ESCOLA POLITÉCNICA DE PERNAMBUCO

d.s.c. UPE Departamento de Sistemas e Computação

UMA FERRAMENTA COMPUTACIONAL PARA ANÁLISE DE ALGORITMOS DE OTIMIZAÇÃO PARA SISTEMAS DINÂMICOS BASEADOS EM INTELIGÊNCIA DE ENXAMES

Trabalho de Conclusão de Curso

Engenharia da Computação

Rodrigo Maia Carneiro de Souza Castro
Orientador: Prof. Carmelo José Albanez Bastos Filho

UNIVERSIDADE DE PERNAMBUCO

RODRIGO MAIA CARNEIRO DE SOUZA CASTRO

UMA FERRAMENTA COMPUTACIONAL PARA ANÁLISE DE ALGORITMOS DE OTIMIZAÇÃO PARA SISTEMAS DINÂMICOS BASEADOS EM INTELIGÊNCIA DE ENXAMES

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Recife, dezembro de 2010.

Rodrigo Maia Carneiro de Souza Castro

Uma Ferramenta Computacional para Análise de Algoritmos de Otimização para Sistemas Dinâmicos Baseados em Inteligência de Enxames

Agradecimentos

Agradeço a toda minha família, em especial aos meus pais, Kleber e Filomena, pela educação que me proporcionaram e que me permitiu chegar até aqui e pelo apoio incondicional que sempre me deram, e à minha irmã, Mariana, por me ajudar em todos os momentos. Sem vocês, eu nunca teria chegado tão longe. À minha namorada, Andréa, por todo carinho, motivação e apoio, essenciais para conclusão deste trabalho.

A todos os professores pelo conhecimento transmitido durante todo o curso. Obrigado ao professor Carmelo Bastos Filho, pela excelente orientação recebida, sempre acompanhada de criticas, incentivos e sugestões para melhorar o trabalho. Obrigado também aos professores Abel Guilhermino e Fernando Castor, pela oportunidade que me deram de desenvolver trabalhos de iniciação científica.

Agradeço a George Cavalcante Júnior, pela orientação e contribuição fornecidas durante todo este trabalho e pela revisão final realizada. Por fim, agradeço a todos os meus amigos de faculdade, com os quais compartilhei esses cinco anos de muito estudo e dedicação.

Resumo

Algoritmos de otimização de problemas dinâmicos vêm sendo cada vez mais estudados pela comunidade científica; e os algoritmos baseados em inteligência de enxames têm se mostrado bastante eficientes na otimização destes problemas. Com o intuito de apoiar os estudos desta área, este trabalho propõe uma ferramenta para simulação e análise de algoritmos de otimização de problemas dinâmicos baseados em inteligência de enxames. A ferramenta proposta provê um *framework* que fornece a infraestrutura básica para implementação e simulação destes algoritmos. Além disso, ele permite também a implementação de problemas de teste dinâmicos e métricas de avaliação de algoritmos. É disponibilizado também um ambiente gráfico para análise dos resultados gerados. As capacidades de simulação e análise da ferramenta proposta são demonstradas em dois estudos de caso apresentados neste trabalho.

Abstract

Optimization algorithms to tackle dynamic problems have gained a lot of attention from the scientific community; and the algorithms based on swarm intelligence have appeared as an efficient alternative to optimize these problems. This work introduces a new tool for simulation and analysis of swarm intelligence algorithms to optimize dynamic problems. The proposed tool provides a framework for implementation and simulation of such algorithms. It also allows the implementation of dynamics problems and metrics to evaluate the performance of the algorithms. Besides, a graphic tool is provided to allow the analysis of the results. We performed two cases of study in this work in order to demonstrate the functionalities of the tool.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

DOSS – *Dynamic Optimization System Simulator.*

DOSA – *Dynamic Optimization System Analyzer.*

FSS – *Fish School Search.*

GDBG – *Generalized Dynamic Benchmark Generator.*

PSO – *Particle Swarm Optimization.*

XML – *Extensible Markup Language.*

Capítulo 1

Introdução

Problemas de otimização dinâmicos são problemas que sofrem alterações com o decorrer do tempo. Essas alterações podem ocorrer tanto nos dados do problema quanto em sua definição. Por isso, algoritmos de otimização para problemas dinâmicos devem não apenas localizar o ótimo da função, mas também devem ser capazes de detectar quando ele muda e encontrar sua nova posição. Weicker [1] sugere que algoritmos dinâmicos devem ser precisos, estáveis e possuir baixo tempo de reação. Por preciso, entende-se a capacidade que o algoritmo tem de encontrar soluções iguais ou próximas do ótimo global; um algoritmo é estável quando mudanças no ambiente não impactam diretamente em sua precisão; e ele possui baixo tempo de reação quando consegue responder rapidamente às mudanças no ambiente, novamente, sem impactar diretamente em sua precisão.

Um outro aspecto importante é a avaliação desses algoritmos. Devido ao ótimo do problema estar constantemente se alterando, não é suficiente que se analise a melhor solução encontrada ao final da execução do algoritmo. Morrison [2] sugere que se analise o desempenho do *fitness* médio [3] encontrado pelo algoritmo durante uma longa exposição do mesmo à dinâmica do problema. Dessa forma, é possível comparar dois algoritmos de forma mais justa.

Os métodos clássicos de otimização de sistemas, apesar de precisos, tendem a aumentar sua complexidade à medida que a complexidade do problema aumenta. Com isso, o custo e o tempo de processamento desses métodos impossibilitam sua aplicação em muitos problemas práticos. Felizmente, para grande parte dos problemas reais, soluções próximas da solução ótima são suficientes, desde que estejam dentro de uma margem aceitável de erro, que varia de acordo com o tipo de problema que se deseja otimizar.

Baseados nisso, algoritmos bio-inspirados têm apresentado bom desempenho para otimização em geral [4]. Mas apesar dos sistemas bio-inspirados serem geralmente adaptativos, a grande maioria dos algoritmos propostos neste sentido são aplicados para resolver problemas estáticos. No entanto, este cenário vem

mudando nos últimos tempos com diversos trabalhos propostos com o intuito de tornar os algoritmos existentes capazes de se adaptar a mudanças no ambiente [5][6][7].

Dentre os algoritmos bio-inspirados, os algoritmos de inteligência de enxames têm se mostrado bastantes promissores na otimização de sistemas dinâmicos [5][7][8]. Esses algoritmos exploram a inteligência coletiva que emerge a partir das interações de cada indivíduo com os demais membros de seu grupo e com o ambiente em que está inserido. Através dela, mesmo os grupos de seres mais simples, com menor capacidade intelectual, são capazes de grandes feitos; como as formigas e as abelhas que conseguem encontrar o menor caminho entre sua posição e uma fonte de alimento.

Tendo em vista o crescente interesse no desenvolvimento de algoritmos baseados em inteligência de enxame capazes de otimizar sistemas dinâmicos, este trabalho propõe uma ferramenta para o desenvolvimento e teste desses algoritmos. Tal ferramenta pretende acelerar a implementação dos algoritmos, ao fornecer um ambiente com uma arquitetura projetada para problemas dinâmicos; facilitar a simulação dos mesmos, principalmente quando se deseja simular várias vezes um algoritmo e armazenar todos os resultados para uma posterior avaliação; e permitir a análise dos resultados gerados, ao realizar um pré-processamento dos mesmos e construir gráficos a partir deles.

Para esse fim, foi desenvolvido um *framework*, que proporciona a infraestrutura necessária para implementação e simulação dos algoritmos, e um ambiente gráfico que suporta a construção de cenários de teste e avaliação de resultados.

Além da implementação e simulação dos algoritmos, o *framework* desenvolvido também fornece suporte a implementação de problemas de teste para ambientes dinâmicos e de métricas para avaliação dos algoritmos.

O ambiente gráfico permite também:

- A construção de cenários de teste, onde é possível escolher o algoritmo a ser testado, o problema de teste e as métricas utilizadas para sua avaliação;

- A visualização em tempo real da simulação para problemas de até três dimensões, onde é possível acompanhar a posição das partículas no espaço de busca;
- A avaliação dos resultados através de gráficos.

1.1 Estrutura do Trabalho

Este trabalho está organizado conforme descrito a seguir.

O capítulo 2 apresenta a fundamentação teórica necessária para o entendimento deste trabalho. Nele, são apresentados os conceitos fundamentais dos algoritmos baseados em inteligência de exames e dos problemas de otimização dinâmicos.

O capítulo 3 apresenta a ferramenta proposta neste trabalho. Para isso, é explicada a arquitetura do sistema e o funcionamento do ambiente gráfico utilizado para simulação e análise dos resultados.

O capítulo 4 apresenta dois estudos de caso com o intuito de explorar as funcionalidades da ferramenta proposta.

O capítulo 5 apresenta as considerações finais do trabalho e os trabalhos futuros sugeridos.

Capítulo 2

Inteligência de Enxames e Problemas de Otimização Dinâmicos

Neste capítulo são apresentados os conceitos fundamentais de inteligência de enxames e os algoritmos de inteligência de enxames selecionados para o estudo de caso abordado neste trabalho. Em seguida, também são apresentados os conceitos fundamentais de problemas de otimização dinâmicos e os problemas selecionados para o estudo de caso.

2.1 Inteligência de Enxames

Algoritmos de inteligência de enxames modelam populações de indivíduos simples, com pouca capacidade de processamento, que interagem entre si e com seu ambiente usando regras também simples. Essas populações de indivíduos formam sistemas descentralizados e auto-organizáveis. O conjunto de interações formados por essas populações, no entanto, fazem emergir padrões de inteligência capazes de resolver problemas complexos, como problemas de busca e otimização.

Os diversos algoritmos de inteligência de enxames são fortemente inspirados na natureza. Por exemplo, existem algoritmos baseados em bandos de pássaros [9], colônias de formigas [10] e cardumes de peixes [11]. Nas seções seguintes, são apresentados os algoritmos de inteligência de enxames que serão testados nos estudos de caso apresentados nesse trabalho.

2.2 Otimização por Enxame de Partículas

Inspirado no comportamento de bandos de pássaros, o PSO (*Particle Swarm Optimization*) é uma técnica de otimização para funções de variáveis contínuas e de alta dimensionalidade. O enxame é formado por uma população de partículas que se movimentam pelo espaço de busca, onde cada partícula representa uma possível solução para o problema que está sendo resolvido. Inicializado com uma população

de soluções aleatórias uniformemente distribuídas pelo espaço de busca, cada partícula da população conhece sua posição atual, a melhor posição já encontrada por ela e a melhor posição encontrada por sua vizinhança [9].

2.2.1 Movimentação das partículas

As partículas que formam o enxame do PSO movimentam-se pelo espaço de busca à procura da melhor solução. Esse movimento é regido pela velocidade da partícula que é formado por três componentes principais: (i) a velocidade atual da partícula, que provoca a inércia do movimento; (ii) a componente cognitiva, que representa a influência da melhor posição já encontrada pela partícula; e (iii) a componente social, que representa a influência da melhor posição encontrada pela vizinhança da partícula [9].

A cada iteração, a velocidade da partícula é, então, atualizada conforme a equação (1).

$$v_{id}(t+1) = wv_{id}(t) + c_1 r_1(t)[y_{id}(t) - x_{id}(t)] + c_2 r_2(t)\left[y_{gd}(t) - x_{id}(t)\right], \qquad (1)$$

onde $v_{id}(t)$ representa a velocidade atual da partícula i na dimensão d; w representa o peso inercial, responsável por controlar o tipo de busca realizada pelo enxame. Um peso inercial alto faz o enxame executar buscas em amplitude e um peso baixo faz o enxame executar busca em profundidade; $c_1 r_1(t)[y_{id}(t) - x_{id}(t)]$ é a componente cognitiva da velocidade e mensura a qualidade da posição atual da partícula em relação à melhor posição já encontrada por ela; $c_2 r_2(t)\left[y_{gd}(t) - x_{id}(t)\right]$ é a componente social da velocidade e mensura a qualidade da posição atual da partícula com base na melhor posição já encontrada por sua vizinhança; c_1 e c_2 são constantes, chamadas de coeficientes de aceleração, utilizadas para ponderar, respectivamente, o grau de contribuição da componente cognitiva e social; e, por fim, r_1 e r_2 são números aleatórios utilizados para introduzir um componente estocástico no algoritmo.

Dessa forma, cada partícula do enxame é representada por três vetores [12]:

- $\overrightarrow{x_i} = (x_{i1}, x_{i2}, \dots, x_{id})$, que representa sua posição no espaço de busca *d*-dimensional;
- $\overrightarrow{y_i} = (y_{i1}, y_{i2}, \dots, y_{id})$, que representa a melhor posição encontrada pela própria partícula;

- $\overrightarrow{v_i} = (v_{i1}, v_{i2}, \ldots, v_{id})$, que representa a velocidade atual da partícula.

A posição de cada partícula, por sua vez, é atualizada a cada iteração do algoritmo conforme a equação (2).

$$\overrightarrow{x_i}(t+1) = \overrightarrow{x_i}(t) + \overrightarrow{v_i}(t+1), \quad (2)$$

onde $\overrightarrow{x_i}(t)$ representa a posição atual da partícula.

2.2.2 Funcionamento do Algoritmo de PSO

Em sua forma original, o PSO guarda a melhor posição encontrada pelas partículas de seu enxame. Essa posição é verificada e atualizada, se for o caso, a cada iteração do algoritmo. Além disso, a cada iteração cada partícula atualiza sua melhor posição e se movimenta em direção à resultante da melhor posição encontrada por ela e da melhor posição encontrada por sua vizinhança, conforme as equações (1) e (2), respectivamente. A Tabela 1 mostra o pseudocódigo do PSO.

Tabela 1. Pseudocódigo do PSO.

```
cria e inicializa um enxame d-dimensional;
repita
   para cada partícula do enxame faça
      para cada dimensão d faça
         se f(x_id) < f(y_id) então
            y_id = x_id;
         fim;
         se f(y_id) < f(y_gd) então
            y_gd = y_id;
         fim;
      fim;
   fim;
   para cada partícula do enxame faça
      atualiza a velocidade atual da partícula conforme equação (1);
      atualiza a posição atual da partícula conforme equação (2);
   fim;
enquanto nenhuma condição de parada for satisfeita;
```

2.2.3 Topologias de Comunicação do Algoritmo de PSO

A interação entre as partículas que formam o enxame do PSO desempenha papel fundamental no algoritmo. Assim como animais da natureza que vivem em comunidades são influenciados, em maior ou menor grau, por seus vizinhos, as partículas do enxame aprendem umas com as outras e tendem a ser mais

influenciadas pelas melhores partículas, ou seja, as que encontraram as melhores soluções [13].

No PSO, cada partícula interage com as partículas que fazem parte de sua vizinhança, que, por sua vez, é determinada pela topologia do algoritmo. Originalmente, foram propostas duas topologias diferentes: a estrela e a anel. Ambas podem ser vistas na Figura 1 (extraída de [13]).

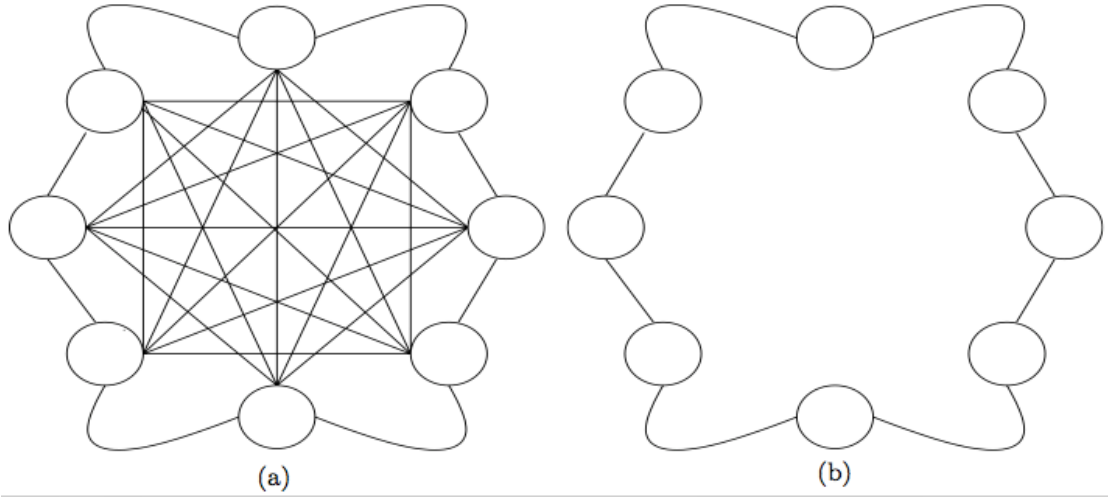

Figura 1. Topologias (a) estrela e (b) anel.

Na topologia estrela, todas as partículas comunicam-se com todas as demais. Dessa forma, as informações sobre as melhores posições encontradas espalham-se muito rapidamente pelo enxame e o algoritmo tende a convergir de forma mais rápida. Por outro lado, devido a rápida convergência, essa topologia torna o algoritmo mais susceptível a ficar preso em ótimos locais.

Já na topologia anel, as partículas comunicam-se com seus 2 vizinhos mais próximos, de acordo com os índices das partículas. A partícula i, por exemplo, irá se comunicar com as partículas $i-1$ e $i+1$. A melhor posição encontrada pelo enxame continua sendo propagada para todas as partículas, uma vez que existem interseções entre as vizinhanças formadas. Porém, a velocidade de propagação das informações é menor que na topologia estrela e, por isso, o algoritmo tende a convergir mais lentamente. Em contra partida, o algoritmo consegue explorar melhor o espaço de busca e, consequentemente, tem menor probabilidade de ficar preso em ótimos locais.

A implementação do PSO que utiliza a topologia estrela, é chamada de *Global Best PSO*. Já a que utiliza a topologia anel, é chamada de *Local Best PSO*.

2.3 *Charged* PSO

O PSO convencional é um algoritmo bastante eficiente para otimização de diversos sistemas estáticos [14]. Porém, ele não possui um bom desempenho em sistemas dinâmicos. Isso porque uma vez que o enxame tenha convergido, ele deixa de realizar buscas em amplitude e, consequentemente, não consegue mais detectar mudanças no ambiente. Para suprir essa deficiência, Blackwell e Bentley [8] propuseram o *Charged PSO*, que utiliza o conceito de cargas eletrostáticas aplicado às partículas.

Neste algoritmo, algumas partículas passam a possuir cargas eletrostáticas de mesma natureza. Dessa forma, partículas carregadas afastam-se umas das outras. O enxame passa, então, a ser formado por partículas carregadas e partículas neutras. As partículas neutras não sofrem influência de nenhuma força de repulsão. Com essa abordagem, as partículas neutras irão tender à melhor posição global do espaço de busca, enquanto as partículas carregadas irão continuar a explorar o ambiente (uma vez que elas sempre são repelidas umas pelas outras ao tentar se aproximar da melhor posição). Como resultado, têm-se um equilíbrio entre a exploração em profundidade e a exploração em amplitude.

A execução do *Charged PSO* continua igual ao pseudocódigo mostrado na Tabela 1. O que precisa ser alterado é a equação (1) que atualiza a velocidade da partícula. Para introduzir a idéia de partículas carregadas eletrostaticamente e evitar a colisão entre as mesmas, deve-se adicionar uma nova aceleração $\boldsymbol{a}_i$ conforme a equação (3).

$$\boldsymbol{a}_i = \sum_{j \neq i} \frac{Q_i Q_j}{{r_{ij}}^3} \boldsymbol{r}_{ij}, \qquad p_{core} < r_{ij} < p, \tag{3}$$

onde $\boldsymbol{r}_{ij} = x_i - x_j$, $r_{ij} = |x_i - x_j|$ e cada partícula possui uma carga igual a Q_i. Partículas neutras possuem carga $Q_i = 0$ e, portanto, não irão contribuir para soma em (3). A equação (1) é então reescrita conforme mostrado na equação (4).

$$v_{id}(t+1) = w v_{id}(t) + c_1 r_1(t)[y_{id}(t) - x_{id}(t)] + c_2 r_2(t)\left[y_{gd}(t) - x_{id}(t)\right] + \boldsymbol{a}_{id}. \tag{4}$$

2.4 *Clan* PSO

O PSO convencional não possui um bom desempenho quando aplicado em problemas de alta dimensionalidade. Então, Carvalho e Bastos-Filho [15] propuseram a topologia *Clan PSO* com o intuito de melhorar o grau de convergência do PSO com foco na distribuição das partículas pelo espaço de busca.

Na topologia proposta, as partículas do enxame são agrupadas em clãs. Dentro de cada clã as partículas são conectadas usando topologia estrela. A Figura 2 (extraída de [15]) mostra um exemplo com quatro clãs. Ao final de cada iteração, cada clã marca, dentre as partículas que o formam, a que possui a melhor posição. Essas partículas são eleitas líderes do clã e irão participar da conferência dos líderes.

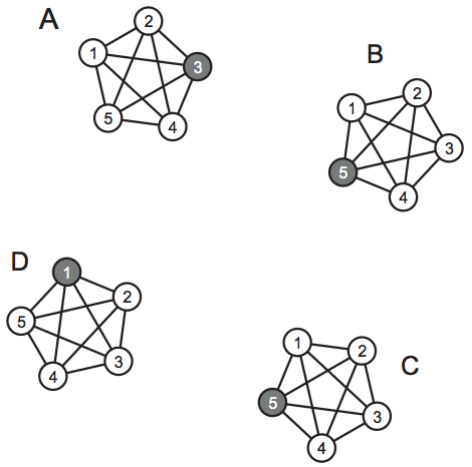

Figura 2. Clãs da topologia e eleição de líderes.

2.4.1 Conferência dos líderes

Uma vez que os líderes foram eleitos, eles formam um novo enxame e executam uma nova iteração do PSO. Isso é feito para garantir que além da melhor partícula do enxame ser eleita, todos os outros líderes terão suas posições ajustadas em direção à melhor partícula.

A topologia a ser utilizada pelo enxame formado durante a conferência dos líderes depende do problema que está sendo resolvido. Pode-se, por exemplo, utilizar a topologia estrela ou a anel. A Figura 3 (extraída de [15]) ilustra uma conferência de líderes que utiliza a topologia estrela.

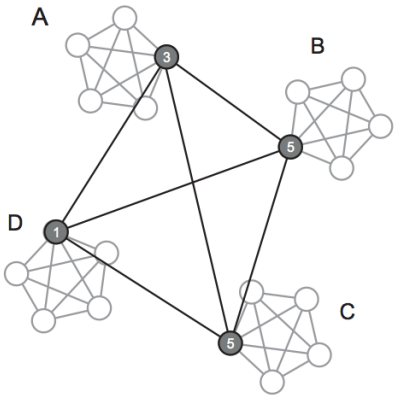

Figura 3. Conferência de líderes utilizando a topologia estrela.

2.4.2 Retro-propagação da informação dentro dos clãs

Após a conferência dos líderes, cada líder retorna para seu clã e dissemina a informação da melhor posição encontrada pelo enxame para as outras partículas. Dessa forma, com o passar da iterações, todas as partículas tenderão a convergir para a melhor posição encontrada.

Através da abordagem proposta para disseminação de informação, cada líder possui influência apenas sobre o seu clã. Consequentemente, a capacidade exploratória de cada clã é preservada.

2.5 Fish School Search

O FSS é um algoritmo de busca e otimização inspirado no comportamento gregário de muitas espécies de peixe. Esse comportamento pode ser explicado pela busca de proteção mútua e sinergia na realização de tarefas coletivas. Por proteção mútua pode-se entender a redução das chances do peixe ser caçado e capturado por predadores; já a sinergia representa uma forma de se alcançar objetivos coletivos, como encontrar comida [11].

Cada peixe representa uma possível solução para o problema que está sendo resolvido. O sucesso de cada peixe durante o processo de busca é representado por seu peso. A função objetivo é mapeada como a densidade de alimento no espaço de busca. Em problemas de minimização, a quantidade de alimento em uma região é inversamente proporcional ao valor da função naquela região. Por fim, o aquário representa o espaço de busca do problema [16].

O algoritmo é formado por quatro operadores que são aplicados em sequência na execução do algoritmo: o operador de movimento individual; o operador de alimentação; o operador de movimento coletivo instintivo; e o operador de movimento coletivo volitivo. Nas subseções seguintes, cada um dos operadores será explicado em mais detalhes.

2.5.1 Operador de movimento individual

Este operador representa o movimento individual de cada peixe. Em sua aplicação, cada peixe calcula, de forma aleatória, uma nova posição vizinha no aquário. Em seguida, essa nova posição é avaliada usando a função-objetivo. Caso a nova posição seja melhor que a sua posição atual, o peixe irá se mover para ela. Caso contrário, ele fica parado. A nova posição é calculada usando a equação (5).

$$n_d(t) = x_d(t) + rand(-1,1)step_{ind}, \tag{5}$$

onde x_i é a posição atual do peixe na dimensão d, n_d é a nova posição do peixe na dimensão d e $rand$ é uma função que gera um número uniformemente distribuído dentro do intervalo [-1, 1]. $step_{ind}$ é uma porcentagem do tamanho do aquário. É interessante que $step_{ind}$ decresça linearmente ao longo das iterações a fim de se garantir que o algoritmo comece realizando busca em amplitude e termine realizando busca em profundidade. O decaimento de $step_{ind}$ pode seguir a equação (6).

$$step_{ind}(t+1) = step_{ind}(t) - \frac{step_{ind\,initial} - step_{ind\,final}}{iterations}, \tag{6}$$

onde $iterations$ é o número máximo de iterações de uma simulação, $step_{ind\ inicial}$ é o tamanho inicial do passo do peixe e $step_{ind\ final}$ é o tamanho final do passo do peixe.

2.5.2 Operador de alimentação

Este operador determina a variação do peso de cada peixe a cada iteração. Através dele, um peixe pode ganhar ou perder peso, dependendo, respectivamente, de seu sucesso ou falha na busca por comida. O peso de cada peixe é calculado conforme a equação (7).

$$W_i(t+1) = w_i(t) + \frac{\Delta f}{\max(|\Delta f|)}, \tag{7}$$

onde $W_i(t)$ é o peso do peixe i, Δf é a variação da função de *fitness* entre a nova posição do peixe e sua posição atual, e $\max(|\Delta f|)$ é o valor absoluto da maior variação Δf encontrada entre todos os peixes do cardume. A fim de se evitar que o peso dos peixes ultrapasse um limite máximo, o parâmetro W_{scale} é utilizado. Dessa forma, o peso de cada peixe varia entre 1 e W_{scale}. Além disso, todos os peixes nascem com peso igual a $\frac{W_{scale}}{2}$.

2.5.3 Operador de movimento coletivo instintivo

Após o movimento individual, espera-se que os peixes que obtiveram maiores sucessos em seus movimentos possuam maior influencia na direção do cardume que os demais. Então, A direção resultante do cardume é calculada usando a equação (8).

$$\vec{I}(t) = \frac{\sum_{i=1}^{N} \Delta\vec{x_i} \Delta f_i}{\sum_{i=1}^{N} \Delta f_i}. \quad (8)$$

Em seguida, a posição de cada peixe é atualizada conforme a equação (9).

$$\vec{x_i}(t+1) = \vec{x_i}(t) + \vec{I}(t). \quad (9)$$

2.5.4 Operador de movimento coletivo volitivo

Este último operador controla a granularidade da busca realizada pelo cardume. Caso o cardume esteja ganhando peso, o raio do cardume é diminuído para que os peixes possam realizar mais busca em profundidade. Por outro lado, se o peso do cardume estiver diminuindo, o raio do cardume é aumentado para que os peixes possam realizar busca em amplitude.

A variação do raio do cardume é aplicada através de uma pequena variação da posição de cada peixe em relação ao baricentro do cardume. O baricentro do cardume é calculado usando a equação (10).

$$\vec{B}(t) = \frac{\sum_{i=1}^{N} \vec{x_i}(t) W_i(t)}{\sum_{i=1}^{N} \vec{x_i}(t)}. \quad (10)$$

Após calcular o baricentro, a posição de cada peixe é atualizada através da equação (11) se o peso do cardume aumentou na iteração atual, caso contrário a posição de cada peixe é atualizada conforme a equação (12).

$$\overrightarrow{x_i}(t+1) = \overrightarrow{x_i}(t) - step_{vol} rand(0,1) \frac{(\vec{x}(t) - \vec{B}(t))}{distance(\vec{x}(t), \vec{B}(t))}, \quad (11)$$

$$\overrightarrow{x_i}(t+1) = \overrightarrow{x_i}(t) + step_{vol} rand(0,1) \frac{(\vec{x}(t) - \vec{B}(t))}{distance(\vec{x}(t), \vec{B}(t))}, \quad (12)$$

onde $distance$ é uma função que calcula a distância euclidiana entre o baricentro e a posição atual do peixe e $step_{vol}$ é o tamanho do passo usado para controlar o deslocamento do peixe.

Para mostrar como todos os operadores mostrados anteriormente são aplicados na prática, a Tabela 2 mostra o pseudocódigo do FSS.

Tabela 2. Pseudocódigo do FSS.

```
inicia aleatoriamente a posição de todos os peixes;
para cada peixe faça
   avalia o fitness de sua posição;
fim;
enquanto nenhuma condição de parada for satisfeita faça
   para cada peixe faça
      executa o movimento individual usando (5);
   fim;
   para cada peixe faça
      alimenta o peixe usando (7);
   fim;
   calcula a resultante do movimento instintivo usando (8);
   para cada peixe faça
      executa o movimento instintivo usando (9);
   fim;
   calcula o baricentro usando (10);
   para cada peixe faça
      execute o movimento volitivo usando (11) ou (12);
   fim;
   atualiza o tamanho dos passos (individual e volitivo) usando (6);
fim;
```

2.6 Problemas de Otimização

Algoritmos de otimização são métodos de busca cujo objetivo é encontrar a solução para um problema cujo o conjunto de variáveis, possivelmente limitadas por um conjunto de restrições, são ótimas.

Todo problema de otimização é formado por [13]:

- Uma **função objetivo**, que representa a quantidade a ser minimizada ou maximizada. Alguns problemas não possuem uma função objetivo definida explicitamente. Nesses casos, o objetivo é encontrar uma solução que satisfaça o conjunto de restrições do mesmo.
- Um **conjunto de variáveis**, que representam uma solução para o problema e cujos valores afetam o valor da função objetivo.
- Um **conjunto de restrições**, que limita os valores que podem ser assumidos pelas variáveis do problema.

2.7 Problemas de Otimização Dinâmicos

Problemas de otimização dinâmicos são problemas cuja função objetivo, instância do problema ou conjunto de restrições mudam com o tempo, fazendo com o que seu ótimo também mude.

O ambiente de um problema dinâmico pode sofrer alterações de diversas formas [5]. De modo geral, pode-se classificar as mudanças de um ambiente dinâmico em dois grupos: (i) mudanças dimensionais e (ii) mudanças não dimensionais [17]. No primeiro caso, o número de dimensões do ambiente muda com o passar do tempo. Já no segundo caso, o número de dimensões permanece constante e os valores das variáveis do problema mudam com o passar do tempo, respeitando-se sempre seu conjunto de restrições.

Ambientes dinâmicos com mudanças não dimensionais podem, ainda, ser divididos em três tipos [18]:

- **Tipo I:** A posição do ótimo global do problema muda com o passar do tempo;
- **Tipo II:** A posição do ótimo global se mantém fixa, mas seu valor muda com o tempo;
- **Tipo III:** Tanto a posição quanto o valor do ótimo global mudam com o tempo.

Para ambientes com mais de uma dimensão, essas mudanças podem ocorrer em uma ou mais dimensões, de forma independente ou simultânea.

2.8 Objetivos de Otimização em Problemas Dinâmicos

Algoritmos de otimização para problemas dinâmicos devem não apenas localizar o ótimo da função, mas também devem ser capazes de detectar quando ele muda e encontrar sua nova posição [6]. Dessa forma, enquanto algoritmos de otimização de problemas estáticos buscam apenas encontrar uma solução satisfatória, os algoritmos de otimização para problemas dinâmicos preocupam-se em ser capazes de seguir o ótimo do problema [7]. Tendo isso em mente, Weicker [1] sugere três objetivos principais para os algoritmos de otimização para problemas dinâmicos: precisão, estabilidade e reação. Cada um desses objetivos são explicados nas subseções seguintes.

2.8.1 Precisão

Mede o quão próxima a solução encontrada pelo algoritmo está da solução global. Dado o algoritmo de otimização AO e uma função de *fitness* F, a precisão deste algoritmo para um dado instante t é definida como:

$$precisao_{F,AO}^{(t)} = \frac{F(melhor_{AO}^{(t)}) - Min_F^{(t)}}{Max_F^{(t)} - Min_F^{(t)}}, \tag{13}$$

onde $melhor_{AO}^{(t)}$ é a melhor solução encontrada no instante de tempo t, $Max_F^{(t)} \in \mathbb{R}$ é o melhor valor de *fitness* do espaço de busca e $Min_F^{(t)} \in \mathbb{R}$ é o pior valor de *fitness* do espaço de busca. A precisão varia entre 0 e 1, onde, para problemas de maximização, quanto mais próximo de 1 mais preciso é o algoritmo.

2.8.2 Estabilidade

Um algoritmo de otimização para problemas dinâmicos é considerado estável se mudanças no ambiente não impactam seriamente em sua precisão. A estabilidade em um dado instante t é definida como:

$$estabilidade_{F,AO}^{(t)} = \max\left\{0, precisao_{F,AO}^{(t-1)} - precisao_{F,AO}^{(t)}\right\}. \tag{14}$$

Seus valores também variam entre 0 e 1. Quanto mais próximo de 0, mais estável é o algoritmo.

2.8.3 Reação

Mede a capacidade de reação do algoritmo às mudanças que ocorrem no ambiente. A capacidade de reação de um algoritmo pode ser medida como:

$$reacao_{F,A,\varepsilon}^{(t)} = \min\{t' - t \mid t < t' \leq maiorgen, t' \in \mathbb{N}, \frac{precisao_{F,A}^{(t')}}{precisao_{F,A}^{(t)}} \geq (1 - \varepsilon)\}, \quad (15)$$

onde $maiorgen$ é o número total de iterações do algoritmo e ε é um valor definido pelo usuário que controla a precisão mínima que o algoritmo deve atingir após uma mudança do ambiente para que se possa considerar que ele reagiu à mesma. Quanto menor o valor, melhor é o poder de reação do algoritmo.

Pode-se observar que a estabilidade e a reação são definidas com base na precisão. O problema apontado por Weicker [1] na definição de precisão dada acima é que ela parte do pressuposto que o ótimo global do problema é conhecido. Especialmente para problemas dinâmicos, não se pode garantir que o ótimo sempre será conhecido, uma vez que ele está sempre mudando.

2.9 Funções de Teste para Problemas Dinâmicos

Funções de teste devem ser simples, fáceis de entender e analisar e configuráveis através de parâmetros. Devem se aproximar de problemas reais, mas ao mesmo tempo ser simples o suficiente para permitir uma análise do algoritmo de otimização que está sendo estudado.

No caso dos problemas dinâmicos, as funções de teste estão sempre se modificando. Essas mudanças não podem, no entanto, sempre gerar ambientes totalmente novos, sem conexão com o anterior. Isso porque, nesses casos, a melhor abordagem sempre seria a reinicialização do algoritmo – já que o que foi aprendido não é mais válido após a mudança do ambiente [19].

Ao se analisar funções de teste para ambientes dinâmicos, deve-se observar dois aspectos principais: o modo como o espaço de busca é atualizado e a dinâmica de atualização empregada.

Com relação ao modo de atualização do espaço de busca, em sua forma mais simples, tem-se uma função com apenas um pico que fica se movimentando

dentro do espaço de busca. Nesse caso, pode-se configurar, por exemplo, o quanto o pico irá se mover a cada mudança do ambiente. Funções de teste mais elaboradas, no entanto, são multimodais, ou seja, têm mais de um pico e a cada atualização alteram a morfologia de seus picos, modificando além de sua posição, sua altura e sua largura.

A dinâmica de atualização irá determinar como o ambiente da função será atualizado. Em geral, as mudanças são contínuas e graduais ou oscilam dentro de um conjunto de valores. Adicionalmente, pode-se inesperadamente gerar uma mudança significativa no ambiente, a fim de se observar como o algoritmo reage a situações extremas.

A seguir, são apresentadas as duas funções de teste para ambientes dinâmicos que serão utilizadas no estudo de caso deste trabalho.

2.10 DF1

Proposto por Morrison *et al*. [19], o DF1 é um gerador de problemas dinâmicos. O ambiente de seus problemas é formado por um conjunto aleatório de cones de alturas e inclinações diferentes. O número de cones, suas alturas e posições são todos configuráveis durante a inicialização do problema. Sua função também pode ser especificada para um número variável de dimensões. A equação (16) mostra a sua função para duas dimensões.

$$f(X,Y) = \max_{i=1,N}[H_i - R_i * \sqrt{(X - X_i)^2 + (Y - Y_i)^2}], \quad (16)$$

onde *N* representa o número de cones do ambiente e cada cone é representado por sua localização (X_i, Y_i), sua altura H_i e sua inclinação R_i. Sempre que o DF1 é executado, ele gera um ambiente de teste com base na função acima e nas seguintes restrições:

$$H_i \in [H_{base}, H_{base} + H_{limite}]. \quad (17)$$

$$R_i \in [R_{base}, R_{base} + R_{limite}]. \quad (18)$$

$$X_i \in [-1,1]. \quad (19)$$

$$Y_i \in [-1,1]. \quad (20)$$

2.10.1 Dinâmica do problema

Os componentes dinâmicos do ambiente são atualizados através de passos discretos. O DF1 fornece um método simples para controlar a geração de passos de diferentes tamanhos.

Para isso, é utilizada a função logística conforme a equação (21).

$$X_i = rX_{(I-1)}\left(1 - X_{(I-1)}\right), \tag{21}$$

onde r é uma constante que varia entre 1 e 4 e X_i é o valor produzido na iteração i.

Um mapa da bifurcação da função logística é fornecido na Figura 4. Esse mapa mostra os valores de X que podem ser gerados para r variando entre 1 e 4. Para valores menores que 3, serão produzidos valores constantes a cada iteração. Já para valores maiores que 3, comportamentos dinâmicos mais complicados começam a ser gerados. Por exemplo, para r igual a 3,5, são gerados os valores {0,3828; 0,5009; 0,8269; 0,8750}. Valores ainda maiores de r geram sequências caóticas de valores para X, conforme pode ser visto na Figura 4.

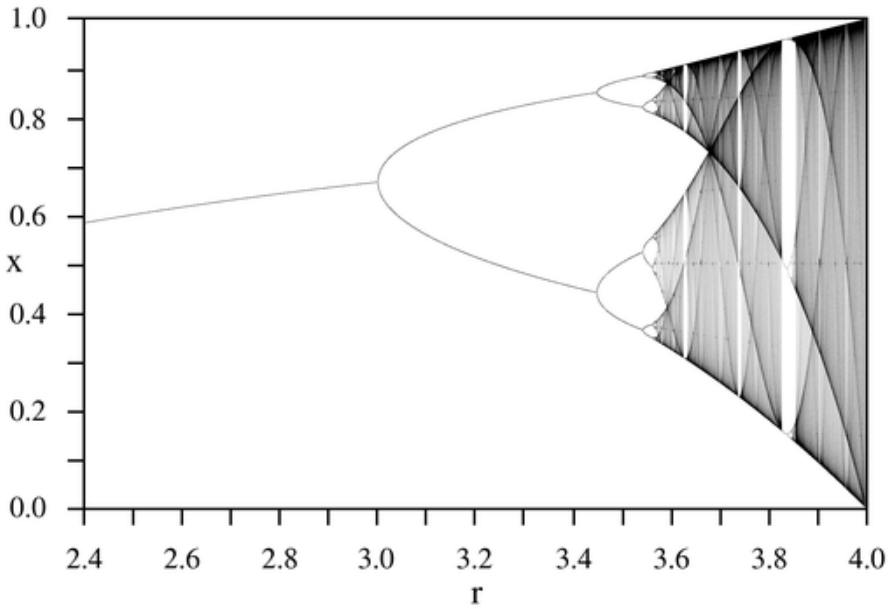

Figura 4. Mapa de bifurcação da função logística.

A cada iteração, o DF1 irá selecionar o tamanho do passo de cada componente dinâmico dentre os valores de X produzidos. Dessa forma, cada componente dinâmico é associado a uma função logística com um valor específico de r.

O valor de r irá determinar se um componente dinâmico do problema será atualizado através de passos: pequenos e constantes; largos e constantes; de tamanhos diferentes; ou de tamanhos caoticamente diferentes.

O valor de X é utilizado então para atualizar o componente dinâmico do problema com o qual está associado. Considerando-se, por exemplo, que o componente dinâmico a ser atualizado seja a altura H de um cone, o primeiro passo é calcular a porcentagem da altura atual do cone em relação à sua altura máxima (H_{pct}), conforme a equação (22).

$$H_{pct} = \frac{H}{H_{base}+H_{limite}}. \quad (22)$$

Em seguida, conforme a equação (23), H_{pct} é somado com o valor de X ajustado por um fator de escala definido pelo usuário.

$$H_{pct} = H_{pct} + Y * H_{escala}. \quad (23)$$

Se o valor final de H_{pct} for menor que 100% do valor máximo permitido para a altura de um cone, então a nova altura é calculada multiplicando-se H_{pct} pela altura máxima de um cone. Por outro lado, se o valor final de H_{pct} for maior que 100%, então o sinal do passo X é invertido e permanece assim a cada iteração até que o valor mínimo para a altura de um cone seja atingido. Nesse momento, o valor do passo é invertido novamente.

Assim, a dinâmica do DF1 é controlada através dos seguintes parâmetros:

- N_{pico}: o número de picos que irão se mover;
- r_h: o valor de r para a dinâmica da altura dos picos;
- r_e: o valor de r para a dinâmica das inclinações dos picos;
- r_x: o valor de r para a dinâmica do movimento dos picos no eixo x;
- r_y: o valor de r para a dinâmica do movimento dos picos no eixo y.

2.11 Moving Peaks

Essa função de teste é formada por uma superfície multidimensional composta por diversos picos, onde a altura, largura e posição de cada pico é alterada a cada atualização do ambiente [20]. Ela pode ser observada na equação (24).

$$F(\vec{x},t) = \max_{i=1\ldots N} \frac{H_i(t)}{1+W_i(t)\sum_{j=1}^{D}(x_j - X_j(t))^2}, \quad (24)$$

onde os vetores $\vec{H}, \vec{W}$ e $\vec{X}$ são definidos inicialmente pelo usuário, N representa o número de picos e D o número de dimensões do problema.

A cada atualização do ambiente, a altura e largura de cada pico é alterada pela soma de uma variável Gaussiana aleatória conforme as equações (23) e (24), onde H_{sev} e W_{sev} são parâmetros que configuram a severidade de cada mudança. A localização de cada pico também é alterada em uma direção aleatória por um vetor $\vec{v}$ de tamanho fixo s, conforme equação (25). Sendo assim, o parâmetro s controla a severidade de cada mudança de posição. O ambiente é atualizado a cada Δe iterações.

$$\sigma \in \mathrm{N}(0,1), \quad (22)$$

$$H_i(t) = H_i(t-1) + H_{sev}\sigma, \quad (23)$$

$$W_i(t) = W_i(t-1) + W_{sev}\sigma, \quad (24)$$

$$\vec{X}(t) = \vec{X}(t-1) + \vec{v}. \quad (25)$$

2.12 Métricas

Conforme exposto na seção 2.7, para comparar diferentes técnicas de otimização para problemas dinâmicos não é suficiente que se compare a melhor solução encontrada por cada técnica, uma vez que o ótimo está sempre mudando com o tempo [3]. Uma boa métrica de avaliação de algoritmos de otimização para problemas dinâmicos deve garantir uma exposição suficientemente abrangente à dinâmica do problema, a fim de reduzir possíveis interpretações errôneas dos resultados causadas pela avaliação de apenas partes da dinâmica do problema [2].

2.12.1 Fitness Médio

Apenas reportar o melhor *fitness* encontrado ao final de uma simulação não é suficiente em ambientes dinâmicos. Isso porque o ambiente está constantemente mudando e, consequentemente, o *fitness* da melhor posição também. O *Fitness* Médio calcula então a média dos melhores valores de *fitness* encontrados ao final de cada iteração, de acordo com a equação (27).

$$F_{medio}(T) = \frac{\sum_{t=1}^{T} F_{melhor}}{T}, \tag{27}$$

onde T representa o número total de iterações e F_{melhor} representa o *fitness* da melhor partícula após a iteração t.

2.12.2 *Fitness* Coletivo

Introduzida por Morrison [2], esta métrica foi criada especialmente para ambientes dinâmicos. Nela, o algoritmo é executado diversas vezes, onde cada execução possui um número suficientemente grande de iterações. Dessa forma, é possível expor o algoritmo à toda dinâmica do problema. O *Fitness* Coletivo é então calculado como a média do *Fitness* Médio obtido em todas as simulações, conforme a equação (28).

$$F_C = \frac{\sum_{m=1}^{M} F_{medio}(T)}{M}, \tag{28}$$

onde T representa o número total de iterações em cada simulação e M representa o número de simulações executadas.

O número de iterações necessário para utilizar a métrica corretamente irá depender do ambiente sendo trabalhado. Se o ambiente é conhecido, pode-se tentar estimar este número. Já quando não se conhece o ambiente, é preciso rodar a métrica diversas vezes até encontrar o número médio de iterações a partir do qual F_C se estabiliza [2].

Capítulo 3

Descrição da Ferramenta

Este capítulo tem como objetivo a apresentação da ferramenta desenvolvida neste trabalho. Com o intuito de facilitar a implementação e os testes de algoritmos de otimização dinâmica baseados em inteligência de enxames, a ferramenta foi dividida em dois projetos: um *framework* chamado DOSS (*Dynamic Optimization System Simulator*), que traz a infraestrutura necessária para implementação e simulação dos algoritmos; e uma interface gráfica chamada DOSA (*Dynamic Optimization System Analyzer*) que permite a criação de cenários de teste e a avaliação dos resultados das simulações.

Para o desenvolvimento da ferramenta, utilizou-se a linguagem de programação *Java* [21]. A escolha da linguagem se deu por ela ser largamente difundida no meio acadêmico e por ser independente de plataforma, permitindo que a ferramenta seja utilizada nos principais sistemas operacionais disponíveis no mercado. Utilizou-se ainda as ferramentas *Eclipse* [22] para a escrita do código; *Maven* [23] para o gerenciamento de dependências e compilação dos projetos; e *NetBeans* [24] para o desenvolvimento das interfaces gráficas. Para construção dos gráficos e exibição das simulações em tempo real utilizou-se, respectivamente, as bibliotecas *JFreeChart* [25] e *ChartDirector* [26].

3.1 Visão Geral do DOSS

O DOSS foi criado para servir como base para implementação e simulação de algoritmos de otimização dinâmica baseados em inteligência de enxames. Por isso, seu enfoque é dado no usuário desenvolvedor. A Figura 5 apresenta seu diagrama de casos de uso. Como pode ser visto no diagrama, o usuário é capaz de implementar novos algoritmos, novas funções de teste e novas métricas para avaliação dos algoritmos. Também é possível implementar condições de parada que, por sua vez, são utilizadas para controlar o momento em que as simulações dos algoritmos são finalizadas.

Para realizar uma simulação, é preciso criar um cenário de teste. Um cenário de teste é formado por um algoritmo, uma função de teste, um conjunto de condições de parada e um conjunto de métricas. Além disso, o DOSS permite a visualização em tempo real da simulação e provê um mecanismo para salvar os resultados das simulações em diferentes formatos.

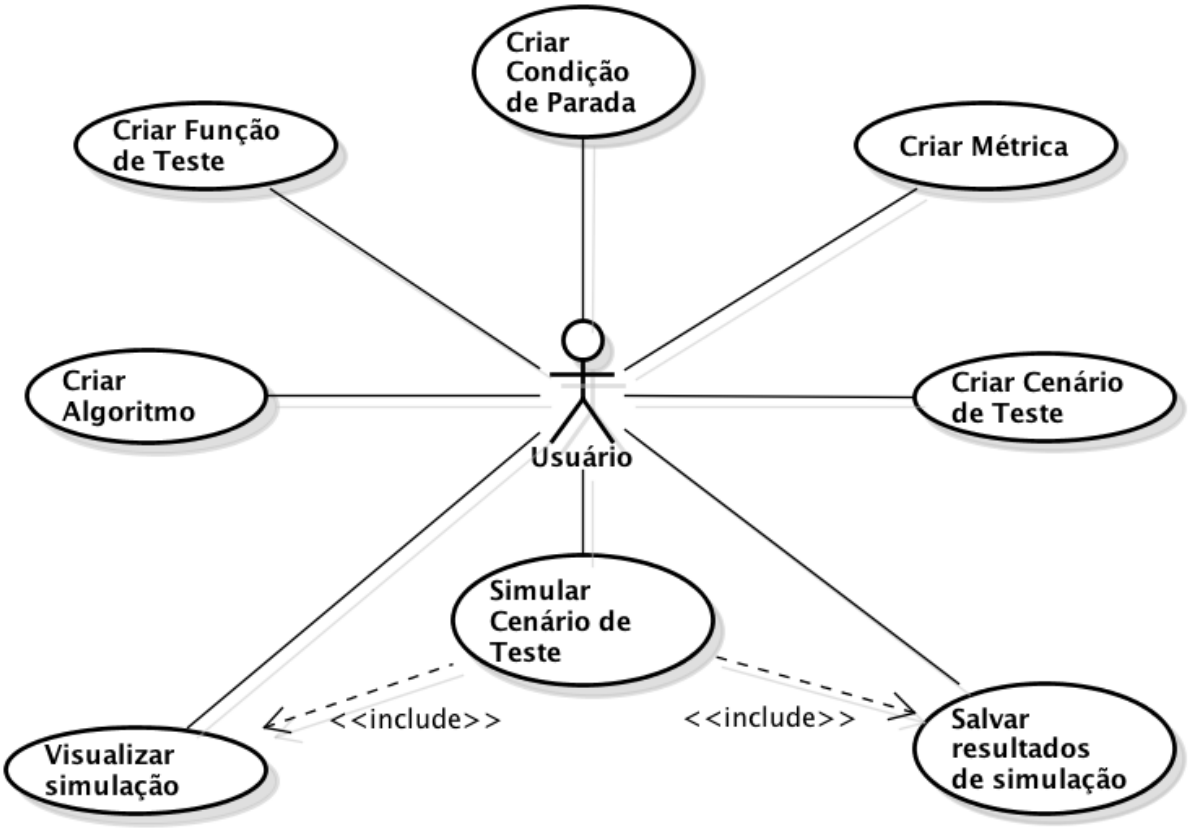

Figura 5. Diagrama de Casos de Uso do DOSS.

3.2 Visão Geral do DOSA

O projeto DOSA foi desenvolvido para facilitar a criação e simulação de cenários de teste ao prover uma interface gráfica para essas tarefas. Uma vez que os algoritmos, as funções de teste, condições de parada e métricas tenham sido implementadas no DOSS, pode-se utilizar o DOSA para, através de sua interface gráfica, criar e gerenciar os cenários de teste e simulá-los. O DOSA permite ainda a avaliação dos resultados das simulações através da criação de gráficos *Box Plot* e de linha, que exibe o conjunto de valores dos resultados conectado por uma única linha. A Figura 6 apresenta o diagrama de casos de uso do projeto.

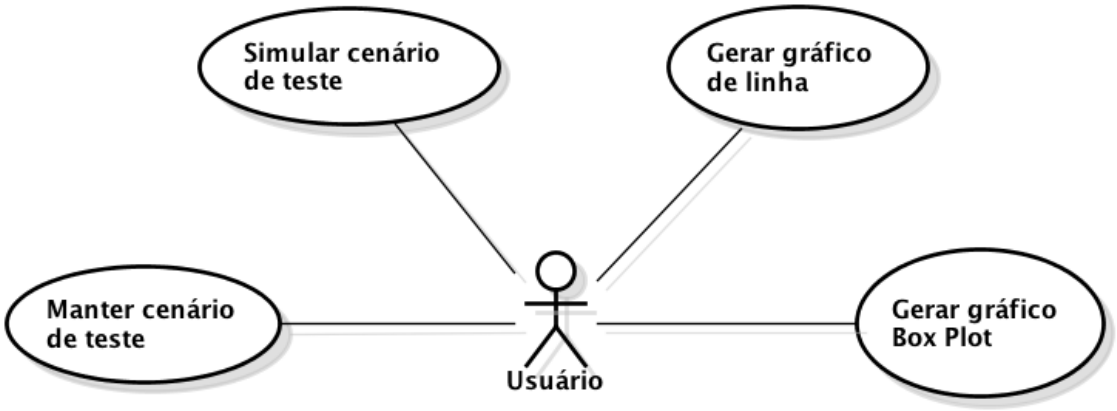

Figura 6. Diagrama de casos de uso do DOSA.

3.3 Módulos da Ferramenta

A ferramenta proposta foi estruturada em três camadas principais, conforme mostrado na Figura 7. A camada de interface gráfica, foi toda desenvolvida no projeto DOSA. Já as de simulação e entidades básicas fazem parte do projeto DOSS. Cada uma delas, por sua vez, é composta por diversos módulos que comunicam-se entre si, conforme pode ser observado na Figura 8. Um módulo do sistema é formado por uma ou mais classes que possuem um objetivo em comum. O módulo de algoritmos, por exemplo, que faz parte da camada de entidades básicas, é formado por todos os algoritmos implementados na ferramenta.

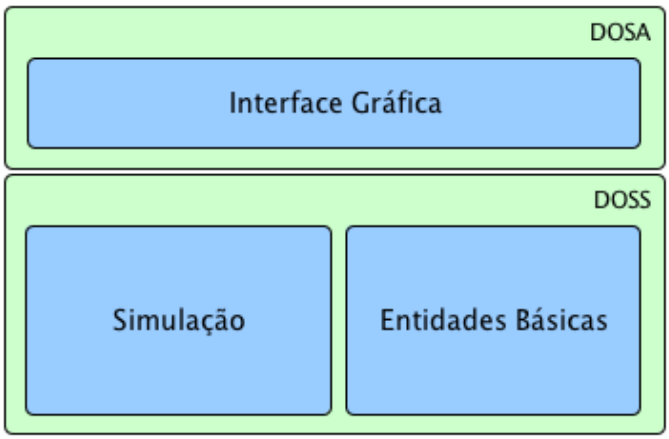

Figura 7. Arquitetura em camadas da ferramenta.

É importante observar que o projeto DOSS é totalmente independente do DOSA. Pode-se construir e executar cenários de teste diretamente através dele e, por ele ter sido implementado como um *framework*, pode-se utilizá-lo de base para

implementação de novos projetos na área de algoritmos de otimização dinâmica baseada em inteligência de enxames.

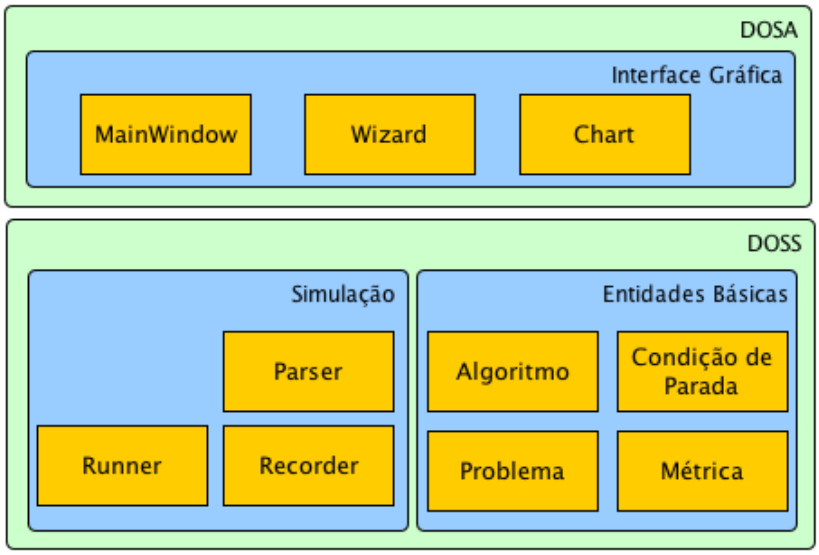

Figura 8. Módulos da ferramenta.

Nas próximas subseções, os módulos da ferramenta são discutidos em mais detalhes.

3.3.1 Módulo Algoritmo

Este módulo contém as implementações dos algoritmos disponibilizados pela ferramenta. Todo algoritmo deve estender a classe abstrata *Algorithm*. Essa classe possui três métodos principais. O método de inicialização, que deve conter as rotinas de inicialização do algoritmo; o método de iteração, responsável por executar todos os passos que formam uma iteração do algoritmo; e o método de execução, responsável por inicializar, executar as iterações e finalizar o algoritmo. A classe *Algorithm* apresenta uma implementação padrão para o método de execução do algoritmo que pode ser vista na Tabela 3. Os métodos de inicialização e iteração devem ser implementados pela classe que estende *Algorithm*. Caso seja necessário uma execução mais elaborada, o método de execução também pode ser sobrescrito.

Tabela 3. Pseudocódigo da execução um algoritmo.

```
inicializa_algoritmo();
repita
   executa_iteracao();
```

```
    atualiza_metricas();
    salva_resultados();
    atualiza_funcao_teste();
enquanto nenhuma condição de parada for satisfeita;
```

3.3.2 Módulo Função de Teste

Contém a implementação das funções de teste fornecidas pela ferramenta. Novas funções de teste também podem ser criadas. Para isso, é preciso apenas estender a classe abstrata *Problem*. Toda função de teste tem o seu número de dimensões e os limites inferior e superior de seu espaço de busca em cada uma de suas dimensão conhecidos. É possível também obter o *fitness* de um ponto específico do espaço de busca e comparar dois valores de *fitness* diferentes para descobrir qual deles é melhor. Para as funções de teste dinâmicas, a classe *Problem* provê o método *update*, que é chamado ao final de cada iteração do algoritmo, como pode ser visto na Tabela 3. Através do método *update*, a função pode ser atualizada.

3.3.3 Módulo Condição de Parada

Agrupa as condições de parada fornecidas pela ferramenta. A condição de parada é a entidade responsável por finalizar a simulação de um algoritmo. Cada cenário de teste tem uma ou mais condições de parada. Atualmente, a ferramenta possui duas condições de parada implementadas: Número máximo de iterações, que irá parar a execução do algoritmo após um número de iterações configurado pelo usuário; e Desvio padrão, que irá parar a execução do algoritmo quando o desvio padrão do *fitness* da melhor posição de cada partícula do algoritmo for menor que um valor especificado pelo usuário.

É possível também implementar novas condições de parada. Para isso, é preciso apenas estender a classe abstrata *StopCondittion*.

3.3.4 Módulo Métrica

Contém todas as métricas implementadas. É através das métricas que os algoritmos são avaliados. Para implementar novas métricas, basta estender a classe abstrata *Measurement*. As métricas que fazem parte de um cenário de teste que está sendo simulado são atualizadas ao final de cada iteração do algoritmo.

3.3.5 Módulo *Runner*

Módulo responsável pela simulação dos algoritmos do sistema. Atualmente, é formado pelas classes *Runner* e *ChartRunner*. A classe *Runner* é a responsável pela execução dos cenários de teste, através dos quais os algoritmos são simulados. Ela permite que sejam configurados o número de vezes que a simulação será executada e se a simulação será exibida em tempo real. É preciso criar uma instância da classe *Runner* para cada cenário de teste que se deseje simular. Para que seja possível visualizar a simulação, é preciso que a função de teste escolhida possua no máximo três dimensões.

A classe *Runner* implementa também o padrão *Observer* [27]. Ele é utilizado para que entidades possam se registrar para serem notificadas sobre o fim da execução das simulações.

Para exibir a simulação é utilizada a classe *ChartRunner*. A cada iteração, essa classe plota a superfície da função de teste e a posição de cada partícula do algoritmo sobre essa superfície.

3.3.6 Módulo *Recorder*

Este módulo é responsável por salvar informações sobre as simulações. Atualmente, o sistema possui as classes *FileRecorder* e *ConsoleRecorder*. A classe *FileRecorder* salva as informações em um arquivo tipo texto. A cada iteração, ela salva as posições das partículas do algoritmo e os valores de todas as métricas que foram registradas no cenário de teste sendo simulado. Já a classe *ConsoleRecorder* realiza as mesmas operações, porém, ao invés de salvar as informações em um arquivo, ela as imprime na saída padrão. Por isso, ela é útil para realização de testes durante a fase de implementação dos algoritmos.

Para criar novos *Recorders* é preciso apenas implementar a interface *IRecorder*.

3.3.7 Módulo *Parser*

Para realizar uma simulação, é preciso criar um cenário de teste. Cada cenário de teste pode ser salvo em um arquivo XML. Assim, é possível recuperar e simular novamente cenários antigos.

Este módulo é formado pela classe *AlgorithmXMLParser*. Ela é responsável por salvar um cenário de teste em um arquivo no formato XML, por ler um arquivo XML e instanciar o cenário de teste descrito por ele.

3.3.8 Módulo *Chart*

Este módulo contém as classes responsáveis por processar as informações geradas pela simulação de um cenário de teste e construir gráficos *Box Plot* a partir delas.

3.3.9 Módulos *MainWindow* e *Wizard*

Contém a implementação da interface gráfica do DOSA. A seção 3.5 irá explicar em detalhes a interface do DOSA.

3.4 DOSS

A seção 3.3 apresentou os principais módulos do DOSS, conforme pode ser visto na Figura 8. A seguir serão apresentados dois pontos fundamentais do projeto: o mecanismo de persistência de cenários de teste; e o mecanismo de simulação de cenários de teste.

3.4.1 Mecanismo de persistência de cenários de teste

Através do DOSS, o usuário é capaz de salvar e recuperar os cenários de teste criados por ele. Para isso, criou-se a classe *AlgorithmXMLParser* que é capaz de salvar os cenários criados em um arquivo XML e de reconstruí-los posteriormente a partir do mesmo arquivo XML.

Para reconstruir um cenário de teste, é preciso criar uma instância de cada entidade que o compõe. Para isso, utiliza-se o mecanismo *Reflection* do *Java* [28], que permite que classes sejam instanciadas em tempo de execução.

Além disso, é preciso configurar cada entidade do cenário de teste com os parâmetros que foram salvos no arquivo XML. Para permitir que os parâmetros de cada entidade fossem descobertos e configurados em tempo de execução, criou-se a anotação *Parameter*. As entidades anotam, então, todos os seus atributos que são configuráveis pelo usuário com esta anotação e, automaticamente, o DOSS será capaz de ler e alterar os valores desses atributos.

3.4.2 Simulação de cenários de teste

A Figura 9 mostra o diagrama de sequência da simulação de um cenário de teste. Para iniciar uma simulação, é preciso instanciar a classe *Runner* informando o cenário de teste que será simulado e o número de vezes que a simulação será executada. Ao iniciar a simulação, a classe *Runner* irá chamar o método *run* do algoritmo. Conforme discutido na subseção 3.3.1, este é o método responsável pela execução do algoritmo. O método *run* irá então inicializar o *Recorder* (ver subseção 3.3.6) e o próprio algoritmo. Em seguida, o método *iterate*, responsável pela execução de uma iteração do algoritmo, é chamado. Logo depois, as métricas registradas no cenário de teste são atualizadas e os valores atuais da simulação são enviados para o *Recorder*. Após isso, a função de teste é atualizada e é verificado se alguma condição de parada foi satisfeita. Por fim, quando alguma condição de parada for satisfeita, o *Recorder* será finalizado e a execução do algoritmo chegará ao fim.

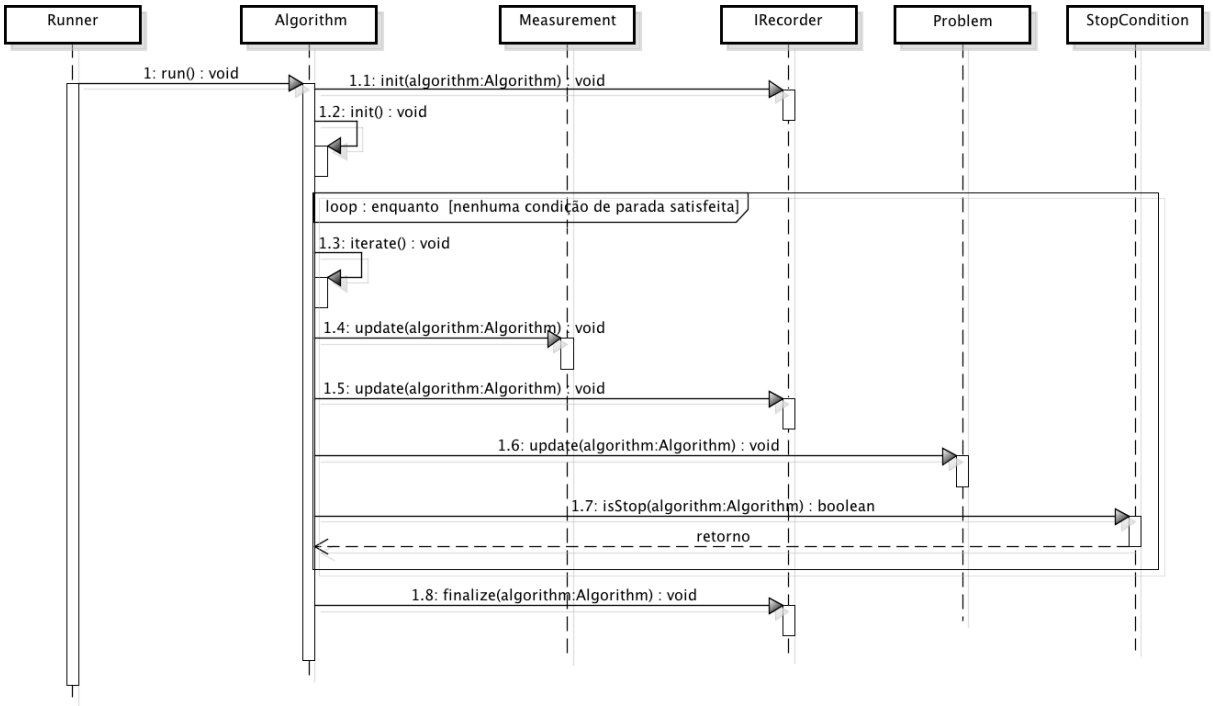

Figura 9. Diagrama de Sequências do DOSS.

3.5 DOSA

A interface gráfica da ferramenta foi desenvolvida no projeto DOSA. A Figura 10 mostra a tela inicial da interface. Nela, o usuário pode selecionar um de seus dois ambientes: o ambiente de simulação, onde pode-se criar cenários de teste e

executá-los; e o ambiente de análise de resultados, onde pode-se analisar os resultados de uma simulação. Nas seções seguintes, os dois ambientes serão explicados em maiores detalhes.

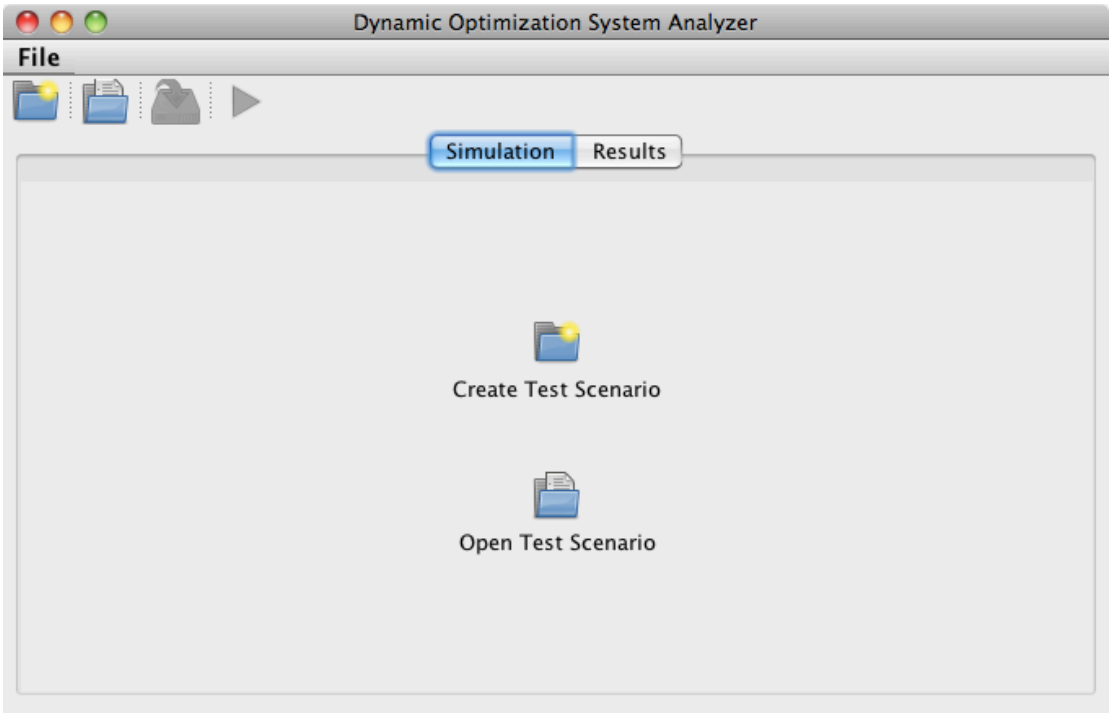

Figura 10. Tela inicial do DOSA.

3.5.1 Criação de cenário de teste

No ambiente de simulação, pode-se criar um novo cenário de teste ou pode-se abrir um cenário já existe. Ao escolher criar um novo cenário de teste, o usuário será apresentado ao Assistente de Criação de Cenários de Teste. Sua tela inicial pode ser vista na Figura 11.

Desenvolvido com o intuito de facilitar o uso da ferramenta, o Assistente irá guiar o usuário na escolha do algoritmo, da função de teste, das condições de parada e das métricas. Ainda na Figura 11, é possível observar que ao selecionar um algoritmo, sua descrição é exibida na parte inferior da tela.

A Figura 12 mostra a tela de seleção de condições de parada. Nela, pode-se escolher dentre as condições de parada existentes, quais serão utilizadas no cenário de teste que está sendo criado. Além disso, é possível escolher a ordem das condições de parada selecionadas através dos botões *Up* e *Down*. As condições de parada selecionadas serão aplicadas durante a execução do algoritmo nessa

mesma ordem. Assim como na tela de seleção de algoritmo, ao selecionar uma condição de parada, sua descrição é exibida na parte inferior da tela.

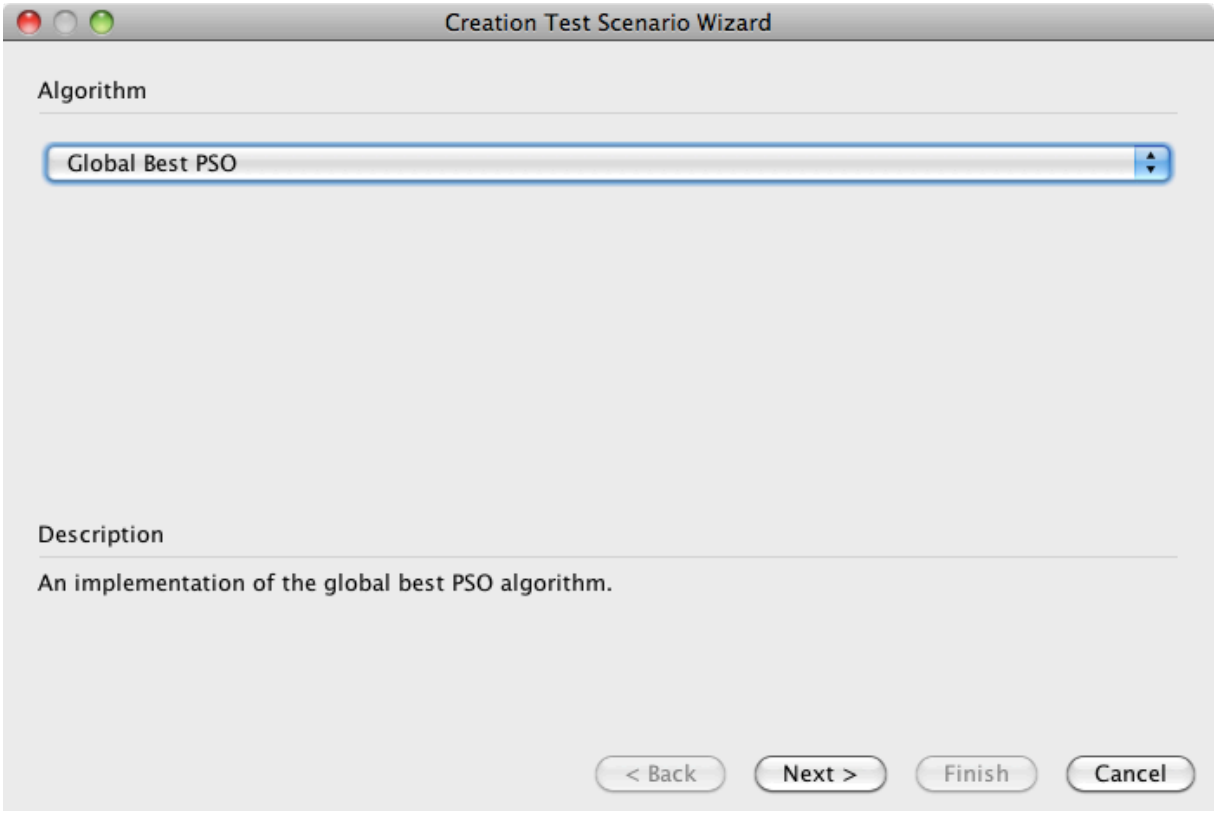

Figura 11. Tela inicial do Assistente de Criação de Cenário de Teste.

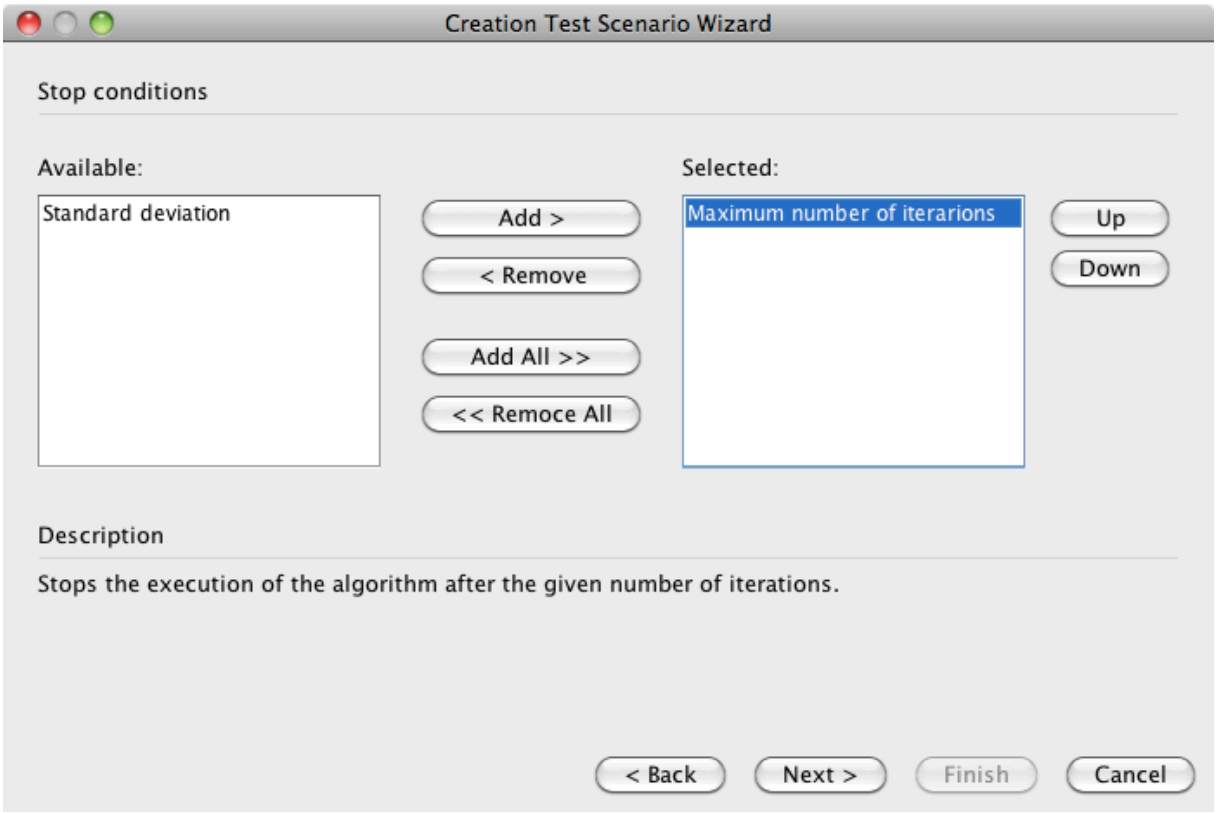

Figura 12. Tela de seleção de condições de parada.

A Figura 13 apresenta a última tela do Assistente. Nela, deve-se escolher o nome do cenário de teste e onde ele será salvo. Ao clicar em *Finish*, o cenário de teste criado pelo usuário será carregado na tela principal do DOSA, conforme pode ser visto na Figura 14. Nesse momento, duas novas opções são habilitadas para o usuário na barra de ferramentas da tela principal. A opção de salvar alterações no cenário de teste e de iniciar sua simulação. A simulação de um cenário de teste será explicada mais a diante.

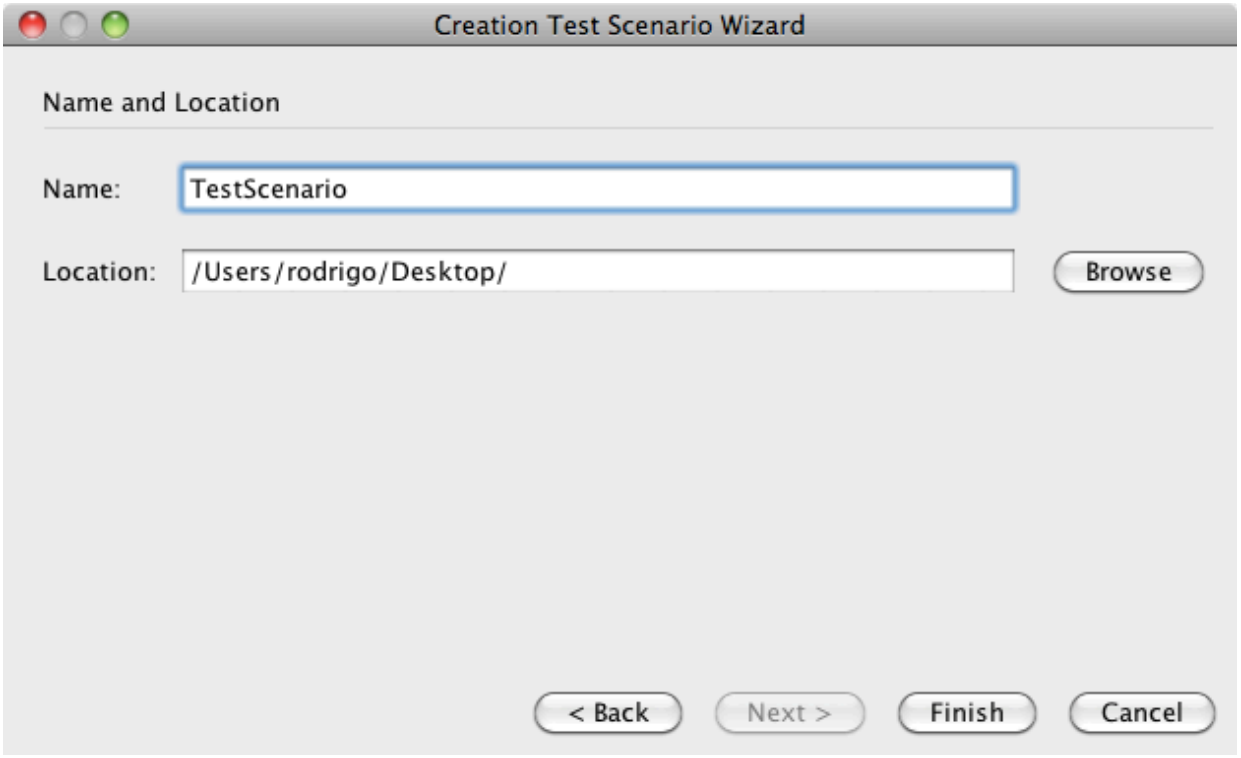

Figura 13. Tela para salvar cenário de teste.

Através do projeto DOSS o usuário pode desenvolver novos algoritmos, novas funções de teste, condições de paradas e métricas; e cada um deles terá parâmetros de configuração específicos. Portanto, um dos principais requisitos no desenvolvimento da interface do DOSA consiste na capacidade de acomodar uma quantidade variável de parâmetros de configuração. Para isso, utilizou-se uma tabela na parte inferior da tela onde os parâmetros são exibidos e podem ser configurados.

Logo acima da tabela de configuração dos parâmetros, o cenário de teste atual é exibido, em uma estrutura de árvore. Ao clicar sobre o algoritmo, por exemplo, seus parâmetros são carregados na tabela de parâmetros e podem ser configurados, conforme ilustrado pela Figura 14.

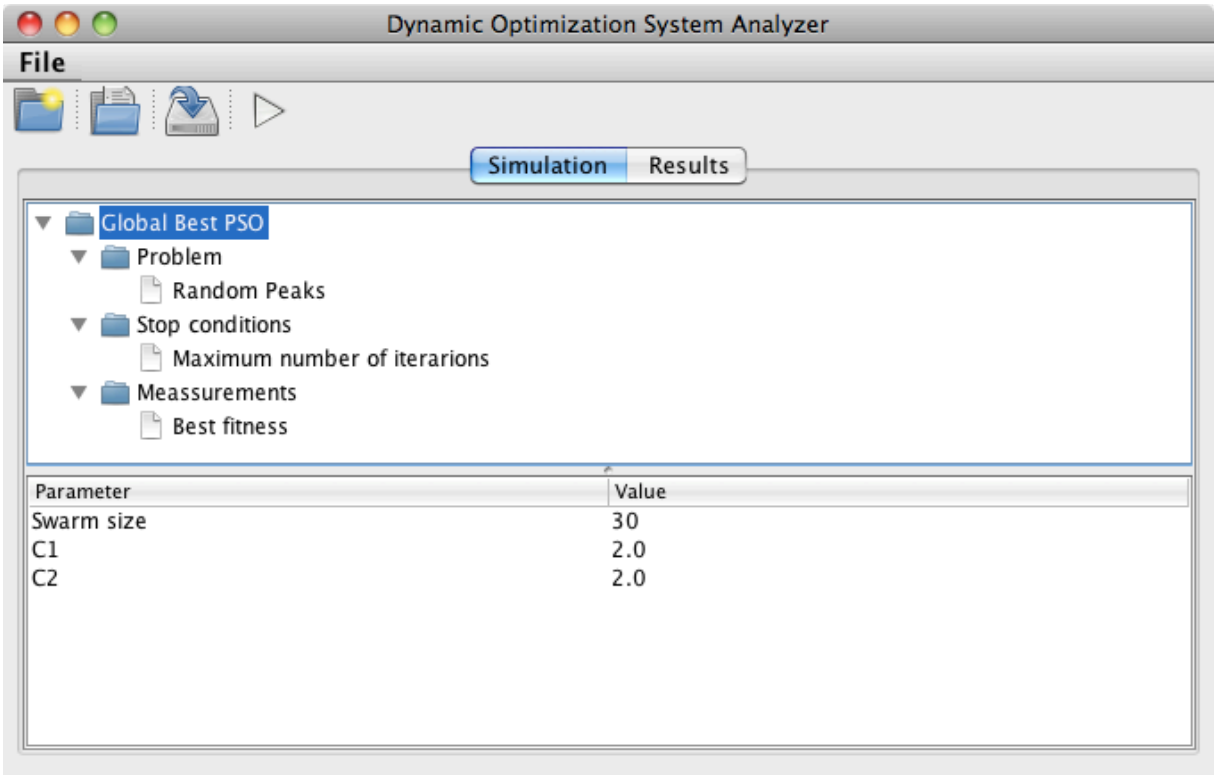

Figura 14. Tela de configuração de um cenário de teste.

1.1.1 Simulação de um cenário de teste

Após criar o cenário de teste e configurá-lo, o usuário pode iniciar sua simulação. Para isso, deve-se clicar no último ícone do lado direito da barra de ferramentas. Ao fazer isso, o usuário será apresentado à tela de configuração de simulação, conforme Figura 15. Nela, é possível escolher o número de simulações que serão executadas. Caso a função de teste que será usada na simulação tenha três dimensões e o usuário escolha executar apenas uma simulação, a opção de exibir a simulação em tempo real estará habilitada para seleção. Se o usuário a selecionar, ao iniciar a simulação, será exibida uma tela similar a Figura 16. Através dela, têm-se uma visão superior da função de teste, onde é possível localizar os pontos de máximo e mínimo da função. É possível também ver a movimentação das partículas em tempo real.

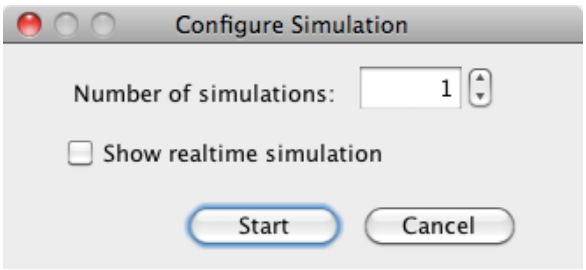

Figura 15. Tela de configuração de simulação.

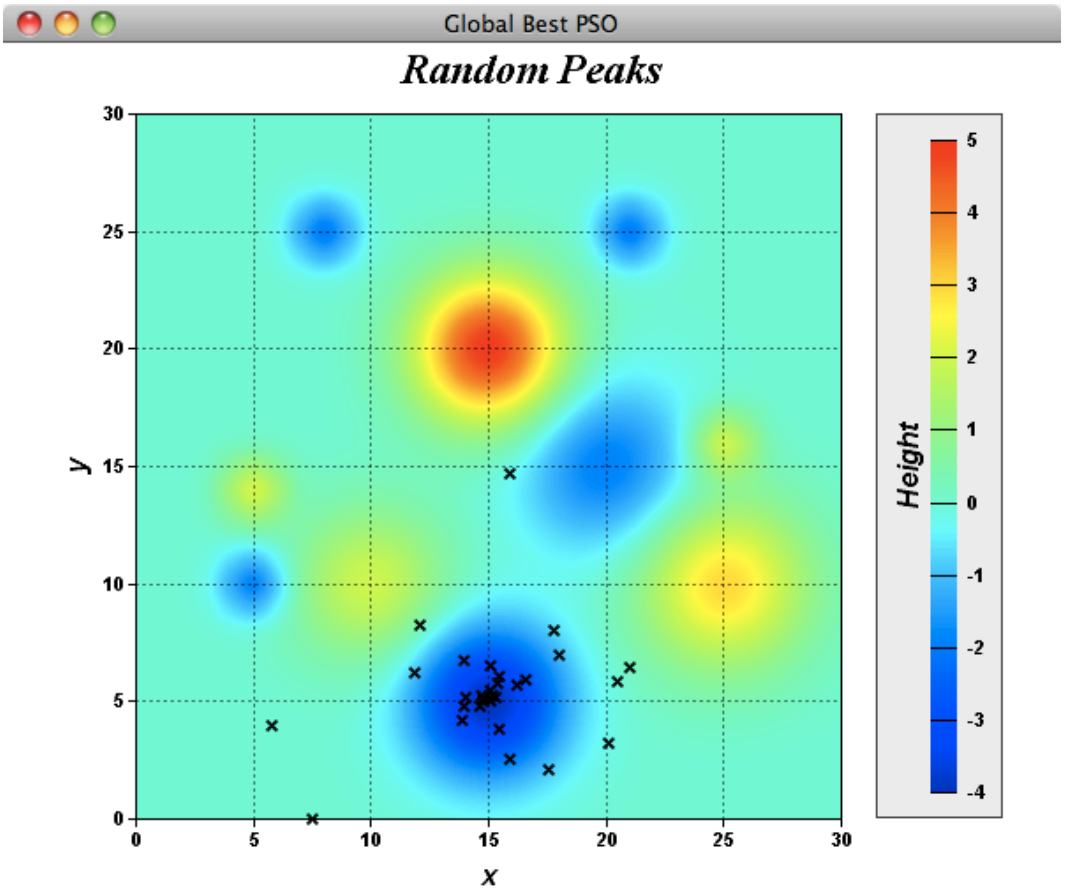

Figura 16. Exibição de uma simulação em tempo real.

3.5.2 Análise de resultados

Atualmente, a ferramenta permite que o usuário analise os resultados de uma simulação através da construção de gráficos *Box Plot* e de linha. Para isso, ao terminar uma simulação, deve-se entrar no ambiente de resultados do DOSA, conforme Figura 17.

Em seguida, deve-se escolher qual tipo de gráfico se deseja construir. O próximo passo é selecionar o diretório onde se encontram os arquivos com os resultados de uma simulação. Para isso, em *Results Directory*, o usuário deve clicar em *Browse* e escolher o diretório. Ao selecionar o diretório, a ferramenta realizará uma análise de todos os arquivos de resultados encontrados para verificar quais métricas foram aplicadas na simulação e qual a última iteração que é comum a todas as simulações. Em seguida, a opção *Measurement* será habilitada e exibirá as métricas encontradas nos resultados da simulação. O usuário deve então selecionar qual métrica deseja analisar e qual o passo que será utilizado na geração do gráfico.

Cada simulação pode ter milhares de iterações. Nesses casos, mostrar todas as iterações no gráfico pode prejudicar seu entendimento. A propriedade passo é então utilizada para mostrar-se apenas um conjunto selecionado de iterações. Ao se escolher um passo igual a 100, por exemplo, apenas as iterações múltiplas de 100 serão mostradas no gráfico, ou seja, {0, 100, 200, ... }. Por fim, basta clicar em *Create Chart* para que o gráfico seja criado.

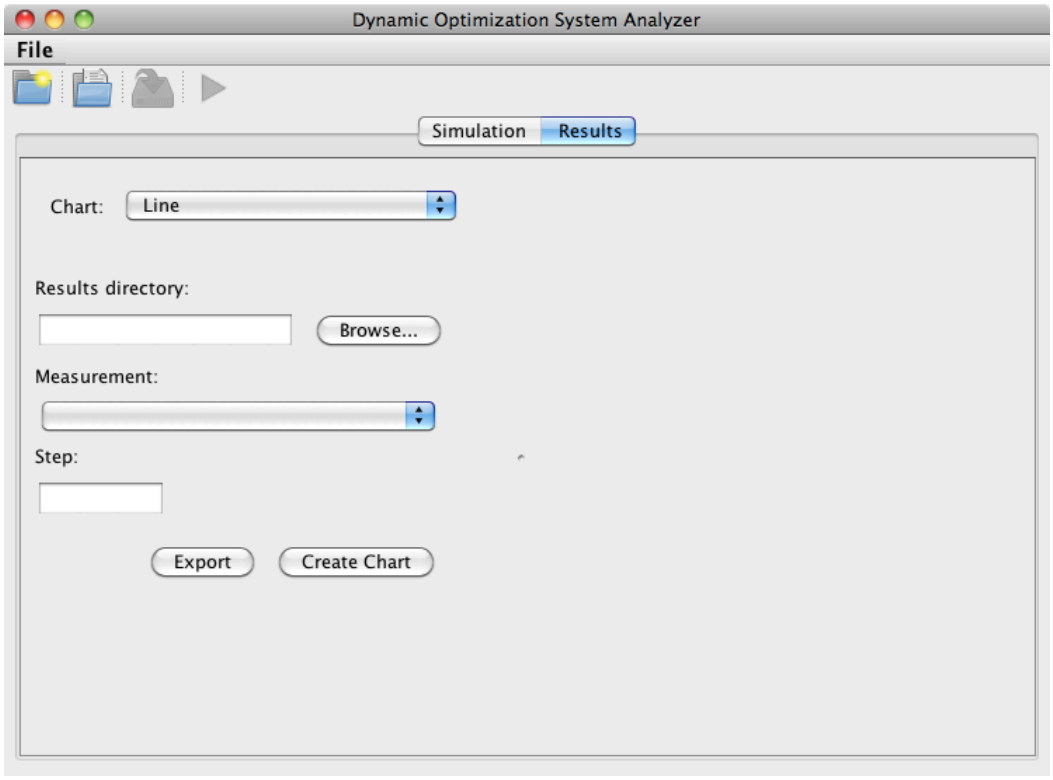

Figura 17. Tela inicial do ambiente de resultados.

A Figura 18 ilustra a análise de um resultado através do gráfico *Box Plot* com a métrica *Best Fitness* e passo igual a 10 em um conjunto de simulações com 100 iterações. Já a Figura 19 ilustra a análise de um resultado através do gráfico de linha com a métrica *Fitness Médio* e passo igual a 20 em um conjunto com 1.000 iterações.

A opção de análise de resultados através do gráfico de linha possibilita ainda que os dados utilizados para geração do gráfico sejam exportados em um arquivo *Microsoft Excel*. Dessa forma, permite-se que os dados sejam facilmente analisados em outras ferramentas. Para isso, basta preencher os campos da tela conforme descrito anteriormente e, em seguida, clicar no botão *Export* localizado ao lado do botão *Create Chart*.

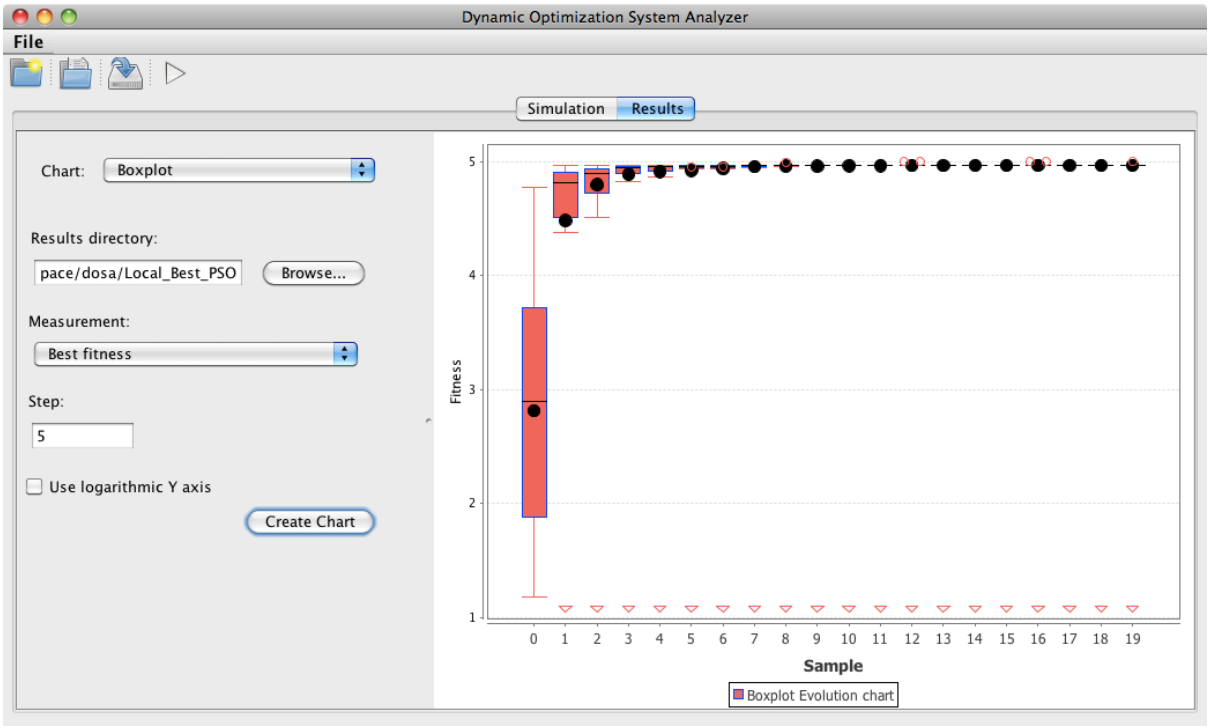

Figura 18. Tela de análise de resultados usando o gráfico *Box Plot*.

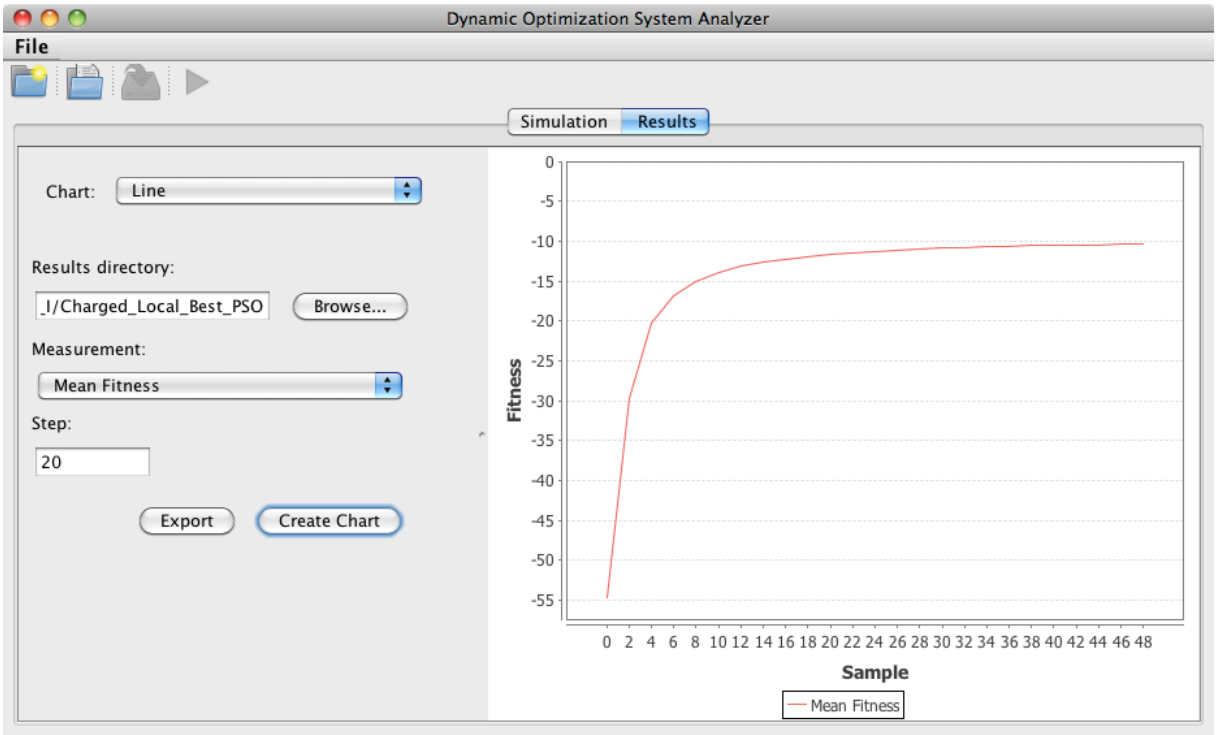

Figura 19. Tela de análise de resultados usando o gráfico de linha.

Capítulo 4

Estudo de Caso

Este capítulo apresenta dois estudos de caso com os algoritmos descritos no capítulo 2 em ambientes dinâmicos. A métrica utilizada para comparar os algoritmos foi a *Fitness* Coletivo, descrita na subseção 2.12.2.

4.1 Análise de Desempenho dos Algoritmos

Nesta seção são analisados os desempenhos dos algoritmos *Charged PSO*, *Clan PSO* e FSS em ambientes dinâmicos. Para isso, são utilizadas as funções DF1 [19] e *Moving Peaks* [20]. A seguir, os experimentos realizados são explicados em mais detalhes.

Os algoritmos foram simulados nos três tipos de ambiente dinâmico descritos na seção 2.7 com o objetivo de maximizar as funções DF1 e *Moving Peaks*. A função DF1 foi configurada conforme a Tabela 4. Já a função *Moving Peaks* foi configurada conforme a Tabela 5.

Tabela 4. Parâmetros de configuração da função DF1.

Parâmetros	Tipo I	Tipo II	Tipo III
Número de Dimensões	10	10	10
Número de picos	10	10	10
Atualiza altura	Não	Sim	Sim
Atualiza superfície	Não	Sim	Sim
Atualiza posição	Sim	Não	Sim
Intervalo entre atualizações	20 iterações	20 iterações	20 iterações
H_{base}	2	2	2
H_{limite}	10	10	10
R_{base}	1	1	1
R_{limite}	7	7	7
X_{base}	-10	-10	-10
X_{limite}	20	20	20
A_h	3,2	3,2	3,2
A_r	1,2	1,2	1,2
A_x	3,2	3,2	3,2

Tabela 5. Parâmetros de configuração da função *Moving Peaks*.

Parâmetros	Tipo I	Tipo II	Tipo III
Número de Dimensões	10	10	10
Número de picos	10	10	10
Intervalo entre atualizações	20 iterações	20 iterações	20 iterações
Tamanho do vetor de mudança s	5	0	5
Severidade de atualização de altura	0	0,5	0,5
Severidade de atualização de largura	0	0,01	0,01

O *Clan* PSO e o *Charged* PSO foram configurados seguindo os valores sugeridos em [8], onde o fator de inércia w é igual a 0,729844 e os coeficientes c_1 e c_2 são iguais a 1,494. Os dois algoritmos foram executados com um enxame de 30 partículas. Além disso, o *Charged* PSO foi configurado com p_{core} e p iguais a 1 e $\sqrt{3}x_{max}$, respectivamente, e com metade do seu enxame formado por partículas carregadas com carga Q igual a 16. A outra metade foi formada por partículas neutras ($Q = 0$). Por fim, o *Charged* PSO utilizou a topologia anel. O *Clan* PSO, por sua vez, foi configurado com três clãs de 10 partículas cada. Ele utilizou a topologia estrela tanto dentro dos clãs quanto na conferência dos líderes.

O FSS foi configurado com um cardume de tamanho 30, peso inicial igual a 500, passo individual inicial e final iguais a 1 e 0,01%, respectivamente, e passo volitivo inicial e final iguais a 1 e 0,1%, respectivamente. Todos os casos foram simulados 30 vezes com 1000 iterações por simulação.

A Tabela 6 apresenta os resultados obtidos com a função DF1. As figuras Figura 20, Figura 21 e Figura 22 apresentam a evolução do *fitness* da função DF1 nos ambientes tipo I, tipo II e tipo III, respectivamente.

Tabela 6. Análise dos algoritmos com a função DF1 através da métrica *Fitness* Coletivo.

Algoritmo	Tipo I	Tipo II	Tipo III
Charged PSO	-10,328 $\pm$ 7,263	-39,427 $\pm$ 8,881	-12,003 $\pm$ 1,624
Clan PSO	-22,080 $\pm$ 8,164	-49,599 $\pm$ 11,647	-23,470 $\pm$ 3,414
FSS	-22,332 $\pm$ 8,030	-55,527 $\pm$ 9,296	-27,941 $\pm$ 3,065

Na Tabela 7 pode-se observar os resultados obtidos com a função *Moving Peaks*. As Figura 23, Figura 24 e Figura 25 mostram a evolução do *fitness* da função *Moving Peaks* nos ambientes tipo I, tipo II e tipo III, respectivamente.

Tabela 7. Análise dos algoritmos com a função *Moving Peaks* através da métrica *Fitness* Coletivo.

Algoritmo	Tipo I	Tipo II	Tipo III
Charged PSO	35,689 ± 4,674	13,765 ± 3,244	13,614 ± 2,525
Clan PSO	30,735 ± 5,962	7,863 ± 2,722	8,230 ± 3,744
FSS	34,064 ± 7,957	7,836 ± 4,000	7,485 ± 3,344

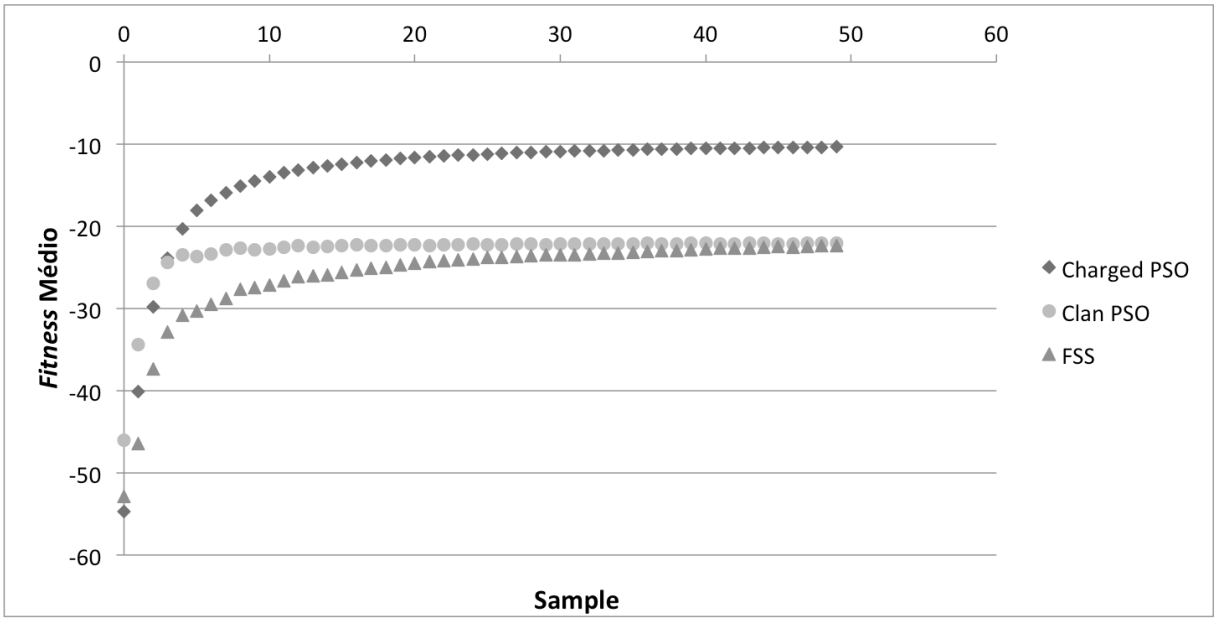

Figura 20. Evolução do *fitness* da função DF1 no ambiente tipo I após 1.000 iterações.

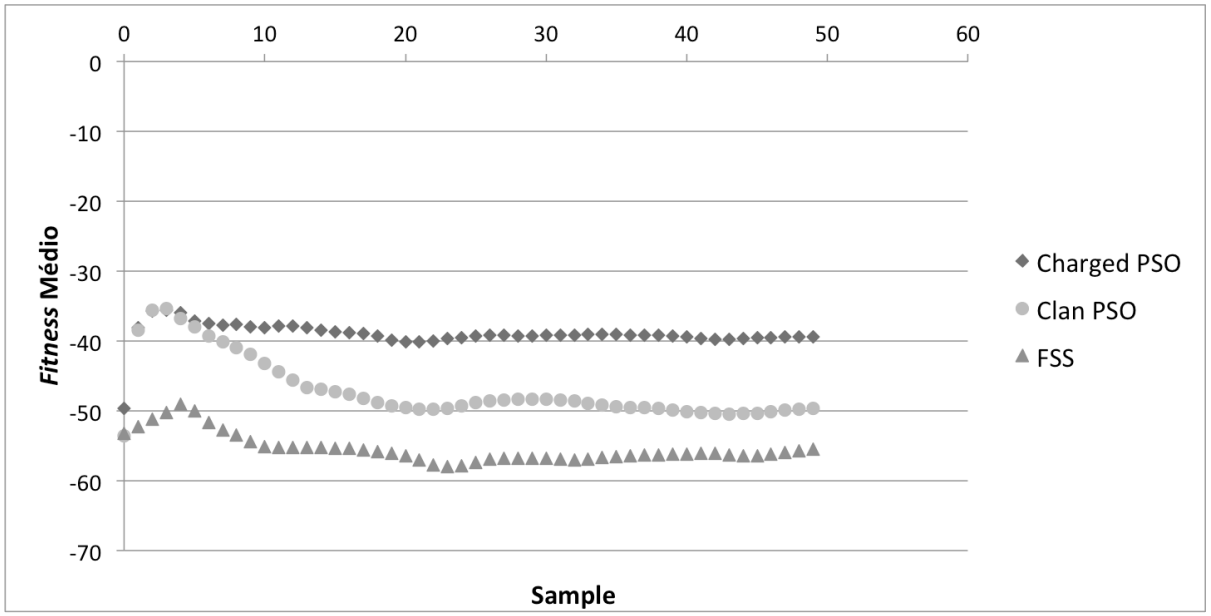

Figura 21. Evolução do *fitness* da função DF1 no ambiente tipo II após 1.000 iterações.

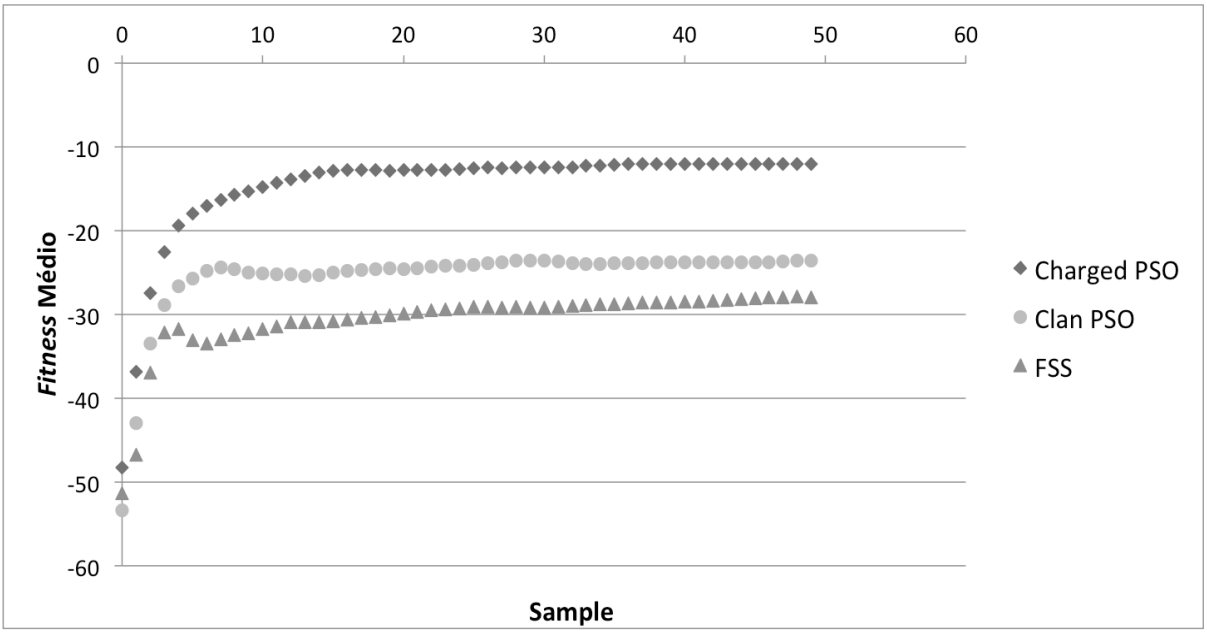

Figura 22. Evolução do *fitness* da função DF1 no ambiente tipo III após 1.000 iterações.

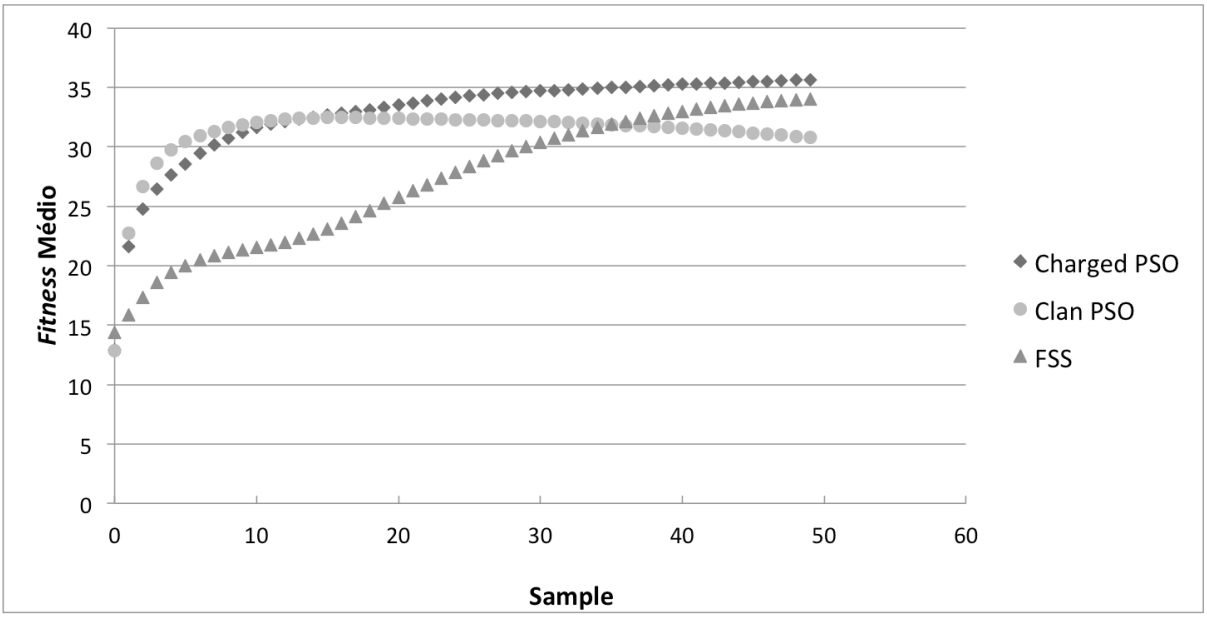

Figura 23. Evolução do *fitness* da função *Moving Peaks* no ambiente tipo I após 1.000 iterações.

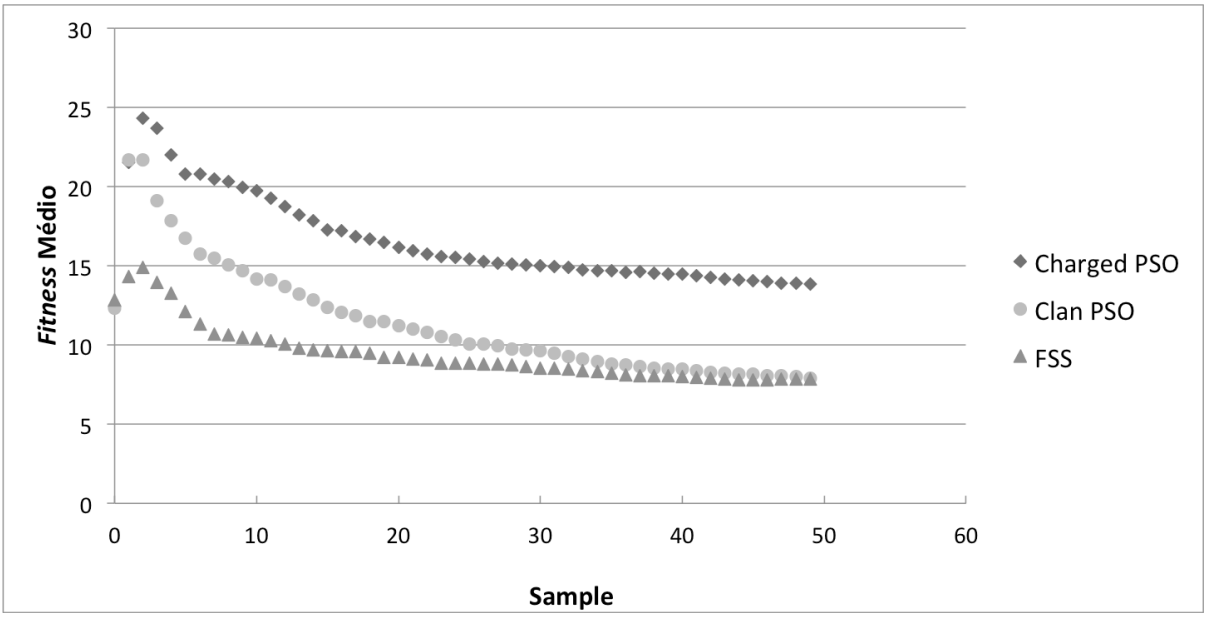

Figura 24. Evolução do *fitness* da função *Moving Peaks* no ambiente tipo II após 1.000 iterações.

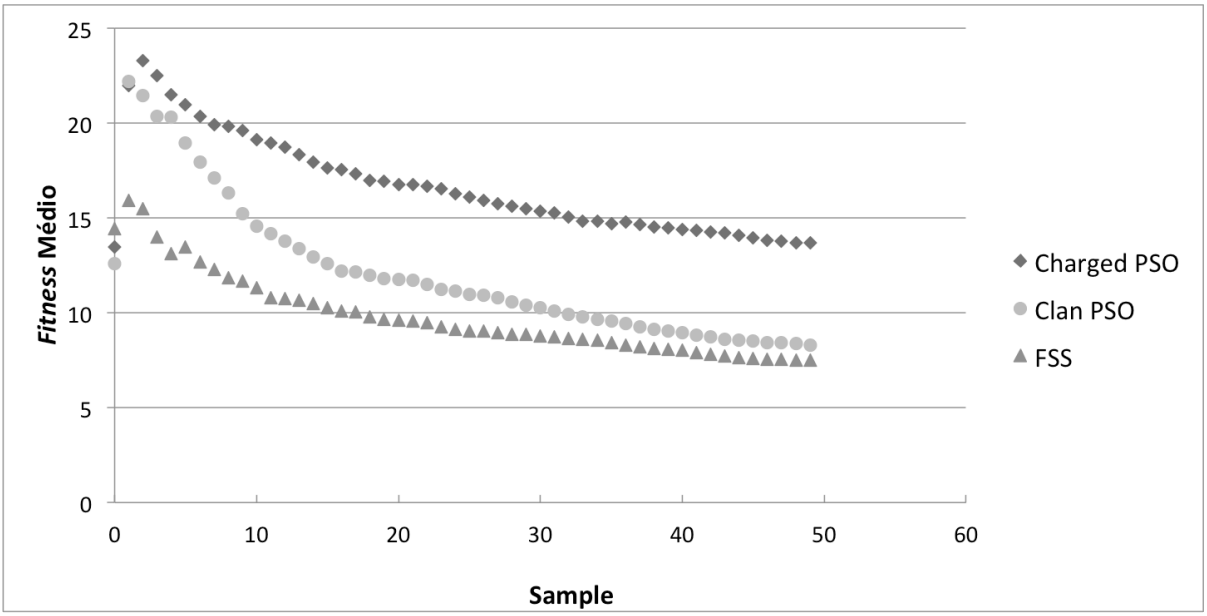

Figura 25. Evolução do *fitness* da função *Moving Peaks* no ambiente tipo III após 1.000 iterações.

O *Charged* PSO teve, na média, um desempenho geral melhor que os demais nos experimentos conduzidos. Porém, ao analisar o desvio padrão de grande parte desses experimentos, nota-se que apenas com este resultado não é possível afirmar com precisão que esse algoritmo sempre terá desempenho superior aos demais nas condições analisadas. Por isso, novos estudos fazem-se necessários, onde os parâmetros dos algoritmos sejam variados e analisados em mais detalhes. Estes estudos, porém, fogem do escopo deste trabalho.

4.2 Análise de Desempenho do *Charged* PSO

Este estudo de caso visa analisar o desempenho do *Charged PSO* ao se alterar o seu número de partículas carregadas. Para isso, simulou-se o algoritmo nos três tipos de ambientes explicados na seção 2.7 utilizando-se a função DF1 [19] configurada de acordo com a Tabela 4.

O *Charged* PSO foi configurado conforme descrito na seção 4.1 e seu percentual de partículas carregadas foi variado entre 10, 50 e 100% do enxame.

Cada tipo de ambiente dinâmico foi simulado 30 vezes, onde cada simulação foi formada por 1.000 iterações.

A Tabela 8 apresenta os resultados obtidos. Observa-se que o algoritmo teve um desempenho geral melhor quando configurado com 10% de suas partículas carregadas. Ao configurá-lo com todas as partículas (100%) do exame carregadas, ele obteve os piores desempenhos. Isso porque nenhuma partícula consegue se aproximar de outra devido às forças de repulsão eletrostática e, consequentemente, o enxame não consegue realizar buscas em profundidade.

Tabela 8. Análise do *Charged* PSO com a função DF1 usando a métrica *Fitness* Coletivo.

Porcentagem de partículas carregadas	**Tipo I**	**Tipo II**	**Tipo III**
10	-8,932 ± 5,436	-38,011 ± 8,912	-12,568 ± 1,863
50	-12,759 ± 8,209	-44,011 ± 9,229	-11,904 ± 1,390
100	-13,131 ± 6,052	-47,701 ± 8,651	-14,213 ± 2,035

As Figura 26, Figura 27 e Figura 28 apresentam a evolução do *fitness* médio da função DF1 com o *Charged* PSO nos ambientes tipo I, tipo II e tipo III, respectivamente.

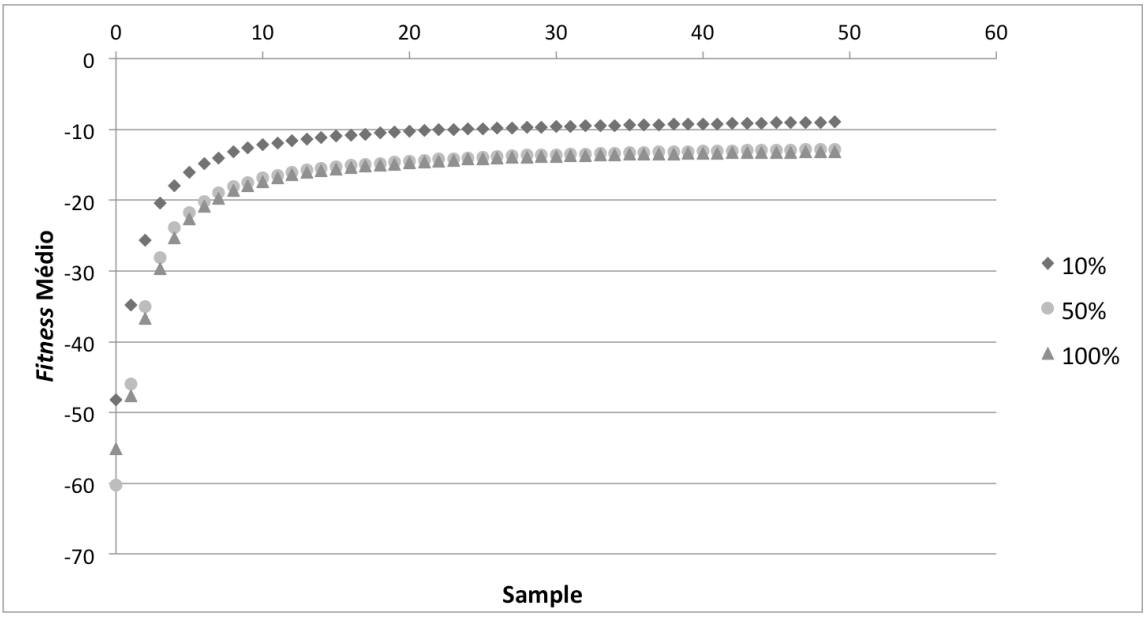

Figura 26. Evolução do *fitness* da função DF1 no ambiente tipo I após 1.000 iterações.

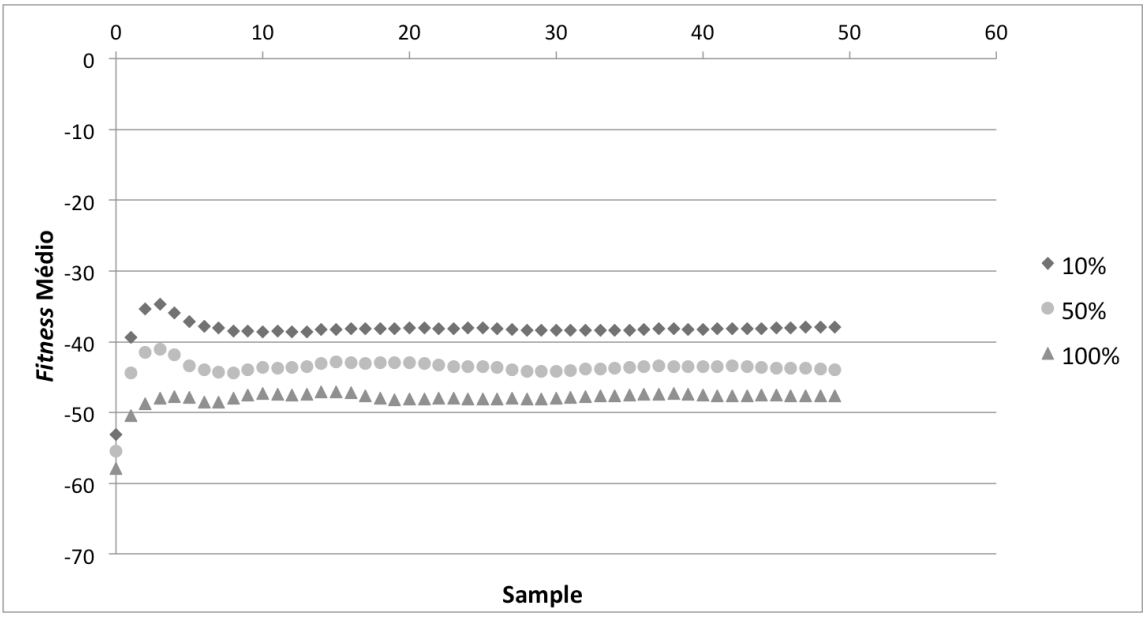

Figura 27. Evolução do *fitness* da função DF1 no ambiente tipo II após 1.000 iterações.

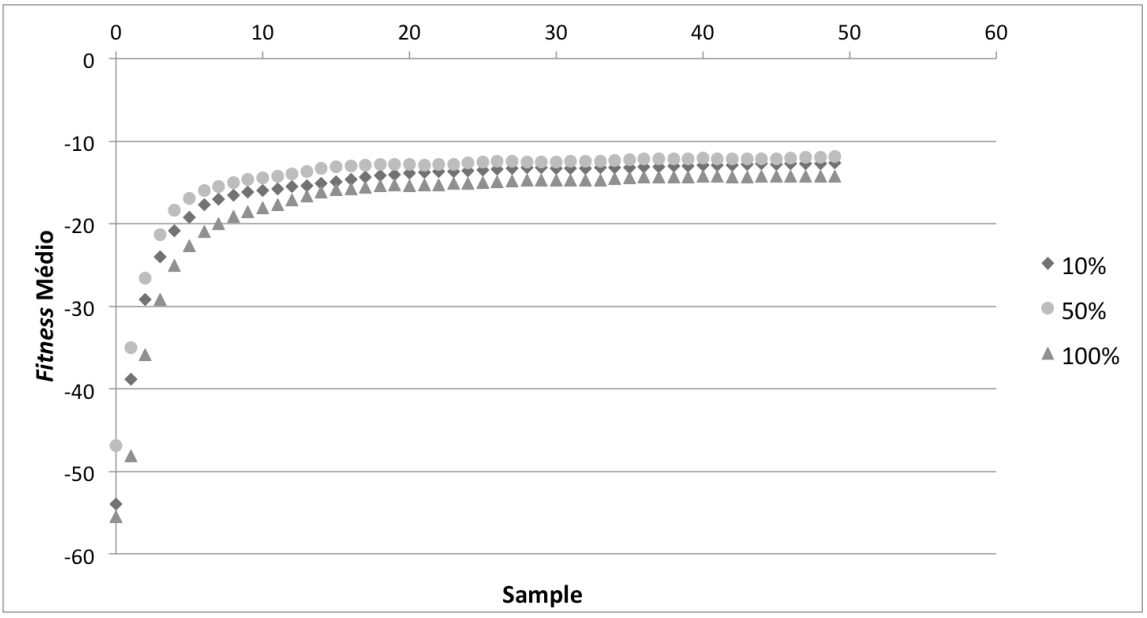

Figura 28. Evolução do *fitness* da função DF1 no ambiente tipo III após 1.000 iterações.

Capítulo 5

Conclusão

5.1 Conclusão e Contribuições

Grande parte das análises realizadas em algoritmos baseados em inteligência de enxames pautam-se em resultados experimentais. Para que esses resultados sejam representativos, é preciso que se realize um grande número de experimentos, a fim de se mostrar que os valores obtidos são consistentes. Com isso, sempre que um novo algoritmo é desenvolvido ou um existente é testado, é preciso que se desenvolva toda uma estrutura para teste e posterior análise dos resultados gerados. Além disso, no caso da otimização de problemas dinâmicos é preciso também implementar as funções dinâmicas ou os geradores de problemas dinâmicos e as métricas que serão utilizadas para coleta dos resultados. Tudo isso demanda esforço e tempo para ser desenvolvido. E, em geral, logo após os experimentos serem realizados, a estrutura de testes desenvolvida é descartada.

Este trabalho se mostra como uma alternativa ao cenário descrito acima uma vez que ele trás um ambiente pronto para implementação e simulação de algoritmos baseados em inteligência de enxames para otimização de problemas dinâmicos e para análise dos resultados gerados. O capítulo 4 demostrou as funcionalidades do sistema proposto através da realização de dois estudos de caso.

Como principais contribuições do sistema proposto, pode-se citar:

1. O desenvolvimento de um *framework* que trás toda a infraestrutura necessária para implementação e simulação de algoritmos baseados em inteligência de enxames e de problemas de otimização dinâmicos, permitindo que o usuário empregue seu tempo apenas no que é importante;
2. A facilidade de criação e simulação de cenários de teste através de um ambiente gráfico simples e amigável;
3. A padronização dos resultados gerados em cada simulação e a facilidade de análise dos mesmos através do ambiente gráfico desenvolvido;

4. A possibilidade de uso didático da ferramenta em cursos de inteligência computacional;

5.2 Trabalhos Futuros

Como trabalho futuro pode-se implementar novos algoritmos baseados em inteligência de exames para otimização de problemas dinâmicos e novos geradores de problemas dinâmicos, como o GDBG [17]. Além disso, pode-se aprofundar o estudo de caso realizado na seção 4.1 a fim de se obter resultados mais precisos sobre o desempenho dos algoritmos analisados e sobre a influência dos parâmetros existentes em cada algoritmo.

Pode-se também melhorar o mecanismo de anotações criado para configuração das entidades do sistema, a fim de permitir que ele suporte tipos de dados complexos, como listas e *arrays,* e de torná-lo mais flexível ao possibilitar que, além de atributos de classes, métodos também sejam anotados.

Atualmente, os resultados podem ser impressos na saída padrão ou em um arquivo texto. Um ponto de melhoria seria imprimir os resultados no formato XML a fim de facilitar o seu processamento. Devido a arquitetura modular desenvolvida, essa é uma melhoria bastante simples de ser realizada. Pode-se, também, melhorar o mecanismo de tratamento de exceção do ambiente gráfico, a fim de se garantir que erros gerados em tempo de execução cheguem até o usuário final através de mensagens intuitivas e, assim, melhorar a usabilidade da ferramenta.

Por fim, é preciso escrever documentações sobre a ferramenta e disponibilizá-las *online* a fim de se divulgar o trabalho desenvolvido e facilitar o uso do mesmo para novos usuários.

Bibliografia

[1] WEICKER, K. ***Performance Measures for Dynamic Environments***. Em: *Parallel Problem Solving from Nature - PPSN VII*, p. 64-73, 2002.

[2] MORRISON, R. W. ***Performance measurement in dynamic environments***. Em: *GECCO 2003: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, p. 99-102, 2003.

[3] JIN, Y. e BRANKE, J. ***Evolutionary Optimization in Uncertain Environments - A Survey***. Em: *IEEE Transactions on Evolutionary Computation*, p. 303-317, 2005.

[4] HENDTLASS T. MOSER, I. e RANDALL, M. ***Dynamic Problems and Nature Inspired Meta-Heuristics***. Em: 2006 *Second IEEE International Conference em e-Science e Grid Computing (e-Science'06)*, p. 111-111, 2006.

[5] EBERHART, R.C. e SHI, Y. ***Tracking and optimizing dynamic systems with particle swarms***. Em: *Proceedings of the 2001 Congress on Evolutionary Computation*, p. 94-100, 2001.

[6] RAKITIANSKAIA, A. e ENGELBRECHT, A. P. ***Cooperative charged particle swarm optimiser***. Em: *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, p. 933-939, 2008.

[7] LIU, L., WANG, D., e YANG, S. ***Compound Particle Swarm Optimization in Dynamic Environments***. Em: *Applications of Evolutionary Computing*, p. 616-625, 2008.

[8] BLACKWELL, T. M. e BENTLEY, P. J. ***Dynamic Search with Charged Swarms.*** Em: Proceedigns of the Genetic and Evolutionary Computation Conference, p. 19-26, 2002.

[9] EBEHART R. e KENNEDY J., ***A New Optimizer Using Particle Swarm Theory***. Em: *Proceedings of the Sixth International Symposium on Micro Machine and Human Science, p. 39-43, 1995.*

[10] DORIGO, M. e DI CARO, G. ***Ant colony optimization: a new meta-heuristic***. Em: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99*, p. 1470–1477, 1999.

[11] BASTOS FILHO, C. J. A. *at al.* ***A Novel Search Algorithm based on Fish School Behavior.*** Em: *IEEE International Conference on Systems, Man, and Cybernetics*, 2008. IEEE SMC2008., p. 2646-2651, 2008.

[12] BRATTON D., KENNEDY J., ***Defining a Standard for Particle Swarm Optimization***. Em: *Swarm Intelligence Symposium, 2007. SIS 2007. IEEE*, p. 120-127, 2007.

[13] ENGELBRECHT, A. P. **Computational Intelligence: An Introduction.** *John Wiley & Sons*, 2007.

[14] EBEHART R. e SHI Y. ***Particle Swarm Optimization: Developments, applications and resources.*** Em: *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2001)*, p. 81-86, 2001.

[15] CARVALHO, D. F. e BASTOS FILHO, C. J. A. ***Clan Particle Swarm Optimization***. Em: *IEEE World Congress on Computational Intelligence*, p. 3044-3051, 2008.

[16] BASTOS FILHO, C. J. A. *at al.* ***Fish School Search***. Em: *Nature-Inspired Algorithms for Optimisation*, p. 261-277, 2009.

[17] LI, C. e YANG, S. ***A Generalized Approach to Construct Benchmark Problems for Dynamic Optimization***. Em: *Simulated Evolution and Learning*, p. 391-400, 2008.

[18] HU, X. e EBERHART, R. C. ***Adaptive Particle Swarm Optimization: Detection and Response to Dynamic Systems***. Em: *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, p. 1666-1670, 2002.

[19] MORRISON, R. W. e DE JONG, K. A. ***A Test Problem Generator for Non-Stationary Environments***. Em: *Proceedings of the 1999 Congress on Evolutionary Computation*, p. 1-7, 2002.

[20] BRANKE, J. ***Memory enhanced evolutionary algorithms for changing optimization problems***. Em: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99*, p. 1875-1882, 1999.

[21] *Oracle and Java.* **Tecnologia *Java*.** Disponível em: <http://www.oracle.com/us/technologies/java>. Último acesso em 26 de outubro de 2010.

[22] *Eclipse Home*. **Site Oficial do *Eclipse***. Disponível em: <http://www.eclipse.org>. Último acesso em 26 de outubro de 2010.

[23] *Maven – Welcome to Apache Maven.* **Site Oficial do *Maven***. Disponível em <http://maven.apache.org>. Último acesso em 26 de outubro de 2010.

[24] *Welcome to NetBeans*. **Site Oficial do *NetBeans***. Disponível em <http://netbeans.org>. Último acesso em 26 de outubro de 2010.

[25] *JFreeChart*. **Site Oficial do *JFreeChart***. Disponível em <http://www.jfree.org/jfreechart>. Último acesso em 26 de outubro de 2010.

[26] *ChartDirector Chart Component and Control Library*. **Site Oficial do *ChartDirector***. Disponível em <http://www.advsofteng.com>. Último acesso em 26 de outubro de 2010.

[27] GAMMA, E., HELM R., JOHNSON R., VLISSIDES J., **Design Patterns - Elements of Reusable Object-Oriented Software**. 1 ed. Editora Addison-Wesley Pub Co, 1995.

[28] *Trail: The Reflection API*. **Tutorial Java sobre Reflection**. Disponível em: <http://download.oracle.com/javase/tutorial/reflect/index.html>. Último acesso em 28 de novembro de 2010.