OPTIMISTIC FRAMEWORK: PROPOSTA DE ARQUITETURA PARA RESOLUÇÃO DE PROBLEMAS DE OTIMIZAÇÃO

Gustavo Silva Semaan[1,2], Flávio Barreiro Landes[1]
[1] Inst. do Noroeste Fluminense de Educação Superior - Universidade Federal Fluminense
[2] Escola de Engenharia Industrial e Metalúrgica de Volta Redonda – UFF
[1] Av. João Jasbick, s/nº - Aeroporto - Santo Antônio de Pádua -RJ
{gustavosemaan, flavio_landes}@id.uff.br

Edelberto Franco Silva
Departamento de Ciência da Computação - Universidade Federal de Juiz de Fora
Campus UFJF, Via Local, 4569, Juiz de Fora – MG
edelberto@ice.ufjf.br

José André de Moura Brito
Escola Nacional de Ciências (ENCE) - Instituto Brasileiro de Geografia e Estatística (IBGE)
Rua André Cavalcanti, 106, Bairro de Fatima – Centro – Rio de Janeiro- RJ
jambrito@gmail.com

Luiz Satoru Ochi
Instituto de Computação - UFF
Av. Gal. Milton Tavares de Souza, s/nº - São Domingos - Niterói – RJ
luiz.satoru@gmail.com

RESUMO

O presente trabalho apresenta a arquitetura do OPTimistic, um framework em java com três camadas para resolução de problemas de otimização, inspirado nos frameworks HyFlex e ECJ. As principais funcionalidades do OPTimistic são: fornece modelos metaheurísticos; é simples desenvolver novas metaheurísticas; possui arquitetura baseada em um mecanismo hiperheurístico; Possui uma engine robusta que interpreta algoritmos sofisticados e fornece uma interface hiperheuristica; As camadas do framework são: (i) Baixo Nível: onde problemas de otimização devem ser implementados; (ii) Alto Nível: modelos metaheurísticos e heurísticas clássicas estão disponíveis para reuso; (iii) Barreira de Domínio: atua centralizando a comunicação; Nesse trabalho foram utilizados o modelo da metaheurística ILS e o clássico Problema da Mochila (0-1). Adicionalmente, em relação ao OPTimistic, são propostos diversos novos caminhos de pesquisa.

Palavra-chave: Metaheurísticas, hiperheurísticas, framework, Otimização, Arquitetura.

ABSTRACT

The present work introduce the OPTimisitc Architecture, a three layers java-based framework for optimization problems inspired by HyFlex and ECJ frameworks. The OPTimistic main features are: providing metaheuristic models; It is easy implements new

metaheuristics; The architecture based on hyperheuristic mechanism; Robust Engine that interprets sophisticated algorithms and provides a hyperheuristic interface; The framework layers are: (i) Low Level where specific optimization problems must be implemented; (ii) High Level: metaheuristic models and classical general heuristics are available. (iii) Domain Barrier: acts centralizing all communication. In this work ILS metaheuristic and a Binary Knapsack Problem were considered. Additionally, the OPTimistic project purpose several new research ways.

Keywords: Metaheuristic, Hyperheuristic, Framework, Optimization, Architecture.

Como Citar:
SEMAAN, Gustavo Silva et al. OPTimistic Framework: proposta de arquitetura para resolução de problemas de otimização. *In*: SIMPÓSIO DE PESQUISA OPERACIONAL E LOGÍSTICA DA MARINHA, 19., 2019, Rio de Janeiro, RJ. **Anais** […]. Rio de Janeiro: Centro de Análises de Sistemas Navais, 2019.

1. INTRODUÇÃO

O OPTimistic é um framework para problemas de otimização desenvolvido em Java, com arquitetura planejada com o objetivo de abstrair o problema a ser resolvido por meio de hiperheurísticas inspiradas, principalmente, no framework HyFlex (*A Flexible Frameworkfor the Design and Analysis of Hyper-heuristics*) [3]. Para isso, ele possui três camadas (ou níveis), denominadas: (i) Baixo Nível, onde os procedimentos para resolver Problemas de Otimização Específicos são implementados; (ii) Alto Nível (ou camada superior) em que algoritmos "genéricos" são formados e invocam os procedimentos da camada de baixo nível e a (iii) Barreira Intermediária, responsável pela efetiva abstração do problema a ser resolvido, não havendo troca de informações entre os níveis Alto e Baixo.

O OPTimistic foi idealizado para atuar em três frentes não mutuamente excludentes: (i) ele fornece um mecanismo hiperheurístico através de sua arquitetura incluindo uma *Engine* e os padrões de projeto adotados; (ii) disponibiliza modelos meta-heurísticos para facilitar a implementação e reutilização códigos e conceitos e (iii) compõe uma arquitetura preparada para o desenvolvimento de técnicas para a Geração Automática de Algoritmos.

Uma hiperheurística diz respeito a qualquer processo heurístico que atua no gerenciamento de outras heurísticas para resolver problemas e, comumente, são definidas como "*heurísticas que gerenciam heurísticas*". Essas heurísticas não podem conter informações específicas sobre o problema de otimização a ser resolvido e, por isso, nas modelagens são apresentadas como um nível mais alto. As hiperheurísticas atuam em um conjunto de heurísticas especialistas (baixo nível) e, conforme o subconjunto de heurísticas selecionado, possibilitam resolver um problema específico. Como parâmetro de entrada são fornecidas basicamente quais heurísticas de baixo nível serão consideradas, a função de avaliação e uma instância associada ao problema [1][2][12].

O framework também foi preparado para suportar funcionalidades de Computação Evolutiva e Geração Automática de Algoritmos. Para isso sua arquitetura também foi inspirada no ECJ, um framework estabelecido e amplamente utilizado [5].

Existem frameworks em Java para implementação de metaheurísticas, Programação Genética e hiperheurísticas com grande aceitação e uso [3][5][13][14]. Entretanto, com base em especificações de pesquisas que irão contemplar também Geração Automática de Algoritmos [7][8], a arquitetura proposta já implementa ou fornece suporte aos principais requisitos necessários. O artigo está organizado da seguinte maneira: a presente seção traz

uma breve introdução do framework; a seção 2 apresenta o OPTimistic, sua arquitetura, detalha suas camadas e introduz o clássico problema da Mochila (0-1); a seção 3 apresenta análises preliminares e resultados computacionais obtidos; por fim a seção 4 apresenta as conclusões e propõe trabalhos futuros.

2. OPTIMISTIC FRAMEWORK

Conforme apresentado na Introdução, o OPTimistic framework foi idealizado para atuar em três frentes não mutuamente excludentes. Sua arquitetura tem como objetivo abstrair o problema a ser resolvido, e foi inspirada principalmente no HyFlex framework (*A Flexible Framework for the Design and Analysis of Hyper-heuristics*) [1][2][3], com o uso de três camadas, conforme a Figura 1.

O OPTimistic teve como frentes principais: (i) uma *Engine* Hiper-heurística em que heurísticas de alto nível podem invocar heurísticas de baixo nível através de um interpretador independente, de modo que não exista nenhuma comunicação entre elas. É natural existir uma abordagem baseada também em framework para facilitar futuras implementações A *Engine* considera conceitos de Teoria da Computação, em especial, de Compiladores; (ii) Modelos Meta-heurísticos para atuar em problemas de otimização com objetivo de escapar de ótimos locais de baixa qualidade; (iii) Geração Automática de Algoritmos com base na *Engine* Hiper-heurística e em sensores, uma vez que a identificação de padrões em soluções pode direcionar a busca a formação de novas soluções e de alta qualidade. Nesse contexto, especificamente, uma solução corresponde a um algoritmo de alto nível. A seção 2.1 detalha as camadas apresentadas na Figura 1.

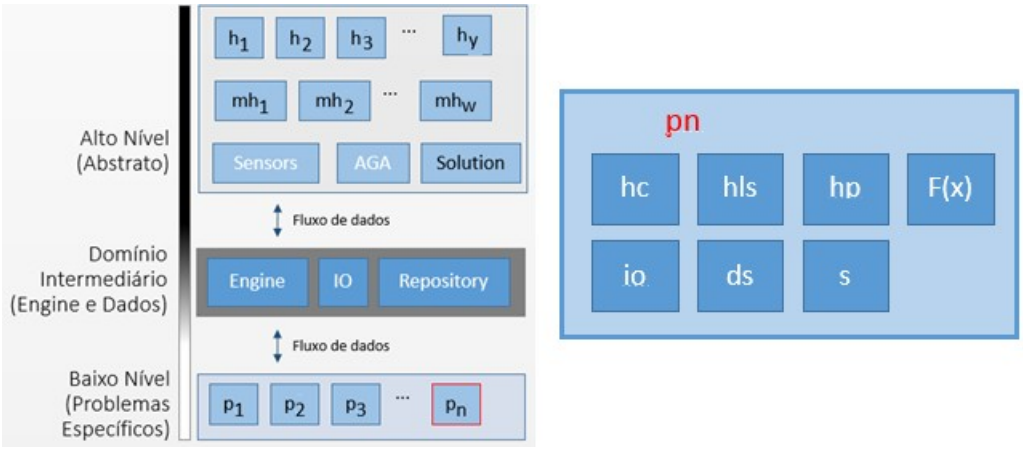

Figura 1: Detalhes das camadas do OPTimistic.

2.1. ARQUITETURA DO OPTIMISTIC

Com o objetivo de abstrair, sua arquitetura foi inspirada principalmente no framework HyFlex [3]. Assim, o OPTimistic possui três camadas denominadas: (i) Baixo Nível; (ii) Alto Nível (ou camada superior) e (iii) Barreira Intermediária. Além disso, com base no ECJ framework (*Evolutionary Computation and Genetic Programming System*) [5], módulos para geração automática de algoritmos também foram especificados com base na *Engine*, no uso de sensores que atuam na obtenção de estatísticas em relação a qualidade das soluções bem como no tempo de processamento e em conceitos de hiperheurísticas.

No Baixo Nível (do inglês, *Low Level*) são implementados os algoritmos heurísticos para a resolução do problema de otimização a ser resolvido. Assim, cada pacote java organiza as classes para atuar em um problema de otimização específico, e possui conjuntos de heurísticas de construção, de busca local e de perturbação para resolvê-lo.

No Alto Nível ocorre a especificação do algoritmo, formado por um vetor de

heurísticas de baixo nível que serão utilizadas no problema. Destaca-se que esse nível não recebe nenhuma informação do Baixo Nível e, portanto, atua de forma independente em relação ao problema de otimização submetido.

A Barreira Intermediária é responsável pela efetiva abstração do problema a ser resolvido. Não deve ocorrer troca de informações entre os níveis Alto e Baixo. Esse nível possui uma *Engine* que interpreta uma solução de alto nível (algoritmo) e executa as heurísticas de baixo nível. Dessa maneira, um dado algoritmo de alto nível pode atuar em conjuntos de heurísticas de diferentes problemas de otimização, que foram implementados no Baixo Nível. Com o objetivo de aumentar o detalhamento sobre as camadas e ilustrar a resolução de um Estudo de Caso, mais informações sobre as camadas do OPTimistic são fornecidas a seguir.

2.1.1. Camada de Baixo Nível

No Baixo Nível são implementados os algoritmos para problemas de otimização específicos. Por exemplo, o clássico Problema da Mochila Binária (BK, do inglês *Binary Knapsack Problem*) é muito utilizado por sua simplicidade, facilidade de entendimento e, ao mesmo tempo, por sua difícil resolução. Embora [4] afirme que o BK talvez seja um dos problemas "fáceis" da classe NP (*Non-Deterministic Polynomial time*), por poder ser resolvido em tempo pseudo-polinomial, após mais de uma década da publicação dessa afirmação, diversos trabalhos ainda consideram o problema [7][8].

O BK consiste em adicionar n itens com pesos pi e valores vi a uma mochila com capacidade máxima w. O objetivo é maximizar o valor total adicionado à mochila, respeitando a sua capacidade máxima, conforme apresentado na formulação de programação inteira [6]. Nessa formulação xi é uma variável binária que assume 1 se o i-ésimo item é adicionado à mochila.

$$Max\ Z\ =\ \sum_{j=1}^{n} v_j x_j$$

Sujeito a

$$\sum_{j=1}^{n} p_j x_j\ \leq\ W$$

$$x_j\ \in\ \{0,\ 1\}, j = 1,\ 2,\ \ldots,\ n$$

Para resolver o BK deve-se criar um pacote na camada Nível Baixo (com nome *Problem.BK*) e implementar heurísticas de construção, busca local e perturbação específicas desse problema.

Métodos de construção têm por objetivo criar soluções, considerando critérios aleatórios, gulosos ou semi-gulosos. Para o problema abordado, a essência dos métodos de construção consiste na ordem de tentativa de inserção de itens à mochila. Exemplo: a heurística (heurística de construção 1) insere itens na mochila conforme o maior benefício (relação valor / peso).

A Busca Local (LS, do inglês *Local Search*) tem como objetivo maximizar o valor total referente aos itens adicionados na mochila. A ideia parte do princípio de que um vizinho de uma solução pode ser melhor do que a própria solução. Assim, deve-se analisar sua estrutura de vizinhança com o intuito de encontrar soluções melhores. Por exemplo, a heurística consiste em remover o item de maior peso e, em seguida, tentar inserir itens que ainda não estão na mochila com base na ordem crescente de peso. Em outras palavras, trata-se de uma tentativa de troca entre itens disponíveis (externos) e itens que já estão na mochila para melhorar a qualidade da solução.

Por fim, perturbações têm por objetivo escapar de ótimos locais de baixa qualidade.

Para isso, uma solução é alterada sem objetivar a melhoria de sua qualidade. Após aplicação de uma perturbação, métodos de busca local podem ser aplicados na busca por novas soluções melhores que a solução antes da perturbação. Com base em conceitos do ILS (*Iterated Local Search*), a perturbação precisa ser forte suficiente para permitir que a busca local explore diferentes soluções, mas também fraca o suficiente para evitar um reinício aleatório. No contexto do problema, trata-se da remoção de itens com o objetivo de aumentar o espaço livre. Espera-se que, ao liberar espaço na mochila, novas inserções possam maximizar o valor acumulado (melhorar a solução anterior). Exemplo: a heurística hp_1 remove o item com menor benefício.

A Figura 2 apresenta uma solução de baixo nível do BK. Para este problema, a representação da solução é feita com o uso de um vetor binário (*boolean*) em que 1 (*true*) indica o item que foi adicionado à mochila, e 0 (*false*) caso contrário.

Conforme o estudo de caso apresentado, conjuntos de heurísticas de construção, busca local e perturbação para o Problema da Mochila foram implementados. Porém, ainda não foi especificado quais e como essas heurísticas serão executadas. Trata-se de uma tarefa da camada superior (Alto Nível).

Item	0	1	2	3	4	5	6
Peso	31	10	20	19	4	3	6
Valor	70	20	39	37	7	5	10

Capacidade máxima 50

Item	0	1	2	3	4	5	6
Solução	1	0	0	1	0	0	0

f(x) = 107 Peso=50

Figura 2: Exemplo de solução para o BK.

2.1.2. Camada de Alto Nível

Essa camada é responsável por determinar como serão os algoritmos, representados como soluções de alto nível. Em uma simples ilustração, essa camada recebe três números inteiros referentes à quantidade de procedimentos (heurísticas de baixo nível) disponíveis do problema abordado: construção (C), busca local (HLS) e perturbação(P). Assim, para |C| = 2, |HLS| = 5 e |P| = 3 indica que existem dois procedimentos construtivos (heurísticas) para formação de soluções iniciais, cinco heurísticas de busca local para refinar soluções e três heurísticas de perturbação.

Uma solução de alto nível (ou algoritmo), por exemplo, seria um vetor $\left[c_2, hls_1, hls_3, p_1, hls_2\right]$. Esse algoritmo utiliza a heurística de construção 2 para formar uma solução inicial que, em seguida, é refinada pelas buscas locais 1 e 3, é modificada pela heurística de perturbação 1 e, por fim, refinada pela busca local 2. Essa solução pode ser utilizada tanto para o BK como para qualquer outro problema de otimização implementado na camada de Baixo Nível que possua (i) |C| ≥ 2, (ii) |HLS| ≥ 3 e (iii) |P| ≥ 1 (para manter a compatibilidade com as chamadas de métodos).

Assim, uma solução de alto nível pode ser obtida tanto de maneira aleatória com base nos valores de |C|, |HLS| e |P|, quanto através do uso de mecanismos inteligentes. Destaca-se, novamente, que essa solução consiste em determinar a ordem em que as implementações que estão no Baixo Nível devem ser realizadas. Destaca-se que uma mesma solução de alto nível pode resultar em diferentes soluções de baixo nível, uma vez que heurísticas são invocadas.

Com o objetivo de aumentar o espaço de busca e formar soluções de alto nível diversificadas, foram implementadas diversas heurísticas clássicas da literatura para atuarem nas soluções de alto nível, como, por exemplo, as heurísticas or-opt e a 2-opt utilizadas em Problemas de Roteamento de Veículos (VRP) (do inglês *Vehicle Routing Problem*) [11].

2.1.3. Barreira intermediária (ou camada intermediária)

Conforme apresentado, para que exista uma efetiva abstração do problema a ser resolvido, não deve ocorrer troca de informações entre os níveis Alto e Baixo. Nesse sentido, a camada intermediária tem como objetivo realizar as manipulações dos dados de entrada e saída, bem como o gerenciamento de soluções geradas (tanto alto quanto de baixo nível). Porém, outra tarefa de grande importância ainda não foi apresentada: *como ocorre execução do algoritmo de alto nível para a resolução dos problemas na camada de baixo nível?*

A camada intermediária possui uma *Engine* que realiza as execuções dos algoritmos de alto nível. Trata-se de um mecanismo de tradução que, através do padrão de projeto reflection, invoca métodos de baixo nível dinamicamente, conforme a demanda apresentada pelo algoritmo. Mais que isso, conforme o tipo de problema (maximização ou minimização), faz a gestão do repositório com as soluções obtidas e a gestão de dados para geração de estatísticas.

Com base nas camadas e suas apresentações até o momento, já existe um processo por completo. Já é possível resolver um problema de otimização se forem fornecidos uma instância e os três conjuntos de heurísticas de baixo nível (C, HLS e P). Porém, para obter soluções de alta qualidade, competitivas em relação a resultados da literatura, torna-se necessário adicionar mecanismos mais inteligentes e sofisticados.

Com o objetivo de tornar o framework mais eficaz, inspirado no ECJ framework [5], a *Engine* evoluiu para suportar os componentes que formam algoritmos, como laços (*FOR e WHILE*), condicional (*IF*) e operadores (*EQUAL, OR, AND e NOT*). Assim, o mecanismo deixou de atuar como uma simples execução de métodos na interface entre as camadas, e tornou-se um interpretador de algoritmos. A Figura 3 apresenta um esboço do diagrama de classes que tratam os novos componentes.

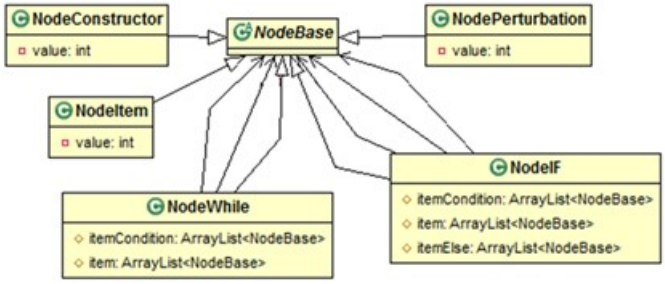

Figura 3: Componentes utilizados pela *Engine* na camada intermediária.

Ainda com base na figura, observa-se que a unidade (Classe) *NodeBase* e suas filhas possibilitam a formação de laços e condicionais aninhados, ou seja, algoritmos mais complexos podem ser formados. Uma vez que a *Engine* suporta esses componentes, tornou-se possível (e de grande relevância) a implementação de modelos meta-heurísticos no Alto Nível. Assim, com base nos mesmos conjuntos de heurísticas C, LS e P, basta especificar qual modelo meta-heurístico deve ser utilizado e nenhuma implementação adicional é necessária. No momento, o OPTimistic possui o modelo da Busca Local Iterada (ILS, do inglês *Iterated Local Search*).

Trata-se de um importante mecanismo, tendo em vista a facilidade de gerar novos algoritmos apenas selecionando uma meta-heurística e subconjuntos de C, LS e P. Indo além, pode ser interessante realizar comparativos entre resultados obtidos com meta-heurísticas mais simples como a Busca Local Iterada (ILS), uma variação mais sofisticada com Método de Descida em Vizinhança Variável (VND, do inglês *Variable Neighborhood Descent*) ILS-VND [11] e meta-heurísticas populacionais como Algoritmos Genéticos. A Figura 4 apresenta uma solução em Alto Nível e seu algoritmo (correspondente ao *Program SolutionHigh*).

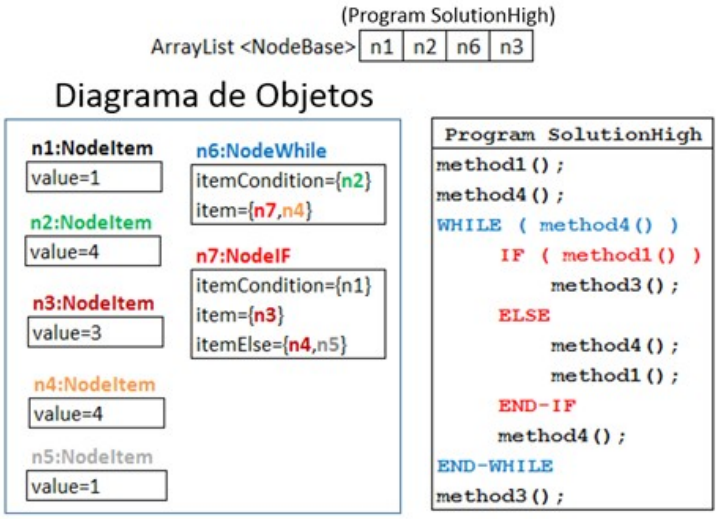

Figura 4: Solução de Alto Nível e o algoritmo correspondente.

2.2. CAMADAS DETALHADAS E ESTRUTURA FÍSICA

Com base na Figura 1 apresentam-se as siglas e suas respectivas descrições, conforme a camada. É possível observar o fluxo de dados entre as camadas, em que a camada superior não tem contato direto com a camada inferior, e elas comunicam-se através da camada intermediária.

No Alto Nível são apresentados os conjuntos de heurísticas clássicas da literatura $H=\{h_1, \ldots, h_y\}$, o conjunto de modelos metaheurísticos $MH=\{mh_1, \ldots, mh_w\}$, e os módulos *Sensors*, *AGA (Geração Automática de Algoritmos*, do inglês *Automatic Generation of Algorithms*) e *Solution*.

O Conjunto H possui heurísticas clássicas para atuar em soluções de alto nível, que desconhecem o problema de otimização abordado no Baixo Nível. Entre elas pode-se citar as heurísticas *Or-opt* (h1) e a *2-opt* (h2), amplamente utilizadas em Problemas de Roteamento de Veículos [11]; O Conjunto possui modelos de meta-heurísticas consagradas da literatura, como ILS (Iterated Local Search), Algoritmos Genéticos e GRASP (*Greedy Randomized Adaptive Search Procedure*) [9].

O módulo *Solution* atua na gestão das soluções de alto nível, no presente contexto são algoritmos. Os módulos *AGA* e *Sensors* são considerados para a geração automática de algoritmos, e foram inspirados no consagrado framework *open-source* para Computação Evolutiva com ênfase em Programação Genética ECJ [5]. Enquanto o módulo *AGA* fornece uma estrutura capaz de modelar algoritmos complexos, o módulo *Sensors* atua na identificação dos métodos e algoritmos mais eficazes e/ou mais eficientes (tempo computacional). Destaca-se que, embora o OPTimistic suporte em sua arquitetura a implementação de modelos para Programação Genética, esse assunto ainda não se apresentou como um objetivo para trabalhos futuros.

O *Domínio Intermediário* possui os módulos *Engine*, *IO* e *Repository*. O módulo *Engine*: trata-se do mecanismo tradutor de algoritmos de alto nível, que invoca métodos implementados no baixo nível e mantém a alta abstração na resolução de problemas.

O módulo *IO*: atua na gestão de entrada e saída de dados para as soluções de alto nível e de baixo nível, enquanto o módulo *Repository*: atua na gestão dessas soluções e das estatísticas relacionadas.

Por fim, *Baixo Nível* apresenta para cada problema de otimização $p_i \in P$, $P=\{p_i, \ldots, p_n\}$ os conjuntos *hc* com heurísticas de construção, *hls* com heurísticas

de busca local e *hp* com heurísticas de perturbação. Além disso, para cada problema de otimização devem ser apresentados o *dataset* (*ds*) com os dados de entrada, *io* a gestão de entrada e saída do problema de otimização, *S* conjunto de solução do problema de otimização e *f(x)* a função de avaliação do problema.

A Figura 5 ilustra a organização dos pacotes em um projeto que possui, além dos pacotes do framework (*Engine*, *HighLevel*, *LowLevel* e *Utils*), dois pacotes (*Problem.TSP* e *Problem.BK*) para resolução dos problemas (i) do Caixeiro Viajante (ii) da Mochila Binária, respectivamente.

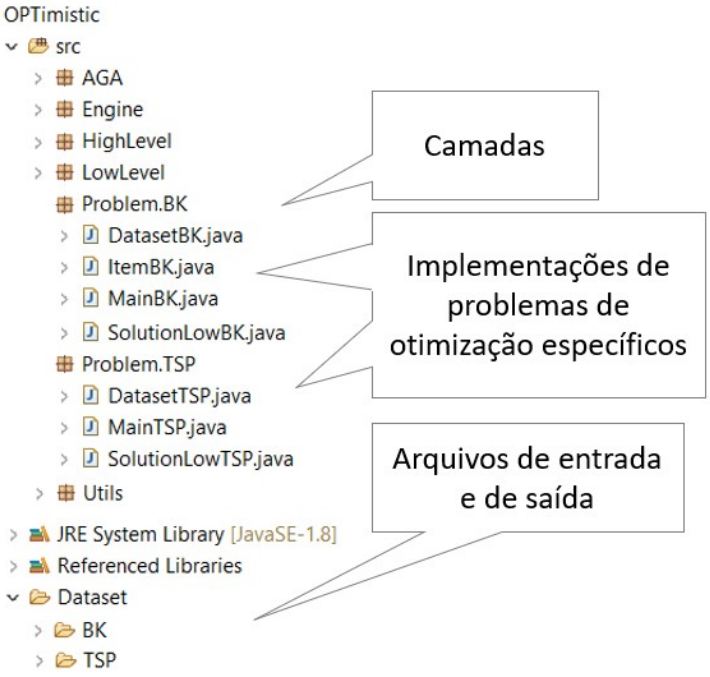

Figura 5: Estrutura física do OPTimistic.

Pouco além, destaca-se a simplicidade da implementação, em que para o TSP (assim como para o problema da mochila (0-1)) foram utilizadas apenas três classes: *DatasetTSP* para leitura e manipulação dos dados do problema, *SolutionLowTSP* onde estão os procedimentos (heurísticas) para resolver o problema e *MainTSP* que corresponde ao programa principal.

Os softwares desenvolvidos serão implementados em linguagens de programação livres (framework e exemplos), as documentações e as instâncias utilizadas nos estudos de caso serão disponibilizadas para toda a comunidade acadêmica.

3. ANÁLISES PREELIMINARES E EXPERIMENTOS COMPUTACIONAIS

Com o objetivo de realizar testes, validações e verificações em relação à arquitetura do OPTimistic, foram implementados alguns problemas clássicos de otimização combinatória, sejam eles: *Problema de Agrupamento, Problema do Caixeiro Viajante e o Problema da Mochila (0-1)*. O Problema da Mochila citado corresponde ao principal exemplo do *Sample Code Library* (SCL) do framework, que também será disponibilizado.

Além dos problemas clássicos no Baixo Nível, um modelo da metaheurística ILS foi implementado para experimentos preliminares. Uma vez que nesse modelo foram considerados os componentes com base na *superclasse NodeBase* (Figura 3), é possível realizar uma série de experimentos e análises, como:

- **Teste de Unidade (*JUnit*) e de Performance (*JUnitPerf*)**: foram realizados com as

heurísticas de baixo nível do BK. Os resultados obtidos foram comparados com os ótimos globais e/ou as melhores soluções da literatura. Foi construído, também, um algoritmo de força bruta (*backtraking*) para instâncias artificiais criadas para testes preliminares.

- **Usabilidade e Manutenibilidade do framework**: códigos foram disponibilizados para especialistas em otimização, com o objetivo de coletar críticas, sugestões e comentários.
- **Teste de Integração e *Engine*** (mecanismo hiperheurístico): Funcionamento da *Engine* em relação à interpretação das soluções de alto nível (algoritmos), inclusive do modelo da metaheurística ILS implementada de maneira preliminar para experimentos (Figura 6). A *Engine* interpreta a solução de maneira semelhante, porém, nesse caso, também são diferenciados os tipos de heurísticas: de construção (*NodeConstructor*), de BuscaLocal (*NodeItem*) e de Perturbação (*NodePerturbation*). É importante ressaltar que os nomes dos objetos e os exemplos apresentados têm como objetivo apenas facilitar o entendimento do funcionamento da *Engine*.
- **GAA**: interpretação de algoritmos gerados automaticamente ou com base em modelos metaheurísticos existentes.
- **Sensores**: experimentos preliminares em relação à obtenção de resultados e uso de estatísticas.
- **Mecanismo de *IO***: os mecanismos de entrada e de saída geridos pela camada intermediária e pelo Baixo Nível, tanto para a problemas de otimização quanto para a gestão de soluções de alto nível; O suporte ao *IO* de soluções de alto nível em relação aos funcionamentos online (gestão em memória principal) e offline (persistir algoritmos em memória secundária).

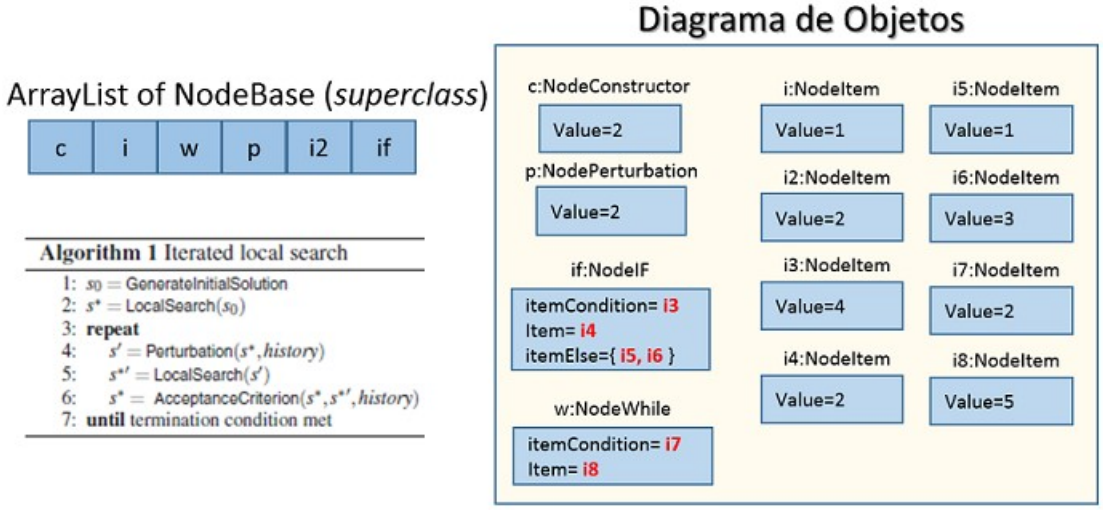

Figura 6: Modelo de Busca Local Iterada (com objetos instanciados).

3.1. SCL – PROBLEMA DA MOCHILA (0-1)

O computador utilizado para os experimentos é dotado de um processador core i3 1,70GHz, 4GB RAM e sistema operacional Kubuntu com kernel 3.16. Foram utilizadas instâncias artificiais geradas de maneira aleatória e oito instâncias da literatura (http://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html).

As instâncias da literatura possuem quantidade de itens variando entre 10 e 24, com ótimos globais relatados. Além disso, em outras instâncias artificiais criadas em experimentos preliminares, foi considerado um algoritmo de força bruta para a obtenção dos ótimos globais. A seleção do problema deve-se ao seu simples entendimento, por ser

considerado clássico, de grande aplicabilidade e de difícil resolução.

Para os experimentos foram utilizados cinco diferentes métodos de construção, que atuam especificamente na adição de itens na mochila, observando as restrições do problema. Assim, são consideradas apenas soluções válidas, em que a capacidade da mochila não excede o limite máximo submetido como parâmetro. Versões implementadas: *Solução válida aleatória* (c_1): seleciona de maneira aleatória um item ainda não inserido na mochila e tenta adicioná-lo; *Modo guloso 1* (c_2): considera a razão valor do item e seu peso (valor/peso), e segue a ordem decrescente para inserção na mochila; *Modo guloso 2* (c_3): segue a ordem crescente do peso de cada item para inserção na mochila; *Modo guloso 3*(c_4): segue a ordem decrescente do peso de cada item para inserção na mochila; *Modo guloso 4* (c_5): segue a ordem decrescente do valor de cada item para inserção na mochila.

Seguem as Buscas Locais: ls1: adiciona os elementos fora da mochila de acordo com a ordem da instância; ls_2: remove o item mais pesado contido na mochila; ls_3: remove o primeiro item contido na mochila de acordo com a para realizar uma inserção em seguida; ls_4: adiciona o item mais pesado fora da mochila; ls_5: adiciona o item mais leve que não está na mochila; ls_6: adiciona o item com o maior benefício (valor/peso) que não está na mochila; ls_7: adiciona o item com o maior valor que não está mochila; ls_8: para todos os itens e de acordo com a ordem da instância, efetua uma troca adjacente em pares diferentes contidos na solução.

Neste estudo um algoritmo de perturbação foi criado (p_1), que aleatoriamente seleciona um fator de perturbação que indicará o quanto será modificada a solução corrente (atual). Em seguida, também de maneira aleatória, seleciona o modo (inserção ou remoção) de itens.

Para os experimentos computacionais o modelo ILS foi executado 50 vezes em cada uma das 8 instâncias da literatura, com 200 iterações em cada execução. A heurística para construção de soluções iniciais é selecionada de maneira aleatória entre as versões disponíveis (5 versões), sendo todos os algoritmos de busca local (8 versões) executados sequencialmente (na mesma ordem) e o único algoritmo de perturbação utilizado.

Destaca-se que todas as implementações resultam em soluções válidas, ou seja, a restrição de capacidade da mochila foi respeitada e nenhum mecanismo de penalidade foi necessário. Assim, nas remoções de itens não há necessidade de verificação, mas itens são incluídos na mochila apenas quando a capacidade máxima for respeitada.

A Tabela 1 contém os resultados alcançados pelo método proposto neste estudo e a comparação desses resultados com os resultados ótimos globais de cada instância. É possível observar que para todas as instâncias em todas as execuções foi obtido o ótimo global. Além disso, destaca-se o baixo tempo de processamento, da ordem de segundos.

Embora as instâncias sejam pequenas em quantidade de itens, variando de 10 a 24, o ILS conseguiu encontrar as soluções ótimas globais para todas as instâncias em todas as execuções. Além disso, o tempo de processamento ficou compreendido entre 1 (um) a 62 segundos para as 200 iterações em cada instância.

Tabela 1: Resultados dos experimentos.

	Função de Avaliação			Tempo (segundos)		
Instância	**Fx (BKS)**	**Fx**	**Desvio(%)**	**Menor**	**Maior**	**Média**
P01	309	309	0	0	23	1,4
P02	51	51	0	0	4	1,1
P03	150	150	0	0	37	4,0
P04	107	107	0	0	1	0,4
P05	900	900	0	0	62	8,5
P06	1735	1735	0	0	24	4,8
P07	1458	1458	0	2	6	4,0

P08	13549094	13549094	0	7	7	7,0

4. CONCLUSÕES E TRABALHOS FUTUROS

Com o objetivo de realizar testes, validações e verificações em relação à arquitetura do OPTimistic foram implementados alguns problemas clássicos de otimização combinatória. Em especial, o Problema da Mochila (0-1) foi abordado como estudo de caso; e trata-se do principal exemplo do *Sample Code Library* (SCL) do framework, e seus fontes também serão disponibilizados.

Os Testes de Unidade e de Performance foram realizados e os objetivos foram alcançados, em que os resultados foram validados conforme a função objetivo em um tempo computacional aceitável. As análises relacionadas à Usabilidade e Manutenibilidade também foram satisfatórias, e as sugestões e críticas dos participantes foram consideradas, e serão implementadas integralmente. Para o teste de Integração foram submetidas soluções de alto nível (algoritmos) para a *Engine* interpretar e processar.

Além de algoritmos para teste, o modelo metaheurístico ILS foi submetido considerando os procedimentos de baixo nível para o problema da Mochila (0-1). Assim, tanto os Sensores quanto os mecanismos de *IO* puderam ser avaliados também. Por fim, os resultados obtidos com o problema de otimização foram comparados com os ótimos globais existentes na literatura, e para todas as instâncias em todas as execuções o ótimo global foi alcançado às expensas de baixo tempo computacional, na ordem de segundos. Com base nos testes, validações e verificações realizados a arquitetura proposta apresenta-se como uma boa alternativa para a resolução de problemas de otimização.

Como trabalhos futuros existem diversos caminhos a serem percorridos, que possuem diferentes graus de complexidade. Entre os temas considerados promissores pelos autores destacam-se:

- **Sensores e GAA:** em sua versão mais recente, os sensores são utilizados apenas para gerar estatísticas com base no tempo de execução de cada classe *NodeBase* (e derivadas) e do valor da função objetivo. Entretanto, eles foram propostos para direcionar o processo de refinamento e de construção de algoritmos, em especial, para o funcionamento do Módulo de Geração Automática de Algoritmos.
- **Identificação de Padrões:** Com base em conjuntos de soluções (tanto de alto nível (algoritmos) quanto de baixo nível (problema de otimização)), a identificação de padrões em soluções pode ser utilizada para formação de novas soluções. Nesse contexto deve-se investigar técnicas de mineração de dados e de comitê de agrupamentos [10].
- **Modelos metaheurísticos:** implementar modelos de outras metaheurísticas que possuem diferentes características, como a gestão de conjunto de soluções (população nos algoritmos evolutivos e BRKGA) ou mesmo refinamento intensivo como no ILS ou VND.
- **Problemas de Otimização:** implementar outros problemas de otimização na camada baixo nível.

5. AGRADECIMENTOS

Os autores agradecem o apoio da PROPPI (Pró-Reitoria de Pesquisa, Pós-Graduação e Inovação) da UFF e da FAPERJ (G. E-26/010.101237/2018), que financiaram esta pesquisa.

6. REFERÊNCIAS BIBLIOGRÁFICAS

[1] ELHAG, E. O. A. (2015). 13th UK *Workshop on Computational Intelligence* (UKCI).

[2] ELHAG, E. O. A. (2015). *A grouping hyper-heuristic framework: Application on graph colouring, Expert Systems with Applications. Expert Systems with Applications*, 42.

[3] BURKE, E.K., CURTOIS, T., HYDE, M., KENDALL, G., OCHOA, G., PETROVIC, S., VÁZQUEZ-RODŔIGUEZ, J.A. (2009). *HyFlex: A Flexible Framework for the Design and Analysis of Hyper-heuristics. Proceedings of the 4th Multidisciplinary International Scheduling Conference: Theory and Applications* (MISTA 2009).

[4] D. PISINGER, (2005) "*Where are the hard knapsack problems?*" *Computers & Operations Research*, vol. 32, no. 9.

[5] (*Genetic and Evolutionary Computation Conference* 2017)

[6] MARTELLO, S. AND TOTH, P. (1990) *Knapsack problems: algorithms and computer implementations*. New York, NY, USA: John Wiley & Sons, Inc.

[7] PARADA, L., HERRERA, C., SEPÚLVEDA, M., AND PARADA, V. (2016). *Evolution of new algorithms for the binary knapsack problem*. *Natural Computing*, vol. 15 Available: http://dx.doi.org/10.1007/s11047-015-9483-8

[8] PARADA, L., SEPULVEDA, M., HERRERA, C., PARADA, V. (2013) *Automatic generation of algorithms for the binary knapsack problem. In: Evolutionary Computation* (CEC), 2013 IEEE Congress on, pp. 3148–3152. IEEE (2013)

[9] RESENDE, M.; RIBEIRO, C. (2010). *Greedy randomized adaptive search procedures*. In Glover, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*, pages 283–319. Kluwer Academic Publishers.

[10] SEMAAN, G S., RODRIGUES, W., BRITO, J. A. M., AND OCHI, L. S. (2014) Método baseado em combinação de soluções com particionamento de grafos para o problema de agrupamento automático. *Learning & Nonlinear Models* (L&NLM), Volume 13.

[11] SUBRAMANIAN, A.; DRUMMOND, L. M. A.; BENTES, C.; OCHI, L. S. E FARIAS, R. (2010) "*A parallel heuristic for the vehicle routing problem with simultaneous pickup and delivery*". *In Computers and Operations Research*.

[12] SUCUPIRA, I. R. (2007). Algoritmos para o Problema de Agrupamento Automático. Dissertação de Mestrado, USP.

[13] LUKASIEWYCZ, M., GLAB, M., REIMANN, F., AND TEICH, J. (2011). *Opt4J: a modular framework for meta-heuristic optimization. In Proceedings of the 13th annual conference on Genetic and evolutionary computation* (GECCO '11), New York, NY, USA DOI: https://doi.org/10.1145/2001576.2001808

[14] DURILLO, J.J, NEBRO, A. (2011) *jMetal: A Java framework for multi-objective optimization*, *Advances in Engineering Software*, Volume 42, Issue 10, 2011. https://doi.org/10.1016/j.advengsoft.2011.05.014.