelo menos uma tarefa mais prioritária e com um período maior. Isso é feito para aplicar a mesma lógica do RM para penalizar as tarefas que não alcançaram os seus *deadline*s. Com isso, o critério de avaliação I se resume a quantidade de tarefas que falharem em respeitar o seu *deadline* somado ao total de situações onde houve uma tarefa menos prioritária que outra com um período maior. Para essa avaliação o melhor valor possível é zero, onde o escalonamento é válido pois a todas as tarefas respeitaram seus respectivos *deadline*s. Esse critério é resumido por meio da equação:

$$A_I = \left(\sum_{\forall i} 1 \quad if\ \forall i : D_i - R_i < 0 \right) + \left(\sum_{\forall i} 1 \quad if\ \forall i : D_i - R_i < 0 \ \wedge\ \exists_{j:}\, PR_j < PR_i \ \wedge\ P_j > P_i \right)$$

Onde D, R e PR, P são *deadline*, instante final de execução, prioridade e período da tarefa, respectivamente.

O critério de avaliação II é o tempo de sobra que as tarefas podem apresentar dentro da sua execução. Quanto maior for a diferença entre o tempo de execução da tarefa e seu *deadline*, melhor será o escalonamento, já que ele oferece melhores possibilidades para aumentar a quantidade de tarefas para o conjunto e também a possibilidade de alterar as características temporais de algumas tarefas sem precisar gerar um novo escalonamento. O segundo critério pode ser consolidado na equação demonstrado a seguir.

$$A_{II} = \sum_{\forall i} D_i - R_i$$

4.6. Cromossomo

O cromossomo foi codificado como um *array* de genes. Cada gene representa uma tarefa que precisa ser escalonada em um processador com determinada prioridade. O conceito de tarefa neste trabalho foi modelado em uma classe chamada de Task. A Task possui as seguintes propriedades:

- *Identifier*: valor literal que identifica unicamente uma tarefa no conjunto de tarefas a serem escalonadas;
- *Period*: valor decimal em ms que define o período onde uma nova instância da tarefa será criada para ser executada;
- *WorstExecutionTime*: valor decimal em que define o pior tempo de execução da tarefa;
- *Deadline*: valor decimal em ms que representa o limite de tempo que a tarefa tem para executar;
- *Processor*: valor inteiro que representa o processador que executará a tarefa.

Destas propriedades somente o processador é alterado durante a execução do AG. A ordem em que cada Task aparece no *array* de genes define a prioridade para cada tarefa. Caso a Task da tarefa T1 seja o primeiro item no *array* de genes, a tarefa T1 terá prioridade execução de 1 no processamento. É importante ressaltar que que essa prioridade é avaliada por cada nó de processamento, ou seja, caso a tarefa de prioridade 1 esteja alocado no nó 1 e a tarefa de prioridade 2 esteja alocada no nó 2, a tarefa de prioridade 2 será a tarefa mais prioritária para o nó 2.

Com isso, o valor do gene neste cromossomo representa o problema de alocação das tarefas nos nós e a ordem das tarefas no *array* lida com o problema de definir as prioridades de execução da cada tarefa.

4.7. Geração das tarefas pelo SimSo

Para gerar as tarefas que serão utilizadas nos testes do algoritmo foi utilizado o SimSo, que possui um módulo para a geração de *task sets* (um conjunto de tarefas). Esse módulo possui

alguns algoritmos prontos para essa função. Para esse trabalho foi escolhido o algoritmo RandomFixedSum, que gera as tarefas com valores aleatórios de período, *deadline* e pior tempo de execução dentro de um período pré-estabelecido de tempo.

O RandomFixedSum recebe os seguintes parâmetros para gerar o *task set*: taxa de utilização, número de tarefas periódicas e número de tarefas aperiódicas. A taxa de utilização é um número real de 1 a 32 que, quando dividido pelo número total de processadores no sistema, indica a taxa de utilização média dos processadores. Quanto maior for essa taxa de utilização, menor será a quantidade de escalonamentos possíveis para o *task set* gerado. O número de tarefas periódicas e o número de tarefas aperiódicas indicam a quantidade de tarefas que será gerada para cada tipo, respectivamente.

4.8. Geração da população Inicial

A geração da população inicial é feita de maneira aleatória. Todas as tarefas com a suas características são carregadas no AG utilizando o arquivo XML fornecido na inicialização. Para criação de cada cromossomo da população inicial o seguinte algoritmo é seguido. Para cada posição do cromossomo é sorteada de forma aleatória uma tarefa do conjunto de tarefas. Neste passo é necessário validar se essa tarefa não foi selecionada em alguma das iterações anteriores. Esta tarefa, então, é adicionada no primeiro espaço vago do cromossomo. E é escolhido, também de forma aleatória, um nó de processamento para ela. Este ciclo é repetido até que todas as tarefas sejam escalonadas no cromossomo.

O sorteio aleatório da tarefa garante que a prioridades das tarefas sejam diferentes entre os indivíduos da população inicial. Os sorteios dos nós também garantem uma diversidade em relação a em qual nó de processamento a tarefa irá executar. Desta forma, é obtida uma boa diversidade nos escalonamentos da população inicial. Para os testes realizados neste trabalho foi utilizado o tamanho de 100 indivíduos que é mantido durante toda a execução do AG.

4.9. Critério de parada

Como critério de parada para o AG foi utilizado um número máximo de gerações. Foi definido, através de testes, o número máximo de gerações de 25. Com esse número de gerações, o AG consegue encontrar escalonamentos válidos e procurar entre eles aqueles que possuem o maior tempo de folga para algumas das tarefas do conjunto a ser escalonado.

4.10. Elitismo

Para evitar a perda das melhores soluções já encontradas foi decidido utilizar a técnica de elitismo para a reinserção do melhor indivíduo na população. A cada nova geração é mantido

o melhor indivíduo da geração anterior. Com isso é possível sempre garantir que o melhor escalonamento não será perdido na nova geração.

4.11. Operador de Seleção

Como operador de seleção foram utilizados *Ranking* e Roleta Viciada. No primeiro passo todos os cromossomos são ordenados de acordo com valor do seu *fitness*. Utilizando esse ordenamento, é calculado um novo *fitness* de acordo com a euqação $0.9 + (1.1 - 0.9) * (i \div (100.0 - 1.0))$. Após o cálculo do novo *fitness* é utilizado o operador de seleção Roleta Viciada.

Nas primeiras versões deste AG foi utilizado o operador torneio de 3 indivíduos com repetição. Estas primeiras versões apresentaram alta pressão seletiva, constantemente alcançando máximos locais durante a sua execução. Com o objetivo de diminuir esta pressão seletiva, foi implementada a Roleta Viciada mencionada anteriormente.

4.12. Mutação

Como operador de mutação foi utilizado o operador Twors. Este operador troca a posição de 2 genes selecionados de forma aleatória. No cromossomo deste trabalho a mutação altera a prioridade das 2 tarefas que trocam de posição entre si.

Este operador foi escolhido com o objetivo de que as tarefas troquem suas prioridades de forma a criar uma nova possibilidade no escalonamento considerando a população atual. É importante ressaltar que a mutação não altera a alocação das tarefas no nó de processamento. Este operador também é importante já que a ele respeita a ordenação do resto do cromossomo (portanto não afeta a alocação das outras tarefas nos processadores) e não existe a possibilidade de gerar um cromossomo inválido.

4.13. Crossover

Como operador de *crossover* foi utilizado o operador OX3. Neste operador, são sorteados dois pontos de corte. Com estes dois pontos de corte os cromossomos dos pais são segmentados em três blocos. O primeiro filho rá receber todos os genes do segundo bloco do primeiro pai. Então o restante do cromossomo desse filho será preenchido com os genes restante do segundo pai, respeitando a ordem em que estão no cromossomo, começando pela primeira posição vazia. O segundo filho será preenchido da mesma forma substituindo a função dos pais no processo. Na Figura 6 é demonstrado o resultado da operação desse *crossover*.

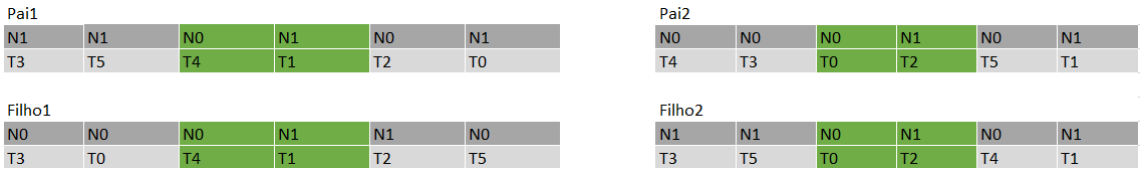


Figura 6 – Exemplo do crossover "*Order Crossover* 3"

Para este operador foi necessário implementar o algoritmo que compara se os genes são iguais. Dois genes são considerados iguais se fazem referência à mesma tarefa. Isso é verificado comparando o valor do atributo Identifier.

Este operador foi utilizado por ser um operador desenvolvido para trabalhar com cromossomos permutacionais, ou seja, ele leva em consideração as ordens relativas dos genes dentro do cromossomo. Por isso ele consegue criar novos escalonamentos com novas prioridades e alocações para as tarefas. Este operador garante a criação de cromossomos válidos, sem permitir repetições ou tarefas não alocadas.

4.14. Cálculo do *Fitness*

O *fitness* é calculado utilizando os critérios I e II de avaliação apresentados na seção 1.4. Como todo o escalonamento gerado deve ser válido, caso o primeiro critério apresente qualquer tarefa que não tenha sido executada dentro do seu *deadline*, o segundo critério não será calculado para este cromossomo.

Para realizar o cálculo do *fitness* é executado uma simulação utilizando os dados do cromossomo conforme descrito na seção 1.3. É importante ressaltar que a avaliação dos 2 critérios é feita para cada nó onde as tarefas serão executadas. Isso é necessário pois é executado uma simulação para cada nó.

4.14.1. Simulação da execução do escalonamento

Para a simulação é gerado um script em Python que é executado pelo SimSo. É utilizado um arquivo que possui um template padrão para geração desse script. Cada nó é simulado separadamente. Assim, é gerado um script para cada um. Na primeira parte do script é definido o parâmetro da duração da simulação. Para os testes realizados a duração da simulação foi definida em 200 ms. Na próxima parte do script são adicionadas todas as tarefas que foram alocadas para o nó a ser simulado, tendo as prioridades das mesmas definida pelo AG. Na terceira parte do script são instanciados todos os processadores do nó juntamente com o escalonador de prioridade fixa. Na última parte é fornecido um algoritmo que gera as informações para a avaliação do escalonamento após a execução da simulação.

O SimSo oferece 2 tipos de simulação para sistemas distribuídos de tempo real. A simulação do tempo médio de execução, ACET, ou simulação do pior tempo de resposta (*WCET)*. A simulação do tempo médio de execução é uma execução mais branda do sistema onde será

considerado o tempo médio de execução de cada tarefa, por isso uma simulação mais simples de se trabalhar. A simulação do pior tempo de resposta, por outro lado, utiliza o pior tempo de execução das tarefas, sendo mais complexa por usar os limites de execução de cada tarefa. Porém essa simulação oferece maiores garantias para o comportamento do sistema, pois cobre uma maior quantidade de cenários quando comparada a simulação ACET. É importante observar que a *WCET* é uma simulação de um cenário teórico considerando o pior caso possível para todas as tarefas. Para esse trabalho todas as simulações estão sendo realizados no tipo *WCET*.

4.14.2. Avaliação

A avaliação do *fitness* é realizada após a simulação de cada nó. Depois que toda a simulação acontece, é verificado no conjunto de tarefas se pelo menos uma delas não alcançou seu *deadline*. Caso isso aconteça, é contabilizado o número de tarefas que não alcançaram seu *deadline*. Esse somatório é utilizado como uma penalização na avaliação do escalonamento, ou seja, quanto maior esse número pior a avaliação no primeiro critério. Caso todas tarefas executem dentro dos seus respectivos *deadline*s, é contabilizado a diferença entre o *deadline* e o tempo de execução, sendo esse número a avaliação positiva no segundo critério. Essa avaliação pode ser resumida na mostrada na equação 4.

$$AC_N = \begin{cases} A_{II} \ \ se \ \forall n: A_I = 0 \\ A_I \ \ se \ \exists n: A_I < 0 \end{cases}$$

Onde AC é avaliação geral do nó. A_I e A_{II} são as avaliações do primeiro e segundo critério respectivamente.

A avaliação de cada nó é adicionada na avaliação geral do escalonamento. Se pelo menos um nó tiver uma avaliação negativa (pelo menos uma tarefa não executou antes do seu *deadline*), a avaliação geral será o somatório das avaliações de todos os nós com uma avaliação negativa. Assim a avaliação do escalonamento será o somatório de todas as tarefas que não alcançaram seus *deadline*s em todos os nós. Caso as avaliações dos nós sejam positivas, a avaliação final será o somatório de todas as diferenças entre o *deadline* e o tempo de execução das tarefas. Então o *fitness* é definido pela seguinte equação:

$$F = \begin{cases} \sum_{n=1} AC_n \ \ se \ \forall n: AC_n \geq 0 \\ \sum_{J=1} AC_j \ \ se \ \exists_J: AC_j < 0 \end{cases}$$

4.15. Processo de desenvolvimento e execução do sistema utilizando o *Genetic Scheduler*

Este AG foi desenvolvido para um sistema de RTS que utiliza o algoritmo de prioridade fixa para o escalonamento das tarefas. Conforme explicado anteriormente é esperado que o *Genetic Scheduler* receba um arquivo com as definições das tarefas que serão executadas. Com

esse arquivo o mesmo gera várias soluções possíveis em seus cromossomos que são válidas através de simulações executadas no SimSo. No final da execução deste AG é gerado o arquivo XML da melhor solução encontrada que pode ser utilizado por um escalonador central para alocar as tarefas em seus respectivos processadores. Todo esse fluxo do desenvolvimento e execução de um RTS utilizando o *Genetic Scheduler* é mostrada na Figura 7.

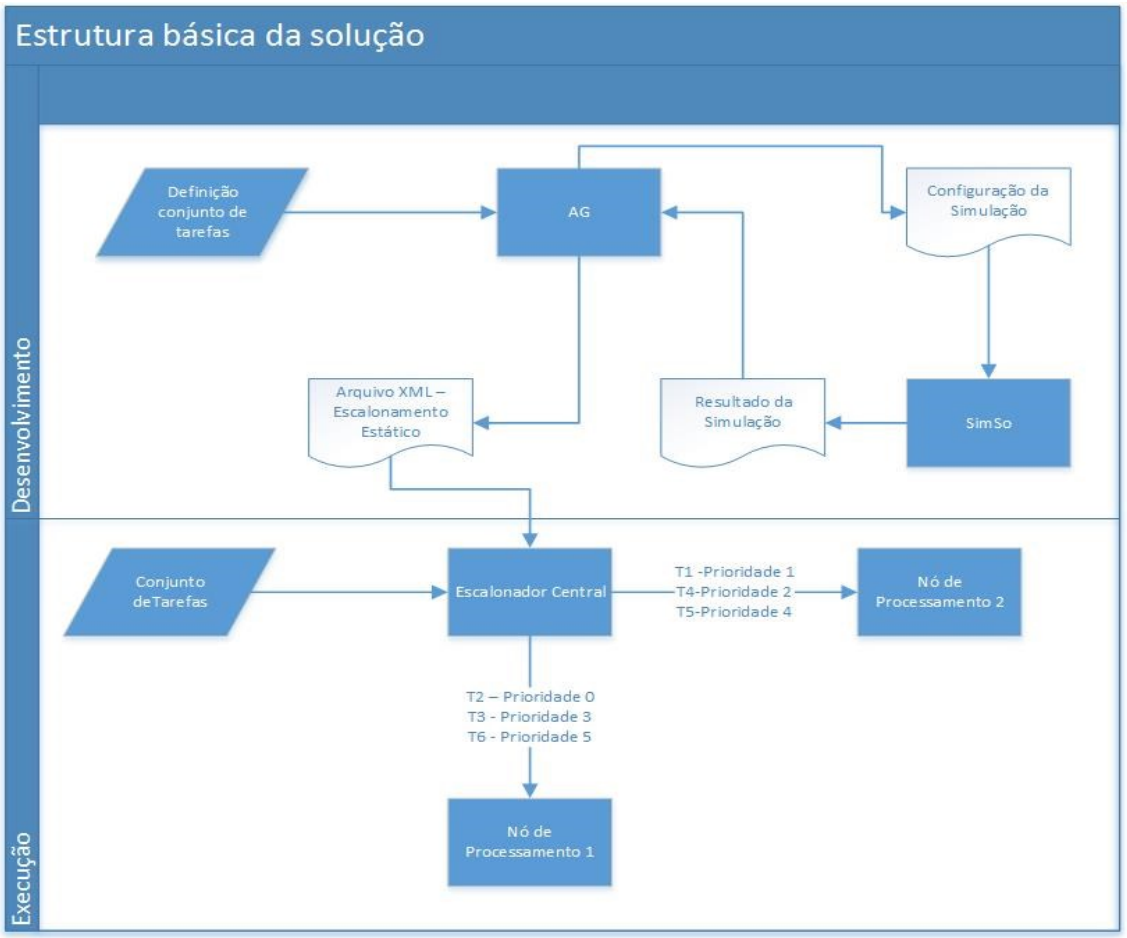


Figura 7 – Etapas do desenvolvimento do STR utilizando o *Genetic Scheduler*

5. Demonstração de Resultados

Para testar o algoritmo desenvolvido foram criados 6 cenários de testes. Para os 4 primeiros cenários foram geradas 50 tarefas periódicas no SimSo, para serem escalonadas em 2 nós de processamento, cada um com 2 processadores, conforme descrito no capítulo 4. Para o quinto cenário, foram geradas 75 tarefas que foram executadas em 3 nós de processamento com 2 processadores cada. Para o último cenário foram executadas 100 tarefas em 4 nós de processamento, também com 2 processadores.

Conforme mencionado na seção 4.6, o algoritmo RandFixSum precisa de uma taxa de utilização esperada para o conjunto de tarefas a serem escalonadas. Em cada um dos cenários de teste foi configurada uma taxa de utilização diferente, conforme descrito na Tabela 4.

Tabela 4 – Taxa de utilização por cenário

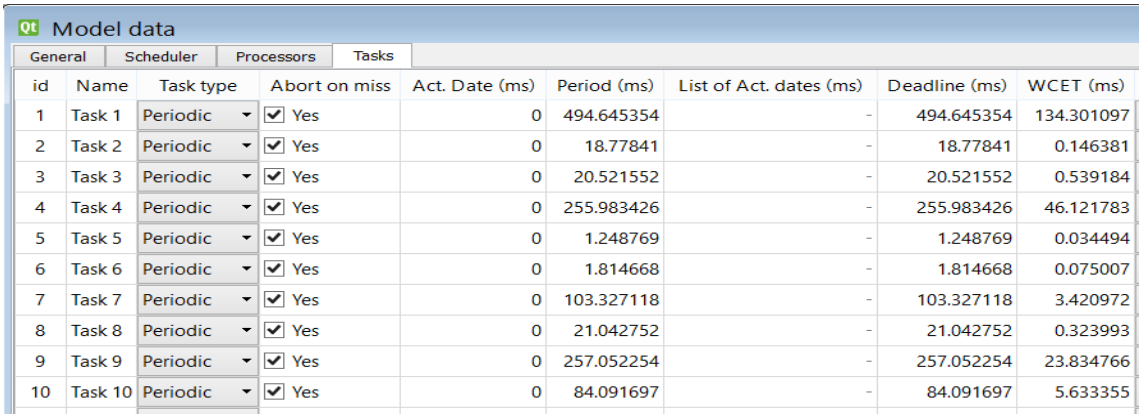
Cenário	Número de Tarefas	Taxa de Utilização (%)	Número de Nós
1	50	75	2
2	50	80	2
3	50	90	2
4	50	93	2
5	75	75	3
6	100	75	4

Para cada execução dos cenários de teste o AG foi calibrado com os seguintes parâmetros descrito na Tabela 5.

Tabela 5 – Configuração do AG na execução dos testes

Cenário	Número de Gerações	Tamanho da População	Probabilidade de Crossover (%)	Probabilidade de Mutação (%)
1	25	100	100	0,5
2	25	100	100	0,5
3	25	100	100	0,5
4	25	100	100	0,5
5	25	100	100	0,5
6	25	100	100	0,5

Todas as tarefas geradas pelo SimSo em todos os cenários de teste não possuem relação de precedência. Seus *deadlines* são iguais ou inferiores aos seus períodos e seus piores tempos de execução são também calculados pelo SimSo. Todas elas também têm seus momentos de ativação igual a zero, ou seja, todas são liberadas no início da execução. Todas também são configuradas para abortar caso seja identificado que não conseguirão executar dentro do seu *deadline*. *Deadline*, período e *WCET* foram configurados em milissegundos. A Figura 8 mostra um conjunto de tarefas geradas no SimSo utilizando sua Interface gráfica.



id	Name	Task type	Abort on miss	Act. Date (ms)	Period (ms)	List of Act. dates (ms)	Deadline (ms)	WCET (ms)
1	Task 1	Periodic	Yes	0	494.645354	-	494.645354	134.301097
2	Task 2	Periodic	Yes	0	18.77841	-	18.77841	0.146381
3	Task 3	Periodic	Yes	0	20.521552	-	20.521552	0.539184
4	Task 4	Periodic	Yes	0	255.983426	-	255.983426	46.121783
5	Task 5	Periodic	Yes	0	1.248769	-	1.248769	0.034494
6	Task 6	Periodic	Yes	0	1.814668	-	1.814668	0.075007
7	Task 7	Periodic	Yes	0	103.327118	-	103.327118	3.420972
8	Task 8	Periodic	Yes	0	21.042752	-	21.042752	0.323993
9	Task 9	Periodic	Yes	0	257.052254	-	257.052254	23.834766
10	Task 10	Periodic	Yes	0	84.091697	-	84.091697	5.633355

Figura 8 – Geração de tarefas no SimSo

Para os testes somente foram executadas simulações no SimSo. Com isso o *Genetic Scheduler* não foi validado em um cenário real.

5.1. Coleta de Dados

Para analisar as execuções do AG foi implementado mais uma funcionalidade. A cada geração na execução do algoritmo, é escrito em um arquivo de log o número da geração atual, o melhor *fitness* encontrado e um identificador numérico que indica o arquivo utilizado para simulação daquele cromossomo no SimSo naquela geração. Com isso, foi possível acompanhar a evolução do *fitness* na execução de cada execução. A Figura 9 mostra um exemplo desse arquivo.

```
Generations: 1
Fitness: -103 - File 32
Generations: 2
Fitness: -80 - File 113
Generations: 3
Fitness: -77 - File 244
Generations: 4
Fitness: -42 - File 315
Generations: 5
Fitness: 638,285985623 - File 418
Generations: 6
Fitness: 831,845288804 - File 510
Generations: 7
Fitness: 831,845288804 - File 510
```

Figura 9 – Arquivo com métricas da execução do AG

5.2. Resultado Individual dos cenários

Nesta seção são demonstradas as execuções de cada cenário. Nas Figuras 10,11,12, 13, 14 e 15 é detalhada a evolução que o algoritmo teve em cada execução. Em todos os gráficos destas figuras o eixo vertical representa o *fitness* o eixo horizontal mostra o número de gerações. Com isso é possível acompanhar a evolução da execução em cada cenário de teste. É importante ressaltar que como a primeira avaliação do *fitness* possui uma ordem de grandeza menor, as figuras a seguir demonstram valores muito próximos de zero.

Na Figura 10 do primeiro cenário de testes, onde a taxa de utilização está em 75%, é notado que o algoritmo encontra uma solução válida até a geração 6. Somente um caso precisou executar até a geração 10 para encontrar uma solução válida.

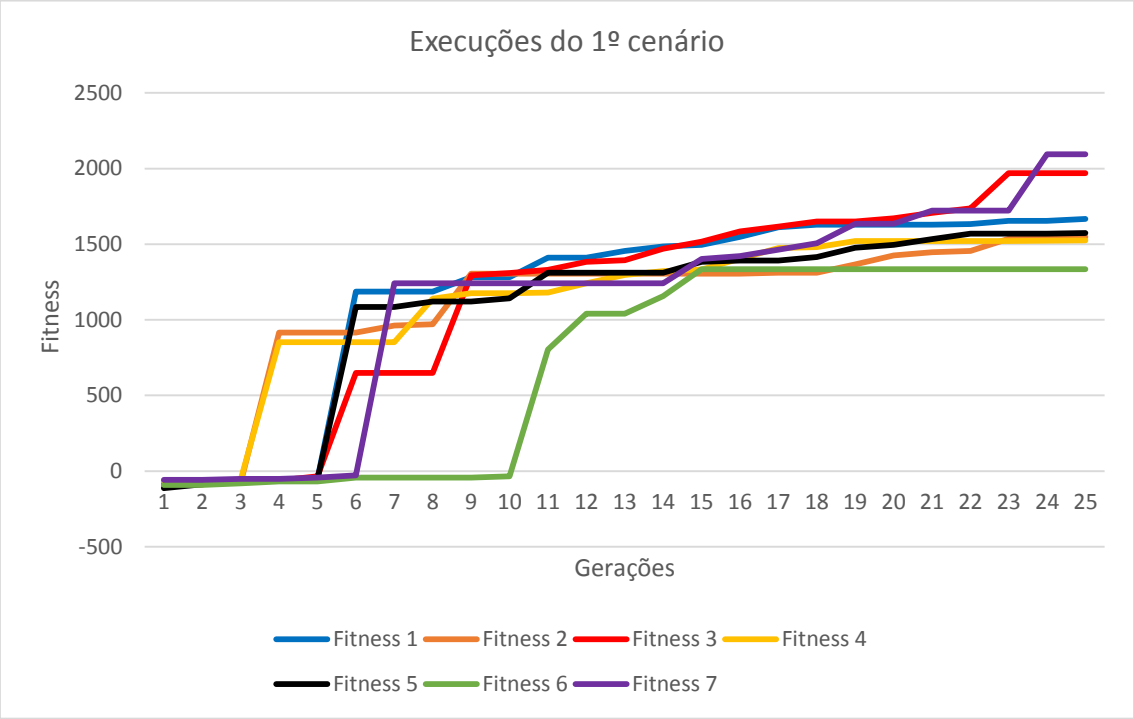


Figura 10 – Execuções do 1º cenário

Na Figura 11 do segundo cenário, onde a utilização está em 80%, temos uma parte das execuções com um comportamento similar ao do primeiro cenário e outros casos precisando executar até a geração 9 para encontrar uma solução válida.

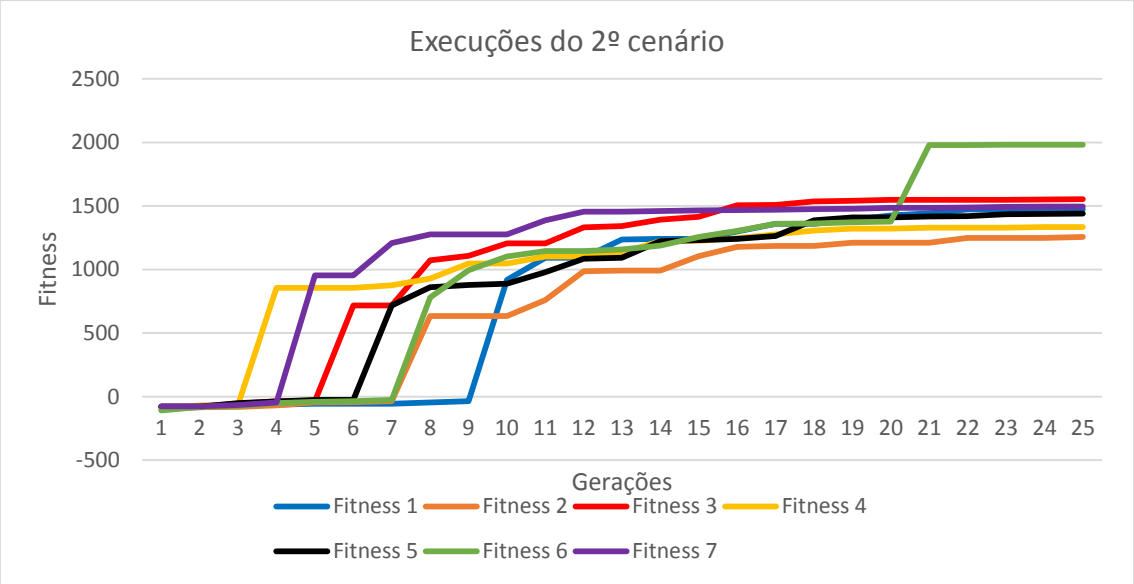


Figura 11 – Execuções do 2º cenário

Na Figura 12 do terceiro cenário, onde a utilização está em 90%, temos uma quantidade maior de execuções que levaram mais gerações para encontrar uma solução válida. Também é notável a queda do tempo de folga total encontrado pelo *Genetic Scheduler*.

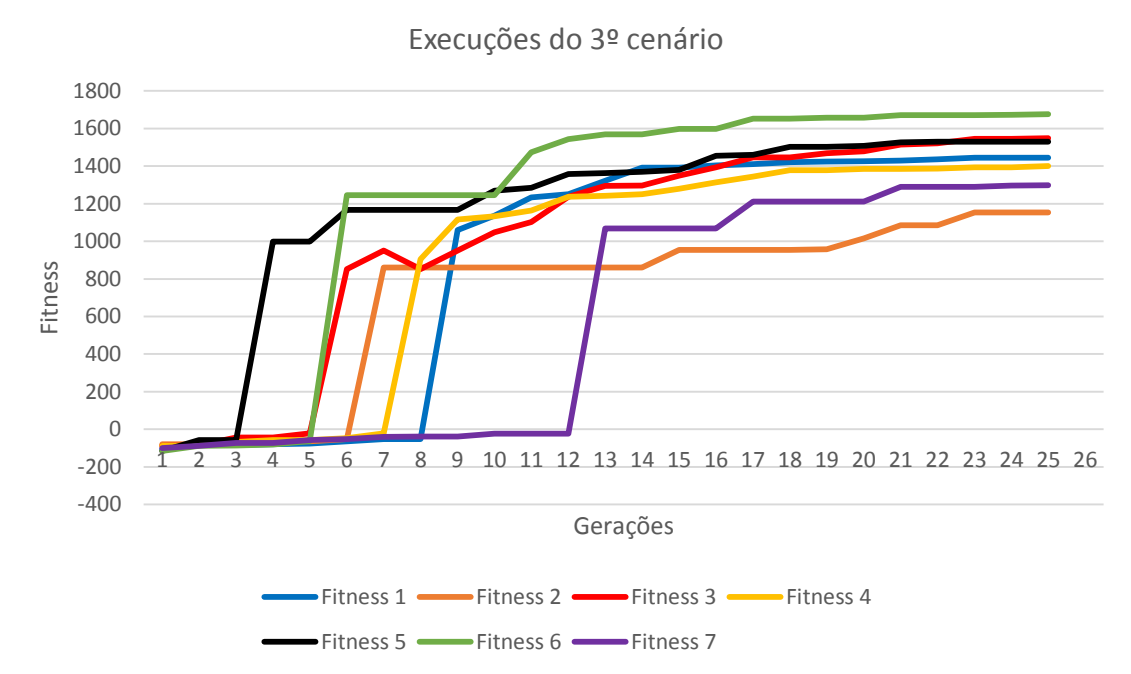


Figura 12 – Execuções do 3º cenário

Na Figura 13 do quarto cenário, onde a taxa de utilização foi aumentada para 93%, temos um caso onde o algoritmo não conseguiu encontrar uma solução válida. É perceptível que o algoritmo começa a levar um maior número de gerações para encontrar uma solução válida e que o tempo total de folga volta a diminuir, assim como aconteceu com o 3º cenário.

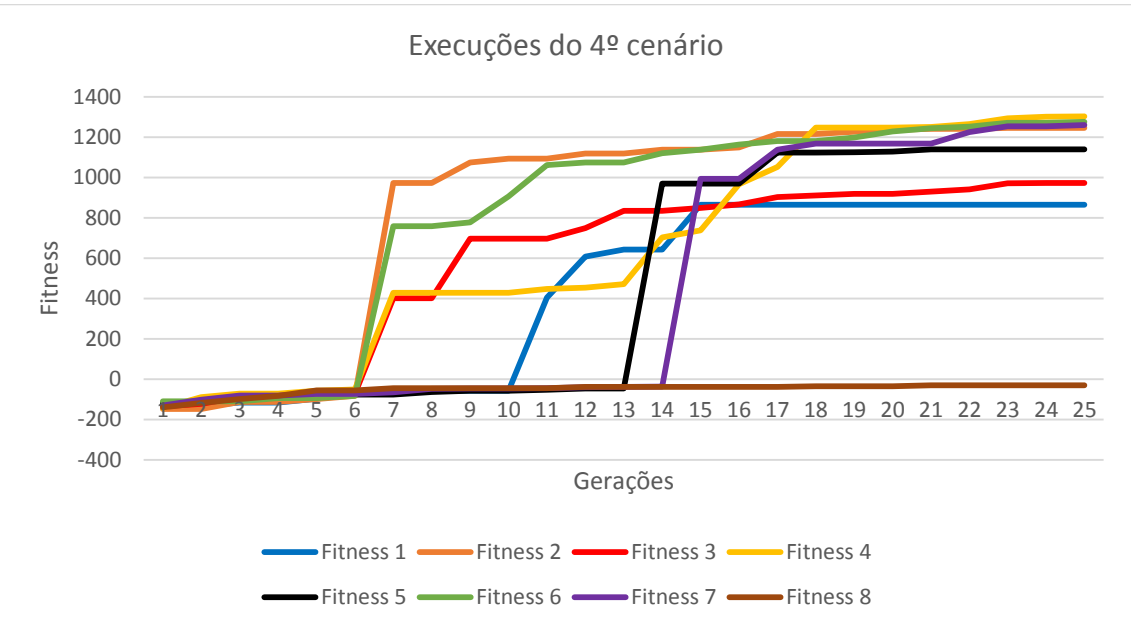


Figura 13 – Execuções do 4º cenário

Na Figura 14 do quinto cenário, onde os números de tarefas e nós de processamento foram aumentados para 75 e 3 respectivamente, é notável que os resultados obtidos são similares ao primeiro cenário que possui a mesma taxa de utilização. Neste caso, a maioria dos testes levaram até a 7 geração para encontrar uma solução válida. Os valores de tempo de folga encontrados são um pouco mais baixos que os encontrados no primeiro cenário.

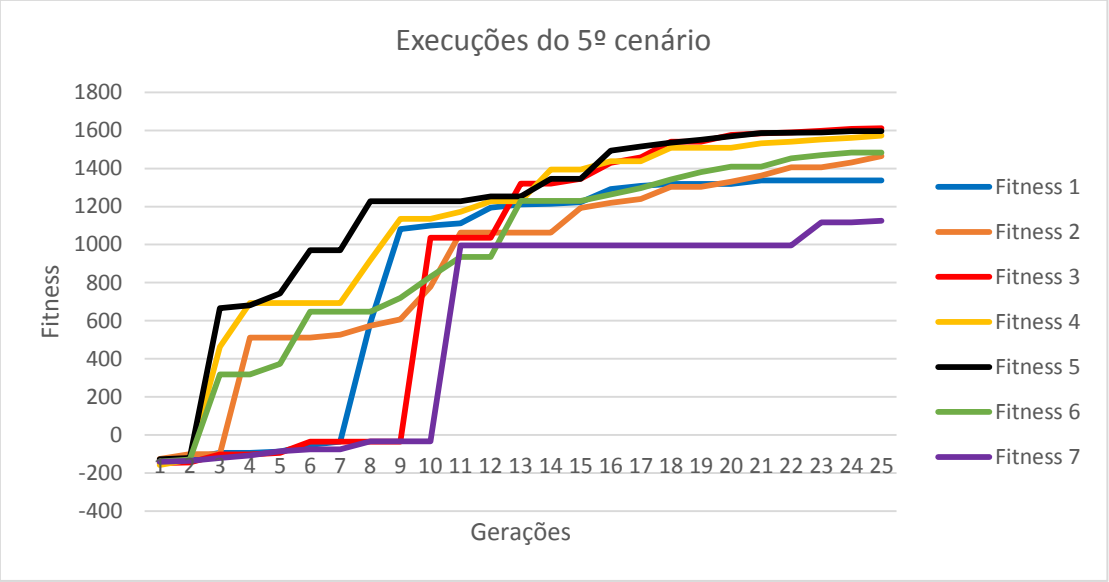


Figura 14 – Execuções do 5º cenário

Na Figura 15 do 6 cenário, que era composto por 100 tarefas para 4 nós de processamento, pode ser notado resultados similares aos cenários 1 e 5. Novamente, a maioria das execuções precisou de até 7 gerações para encontrar uma solução que respeitasse todos os deadlines. Assim como o cenário 5, este apresentou valores menores de tempo de folga quando comparado ao primeiro cenário.

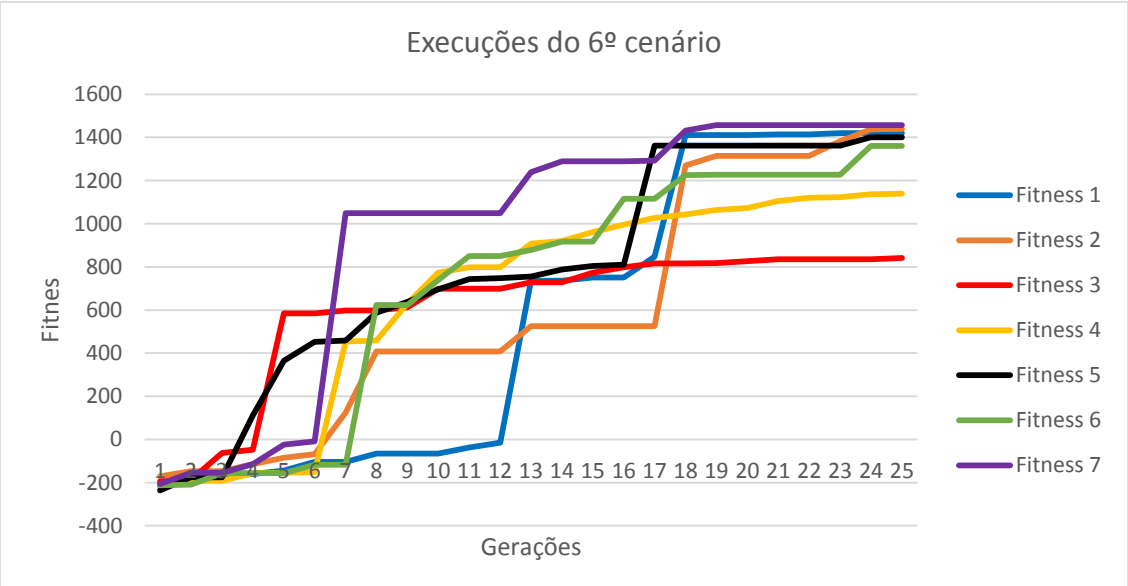


Figura 15 – Execuções do 6º cenário

5.3. Análise dos Resultados Consolidados dos Cenários com Aumento da Taxa de Utilização

Nesta seção são demonstrados os resultados consolidados obtidos na execução dos cenários com o mesmo número de tarefas e nós de processamento, sendo somente a taxa de utilização diferente. Os 3 primeiros cenários foram executados 7 vezes e o último cenário foi executado 8 vezes pelo AG. O último cenário possui uma execução a mais porque em uma de suas execuções o AG não encontrou um escalonamento válido. A Tabela 6 mostra a consolidação do resultado dessas execuções.

Tabela 6 – Consolidação das execuções de teste com aumento da taxa de utilização

Cenário	Nº Execuções	Nº Execuções Encontraram Solução Válida	Melhor Fitness
1	7	7	2094,094849
2	7	7	1981,096775
3	7	7	1676,200414
4	8	7	1302,976756

Conforme observado na Tabela 6 é possível identificar que em todos os cenários foi possível encontrar uma solução válida. Porém, no cenário 4 onde a taxa de utilização estava em 93%, houve um caso onde o AG falhou em encontrar um escalonamento válido. Pode-se observar, também, que a medida em que a taxa de utilização aumentou, menores foram os tempos de folga que o algoritmo conseguiu encontrar, principalmente com uma taxa de utilização igual ou maior a 90%.

Na Figura 16 isso é demonstrado com a consolidação de todas as execuções em um único gráfico. Para esse gráfico foi utilizado a média dos *fitness* encontrada em cada geração. Nele é possível perceber que o algoritmo tem um comportamento padrão com uma taxa de utilização de até 90%. Porém quando a taxa de utilização foi aumentada além disso, os *fitness* encontrados foram menores.

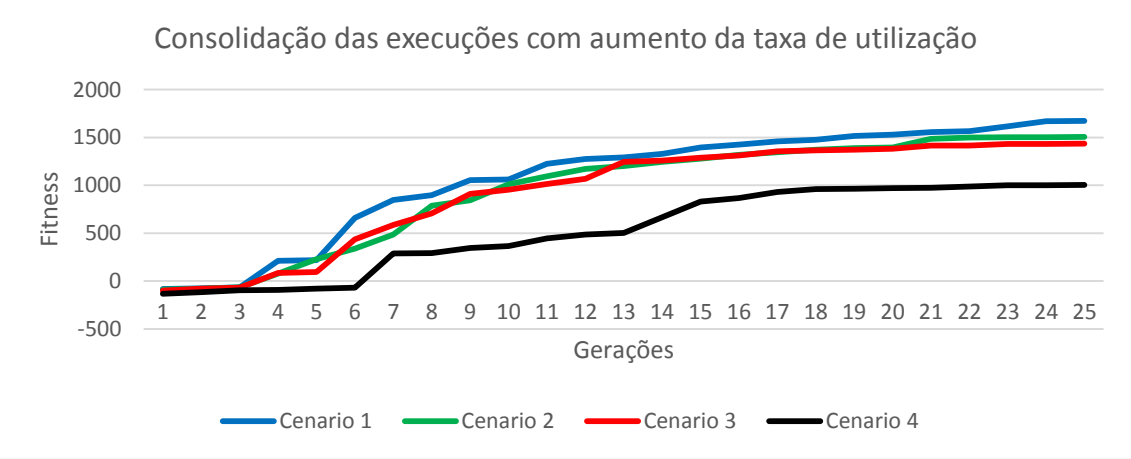


Figura 16 – Consolidação das execuções com aumento da taxa de utilização

5.4. Análise dos Resultados Consolidados dos Cenários com Aumento no Número de Tarefas e Nós de Processamento

Esta seção tem como objetivo avaliar os resultados consolidados dos cenários que possuem a mesma taxa de utilização, porém tem quantidades diferentes de tarefas e nós de processamento. A Tabela 7 mostra a consolidação do resultado dessas execuções.

Tabela 7 – Consolidação das execuções de teste com maior quantidade de tarefas e nós de processamento

Cenário	Nº Execuções	Nº Execuções Encontraram Solução Válida	Melhor Fitness
1	7	7	2094,094849
4	7	7	1610,938112
5	7	7	1457,235449

Assim como na análise dos cenários com diferentes taxas de utilização, foi encontrada uma solução válida em todos os cenários. Porém, nestes casos, todas as execuções encontraram soluções que cumpriam seus deadlines. Na medida em que os números de tarefas e nós de processamento aumentaram, menores foram os melhores tempos de folga encontrados pelo algoritmo. Isso acontece porque o espaço de busca aumenta com maiores números de tarefas e nós de processamento, e sendo assim, com mais possibilidades a serem verificadas.

Assim como na Figura 16, a Figura 17 demonstra as execuções destes cenários com as médias dos fitness encontradas em cada geração. Analisando os dados é notável que nestes cenários o AG encontrou resultados com uma maior quantidade de tarefas que não respeitaram seus deadlines no início. A partir do momento em que encontrou soluções válidas, os tempos de folga encontrados foram geralmente mais baixos para os cenários com maiores quantidades de tarefas e nós de processamento.

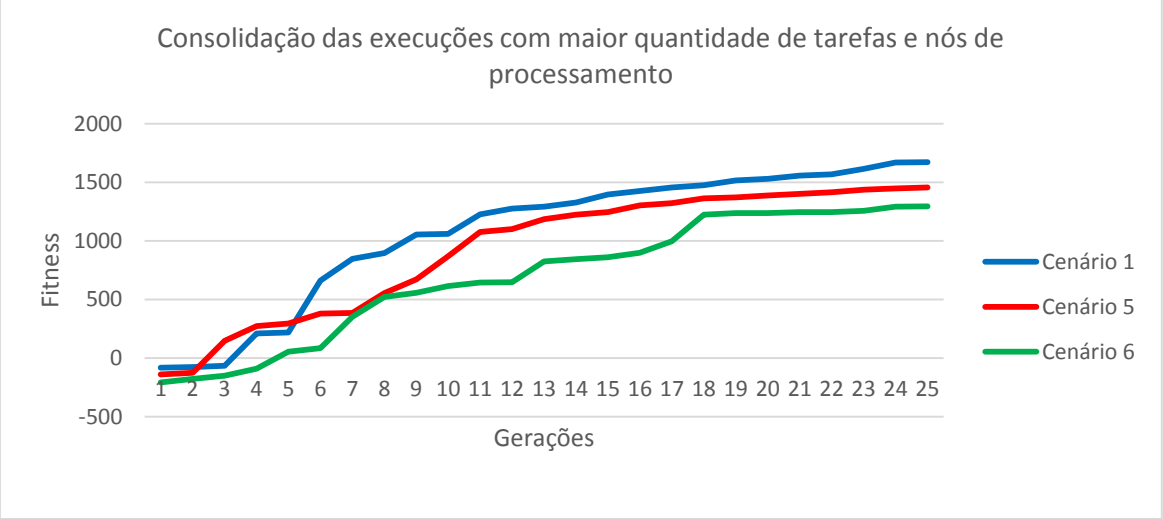


Figura 17 – Consolidação das execuções com maior quantidade de tarefas e nós de processamento

5.5. Análise da confiabilidade do *Genetic Scheduler*

Nesta seção são demonstrados dados relacionados à confiabilidade da execução *Genetic Scheduler*. Essa análise é feita com a intenção de demonstrar a eficiência do algoritmo em encontrar um escalonamento válido. Para isso, foram analisados a média, o desvio padrão e o percentil de 85% do número de gerações necessárias para o *Genetic Scheduler* encontrar o primeiro escalonamento válido em cada execução. Com isso podemos fazer uma análise da confiabilidade do algoritmo identificar alguns padrões. A Tabela 8 e a Figura 18 a seguir demonstram os dados consolidados para essa análise.

Tabela 8 – Tabela com as medidas em qual geração foi encontrado uma solução válida

Cenário	Média	Desvio Padrão	Percentil 85%
1	6,285714286	2,185294077	6,8
2	6,857142857	1,884415137	8
3	7,428571429	2,664965444	8,6
4	9,714285714	3,325841922	13,4
5	6,571428571	3,245090483	9,6
6	7,285714286	2,657296463	7,8

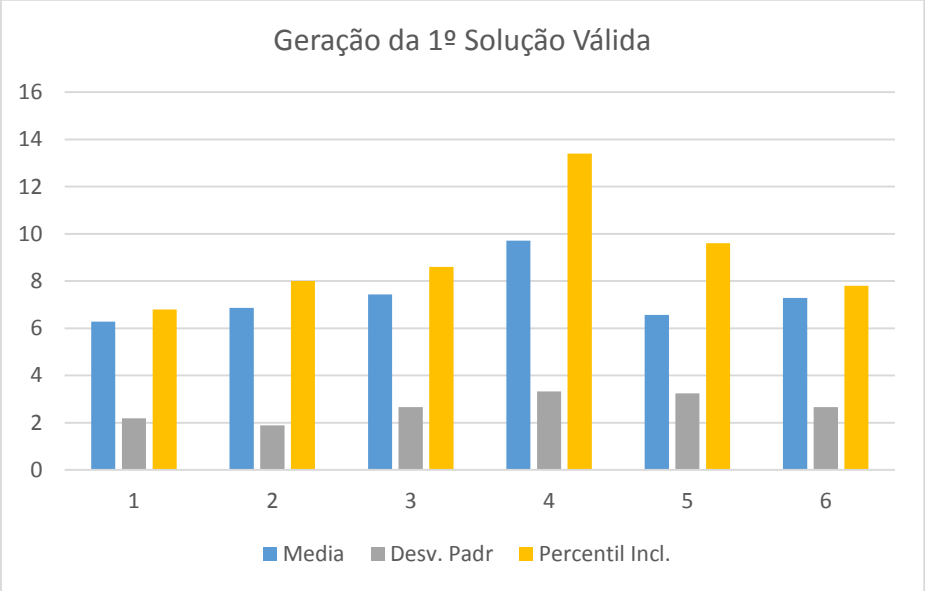


Figura 18 – Gráfico referente a medidas de quando foi encontrado uma solução válida

Como pode ser observado nos dados acima, o *Genetic Scheduler* consegue encontrar uma solução válida em 85% das execuções de cada cenário utilizando apenas metade do número de gerações limite. A variação demonstrada pelo desvio padrão embora não seja muito constante também apresenta valores dentro de um limite aceitável.

Conforme visto na seção 5.3, uma maior taxa de atualização aumenta a quantidade de gerações que o AG precisa para encontrar uma solução válida. Já o aumento no número de

tarefas e nós de processamento visto na seção 5.4 aumentaram também a média do número de gerações necessárias para encontrar a primeira solução válida. O desvio padrão para os cenários 5 e 6 são bastante altos.

Com isso, o algoritmo demonstra resultados satisfatórios no seu principal objetivo de encontrar escalonamentos válidos com um bom grau de confiança, embora ainda apresente uma certa disparidade em qual geração essa solução será encontrada entres as suas execuções com taxas de utilização diferentes. O *Genetic Scheduler* foi capaz de encontrar soluções para cenários com taxa de utilização, número de tarefas e quantidades de nós de processamento diferentes.

6. Conclusões e trabalhos futuros

6.1. Conclusões

Neste trabalho foi apresentado uma revisão conceitual sobre AG e STR. Essa revisão em conjunto com os trabalhos relacionados apresentados serviu de embasamento para o desenvolvimento de um AG capaz de receber um conjunto de tarefas com restrições temporais que devem ser executadas em um cluster simulado. Utilizando desse embasamento foi desenvolvido um AG que consegue definir a alocação das tarefas em processadores e a prioridade que as mesmas terão durante a sua execução. Esse algoritmo consegue encontrar esses escalonamentos sem precisar alterar as características definidas inicialmente para as tarefas como *deadline*, período e pior tempo de execução.

Fez-se necessário a pesquisa pela melhor forma para avaliar os resultados encontrados no AG. Foi definido a utilização de simulações para avaliar os escalonamentos gerados. Para isso foi utilizado a ferramenta SimSo capaz de receber os escalonamentos gerados em arquivo XML e simular a sua execução. Essa ferramenta gera os dados necessários para a avaliação do *fitness*, sendo esse utilizado para verificar a viabilidade e a qualidade do escalonamento.

Com o embasamento teórico e uma ferramenta capaz de prover informações para avaliação dos escalonamentos encontrados este AG desenvolvido neste projeto foi capaz de encontrar escalonamentos válidos para um modelo simplificado de tarefas. Utilizando de técnicas como cromossomo permutacional para representar o problema de uma forma computacional o problema. Com isso o algoritmo conseguiu encontrar boas soluções no espaço de busca.

Utilizando os dados gerados durante a execução do *Genetic Scheduler* foi possível executar teste qualitativos no AG e os escalonamentos gerados pelo mesmo. Podendo assim confirmar que o algoritmo consegue alcançar seu objetivo principal de encontrar escalonamentos válidos e ainda consegue melhorar esses para que o tempo de folga seja o maior possível, gerando com isso escalonamentos mais robustos e flexíveis.

Pode ser concluído que a utilização de AG para a geração de escalonamento de STR se mostra uma opção interessante. Porém ainda necessário realizar testes em cenários reais e com um modelo mais complexo de tarefas para confirmar a viabilidade dessa solução.

6.2. Trabalhos Futuros

Esta seção se destina a identificar sugestões de melhoria e continuidade para área de escalonamentos de tarefas com restrição temporal gerados por um AG. Considerando o trabalho realizado, junto com os trabalhos relacionados, 3 pontos de melhorias foram identificados:

- Tempo da simulação: Para um AG o tempo de execução é uma questão crítica com relação direta a sua viabilidade. Atualmente o tempo de execução da simulação tem um

alto custo para o tempo de execução total do AG. Encontrar alternativas ou desenvolver uma ferramenta própria, focada na performance para melhorar esse ponto no AG seria uma possibilidade de melhoria para este trabalho.

- Evolução do modelo: O modelo de tarefas usado neste trabalho pode ser melhorado incluindo a parte de mensagens entre outras variáveis para ter um modelo que se aproxima mais dos cenários reais para sua aplicação.
- Utilização de algoritmos de escalonamentos para multiprocessadores: Na área de STR novos algoritmos de escalonamentos foram criados para levar em consideração as características de sistemas multiprocessador, obtendo uma performance melhor durante o escalonamento. Alterar o *Genetic Scheduler* para utilizar um destes algoritmos para suportar o ambiente de sistemas multiprocessador é um possível trabalho futuro.

7. Bibliografia

LINDEN, Ricardo. **Algoritmos genéticos** .3a ediçao. Rio de Janeiro: Ciência Moderna, 2012.

FARINES, Jean-Marie; FRAGA, Joni da Silva; OLIVEIRA, RS de. Sistemas de tempo real. **Escola de Computação**, v. 2000, p. 201, 2000.

YOO, Myungryun; GEN, Mitsuo. Scheduling algorithm for real-time tasks using multiobjective hybrid genetic algorithm in heterogeneous multiprocessors system. **Computers & Operations Research**, v. 34, n. 10, p. 3084-3098, 2007.

GHARSELLAOUI, Hamza et al. Real-time reconfigurable scheduling of multiprocessor embedded systems using hybrid genetic based approach. In: **Computer and Information Science (ICIS), 2015 IEEE/ACIS 14th International Conference on**. IEEE, 2015. p. 605-609.

AZKETA, Ekain et al. Permutational genetic algorithm for the optimized assignment of priorities to tasks and messages in distributed real-time systems. In: **Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on**. IEEE, 2011. p. 958-965.

AZKETA, Ekain et al. An empirical study of permutational genetic crossover and mutation operators on the fixed priority assignment in distributed real-time systems. In: **Industrial Technology (ICIT), 2012 IEEE International Conference on**. IEEE, 2012. p. 598-605.

SEBESTYEN, Gheorghe; HANGAN, Anca. Genetic approach for real-time scheduling on multiprocessor systems. In: **Intelligent Computer Communication and Processing (ICCP), 2012 IEEE International Conference on**. IEEE, 2012. p. 267-272.

OH, Jaewon; WU, Chisu. Genetic-algorithm-based real-time task scheduling with multiple goals. **Journal of systems and software**, v. 71, n. 3, p. 245-258, 2004.

KONAR, Debanjan et al. An Improved Hybrid Quantum-Inspired Genetic Algorithm (HQIGA) for Scheduling of Real-Time Task in Multiprocessor System. **Applied Soft Computing**, 2017.

HAO, Lu; YANG, Xiaopeng; HU, Shangkun. Task scheduling of improved time shifting based on genetic algorithm for phased array radar. In: **Signal Processing (ICSP), 2016 IEEE 13th International Conference on**. IEEE, 2016. p. 1655-1660.

CHÉRAMY, Maxime et al. Simulation of real-time scheduling with various execution time models. In: **Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on**. IEEE, 2014. p. 1-4.

GeneticSharp. (07 do 03 de 2017). Acesso em 07 do 03 de 2017, disponível em https://github.com/giacomelli/GeneticSharp

Simso. (07 do 03 de 2017). Acesso em 07 do 03 de 2017, disponível em http://projects.laas.fr/simso/

8. Anexo

8.1. Artigo

Genetic Scheduler: Um Algoritmo Genético para Escalonamento de Tarefas com Restrição Temporal em Sistemas Distribuídos

Ruann M. Homem[1], Luciana R. Reich[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC) Caixa Postal 476 – 88040-900 – Florianopolis – SC – Brazil

ruann.homem@grad.ufsc.br, luciana.rech@inf.ufsc.br

Resumo. *Encontrar escalamentos que definem a alocação das tarefas e a prioridade que essas terão no processamento da execução de um escalonamento para sistemas reais é umas das principais tarefas que deve ser feita na desenvolvimento e execução de um sistema de tempo real. O escalonamento gerado tem como objetivo garantir as restrições temporais na execução destas tarefas. Neste contexto este artigo apresenta o Genetic Scheduler, um algoritmo genético capaz de escalonar um conjunto de tarefas com restrições temporais. Esse trabalho demonstra como esse algoritmo encontrar esses escalonamentos e os valida através de simulações.*

1. Introdução

Sistemas de Tempo Real apresentam a necessidade de executar suas tarefas dentro de um limite de tempo conhecido como deadline, caracterizando assim um limite temporal em sua execução. Devido a essa restrição a ordenação na execução de suas tarefas é um fator crítico para este tipo de sistema. A essa ordenação é dado o nome de escalonamento das tarefas. Encontrar uma lógica capaz de gerar escalonamentos válidos, ou seja, escalonamentos que possam garantir as limitações temporais na execução das tarefas, é uma das principais atividades que devem ser executadas no desenvolvimento de um sistema de tempo real. Como exemplos desse tipo de sistema pode ser citado: sistemas de controle de tráfego aéreo e sistemas de streaming, entre outros (KONAR, 2017).

Uma das abordagens que podem ser utilizadas na resolução do problema de escalonamento de tarefas com restrições temporais é a técnica Algoritmos Genéticos. Um Algoritmo Genético simula um ambiente de competição entre as soluções já conhecidas, chamadas de cromossomos que formam uma população, para obter soluções melhores através da utilização de operadores genéticos nos melhores cromossomos conhecidos. A utilização de um Algoritmo Genético pode ser realizada em conjunto com uma ferramenta de simulação para avaliar as possíveis soluções que o algoritmo gerou (SEBESTYEN E HANGAN, 2012). Com isso sendo uma alternativa para validar os escalonamentos além das técnicas de análise de escalabilidade.

Algoritmos Genéticos é uma abordagem interessante na solução do problema de encontrar escalonamentos factíveis para tarefas em Sistemas de tempo real em ambientes distribuídos (AZKETA, 2011) (SEBESTYEN E HANGAN, 2012).

Com os trabalhos relacionados analisados, constatou-se que o problema de geração de escalonamentos que respeitem as restrições temporais pode ser abordado de diversas formas. Trabalhos como (SEBESTYEN E HANGAN, 2012), (KONAR, 2017), (GHARSELLAOUI, 2015) alteraram as características temporais das tarefas utilizando em conjunto com técnicas de escalonamento conhecidas como EDF (Earliest Deadline First) para gerar escalonamentos válidos. Poucos trabalhos como (AZKETA, 2011), geraram escalonamentos utilizando apenas Algoritmos Genéticos analisando as características já definidas para as tarefas, ou seja, sem alterar o deadline ou o tempo de execução.

Em (AZKETA, 2011) e (GHARSELLAOUI, 2015) foram utilizadas técnicas de análise de escalabilidade para verificar seus resultados obtidos ou não especificaram como foram obtidos os dados utilizados no fitness. (SEBESTYEN E HANGAN, 2012) testaram seus resultados com ferramentas de simulação tentando executar os escalonamentos gerados em um sistema controlado e mais próximo de um cenário real. Com isso, a motivação desse trabalho foi aplicar as técnicas de Algoritmos Genéticos para criar um algoritmo que possa gerar escalonamentos válidos para um conjunto de tarefas, validando-os através de simulações.

2. Trabalhos Relacionados

Em todos os trabalhos relacionados é verifica-se que a representação mais comum para o cromossomo para este problema é um array onde um dos valores representa o processador e outro representa a tarefa ou uma característica temporal. Geralmente o cromossomo também é permutacional, onde a ordem dos genes é utilizada para indicar a prioridade das tarefas no escalonamento. Seguindo esse padrão, o presente trabalho também utiliza um cromossomo permutacional onde os genes representam as tarefas e, suas posições no cromossomo a prioridade que elas possuem no escalonamento. A diferença é que para este cromossomo uma tarefa não é alocada para um processador e sim para um nó, que possui N processadores.

Para a avaliação do cromossomo o fato de uma tarefa executar ou não dentro do escalonamento é tratado como uma penalidade no fitness. Algumas características, como a taxa de utilização, também foram utilizadas na avaliação. Neste trabalho é usada uma abordagem similar à (AZKETA, 2011), onde uma função contabiliza quantas tarefas não alcançaram seus deadlines e o tempo de folga total do escalonamento. Como diferencial, foi preciso calcular a mesma função para todos os nós que temos no sistema. Caso exista um nó que possua pelo menos uma tarefa que não executou dentro do seu deadline, todo o escalonamento é considerado inválido.

Trabalhos como (SEBESTYEN E HANGAN, 2012), (KONAR, 2017) e (GHARSELLAOUI, 2015) utilizaram o algoritmo de escalonamento EDF, enquanto (AZKETA, 2011), (YOO e MITSUO, 2007) utilizaram um algoritmo de prioridade fixa. Este trabalho é similar ao último grupo, utilizando o algoritmo de prioridade fixa.

Para os operadores genéticos a maioria dos trabalhos utilizaram operadores básicos e bastante conhecidos como: crossover de dois pontos e mutação clássica. Alguns trabalhos se destacam por utilizar outras técnicas em conjunto com os seus algoritmos.

Em (GHARSELLAOUI, 2015) a busca tabu foi utilizada no lugar do operador de mutação. Em (KONAR, 2017) a modelagem do cromossomo e os operadores foram feitos com técnicas da computação quântica. (YOO e MITSUO, 2007) utilizou a técnica de SA em conjunto com o seu Algoritmo Genético para melhorar a convergência do algoritmo. Para este trabalho foram utilizados operadores voltados para cromossomos permutacionais: OX3 assim como em (AZKETA, 2011) para crossover e Twor para a mutação.

Para a seleção de indivíduos foi utilizado o mesmo operador dos trabalhos de (Yoo e Mitsu, 2007) e (GHARSELLAOUI, 2015): A roleta viciada. Porém, ela foi utilizada em conjunto a técnica de Ranking.

Todos os trabalhos relacionados solucionavam o problema definindo o escalonamento das tarefas nos processadores disponíveis e a prioridade das tarefas no mesmo. Alguns trabalhos como (SEBESTYEN E HANGAN, 2012) não alteravam diretamente a prioridade, e sim alguma característica que fora utilizada para a definição da mesma. No trabalho, citado o mesmo alterava os deadlines das tarefas para alterar as prioridades utilizando o algoritmo EDF (que privilegia um deadline menor). Neste algoritmo, foi trabalhado com a prioridade da tarefa e a sua alocação em um nó específico.

Para os testes das soluções obtidas, dois trabalhos se destacam. (SEBESTYEN E HANGAN, 2012) utilizaram a ferramenta RTMultSim para realizar simulações com os escalonamentos gerados, testando as suas soluções em um cenário mais próximo do cenário real. (AZKETA, 2011) utilizou um modelo matemático para verificar se as tarefas alcançavam os seus deadlines e as folgas que elas apresentavam em suas execuções. No presente trabalho foi utilizado uma ferramenta de simulação como foi feito em (SEBESTYEN E HANGAN, 2012) para gerar os insumos necessários para o cálculo do fitness. Para o cálculo do fitness foi utilizado a função "scheduling index factor" apresentada em (AZKETA, 2011).

3. Algoritmos Genéticos

Algoritmos Genéticos são algoritmos que utilizam modelos computacionais inspirados nos processos naturais de evolução das espécies para resolver problemas computacionalmente complexos (LINDEN, 2012). Esses algoritmos são técnicas heurísticas de otimização global que tentam maximizar uma função de avaliação (Fitness) para encontrar a solução para um problema. Para isso o Algoritmos Genéticos cria uma população contendo diversas soluções possíveis para esse problema. Cada membro dessa população é denominado indivíduo. A cada interação do algoritmo o mesmo seleciona um conjunto de indivíduos baseado na função de avaliação, que mede a qualidade da solução, para aplicar operadores genéticos (de crossover e mutação). Os novos indivíduos resultantes compõem a população da próxima geração, que tendem a conter soluções "evoluídas", ou seja, soluções que resolvem melhor o problema. Essas soluções evoluídas irão se aproximar de soluções ótimas a cada passo do algoritmo.

4. Sistemas de Tempo Real

Sistemas de Tempo Real são aplicações que apresentam requisitos de tempo real na execução de suas tarefas, principalemente restrições temporais. Portanto, um Sistema de tempo real é um sistema computacional que deve reagir a estímulos oriundos do seu

ambiente em prazos específicos (FARINES et al., 2000). Com isso, um Sistema de tempo real precisa apresentar dois comportamentos importantes: o primeiro comportamento é a execução lógica. Ou seja, o sistema precisa apresentar saídas que estejam logicamente corretas para as entradas que recebeu. O segundo comportamento esperado é a previsibilidade temporal. Toda a ação dentro do sistema precisa ser executada dentro de um tempo controlado. Uma falha em executar dentro deste tempo é considerada uma falha temporal. Existem sistemas como controle de tráfego aéreo e sistemas militares de defesa que possuem restrições temporais mais rígidas do que sistemas como teleconferências através da Internet e as aplicações de multimídia em geral, que possuem restrições temporais mais brandas.

5. Cromossomo

O cromossomo foi codificado como um array de genes. Cada gene representa uma tarefa que precisa ser escalonada em um processador com determinada prioridade. O conceito de tarefa neste trabalho foi modelado em uma classe chamada de Task. A Task possui as seguintes propriedades:

- Identifier: valor literal que identifica unicamente uma tarefa no conjunto de tarefas a serem escalonadas.
- Period: valor decimal em ms que define o período onde uma nova instância da tarefa será criada para ser executada.
- WorstExecutionTime: valor decimal em que define o pior tempo de execução da tarefa.
- Deadline: valor decimal em ms que representa o limite de tempo que a tarefa tem para executar.
- Processor: valor inteiro que representa o processador que executará a tarefa.

Destas propriedades somente o processador é alterado durante a execução do Algoritmos Genéticos. A ordem em que cada Task aparece no array de genes define a prioridade para cada tarefa. Caso a Task da tarefa T1 seja o primeiro item no array de genes, a tarefa T1 terá prioridade execução de 1 no processamento. É importante ressaltar que que essa prioridade é avaliada por cada nó de processamento, ou seja, caso a tarefa de prioridade 1 esteja alocado no nó 1 e a tarefa de prioridade 2 esteja alocada no nó 2, a tarefa de prioridade 2 será a tarefa mais prioritária para o nó 2.

Com isso, o valor do gene neste cromossomo representa o problema de alocação das tarefas nos nós e a ordem das tarefas no array lida com o problema de definir as prioridades de execução da cada tarefa.

6. Implementação

No genetic Scheduler foram utilizados os seguintes métodos e operadores geneticos:

1) Geração da população inicial

A geração da população inicial é feita de maneira aleatória. Todas as tarefas com a suas características são carregadas no Algoritmos Genéticos utilizando o arquivo XML fornecido na inicialização. Para criação de cada cromossomo da população inicial o seguinte algoritmo é seguido. Para cada posição do cromossomo é sorteada de forma aleatória uma tarefa do conjunto de tarefas. Neste passo é necessário validar se essa tarefa não foi selecionada em alguma das iterações anteriores. Esta tarefa, então, é adicionada no primeiro espaço vago do cromossomo. E é escolhido, também de forma aleatória, um nó de processamento para ela. Este ciclo é repetido até que todas as tarefas sejam escalonadas no cromossomo.

2) Critérios de parada

Como critério de parada para o Algoritmos Genéticos foi utilizado um número máximo de gerações. Foi definido, através de testes, o número máximo de gerações de 25. Com esse número de gerações, o Algoritmo Genético consegue encontrar escalonamentos válidos e procurar entre eles aqueles que possuem o maior tempo de folga para algumas das tarefas do conjunto a ser escalonado.

3) Elitismo

Para evitar a perda das melhores soluções já encontradas foi decidido utilizar a técnica de elitismo para a reinserção do melhor indivíduo na população. A cada nova geração é mantido o melhor indivíduo da geração anterior. Com isso é possível sempre garantir que o melhor escalonamento não será perdido na nova geração.

4) Seleção

Como operador de seleção foram utilizados Ranking e Roleta Viciada. No primeiro passo todos os cromossomos são ordenados de acordo com valor do seu fitness. Utilizando esse ordenamento, é calculado um novo fitness de acordo com a equação $0.9 + (1.1 - 0.9) * (\mathrm{i} \div (100.0 - 1.0))$. Após o cálculo do novo fitness é utilizado o operador de seleção Roleta Viciada.

5) Crossover

Como operador de crossover foi utilizado o operador OX3. Neste operador são sorteados 2 pontos de corte. Com estes 2 pontos de corte os cromossomos dos pais são segmentados em 3 blocos. O primeiro filho rá receber todos os genes do segundo bloco do primeiro pai. Então o restante do cromossomo desse filho será preenchido com os genes restante do segundo pai, respeitando a ordem em que estão no cromossomo, começando pela primeira posição vazia. O segundo filho será preenchido da mesma forma substituindo a função dos pais no processo. A figura 1 a seguir mostra um exemplo desse operador

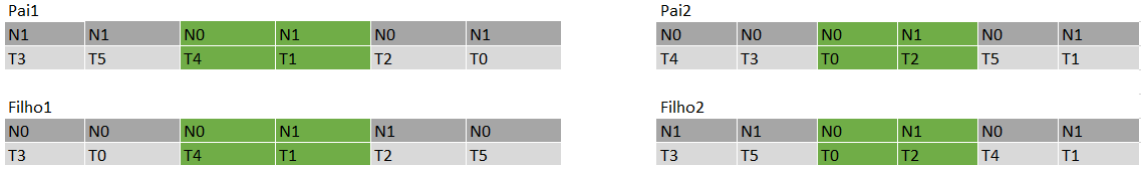


Figura 1. Exemplo do crossover "Order Crossover 3".

Para este operador foi necessário implementar o algoritmo que compara se os genes são iguais. Para este AG, o gene é considerado o mesmo se fazem referência a mesma tarefa. Isso é verificado comparando o valor do atributo Identifier.

6) Mutação

Como operador de mutação foi utilizado o operador Twors. Este operador troca a posição de 2 genes selecionados de forma aleatória. No cromossomo deste trabalho a mutação altera a prioridade das 2 tarefas que trocam de posição entre si.

7. Critério de qualidade do escalonamento

Para avaliar os escalonamentos gerados foram utilizados dos critérios de avaliação. O primeiro critério verifica se todas as tarefas foram executadas dentro dos seus deadlines. O segundo critério avalia a folga entre o deadline e a execução da tarefa.

No cenário proposto para este trabalho, todas as tarefas a serem escalonadas precisam necessariamente alcançar seus deadlines. Ainda nesse critério é verificado se, para as tarefas que falharam, existe pelo menos uma tarefa mais prioritária e com um período maior. Isso é feito para aplicar a mesma lógica do RM para penalizar as tarefas que não alcançaram os seus deadlines. Com isso, o critério de avaliação I se resume a quantidade de tarefas que falharem em respeitar o seu deadline somado ao total de situações onde houve uma tarefa menos prioritária que outra com um período maior. Para essa avaliação o melhor valor possível é zero, onde o escalonamento é válido pois a todas as tarefas respeitaram seus respectivos deadlines. Esse critério é resumido por meio da equação:

$$A_I = \left(\sum_{\forall i} 1 \quad \text{if } \forall i\colon D_i - R_i < 0\right) + \left(\sum_{\forall i} 1 \quad \text{if } \forall i\colon D_i - R_i < 0 \wedge \exists_J\colon PR_j < PR_i \wedge P_j < P_i\right)$$

Onde D, R e PR, P são deadline, instante final de execução, prioridade e período da tarefa, respectivamente.

O critério de avaliação II é o tempo de sobra que as tarefas podem apresentar dentro da sua execução. Quanto maior for a diferença entre o tempo de execução da tarefa e seu deadline, melhor será o escalonamento, ja que ele oferece melhores possibilidades para aumentar a quantidade de tarefas para o conjunto e também a possibilidade de alterar as características temporais de algumas tarefas sem precisar gerar um novo escalonamento. O segundo critério pode ser consolidado na equação demonstrado a seguir.

$$A_{II} = \sum_{\forall i} D_i - R_i$$

8. Fitness

O fitness é calculado utilizando os critérios I e II de avaliação apresentados Anteriormente. Como todo o escalonamento gerado deve ser válido, caso o primeiro critério apresente qualquer tarefa que não tenha sido executada dentro do seu deadline, o segundo critério não será calculado para este cromossomo.

Para a simulação é gerado um script em Python que é executado pelo SimSo. Nesta ferramenta são executadas simulações de escalonamentos para Sistemas de tempo

real em um ambiente distribuído. É utilizado um arquivo que possui um template padrão para geração desse script. Cada nó é simulado separadamente. Assim, é gerado um script para cada um. Na primeira parte do script é definido o parâmetro da duração da simulação. Para os testes realizados a duração da simulação foi definida em 200 ms. Na próxima parte do script são adicionadas todas as tarefas que foram alocadas para o nó a ser simulado, tendo as prioridades das mesmas definida pelo Algoritmo Genetico. Na terceira parte do script são instanciados todos os processadores do nó juntamente com o escalonador de prioridade fixa. Na última parte é fornecido um algoritmo que gera as informações para a avaliação do escalonamento após a execução da simulação.

O SimSo oferece 2 tipos de simulação para sistemas distribuídos de tempo real. A simulação do tempo médio de execução, ACET, ou simulação do pior tempo de resposta, (WCET). A simulação do tempo médio de execução é uma execução mais branda do sistema onde será considerado o tempo médio de execução de cada tarefa, por isso uma simulação mais simples de se trabalhar. A da simulação do pior tempo de resposta, por outro lado, utiliza o pior tempo de execução das tarefas, sendo mais complexa por usar os limites de execução de cada tarefa. Porém essa simulação oferece maiores garantias para o comportamento do sistema, pois cobre uma maior quantidade de cenários quando comparada a simulação ACET. É importante observar que a WCET é uma simulação de um cenário teórico considerando o pior caso possível para todas as tarefas. Para esse trabalho todas as simulações estão sendo realizados no tipo WCET.

A avaliação do fitness é realizada após a simulação de cada nó. Depois que toda a simulação acontece, é verificado no conjunto de tarefas se pelo menos uma delas não alcançou seu deadline. Caso isso aconteça, é contabilizado o número de tarefas que não alcançaram seu deadline. Esse somatório é utilizado como uma penalização na avaliação do escalonamento, ou seja, quanto maior esse número pior a avaliação no primeiro critério. Caso todas tarefas executem dentro dos seus respectivos deadlines, é contabilizado a diferença entre o deadline e o tempo de execução, sendo esse número a avaliação positiva no segundo critério. Essa avaliação pode ser resumida na mostrada na equação.

$$\mathrm{AC_N} = \begin{cases} \mathrm{A_{II}} \ \text{se} \ \forall \mathrm{n}: \mathrm{A_I} = 0 \\ \mathrm{A_I} \ \text{se} \ \exists \mathrm{n}: \mathrm{A_I} < 0 \end{cases}$$

AC é avaliação geral do nó. $\mathrm{A_I}$ e $\mathrm{A_{II}}$ são as avaliações do primeiro e segundo critério respectivamente.

A avaliação de cada nó é adicionada na avaliação geral do escalonamento. Se pelo menos um nó tiver uma avaliação negativa (pelo menos uma tarefa não executou antes do seu deadline), a avaliação geral será o somatório das avaliações de todos os nós com uma avaliação negativa. Assim a avaliação do escalonamento será o somatório de todas as tarefas que não alcançaram seus deadlines em todos os nós. Caso as avaliações dos nós sejam positivas, a avaliação final será o somatório de todas as diferenças entre o deadline e o tempo de execução das tarefas. Então o fitness é definido pela seguinte equação.

$$F = \begin{cases} \sum_{n=1} AC_n \ \ se \ \forall n: AC_n \geq 0 \\ \sum_{J=1} AC_j \ \ \ se \ \exists_J: AC_j < 0 \end{cases}$$

9. Testes

Para testar o algoritmo desenvolvido foram criados 6 cenários de testes. Para os 4 primeiros cenários foram geradas 50 tarefas periódicas no SimSo, para serem escalonadas em 2 nós de processamento, cada um com 2 processadores, conforme descrito no capítulo 4. Para o quinto cenário, foram geradas 75 tarefas que foram executadas em 3 nós de processamento com 2 processadores cada. Para o último cenário foram executadas 100 tarefas em 4 nós de processamento, também com 2 processadores.

Na tabela 1 são demonstrados os resultados consolidados obtidos na execução dos cenários com o mesmo número de tarefas e nós de processamento, sendo somente a taxa de utilização diferente.

Tabela 1. Consolidação das execuções de teste com aumento da taxa de utilização

Cenário	Nº Execuções	Nº Execuções Encontraram Solução Válida	Melhor Fitness
1	7	7	2094,094849
2	7	7	1981,096775
3	7	7	1676,200414
4	8	7	1302,976756

Conforme observado na Tabela 1 é possível identificar que em todos os cenários foi possível encontrar uma solução válida. Porém no cenário 4 onde a taxa de utilização estava em 93%, houve um caso onde o AG falhou em encontrar um escalonamento válido. Podemos observar também que a medida em que a taxa de utilização aumentou, menores foram os tempos de folga que o algoritmo conseguiu encontrar, principalmente com uma taxa de utilização igual ou maior a 90%.

Na figura 2 a seguir isso é demonstrado com a consolidação de todas as execuções em um único gráfico. Para esse gráfico foi utilizado a média dos *fitness* encontrada em cada geração. Nele é possível perceber que o algoritmo tem um comportamento padrão com uma taxa de utilização de até 90%. Porém quando a taxa de utilização foi aumentada além disso, os *fitness* encontrados foram menores.

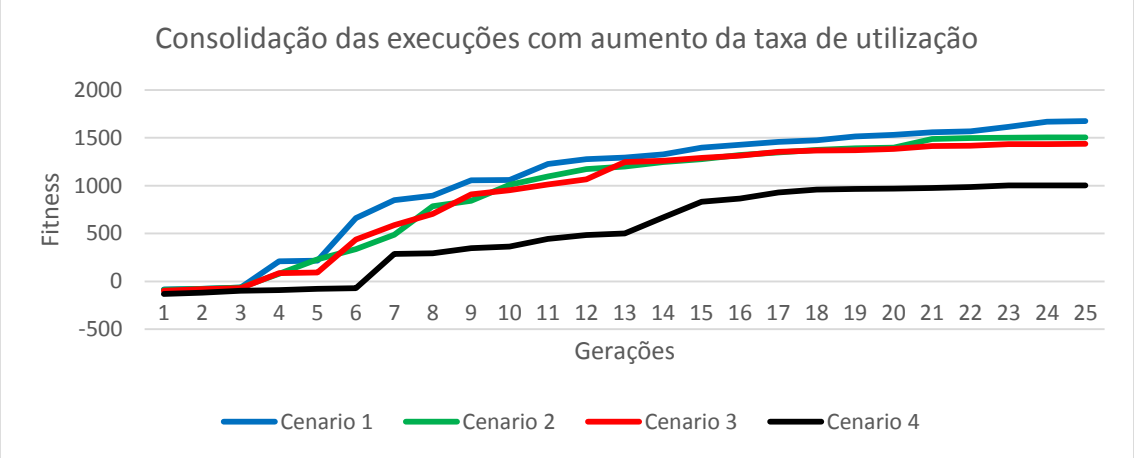


Figura 2. Consolidação das execuções com aumento da taxa de utilização

Na proxima tabela são demonstrado resultados consolidados dos cenários que possuem a mesma taxa de utilização, porém tem quantidades diferentes de tarefas e nós de processamento. A Tabela 2 demonstra a consolidação do resultado dessas execuções.

Tabela 2. Consolidação das execuções de teste com maior quantidade de tarefas e nós de processamento

Cenário	Nº Execuções	Nº Execuções Encontraram Solução Válida	Melhor Fitness
1	7	7	2094,094849
4	7	7	1610,938112
5	7	7	1457,235449

Assim como na análise dos cenários com diferentes taxas de utilização, foi encontrada uma solução válida em todos os cenários. Porém, nestes casos, todas as execuções encontraram soluções que cumpriam seus deadlines. Na medida em que os números de tarefas e nós de processamento aumentaram, menores foram os melhores tempos de folga encontrados pelo algoritmo. Isso acontece, pois, o espaço de busca aumenta com maiores números de tarefas e nós de processamento, e sendo assim, com mais possibilidades a serem verificadas.

Assim como na Figura 2, a Figura 3 demonstra as execuções destes cenários com as médias dos fitness encontradas em cada geração. Analisando os dados é notável que nestes cenários o AG encontrou resultados com uma maior quantidade de tarefas que não respeitaram seus deadlines no início. A partir do momento em que encontrou soluções válidas, os tempos de folga encontrados foram geralmente mais baixos para os cenários com maiores quantidades de tarefas e nós de processamento.

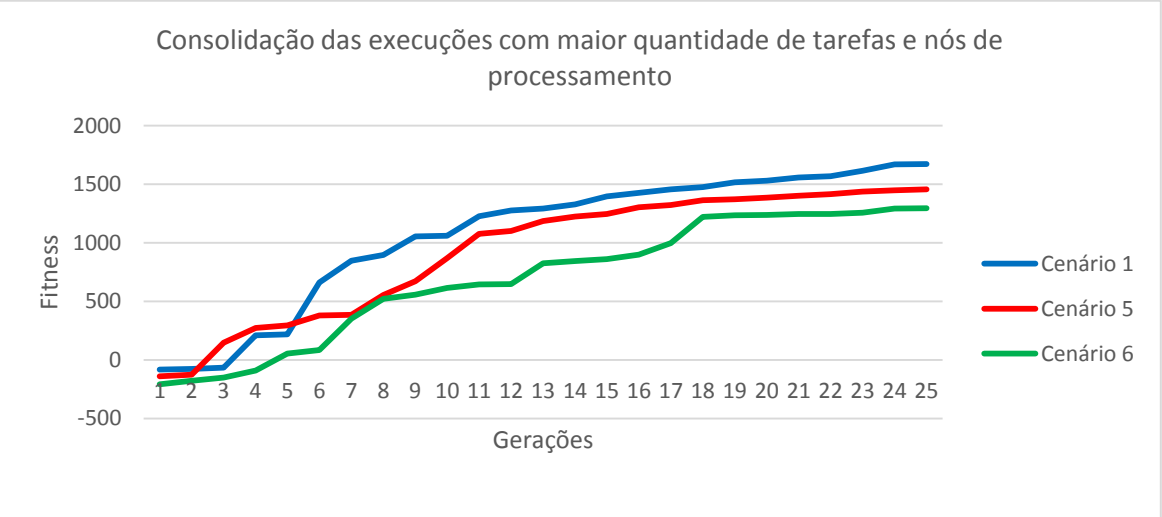


Figura 3. Consolidação das execuções com maior quantidade de tarefas e nós de processamento.

10. Conclusões

Neste artigo foi apresentado uma revisão conceitual sobre AG e STR. Essa revisão em conjunto com os trabalhos relacionados apresentados serviu de embasamento para o desenvolvimento de um AG capaz de receber um conjunto de tarefas com restrições temporais que devem ser executadas em um cluster simulado. Utilizando desse embasamento foi desenvolvido um AG que consegue definir a alocação das tarefas em processadores e a prioridade que as mesmas terão durante a sua execução. Esse algoritmo consegue encontrar esses escalonamentos sem precisar alterar as características definidas inicialmente para as tarefas como *deadline*, período e pior tempo de execução.

Fez-se necessário a pesquisa pela melhor forma para avaliar os resultados encontrados no AG. Foi definido a utilização de simulações para avaliar os escalonamentos gerados. Para isso foi utilizado a ferramenta SimSo capaz de receber os escalonamentos gerados em arquivo XML e simular a sua execução. Essa ferramenta gera os dados necessários para a avaliação do *fitness*, sendo esse utilizado para verificar a viabilidade e a qualidade do escalonamento.

Com o embasamento teórico e uma ferramenta capaz de prover informações para avaliação dos escalonamentos encontrados este AG desenvolvido neste projeto foi capaz de encontrar escalonamentos válidos para um modelo simplificado de tarefas. Utilizando de técnicas como cromossomo permutacional para representar o problema de uma forma computacional o problema. Com isso o algoritmo conseguiu encontrar boas soluções no espaço de busca.

Utilizando os dados gerados durante a execução do *Genetic Scheduler* foi possível executar teste qualitativos no AG e os escalonamentos gerados pelo mesmo. Podendo assim confirmar que o algoritmo consegue alcançar seu objetivo principal de encontrar escalonamentos válidos e ainda consegue melhorar esses para que o tempo de folga seja o maior possível, gerando com isso escalonamentos mais robustos e flexíveis.

Pode ser concluído que a utilização de AG para a geração de escalonamento de STR se mostra uma opção interessante. Porém ainda necessário realizar testes em cenários reais e com um modelo mais complexo de tarefas para confirmar a viabilidade dessa solução.

11. Referências

LINDEN, Ricardo. Algoritmos genéticos .3a ediçao. Rio de Janeiro: Ciência Moderna, 2012.

FARINES, Jean-Marie; FRAGA, Joni da Silva; OLIVEIRA, RS de. Sistemas de tempo real. Escola de Computação, v. 2000, p. 201, 2000.

YOO, Myungryun; GEN, Mitsuo. Scheduling algorithm for real-time tasks using multiobjective hybrid genetic algorithm in heterogeneous multiprocessors system. Computers & Operations Research, v. 34, n. 10, p. 3084-3098, 2007.

GHARSELLAOUI, Hamza et al. Real-time reconfigurable scheduling of multiprocessor embedded systems using hybrid genetic based approach. In: Computer and Information Science (ICIS), 2015 IEEE/ACIS 14th International Conference on. IEEE, 2015. p. 605-609.

AZKETA, Ekain et al. Permutational genetic algorithm for the optimized assignment of priorities to tasks and messages in distributed real-time systems. In: Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on. IEEE, 2011. p. 958-965.

AZKETA, Ekain et al. An empirical study of permutational genetic crossover and mutation operators on the fixed priority assignment in distributed real-time systems. In: Industrial Technology (ICIT), 2012 IEEE International Conference on. IEEE, 2012. p. 598-605.

SEBESTYEN, Gheorghe; HANGAN, Anca. Genetic approach for real-time scheduling on multiprocessor systems. In: Intelligent Computer Communication and Processing (ICCP), 2012 IEEE International Conference on. IEEE, 2012. p. 267-272.

OH, Jaewon; WU, Chisu. Genetic-algorithm-based real-time task scheduling with multiple goals. Journal of systems and software, v. 71, n. 3, p. 245-258, 2004.

KONAR, Debanjan et al. An Improved Hybrid Quantum-Inspired Genetic Algorithm (HQIGA) for Scheduling of Real-Time Task in Multiprocessor System. Applied Soft Computing, 2017.

HAO, Lu; YANG, Xiaopeng; HU, Shangkun. Task scheduling of improved time shifting based on genetic algorithm for phased array radar. In: Signal Processing (ICSP), 2016 IEEE 13th International Conference on. IEEE, 2016. p. 1655-1660.

CHÉRAMY, Maxime et al. Simulation of real-time scheduling with various execution time models. In: Industrial Embedded Systems (SIES), 2014 9th IEEE International Symposium on. IEEE, 2014. p. 1-4.

GeneticSharp. (07 do 03 de 2017). Acesso em 07 do 03 de 2017, disponível em https://github.com/giacomelli/GeneticSharp

Simso. (07 do 03 de 2017). Acesso em 07 do 03 de 2017, disponível em http://projects.laas.fr/simso/

8.2. Código Fonte

```
using System;
using GeneticSharp.Domain.Selections;
using GeneticSharp.Domain.Mutations;
using GeneticSharp.Domain.Populations;
using GeneticSharp.Domain;
using GeneticSharp.Domain.Terminations;
using GeneticSharp.Infrastructure.Threading;
using GeneticScheduler.Chromosome;
using GeneticScheduler.Fitness;
using GeneticScheduler.Crossover;
using GeneticScheduler.Reinsertion;
using System.Text;
using System.IO;

namespace GeneticScheduler
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var selection = new TournamentSelection(3, true);
            var crossover = new CustomOrderedCrossover(new GeneComparer());
            var mutation = new TworsMutation();
            var fitness = new ScheduleFitness();
            var chromosome = new ScheduleChromosome();
            var population = new Population(100, 100, chromosome);
            population.GenerationStrategy = new PerformanceGenerationStrategy();

            var taskExecutor = new SmartThreadPoolTaskExecutor();
            taskExecutor.MinThreads = 50;
            taskExecutor.MaxThreads = 50;


            var ga = new GeneticAlgorithm(population, fitness, selection,
crossover, mutation);

            ga.TaskExecutor = taskExecutor;

            ga.CrossoverProbability = 1;
            ga.MutationProbability = 0.05f;
            ga.Reinsertion = new CustomElitistReinsertion(1);
            ga.Termination = new GenerationNumberTermination(25);

            StringBuilder result = new StringBuilder();

            ga.GenerationRan += delegate
            {
                var bestChromosome = ga.Population.BestChromosome;
                Console.WriteLine("Generations: {0}",
ga.Population.GenerationsNumber);
                Console.WriteLine("Fitness: {0} - File {1}",
bestChromosome.Fitness, ((ScheduleChromosome)bestChromosome).FileIdentifier);
                result.AppendLine(string.Format("Generations: {0}",
ga.Population.GenerationsNumber));
                result.AppendLine(string.Format("Fitness: {0} - File {1}",
bestChromosome.Fitness, ((ScheduleChromosome)bestChromosome).FileIdentifier));
```

```
                if(ga.GenerationsNumber == 25)
                {
                    StreamWriter file = new StreamWriter(@"");
                    file.Write(result.ToString());
                    file.Close();
                }
            };

            Console.WriteLine("GA running...");
            ga.Start();

            Console.WriteLine("Best solution found has {0} fitness.",
ga.BestChromosome.Fitness);

            GeneticAlgorithmResultWritter resultWritter = new
GeneticAlgorithmResultWritter();
            resultWritter.createResult((ScheduleChromosome)ga.BestChromosome);

            Console.Read();
        }
    }
}


using System;
using System.Collections.Generic;
using GeneticSharp.Domain.Chromosomes;

namespace GeneticScheduler.Chromosome
{
    public class GeneComparer : IEqualityComparer<Gene>
    {
        public bool Equals(Gene x, Gene y)
        {
            //Check whether the objects are the same object.
            if (Object.ReferenceEquals(x, y)) return true;

            //Check whether the products' properties are equal.
            return x != null && y != null && ((TaskValue)(x.Value)).Identifier ==
((TaskValue)(y.Value)).Identifier;
        }

        public int GetHashCode(Gene obj)
        {
            TaskValue value = obj.Value as TaskValue;
            return value.Identifier.GetHashCode();
        }
    }
}


using GeneticSharp.Domain.Chromosomes;
using GeneticSharp.Domain.Randomizations;
using System;
using GeneticScheduler.Configuration;

namespace GeneticScheduler.Chromosome
{
    public class ScheduleChromosome : ChromosomeBase
    {
```

```
        private static int chromosomeCount = 0;
        private int[] taskList;

        public int FileIdentifier = 0;

        public ScheduleChromosome() : base(100)
        {
            TaskSetLoader setLoader = TaskSetLoader.GetLoader();

            this.taskList =
RandomizationProvider.Current.GetUniqueInts(setLoader.TasksList.Count, 1,
setLoader.TasksList.Count + 1);
            for (int i =0; i < Length; i++)
            {
                ReplaceGene(i, GenerateNewGene(i));
            }

            FileIdentifier = chromosomeCount;
            chromosomeCount++;
        }

        public override IChromosome CreateNew()
        {
            return new ScheduleChromosome();
        }

        public override Gene GenerateGene(int geneIndex)
        {
            throw new NotImplementedException();
        }

        public Gene GenerateNewGene(int geneIndex)
        {
            TaskSetLoader setLoader = TaskSetLoader.GetLoader();
            int taskIdentifier = this.taskList[geneIndex];
            int cluster = RandomizationProvider.Current.GetInt(0,
setLoader.ClusterNumber());
            TaskValue value = new TaskValue(taskIdentifier, cluster);
            return new Gene(value);
        }
    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GeneticScheduler.Chromosome
{
    public class TaskValue
    {
        public int Identifier { get; set; }
        public int ClusterIdentifier { get; set; }

        public TaskValue(int taskidentifier)
        {
            this.Identifier = taskidentifier;
```

```
        }

        public TaskValue(int taskidentifier, int clusterIdentifier)
        {
            this.Identifier = taskidentifier;
            this.ClusterIdentifier = clusterIdentifier;
        }
    }
}


using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Xml;
using GeneticScheduler.Model;

namespace GeneticScheduler.Configuration
{
    public  class TaskSetLoader
    {
        private const string fileName = @"";
        private const string simulationKey = @"simulation";
        private const string tasksKey = @"tasks";
        private const string taskKey = @"task";

        private const string taskIdKey = @"id";
        private const string taskPeriodKey = @"period";
        private const string taskWCETKey = @"WCET";
        private const string taskDeadlineKey = @"deadline";

        private const int clusterNumber = 4;
        private const int processorNumberCluster = 2;

        private static TaskSetLoader Singleton;

        public List<Model.Task> TasksList { get; set; }

        public static TaskSetLoader GetLoader()
        {
            if(TaskSetLoader.Singleton == null)
            {
                TaskSetLoader.Singleton = new TaskSetLoader();
                TaskSetLoader.Singleton.LoadTasks();
            }
            return TaskSetLoader.Singleton;
        }

        private TaskSetLoader()
        {
            this.TasksList = new List<Model.Task>();
        }

        private void LoadTasks()
        {
            XmlReader xmlReader = XmlReader.Create(fileName);
            while (xmlReader.Read())
            {
                if(xmlReader.NodeType == XmlNodeType.XmlDeclaration)
                {
```

```
                    while(xmlReader.Read())
                    {
                        if ((xmlReader.NodeType == XmlNodeType.Element) &&
(xmlReader.Name == simulationKey))
                        {
                            while (xmlReader.Read())
                            {
                                if ((xmlReader.NodeType == XmlNodeType.Element)
&& (xmlReader.Name == tasksKey))
                                {
                                    while (xmlReader.Read())
                                    {
                                        if ((xmlReader.NodeType ==
XmlNodeType.Element) && (xmlReader.Name == taskKey))
                                        {
                                            int identifier =
int.Parse(xmlReader.GetAttribute(taskIdKey));
                                            double period =
double.Parse(xmlReader.GetAttribute(taskPeriodKey).Replace(".", ","));
                                            double worstExecTime =
double.Parse(xmlReader.GetAttribute(taskWCETKey).Replace(".", ","));
                                            double deadline =
double.Parse(xmlReader.GetAttribute(taskDeadlineKey).Replace(".", ","));
                                            Model.Task task = new
Model.Task(identifier, period, worstExecTime, deadline, 0);
                                            this.TasksList.Add(task);
                                        }
                                    }
                                }
                            }
                        }

                    }
                }
            }
        }

        public Model.Task getTaskById(int taskIndex)
        {
            return this.TasksList.ElementAt(taskIndex).Duplicate();
        }

        public int ClusterNumber()
        {
            return clusterNumber;
        }

        public int ProcessorsInClusterNumber()
        {
            return processorNumberCluster;
        }

        public List<int> Clusters()
        {
            List<int> clusters = new List<int>();
            int clusterNumber = this.ClusterNumber();
            for (int i = 0; i < clusterNumber; i++)
            {
                clusters.Add(i);
            }
            return clusters;
```

```
        }
    }
}


using GeneticSharp.Domain.Chromosomes;
using GeneticSharp.Domain.Crossovers;
using GeneticSharp.Domain.Randomizations;
using System;
using System.Collections.Generic;
using System.Linq;

using GeneticScheduler.Chromosome;

namespace GeneticScheduler.Crossover
{
    public class CustomOrderedCrossover : CrossoverBase
    {
        private IEqualityComparer<Gene> comparer;

        #region Constructors
        /// <summary>
        /// Initializes a new instance of the <see
cref="GeneticSharp.Domain.Crossovers.OrderedCrossover"/> class.
        /// </summary>
        public CustomOrderedCrossover(IEqualityComparer<Gene> comparer)
            : base(2, 2)
        {
            IsOrdered = true;
            this.comparer = comparer;
        }
        #endregion

        #region Methods
        /// <summary>
        /// Performs the cross with specified parents generating the children.
        /// </summary>
        /// <param name="parents">The parents chromosomes.</param>
        /// <returns>The offspring (children) of the parents.</returns>
        protected override IList<IChromosome> PerformCross(IList<IChromosome>
parents)
        {
            var firstParent = parents[0];
            var secondParent = parents[1];

            if (parents.AnyHasRepeatedGene())
            {
                throw new CrossoverException(this, "The Ordered Crossover (OX1)
can be only used with ordered chromosomes. The specified chromosome has repeated
genes.");
            }

            var middleSectionIndexes =
RandomizationProvider.Current.GetUniqueInts(2, 0, firstParent.Length);
            Array.Sort(middleSectionIndexes);
            var middleSectionBeginIndex = middleSectionIndexes[0];
            var middleSectionEndIndex = middleSectionIndexes[1];
            var firstChild = CreateChild(firstParent, secondParent,
middleSectionBeginIndex, middleSectionEndIndex, this.comparer);
```

```
            var secondChild = CreateChild(secondParent, firstParent,
middleSectionBeginIndex, middleSectionEndIndex, this.comparer);

            return new List<IChromosome>() { firstChild, secondChild };
        }

        /// <summary>
        /// Creates the child.
        /// </summary>
        /// <returns>The child.</returns>
        /// <param name="firstParent">First parent.</param>
        /// <param name="secondParent">Second parent.</param>
        /// <param name="middleSectionBeginIndex">Middle section begin
index.</param>
        /// <param name="middleSectionEndIndex">Middle section end index.</param>
        private static IChromosome CreateChild(IChromosome firstParent,
IChromosome secondParent, int middleSectionBeginIndex, int middleSectionEndIndex,
IEqualityComparer<Gene> comparer)
        {
            var middleSectionGenes =
firstParent.GetGenes().Skip(middleSectionBeginIndex).Take((middleSectionEndIndex
- middleSectionBeginIndex) + 1);
            var secondParentRemainingGenes =
secondParent.GetGenes().Except(middleSectionGenes, comparer).GetEnumerator();
            var child = firstParent.CreateNew();

            for (int i = 0; i < firstParent.Length; i++)
            {
                var firstParentGene = firstParent.GetGene(i);

                if (i >= middleSectionBeginIndex && i <= middleSectionEndIndex)
                {
                    child.ReplaceGene(i, firstParentGene);
                }
                else
                {
                    secondParentRemainingGenes.MoveNext();
                    child.ReplaceGene(i, secondParentRemainingGenes.Current);
                }
            }

            return child;
        }
        #endregion
    }
}

using GeneticScheduler.SimSo;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GeneticScheduler.Fitness
{
    interface ISimulator
    {
        SimulationResult Simulate(int processorsNumbers, List<Model.Task> tasks,
string filePath);
    }
```

```
}


using GeneticSharp.Domain.Fitnesses;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using GeneticSharp.Domain.Chromosomes;
using System.Threading;
using GeneticScheduler.Configuration;
using GeneticScheduler.Chromosome;
using GeneticScheduler.SimSo;

namespace GeneticScheduler.Fitness
{
    public class ScheduleFitness : IFitness
    {

        public double Evaluate(IChromosome chromosome)
        {
            TaskSetLoader setLoader = TaskSetLoader.GetLoader();
            int processorsNumber = setLoader.ProcessorsInClusterNumber();
            List<int> clusters = setLoader.Clusters();

            double scheduleIndex = 0;
            double fitness = 0;
            Gene[] genes = chromosome.GetGenes();

            Parallel.ForEach(clusters, (cluster) =>
            {
                string filePath =
string.Format(@"C:\Users\RuannM\Desktop\Analysys\File_{0}_{1}.py",
((ScheduleChromosome)chromosome).FileIdentifier, cluster);
                List<Model.Task> tasks = new List<Model.Task>();
                int priority = 0;
                double utilizationFactor = 0.0;
                for (int j = 0; j < chromosome.Length; j++)
                {
                    TaskValue value = (TaskValue)genes[j].Value;
                    if (value.ClusterIdentifier == cluster)
                    {
                        Model.Task task = setLoader.getTaskById(value.Identifier
- 1);
                        task.Priority = priority++;

                        utilizationFactor += task.WorstExecutionTime /
task.Period;

                        tasks.Add(task);
                    }
                }

                ISimulator simulator = new SimSoModelGenerator();

                SimulationResult simulationResult =
simulator.Simulate(processorsNumber, tasks, filePath);

                double newscheduleIndex = simulationResult.scheduleIndex;
```

```
                if (newscheduleIndex < 0)
                {
                    for (int j = 0; j < chromosome.Length; j++)
                    {
                        bool hasNotFoundLowerPeriod = true;
                        TaskValue value = (TaskValue)genes[j].Value;
                        if (value.ClusterIdentifier == cluster &&
simulationResult.missTasks.Contains(value.Identifier))
                        {
                            Model.Task task =
setLoader.getTaskById(value.Identifier - 1);
                            for (int i = 0; i < j; i++)
                            {
                                TaskValue previousValue =
(TaskValue)genes[i].Value;
                                if (previousValue.ClusterIdentifier == cluster)
                                {
                                    Model.Task previousTask =
setLoader.getTaskById(previousValue.Identifier - 1);
                                    if (hasNotFoundLowerPeriod && task.Period <
previousTask.Period)
                                    {
                                        hasNotFoundLowerPeriod = false;
                                        newscheduleIndex -= 1;
                                    }
                                }
                            }
                        }
                    }


                }

                if (newscheduleIndex < 0 && scheduleIndex > 0)
                {
                    scheduleIndex = 0;
                }

                scheduleIndex += newscheduleIndex;
            }
           );

            fitness = scheduleIndex;

            return fitness;
        }
    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GeneticScheduler.Model
{
    public class Task
```

```
    {
        public int Identifier { get; set; }
        public double Period { get; set; }
        public double WorstExecutionTime { get; set; }
        public double Deadline { get; set; }
        public int Priority  { get; set; }
        public int ClusterIdentifier { get; set; }

        public Task()
        {

        }

        public Task(int identifier, double period, double worstExecutionTime,
double deadline, int priority)
        {
            this.Identifier = identifier;
            this.Period = period;
            this.WorstExecutionTime = worstExecutionTime;
            this.Deadline = deadline;
            this.Priority = priority;
        }

        public Task Duplicate()
        {
            Task task = new Task();
            task.Identifier = this.Identifier;
            task.Period = this.Period;
            task.WorstExecutionTime = this.WorstExecutionTime;
            task.Deadline = this.Deadline;
            task.Priority = this.Priority;
            task.ClusterIdentifier = this.ClusterIdentifier;
            return task;
        }
    }
}


using GeneticSharp.Domain.Chromosomes;
using GeneticSharp.Domain.Populations;
using GeneticSharp.Domain.Reinsertions;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GeneticScheduler.Reinsertion
{
    class CustomElitistReinsertion : ReinsertionBase
    {
        private int ParentNumber;

        #region Constructors
        /// <summary>
        /// Initializes a new instance of the <see
cref="GeneticSharp.Domain.Reinsertions.ElitistReinsertion"/> class.
        /// </summary>
        public CustomElitistReinsertion() : base(false, true)
        {
```

```
        }

        public CustomElitistReinsertion(int parentsNumber) : base(false, true)
        {
            this.ParentNumber = parentsNumber;
        }
        #endregion

        #region Methods
        /// <summary>
        /// Selects the chromosomes which will be reinserted.
        /// </summary>
        /// <returns>The chromosomes to be reinserted in next
generation..</returns>
        /// <param name="population">The population.</param>
        /// <param name="offspring">The offspring.</param>
        /// <param name="parents">The parents.</param>
        protected override IList<IChromosome>
PerformSelectChromosomes(IPopulation population, IList<IChromosome> offspring,
IList<IChromosome> parents)
        {
            if (this.ParentNumber > 0)
            {
                var bestParents = parents.OrderByDescending(p =>
p.Fitness).Take(this.ParentNumber);

                foreach (var p in population.CurrentGeneration.Chromosomes)
                {
                    offspring.Add(p);
                }
            }

            return offspring;
        }

        public IChromosome GetWorstOffspring(IList<IChromosome> offspring)
        {
            IChromosome worstOffspring = offspring.ElementAt(0);
            foreach (var nextOffspring in offspring)
            {
                if (worstOffspring.Fitness > nextOffspring.Fitness)
                {
                    worstOffspring = nextOffspring;
                }
            }
            return worstOffspring;
        }
        #endregion
    }
}



using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using GeneticScheduler.Model;
```

```
using GeneticScheduler.Fitness;

namespace GeneticScheduler.SimSo
{
    public class SimSoModelGenerator : ISimulator
    {
        private const string pythonIndent = "    ";
        public SimulationResult Simulate(int processorsNumbers, List<Model.Task>
tasks, string filePath)
        {
            StringBuilder builder = new StringBuilder();
            builder.AppendLine(GetHeader());

            foreach(Model.Task task in tasks)
            {
                builder.Append(pythonIndent); builder.Append(pythonIndent);
                builder.AppendLine(GetTask("T" + task.Identifier,
task.Identifier, task.Period, task.WorstExecutionTime, task.Deadline,
task.Priority, task.ClusterIdentifier));
            }

            for(int i = 0; i < processorsNumbers; i++)
            {
                builder.Append(pythonIndent); builder.Append(pythonIndent);
                builder.AppendLine(GetProcessor("CPU" + i, i));
            }

            builder.Append(GetFinalPart());

            string fileString = builder.ToString();

            StreamWriter file = new StreamWriter(filePath);
            file.Write(fileString);

            //Console.WriteLine(filePath);

            file.Close();
            return this.run(filePath);
        }

        private SimulationResult run(string filePath)
        {
            ProcessStartInfo start = new ProcessStartInfo();
            start.FileName = "python";
            start.Arguments = filePath;
            start.UseShellExecute = false;
            start.RedirectStandardOutput = true;
            start.WorkingDirectory = @"C:\Users\RuannM";
            using (Process process = Process.Start(start))
            {
                using (StreamReader reader = process.StandardOutput)
                {
                    string result = reader.ReadToEnd();
                    string abortCount = result.Split('*', '*')[1].Replace('.',
',');
                    List<int> missTask = new List<int>();
                    if (result.IndexOf("[") > 0)
                    {
                        string missTaskString = result.Split('[', ']')[1];
                        missTask = missTaskString.Substring(0,
missTaskString.Count() - 1).Split(',').Select(Int32.Parse).ToList();
```

```
                    }

                    return new SimulationResult(Convert.ToDouble(abortCount),
missTask);
                }
            }
        }

        static void cmd_DataReceived(object sender, DataReceivedEventArgs e)
        {
            Console.WriteLine("Output from other process");
            Console.WriteLine(e.Data);
        }


        private string GetHeader()
        {
            StringBuilder builder = new StringBuilder();
            builder.AppendLine("import sys");
            builder.AppendLine("from simso.core import Model");
            builder.AppendLine("from simso.configuration import Configuration");
            builder.AppendLine("def main(argv):");
            builder.Append(pythonIndent);
            builder.AppendLine("if len(argv) == 2:");
            builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine("configuration = Configuration(argv[1])");
            builder.Append(pythonIndent);
            builder.AppendLine("else:");
            builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine("configuration = Configuration()");
            builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine("configuration.duration = 200 *
configuration.cycles_per_ms");

            string header = builder.ToString();

            return header;
        }

        private string GetTask(string name, double identifier, double period,
double worstExecutionTime, double deadline, double priority, double processor)
        {
            string task = string.Format(this.GetTaskConfigurationString(),
                name,
                identifier,
                period.ToString("0.00",
System.Globalization.CultureInfo.InvariantCulture),
                worstExecutionTime.ToString("0.00",
System.Globalization.CultureInfo.InvariantCulture),
                deadline.ToString("0.00",
System.Globalization.CultureInfo.InvariantCulture),
                priority);
            return task;
        }

        private string GetTaskConfigurationString()
        {
            return @"configuration.add_task(name=""{0}"", abort_on_miss=True,
identifier={1}, period={2}, activation_date=0, wcet={3}, deadline={4},
data={{""priority"":{5}}})";
        }
```

```
        private string GetProcessor(string name, int identifier)
        {
            string processor =
string.Format(@"configuration.add_processor(name=""{0}"", identifier={1})", name,
identifier);
            return processor;
        }

        private string GetFinalPart()
        {
            StringBuilder builder = new StringBuilder();
            builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine(@"configuration.scheduler_info.clas =
""simso.schedulers.FP""");
            builder.Append(pythonIndent);
            builder.AppendLine(@"configuration.check_all()");
            builder.Append(pythonIndent);
            builder.AppendLine(@"model = Model(configuration)");
            builder.Append(pythonIndent);
            builder.AppendLine(@"model.run_model()");
            builder.Append(pythonIndent);
            builder.AppendLine(@"scheduleIndex = 0");
            builder.Append(pythonIndent);
            builder.AppendLine(@"negativeScheduleIndex = 0");
            builder.Append(pythonIndent);
            builder.AppendLine(@"missTasks = """""");

            builder.Append(pythonIndent);
            builder.AppendLine(@"for task in model.results.tasks.values():");
            builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine(@"for job in task.jobs:");
            builder.Append(pythonIndent); builder.Append(pythonIndent);
builder.Append(pythonIndent);
            builder.AppendLine(@"if job.aborted:");
            builder.Append(pythonIndent); builder.Append(pythonIndent);
builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine(@"negativeScheduleIndex -= 1");
            builder.Append(pythonIndent); builder.Append(pythonIndent);
builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine(@"missTasks += str(job.task.identifier) + "","""");

            builder.Append(pythonIndent);
            builder.AppendLine(@"if negativeScheduleIndex < 0:");

            builder.Append(pythonIndent); builder.Append(pythonIndent);

builder.AppendLine(@"print(""scheduleIndex:*{0}*"".format(negativeScheduleIndex))
");
            builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine(@"print(""missTasks:[{0}]"".format(missTasks))");

            builder.Append(pythonIndent);
            builder.AppendLine(@"else:");

            builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine(@"for task in model.results.tasks.values():");
            builder.Append(pythonIndent); builder.Append(pythonIndent);
builder.Append(pythonIndent);
            builder.AppendLine(@"for job in task.jobs:");
```

```
            builder.Append(pythonIndent); builder.Append(pythonIndent);
builder.Append(pythonIndent); builder.Append(pythonIndent);
            builder.AppendLine(@"if job.computation_time and job.end_date:");
            builder.Append(pythonIndent);  builder.Append(pythonIndent);
builder.Append(pythonIndent); builder.Append(pythonIndent);
builder.Append(pythonIndent);
            builder.AppendLine(@"scheduleIndex += job.normalized_laxity");
            builder.Append(pythonIndent); builder.Append(pythonIndent);

builder.AppendLine(@"print(""scheduleIndex:*{0}*"".format(scheduleIndex))");

            builder.AppendLine(@"main(sys.argv)");
            string finalPart = builder.ToString();
            return finalPart;
        }
    }
}


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace GeneticScheduler.SimSo
{
    public class SimulationResult
    {
        public double scheduleIndex { get; set; }
        public List<int> missTasks { get; set; }

        public SimulationResult(double scheduleIndex, List<int> missTasks)
        {
            this.scheduleIndex = scheduleIndex;
            this.missTasks = missTasks;
        }
    }
}


<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5.2" />
    </startup>
</configuration>

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using GeneticScheduler.Chromosome;
using GeneticScheduler.Configuration;
using System.Xml;
using GeneticSharp.Domain.Chromosomes;

namespace GeneticScheduler
{
```

```
    class GeneticAlgorithmResultWritter
    {
        private const string fileName =
@"C:\Users\RuannM\Desktop\results\runResult_10.xml";

        public void createResult(ScheduleChromosome chromosome)
        {
            XmlDocument fileResult = new XmlDocument();
            XmlNode docNode = fileResult.CreateXmlDeclaration("1.0", "UTF-8",
null);
            fileResult.AppendChild(docNode);

            XmlNode scheduleNode = fileResult.CreateElement("schedule");
            fileResult.AppendChild(scheduleNode);

            TaskSetLoader setLoader = TaskSetLoader.GetLoader();

            int clusterNumber = setLoader.ClusterNumber();
            int processorNumber = setLoader.ProcessorsInClusterNumber();
            Gene[] genes = chromosome.GetGenes();

            for (int clusterIdentifier = 0; clusterIdentifier < clusterNumber;
clusterIdentifier++)
            {
                XmlNode clusterNode = fileResult.CreateElement("cluster");
                XmlAttribute clusterAttribute = fileResult.CreateAttribute("id");
                clusterAttribute.Value = clusterIdentifier.ToString();
                clusterNode.Attributes.Append(clusterAttribute);
                scheduleNode.AppendChild(clusterNode);

                for (int j = 0; j < processorNumber; j++)
                {
                    XmlNode processorNode =
fileResult.CreateElement("processor");
                    XmlAttribute processorAttribute =
fileResult.CreateAttribute("id");
                    processorAttribute.Value = j.ToString();
                    processorNode.Attributes.Append(processorAttribute);
                    clusterNode.AppendChild(processorNode);
                }

                for (int priority = 0; priority < chromosome.Length; priority++)
                {
                    TaskValue task = (TaskValue)genes[priority].Value;
                    if (task.ClusterIdentifier == clusterIdentifier)
                    {
                        XmlNode taskNode = fileResult.CreateElement("task");

                        XmlAttribute taskAttribute =
fileResult.CreateAttribute("id");
                        taskAttribute.Value = task.Identifier.ToString();
                        taskNode.Attributes.Append(taskAttribute);

                        XmlAttribute taskPriorityAttribute =
fileResult.CreateAttribute("priority");
                        taskPriorityAttribute.Value = priority.ToString();
                        taskNode.Attributes.Append(taskPriorityAttribute);

                        clusterNode.AppendChild(taskNode);
                    }
                }
```

```
            }

            fileResult.Save(fileName);
        }
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="GeneticSharp" version="1.0.94" targetFramework="net452" />
  <package id="HelperSharp" version="0.0.4.2" targetFramework="net452" />
  <package id="SmartThreadPool.dll" version="2.2.3" targetFramework="net452" />
</packages>
```