HGVPRLB: a hybrid algorithm for solving binary problems

Josiane da Costa Vieira Rezende
Marcone Jamilson Freitas Souza
Departamento de Computação
Universidade Federal de Ouro Preto (UFOP)
CEP: 35.400-000 – Ouro Preto – MG – Brasil
Email: josianecvieira@gmail.com, marcone@iceb.ufop.br

Rone Ilídio da Silva
Departamento de Tecnologia, Engenharia Civil e Computação
Universidade Federal de São João Del Rei (UFSJ)
CEP: 36.420-000 – Ouro Branco – MG – Brasil
Email: rone@ufsj.edu.br

***Abstract*—In this paper it is proposed a hybrid algorithm, so-called HGVPRLB, for solving generic binary problems. The algorithm HGVPRLB combines the heuristic procedures GRASP, Variable Neighborhood Descent, Constraint Propagation and Local Branching Cuts. It was tested in a set of binary problems from MIPLIB 2010 in order to check both its ability to obtain feasible solutions as its ability to improve the value of these solutions varying the processing time. Computational experiments showed that when the processing time increases the algorithm can increase the number of feasible solutions found in the set as well the quality of the solutions. Besides it, the proposed algorithm outperforms another algorithm of literature, as well as two other open source solvers.**

Keywords–Linear Binary Programming, Heuristics, GRASP, Variable Neighborhood Descent, Local Branching, Constraint Propagation

***Resumo*—Neste trabalho é proposto um algoritmo híbrido, nomeado HGVPRLB, para resolver problemas binários genéricos. O algoritmo HGVPRLB combina os procedimentos heurísticos GRASP, *Variable Neighborhood Descent*, propagação de restrições e cortes *Local branching*. Ele foi testado em um conjunto de problemas binários da biblioteca MIPLIB 2010 para verificar tanto sua capacidade de obter soluções viáveis quanto sua capacidade de melhorar o valor dessas soluções ao se variar o tempo de processamento. Os experimentos computacionais realizados mostraram que nesses dois quesitos o algoritmo proposto se mostrou superior a outro algoritmo da literatura, bem como a dois outros resolvedores de código aberto.**

Palavras-chave–Programação Linear Binária, Heurística, GRASP, Variable Neighborhood Descent, Local Branching, Propagação de Restrições

I. Introdução

Problemas de Programação Linear Binária (PLB) são casos particulares da Programação Linear Inteira (PLI), em que todas as variáveis de decisão são binárias [1].

Dois tipos de métodos de solução para os problemas de PLB são encontrados na literatura: exatos e heurísticos. No primeiro busca-se a melhor solução, dita solução ótima, para o problema, quando ela existe, satisfazendo a todas as restrições impostas. Porém, para muitos problemas, tal tipo de método pode requerer um tempo computacional inviável se o problema for da classe NP-difícil. Por outro lado, o segundo tipo procura uma solução sem a garantia de que ela seja ótima; entretanto, a solução pode ser encontrada em tempo de tomada de decisão, e estar muito próxima da ótima [2].

Na literatura encontram-se diversos softwares que utilizam uma dessas estratégias para solucionar os problemas de programação linear, os chamados resolvedores. Eles podem ser tanto de código aberto, quanto comerciais. Dentre eles, citamos: CPLEX [3], COIN-OR-CBC [4], Symphony [5], MINTO [6], SCIP [7], GLPK [8], LP Solve [9], *Local Solver* [10] e *Local Branching* [11].

A literatura também apresenta diversas estratégias para a solução de problemas de PLB, entre elas: Relaxamento LP [12] e Programação por Restrições [13].

Relaxamentos LP consistem em relaxar as condições de integralidade das variáveis de decisão, isto é, os valores das variáveis passam a ser números reais compreendidos entre 0 e 1. Em programação por restrições é definido um espaço de busca em um conjunto de restrições e deseja-se encontrar soluções viáveis, isto é, pontos que pertencem ao espaço de busca e que satisfazem as restrições.

Neste trabalho é proposto um algoritmo híbrido, nomeado HGVPRLB, para resolver problemas de PLB baseado na metaheurística *Greedy Randomized Adaptive Search Procedures* – GRASP [14]. Em sua fase construtiva ele mescla relaxamentos LP e programação por restrições para a obtenção de uma solução inicial. Em seguida, é realizada uma busca local, guiada por uma heurística *Variable Neighborhood Descent* – VND [15], a qual, por sua vez, faz uso de cortes *Local Branching* [11].

Experimentos computacionais realizados com problemas binários da biblioteca MIPLIB 2010 [16] mostram que o algoritmo HGVPRLB é capaz de produzir, em tempo reduzido, um número maior de soluções viáveis nessa biblioteca quando comparado com outro trabalho da literatura e com dois resolvedores de código aberto. Além disso, a qualidade das soluções produzidas pelo algoritmo proposto é de qualidade superior.

O restante deste artigo está organizado como segue. A seção II descreve o problema objeto deste trabalho. A seção III apresenta os trabalhos relacionados. Na seção IV o algoritmo proposto é apresentado. Na seção V são descritos e analisados os resultados dos experimentos computacionais realizados com

978-1-4673-9143-6/15/$31.00 ©2015 IEEE

problemas da biblioteca MIPLIB 2010. Finalmente, na seção VI são apresentadas as conclusões deste trabalho.

II. Descrição Formal do Problema

Nesta seção é descrito, de forma matemática, um problema de programação linear binária (PLB). Para isso, seja $X = \{x_1, x_2, ..., x_n\}$ um conjunto de n variáveis binárias, isto é, $x_j \in \{0,1\}\ \forall j = 1, \cdots, n$, estando a cada qual associado um custo $c_j \in \mathcal{R}$. Sejam, também, b um vetor m-dimensional com elementos $b_i \in \mathcal{R}$ e A uma matriz de dimensões $m \times n$, com elementos $a_{ij} \in \mathcal{R}$, sendo $1 \leq i \leq m$ e $1 \leq j \leq n$. A formulação matemática do problema de PLB pode, então, ser expressa pelas equações (1) a (3):

$$\max\,(ou\ \min)\ \sum_{j=1}^{n} c_j x_j \tag{1}$$

Sujeito a:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i\ \forall i = 1, \cdots, m \tag{2}$$

$$x_j \in \{0,1\}\ \ \forall j = 1, \cdots, n \tag{3}$$

III. Trabalhos Relacionados

Dada a complexidade inerente à resolução de um problema de PLI, são encontrados na literatura vários trabalhos que procuram resolvê-los de forma mais eficiente com relação ao tempo de processamento. Dentre os diversos métodos utilizados para a resolução desses problemas, alguns dos principais são descritos a seguir.

Os primeiros estudos para a resolução de funções lineares sujeitas a restrições foram desenvolvidos por Fourier [17], em seu trabalho sobre sistemas lineares de inequações. Entretanto, somente em 1947 foi desenvolvido por George Dantzig [18] o método SIMPLEX, que foi o primeiro algoritmo para solução de problemas de programação linear. O SIMPLEX explora o fato de que a solução ótima do problema é uma solução básica viável, o que significa do ponto de vista geométrico, que ela é um ponto na extremidade de um politopo gerado pelas restrições. Assim, a partir de um vértice inicial, o método move-se para um vértice adjacente a este enquanto forem encontradas soluções de melhora.

Em [11] os autores propõem o uso do resolvedor de programação linear inteira como uma ferramenta caixa-preta para explorar eficientemente subespaços das soluções (vizinhanças) definidas e controladas por um algoritmo de *branching* externo simples. A vizinhança é obtida por meio da introdução, no modelo de programação linear inteira, de desigualdades inválidas, chamadas de cortes *local branching*. O ponto crítico em métodos de fixação de variáveis está relacionado à escolha das variáveis a serem fixadas em cada passo para a geração dos cortes. Para problemas mais complexos, só se pode resolvê-los após vários ciclos de fixação.

Em [19] é sugerida uma técnica de planos de corte globalmente válidos para algoritmos de busca 0-1 a partir de informações aprendidas por meio de propagação de restrições. Os cortes são gerados não somente quando há inviabilidade, mas também quando existe a necessidade de fixar uma variável em um valor 0 ou 1. Porém, os experimentos realizados pelos autores mostraram que isoladamente tais cortes não ajudam a PI, pois alguns dos planos de corte gerados são fracos.

No trabalho [10] é apresentado um resolvedor comercial que utiliza uma heurística de busca local para otimizar problemas binários lineares e não-lineares. Tal software, chamado de *LocalSolver*, alcança boas soluções em alguns problemas da biblioteca MIPLIB 2010 em casos classificados como difícil. Entretanto, devido à natureza comercial do produto, os algoritmos implementados são apenas superficialmente descritos pelos autores. Por outro lado, existe uma grande necessidade de programas resolvedores de alta qualidade, porém livres, de modo que o usuário possa modificá-los de acordo com suas necessidades a fim de obter melhores resultados de forma mais rápida.

Em [20] é desenvolvido um algoritmo, denominado pRINS, que explora técnicas de pré-processamento, procurando por um número ideal de fixações, visando a produzir sub-problemas de tamanho controlado. As variáveis fixadas são organizadas por um vetor de prioridades. Posteriormente, os problemas são criados e resolvidos de modo semelhante ao método *Variable Neighborhood Descent* até que um critério de parada seja satisfeito.

No trabalho [21] os autores propõem uma abordagem *Local Search* para problemas binários. O método desenvolvido considera a estrutura do problema como um grafo de conflitos e utiliza um procedimento de geração de vizinhos para percorrer as soluções usando cadeias de movimentos. O método visa a produção rápida de soluções viáveis, resultado que é comprovado nos experimentos realizados.

O COIN-OR-CBC [4] é um resolvedor de código aberto voltado à resolução de problemas lineares e inteiros. Ele apresenta bons resultados quando comparado aos demais de sua categoria; entretanto, em PLI, soluções ótimas são dificilmente encontradas em tempo de tomada de decisão para grande parte dos problemas reais.

IV. O Algoritmo HGVPRLB proposto para Problemas Binários

O algoritmo híbrido proposto, nomeado HGVPRLB, é um algoritmo **H**íbrido baseado em **G**RASP, **V**ND, **P**ropagação de **R**estrições e ***L**ocal **B**ranching*, destinado a resolver Problemas Lineares Binários. Seu pseudocódigo é apresentado no Algoritmo 1.

No Algoritmo 1 tem-se como dados de entrada o programa linear *LP*, o tempo limite *tempoLimite* de execução, o valor $\beta \in [0,1]$, o valor de γ definido em 0.01, o número máximo de iterações *max_iter* a ser passado para o Algoritmo 2, o tempo de execução *t_vnd* utilizado no Algoritmo 6, o valor θ utilizado para liberar as variáveis fixas com valores 0 e 1 quando o resolvedor encontrar dificuldade na relaxação das variáveis.

Algoritmo 1: HGVPRLB

Entrada: *LP, tempoLimite,* β, γ, *max_iter, t_vnd,* θ
Saída: $s^\star$

```
1  f* ← +∞;
2  t ← 0;
3  enquanto t < tempoLimite faça
4      s ← construaSolucao(LP, β, γ, max_iter, θ);
5      tempoVND ← min{tempoRestante, t_vnd};
6      s ← VND(LP, s, tempoVND);
7      se f(s) < f* então
8          s* ← s;
9          f* ← f(s*);
10     fim
11     Atualize t;
12 fim
13 Retorne s*;
```

Tal como um algoritmo clássico GRASP, há duas fases: uma construtiva e outra de refinamento da solução construída. Essas duas fases são aplicadas iterativamente até que o tempo limite seja atingido.

Na linha 4 do Algoritmo 1 uma solução é construída por meio do procedimento *construaSolucao(LP,*β, γ*,max_iter,*θ*)*, definido na Seção IV-A. Na linha 5 é calculado o menor valor entre o tempo restante e o tempo t_vnd. Este último é o tempo de execução do procedimento *VND(LP,s,tempoVND)*, acionado na linha 6 e descrito na Seção IV-B, onde a solução contruída é refinada. Na linha 7 é verificado se essa solução refinada é de qualidade superior à melhor solução $s^\star$ encontrada até então. Se a resposta for positiva, $s^\star$ é atualizada. Na linha 11 o tempo é atualizado.

Nas subseções seguintes o Algoritmo 1 é detalhado.

A. Fase Construtiva

O procedimento *construaSolucao(LP,*β, γ*,max_iter,*θ*)* tem como objetivo construir uma solução inicial para o Algoritmo HGVPRLB. Seu pseudocódigo é definido pelo Algoritmo 2.

Este procedimento recebe como parâmetros o problema linear *LP*, o valor β necessário para calcular a lista restrita de candidatos, o valor γ que define como candidatas somente variáveis cujo valor é maior ou igual a γ, o valor *max_iter*, que define o número máximo de iterações na fase de construção e o valor θ que define a porcentagem de variáveis que serão liberadas quando o resolvedor encontrar solução viável na relaxação.

Na linha 3 do Algoritmo 2 é acionado o resolvedor *CBC* aplicado ao problema *LP* relaxando as variáveis binárias, isto é, considerando-as no intervalo $[0, 1]$. Nas linhas 4 a 8 é verificado se o resolvedor não encontrou uma solução viável para problema, se isto ocorrer são liberadas tanto do problema *LP* quanto da solução s as θ variáveis anteriormente fixadas. Esta verificação se repete até se obter uma solução viável relaxada para o problema. A seguir, na linha 9, se foi encontrada uma solução viável, esta é atribuída ao vetor v. Na linha 10 é chamado o procedimento *ConstruaRCL(LP,v,*γ, β*)* definido na Subseção IV-A1, cujo objetivo é retornar o índice de uma variável que será fixada no valor 1, tanto no problema *LP* quanto na solução s do problema. Fixada essa variável $s_{j^\star}$ é chamado o procedimento *PropagacaodeRestricoes(LP,l,u,C)* descrito na Seção IV-A2. O objetivo desse procedimento é verificar inicialmente se existe solução viável fixando-se a variável $s_{j^\star}$ no valor 1. Se esta solução não existir, na linha 14 a fixação da variável $s_{j^\star}$ é liberada tanto no problema *LP* quanto na solução s. Caso ela exista, ele retornará também um conjunto I de variáveis $s_j \neq s_{j^\star}$ que devem ser fixadas em valores 0 ou 1 em função da fixação da variável $s_{j^\star}$. Essas fixações são realizadas na linha 19. Na linha 21 é verificado se a solução corrente tem qualidade superior à da melhor solução encontrada até o momento. Na determinação da solução corrente s as variáveis ainda não fixadas recebem o valor 0. As linhas 3 a 23 são executadas até o número máximo de iterações *max_iter*. Na linha 25 a melhor solução encontrada é retornada.

Algoritmo 2: construaSolucao

Entrada: *LP,* β, γ, *max_iter,* θ
Saída: $s^\star$

```
1  s ← s* ← ∅;
2  enquanto −−max_iter ≥ 0 faça
3      result ← CBCRelax(LP,v);
4      enquanto result ≠ LP_factivel faça
5          Selecione aleatoriamente θ% das variáveis fixas
           em LP e s;
6          Libere de LP e s as variáveis selecionadas;
7          result ← CBCRelax(LP,v);
8      fim
9      v ← solução do problema LP relaxado;
10     j* ← ConstruaRCL(LP, v, β, γ);
11     s_{j*} ← 1;
12     Fixe s_{j*} = 1 em LP;
13     I ← PropagacaodeRestricoes(LP,l,u,C);
14     se I = inviável então
15         Libere s_{j*} em s;
16         Libere s_{j*} em LP;
17     fim
18     senão
19         Fixe implicações I em s e em LP;
20     fim
21     se f(s) for melhor que f(s*) então
22         s* ← s;
23     fim
24 fim
25 Retorne s*;
```

1) Lista Restrita de Candidatos (RCL): O procedimento *ConstruaRCL(LP,v,*β, γ*)* tem por objetivo selecionar o índice de uma variável a ser fixada no valor 1. Ele é acionado após a chamada ao procedimento *CBCRelax(LP,v)*, o qual gera uma solução relaxada v para o problema LP. Este procedimento

é descrito no Algoritmo 3 e recebe como dados de entrada o problema *LP*, o vetor fracionário v que possui o valor da relaxação das variáveis do problema, o valor de β e o valor γ.

Algoritmo 3: ConstruaRCL

Entrada: *LP, v, β, γ*
Saída: $j^\star$
1 $A \leftarrow \{(j, s_j) : s_j \geq \gamma$ e j não fixado em *LP*$\}$;
2 *max* $\leftarrow \max_{\forall j \in A}\{s_j\}$;
3 *min* $\leftarrow \min_{\forall j \in A}\{s_j\}$;
4 *RCL* $\leftarrow \{j : j \in A$ e $s_j \geq$ *max* - $\beta\times$ (*max* - *min*)$\}$;
5 $j^\star \leftarrow$ Elemento j selecionado aleatoriamente da *RCL*;
6 **Retorne** j^*;

Na linha 1 do Algoritmo 3 A recebe um conjunto de índices de variáveis não fixadas em *LP*. Destas são selecionadas as variáveis que possuem valor de relaxação maior ou igual a γ. Essas variáveis formam uma lista de variáveis candidatas a serem incorporadas a uma lista restrita de candidatos, chamada RCL (das iniciais em inglês de *Restricted Candidate List*) escolhidas de forma a se obter uma lista com diversidade e de boa qualidade. Uma vez que a escolha de β influencia o tamanho da RCL e, consequentemente, a qualidade das soluções obtidas nesta fase, à medida que o tamanho da RCL aumenta, a qualidade da solução piora; entretanto, em contrapartida, é aumentada a diversidade das soluções geradas. A lista restrita de candidatos *RCL* é formada pelos índices j das variáveis s_j que satisfazem a Equação (4).

$$s_j \geq max - \beta \times (max - min) \tag{4}$$

sendo *max* e *min*, o maior e menor valor da relaxação das variáveis, respectivamente, e obtidos nas linhas 2 e 3. Na linha 4 o vetor RCL recebe os índices das variáveis que satisfazem a Equação 4. A partir desta lista de candidatos, na linha 5 é selecionado de forma aleatória um índice de uma variável que será o retorno do algoritmo.

2) Programação por Restrições: O procedimento *PropagacaodeRestricoes(LP,l,u,C)* possui a finalidade de verificar se existe solução viável fixando uma variável $s_{j^\star}$ no valor 1. Se essa solução existir, ele retornará um conjunto I de fixações das demais variáveis $s_j \neq s_{j^\star}$ nos valores 0 ou 1. Caso contrário, ele retornará que fixando-se a variável $s_{j^\star}$ no valor 1 a solução s será inviável.

Para a adequação a este modelo de propagação por restrições, cada restrição i é reescrita na forma da Eq. (5):

$$\sum_{j \in N} a_{ij} s_j \leq b_i \tag{5}$$

em que N é o conjunto de variáveis.

Assim, seja a medida de inviabilidade U_{ij} determinada pela Eq. (6):

$$U_{ij^\star} = b_i - \left(\sum_{j \in N_i^+, j \neq j^\star} |a_{ij}|\, l_j + \sum_{j \in N_i^-, j \neq j^\star} |a_{ij}|\, u_j \right) \tag{6}$$

em que $N_i^+ = \{j \in N : a_{ij} \geqslant 0\}$ e $N_i^- = \{j \in N : a_{ij} < 0\}$.

Em cada restrição i, as variáveis s_j a ela pertencentes possuem limites superior e inferior, respectivamente, l_j e u_j, definidos como parâmetros. Se a j-ésima variável é fixada, $l_j = u_j = s_{j^\star}$. Caso contrário, $l_j = 0$ e $u_j = 1$.

O procedimento 5, *avaliaInviabilidade(LP,l,u,c,j*)*, usa a medida U_{ij} para verificar se a variável s_j pode ou não ser fixada nos valores 0 ou 1. Se não puder ser fixada por existência de inviabilidade, ele retorna a existência de conflito. Caso contrário, ele pode retornar: 1) O valor binário que a variável s_j deve ser fixada, ou então, 2) a indefinição do valor a ser fixado para essa variável.

Assim este procedimento retorna o par *status,limite* sendo que *status* assume o valor CONFLITO e *limite* assume o valor NULO quando s_j não puder assumir um valor binário. Por outro lado, *status* assume o valor FIXA quando s_j pode assumir um valor binário e *limite* assume o valor 0 ou 1. Finalmente *status* assume o valor SEM_IMPLICACOES e *limite* assume o valor NULO quando há indefinição do valor a ser fixado para essa variável.

Os Algoritmos 4 e 5 apresentam os pseudocódigos baseados na implementação do algoritmo de [19] para detectar conflitos e implicações.

No Algoritmo 4 são dados de entrada o problema *LP*, os limites superior e inferior definidos anteriormente como l e u, e um conjunto de restrições C em cada uma delas aparece o índice j da variável s_j que foi fixada. As linhas 1 e 2 recebem, a princípio, um conjunto vazio de fixações, e o *status* definido como SEM_IMPLICACOES. Na linha 4 é atribuída o valor *falso* a uma variável, chamada de *novas_implicacoes*. C é o conjunto das restrições em que a $j^\star$-ésima variável fixada aparece.

Nas linhas 5 a 16 é realizado um laço iterativo em cada uma dessas restrições.

Nas linhas 6 a 16, para cada uma das variáveis desta restrição, é avaliado se ela pode ser fixada em algum valor de acordo com as fixações anteriores. Para isso é chamado o método *avaliaInviabilidade(LP,l,u,c,j*)* apresentado no Algoritmo 5. Caso o resultado retornado seja igual a FIXA, esta variável e seu limite são adicionados ao conjunto I, e a variável *novas_implicacoes* recebe *verdadeiro*. Se o *status* for igual a CONFLITO, este valor é retornado pela função.

Na linha 17, se não houve fixações, a função retorna o *status* SEM_IMPLICACOES.

Nas linhas 20 a 25 para toda fixação realizada anteriormente adicionada ao conjunto I, os limites l_j e l_u dessas variáveis são atualizados. Cada restrição em que essas variáveis aparecem são adicionadas ao conjunto C.

Nas linhas 26 a 30 são excluídas as restrições pertencentes a C em que todas as variáveis já foram fixadas. Esta função é

Algoritmo 4: PropagacaodeRestricoes

```
   Entrada: LP, l, u, C
   Saída: (status, I)
1  status ⟵ SEM_IMPLICACOES;
2  I ⟵ ∅;
3  repita
4      novas_implicacoes ← falso;
5      para todo restrição c em C faça
6          para todo índice j* de variável não fixada em c faça
7              (limite,status) ⟵ avaliaInviabilidade(LP,l,u,c,j*);
8              se status = FIXA então
9                  I ⟵ I ∪ {(j*, limite)};
10                 novas_implicacoes ← verdadeiro;
11             fim
12             senão se status = CONFLITO então
13                 return (CONFLITO, I);
14             fim
15         fim
16     fim
17     se |I| = ∅ então
18         return (SEM_IMPLICACOES, I);
19     fim
20     para todo (j, limite) ∈ I faça
21         l_j ← u_j ← limite;
22         para todo
           c em C que contém uma ou mais variáveis fixadas
           faça
23             C ⟵ C ∪ {c};
24         fim
25     fim
26     para todo c em C faça
27         se todas as variáveis da restrição c estão fixadas então
28             C ⟵ C \ {c};
29         fim
30     fim
31 até (C ≠ 0) e novas_implicacoes = verdadeiro;
32 Retorne (status, I);
```

Algoritmo 5: avaliaInviabilidade

```
   Entrada: LP, l, u, i, j*
   Saída: (status,limite)
1  U_ij* ⟵ Calculo_U;
2  se j* ∈ N_i^+ então
3      se U_ij* < 0 então
4          limite ⟵ NULO;
5          status ⟵ CONFLITO;
6      fim
7      senão se a_ij* > U_ij* então
8          limite ⟵ 0;
9          status ⟵ FIXA;
10     fim
11     senão
12         limite ⟵ NULO;
13         status ⟵ SEM_IMPLICACOES;
14     fim
15 fim
16 se U_ij* ∈ N_i^- então
17     se a_ij* > U_ij* então
18         limite ⟵ NULO;
19         status ⟵ CONFLITO;
20     fim
21     senão se U_ij* < 0 então
22         limite ⟵ 1;
23         status ⟵ FIXA;
24     fim
25     senão
26         limite ⟵ NULO;
27         status ⟵ SEM_IMPLICACOES;
28     fim
29 fim
30 Retorne (status,limite);
```

repetida até que o conjunto de restrições C seja igual a vazio, ou não houver novas implicações.

No final, linha 32 do Algoritmo 4, o procedimento *PropagacaodeRestricoes(LP,l,u,C)* retorna o par *status* e o conjunto I de fixações realizadas.

O Algoritmo 5 tem como dados de entrada o problema binário *LP*, os limites superior e inferior l e u de cada variável, a restrição i onde está contida a variável $s_{j^\star}$, e o índice $j^\star$ dessa variável a ser verificada.

Na linha 1 do Algoritmo 5 a inviabilidade $U_{ij^\star}$ é calculada de acordo com a Equação (6).

Nas linhas 2 a 15, se a variável $s_{j^\star}$ possuir um coeficiente positivo, ela é analisada da seguinte forma: se a inviabilidade $U_{ij^\star}$ for menor que zero, o *status* recebe a definição CONFLITO e o *limite* recebe NULO. Se o coeficiente da variável $s_{j^\star}$ for maior que o valor $U_{ij^\star}$, o *status* é definido como FIXA, e o *limite* será 0. Se nenhuma das condições anteriores forem válidas, o *status* será definido como SEM_IMPLICACOES e o *limite* será NULO.

Nas linhas 16 a 29, se a variável analisada possuir um coeficiente negativo, ela é analisada de forma análoga ao caso anterior, sendo que: se o coeficiente da variável $s_{j^\star}$ for maior que o valor $U_{ij^\star}$, o *status* é definido como CONFLITO, e o *limite* como NULO. Se o valor $U_{ij^\star}$ for menor que zero, o *status* recebe a definição FIXA e o *limite* recebe o valor 1. Se nenhuma das condições anteriores forem válidas, o *status* será definido como SEM_IMPLICACOES e o *limite* será NULO.

Na linha 30 é retornado o par *status* e *limite*.

B. Refinamento - VND

Nesta fase, uma solução inicial é refinada diversas vezes durante um tempo determinado. Tal fase é controlada pelo procedimento *Variable Neighborhood Descent* – VND [15]. VND é um método de busca local que consiste em explorar o espaço de soluções por meio de trocas sistemáticas de estruturas de vizinhança previamente ordenadas.

O procedimento utiliza a primeira estrutura de vizinhança visando a melhora da solução corrente. Quando não é mais possível essa melhora na vizinhança corrente, o método inicia sua busca na próxima vizinhança. O procedimento retorna à primeira vizinhança quando uma melhor solução é encontrada. O método se encerra após aplicar todas as estruturas de vizinhança sem conseguir melhorar a solução corrente. Neste trabalho consideramos um conjunto de estruturas de vizinhança, cada qual formada pela fixação de um certo percentual de variáveis da solução oriunda da fase de construção que possuem valor igual a 1, variando-se em 5% o número

de variáveis a serem fixadas no intervalo 95% a 70%. Assim, a primeira vizinhança contém 95% das variáveis fixadas no valor 1, a segunda contém 90% e assim por diante. Para exemplificar, sejam $N1 = \{j : s_j = 1\}$ na fase de construção e $n_1 = |N1|$. Se estivermos na primeira vizinhança, isto é, com 95% das variáveis fixadas no valor 1, então serão adicionadas duas restrições ao problema LP, dadas pelas equações 7:

$$\lceil 0.90 \times n_1 \rceil \leq \sum_{j \in N1} s_j \leq \lceil 0.95 \times n_1 \rceil \quad (7)$$

Desta maneira, a princípio se busca apertar a solução fixando um número maior de variáveis, e à medida que o resolvedor encontra dificuldade em encontrar uma solução a partir desses limites, eles são relaxados e se inicia uma nova busca com um menor número de variáveis fixadas. Assim, procuramos fixar um número grande de variáveis, mas com um grau de liberdade razoável, possibilitando ao resolvedor encontrar melhores soluções.

O Algoritmo 6 mostra o pseudocódigo do procedimento VND usado na fase de busca local do algoritmo HGVPRLB implementado para gerar os cortes *Local Branching*.

Algoritmo 6: VND

Entrada: *LP, s, tempoVND*
Saída: $s^\star$

```
1  nLvizinhanca[] ← {0.95, 0.90, 0.85, 0.80, 0.75, 0.70};
2  t ← 0;
3  k ← 0;
4  repita
5      n1 ← número de variáveis de s fixadas no valor 1;
6      αl ← nLvizinhanca[k] ;
7      αu ← nLvizinhanca[k + 1];
8      coef_sup ← ⌈n1 × αl⌉;
9      coef_inf ← ⌈n1 × αu⌉;
10     tempoLB ← tempoVND − t;
11     s ← LocalBranching (LP, tempoLB, coef_sup, coef_inf, s) ;
12     se f(s) é melhor que f(s') então
13         s* ← s;
14         k ← 0;
15     fim
16     senão
17         k++;
18     fim
19 até t > tempoVND;
20 Retorne s*;
```

No Algoritmo 6 são dados de entrada o problema linear *LP*, a solução *s*, e o tempo de execução *tempoVND*.

Na linha 1 é definido o vetor que contém os intervalos da estrutura de vizinhança, definida como *nLvizinhanca*[].

Nas linhas 2 e 3 são inicializados o tempo de execução do procedimento e o valor de *k* é setado para iniciar na primeira estrutura de vizinhança.

Na linha 5 a variável *n* recebe o número de variáveis que na solução possuem valor igual a 1. Estas serão as variáveis utilizadas para os cortes *Local Branching*. Estes cortes possuem limites inferior e superior que serão definidos a partir de valores da estrutura de vizinhança corrente.

Na linha 6 α_l recebe o valor da posição do vetor *nLvizinhanca*[], referente à posição atual de *k*.

Na linha 7 α_u recebe o valor da posição do vetor *nLvizinhanca*[], referente à posição *k + 1*.

Nas linhas 8 e 9 são calculados os valores de $coef_{sup}$ e $coef_{inf}$, de acordo com o valor de n, α_l e α_u, respectivamente. Os referidos coeficientes assumem o valor da soma de todas as variáveis que possuem valor 1 na solução, multiplicados por α_l e α_u, respectivamente.

Na linha 10 é calculado o tempo disponível para a execução do procedimento *LocalBranching (LP,tempoLB,*$coef_{sup}$*,*$coef_{inf}$*,s)*, apresentado na Seção IV-C. Assim, a variável *tempoLB* recebe o tempo de entrada *tempoVND* diminuído do tempo corrente *t*. Este tempo pode ser gasto totalmente ou parcialmente pelo procedimento *LocalBranching (LP,tempoLB,*$coef_{sup}$*,*$coef_{inf}$*,s)*.

Na linha 12 é verificado se a solução melhorou, e caso afirmativo, a nova solução é armazenada, e a variável *k* recebe o valor 0. Se isto não ocorrer, na linha 17 o valor de *k* é incrementado em uma unidade.

As linhas 4 a 19 são repetidas enquanto houver tempo. Na linha 20 é retornada a melhor solução encontrada.

C. Local Branching

O procedimento aqui proposto é baseado no trabalho [11], que propõe o uso do resolvedor de programação linear inteira como uma ferramenta caixa-preta para explorar eficientemente subespaços das soluções (vizinhanças). Neste procedimento foi utilizado como resolvedor o CBC.

Para mostrar o funcionamento dos cortes *local branching*, seja a solução corrente s com valores binários já definidos para suas variáveis s_j. Então, para cada restrição i não satisfeita é introduzida uma variável, com coeficiente adequado, de sorte a torná-la satisfeita. Por exemplo, se a restrição for $2s_1+s_2 \leq 1$, e s_1 e s_2 estão assumindo valor 1 na solução corrente, então à esta restrição será adicionada a variável s_3 com coeficiente -2, de modo a torná-la satisfeita. Assim, ela passa a ser escrita como $2s_1 + s_2 - 2s_3 \leq 1$.

Genericamente, é subtraído do lado esquerdo de cada restrição i não satisfeita, do tipo $\leq$, dada por $\sum_{j=1}^{n} a_{ij}s_j \leq b_i$, uma variável s_{j+1} cujo coeficiente $a_{i,n+i}$ é definido pela Equação (8):

$$a_{i,n+i} = b_i - \sum_{j=1}^{n} a_{ij}s_j \quad (8)$$

resultando, assim, na nova restrição i válida, dada pela Equação (9):

$$\sum_{j=1}^{n} a_{ij}s_j - a_{i,n+i}s_{n+i} \leq b_i \quad (9)$$

De forma análoga, a cada restrição i não satisfeita do tipo $\geq$ é adicionada ao lado esquerdo da restrição uma variável s_{n+i}, cujo coeficiente $a_{i,n+i}$ é dado pela Equação (10):

$$a_{i,n+i} = \sum_{j=1}^{n} a_{ij} s_j - b_i \quad (10)$$

No caso de a restrição i ser do tipo $=$, então é adicionada ou subtraída uma variável s_{j+1}, conforme o caso, de sorte a transformá-la em uma restrição satisfeita.

Tais variáveis também são acrescentadas à função objetivo do problema e recebem coeficientes positivos altos em relação aos demais.

Tornadas satisfeitas todas as restrições, a seguir são criadas duas novas restrições, definidas pela Equação (11). Essas restrições são adicionadas ao modelo original de programação linear inteiro e cumprem a função do *local branching*.

$$\lceil \alpha_l \times n_1 \rceil \leq \sum_{j \in N1} s_j \leq \lceil \alpha_u \times n_1 \rceil \quad (11)$$

Na Equação (11), α_l e α_l são valores contíguos pertencentes ao conjunto de vizinhanças *nLvizinhanca* = $\{0.95, 0.90, \cdots, 0.75, 0.70\}$, $N1 = \{j : s_j = 1\}$ e n_1 é o número de variáveis da solução corrente que assumem o valor binário 1, incluindo as variáveis que foram adicionadas às restrições anteriormente não satisfeitas.

A ideia do método é que a k-ésima vizinhança *nLvizinhanca* de uma solução s deve ser "suficientemente pequena" para ser otimizada em um curto tempo de processamento, mas "grande o bastante" para conter soluções melhores do que s, sendo os valores de k controlados pela heurística VND, com $k \in$ *nLvizinhanca*.

O Algoritmo 7 apresenta o pseudocódigo do procedimento de busca local *LocalBranching* utilizado para resolver o problema binário descrito.

Algoritmo 7: *LocalBranching*

Entrada: *LP, t, coef$_{sup}$, coef$_{inf}$, s*
Saída: $s^\star$

1 **se** *solução é inviável* **então**
2 | *LP* ← variáveis de folga;
3 **fim**
4 *LP* ← adiciona cortes *local branching* considerando *coef$_{inf}$* ;
5 *LP* ← adiciona cortes *local branching* considerando *coef$_{sup}$* ;
6 *LP* ← adiciona variáveis de folga com valor alto de coeficiente;
7 define limite de tempo *t* para otimização;
8 $s^\star \leftarrow$ otimiza();
9 **Retorne** $s^\star$;

O Algoritmo 7 recebe como dados de entrada o problema linear *LP* e um tempo *tempoLB* de execução. No que se refere à linha 1 é verificada se a solução é inviável, e caso afirmativo, na linha 2 são acrescidas as variáveis de folga às restrições do problema que não foram obedecidas, de forma que todas as restrições passem a ser respeitadas.

A partir das variáveis de folga são acrescidas ao problema duas novas restrições nas linhas 4 e 5, os chamados cortes *Local Branching* de [11]. Por se tratarem de problemas de minimização, com o objetivo de as variáveis de folga possuirem valor igual a 0, na linha 6, estas são adicionadas à função objetivo do problema com valores de coeficientes altos.

Na linha 7 é definido o tempo de otimização. Na linha 8 o problema é otimizado, e finalmente, na linha 9, a solução modificada é passada ao resolvedor *CBC* para que ele retorne uma solução binária. Posteriormente, se não houver dificuldade de ser encontrada a solução, o procedimento retorna a solução obtida pelo resolvedor *CBC*.

V. Experimentos Computacionais

O algoritmo HGVPRLB proposto, descrito pelo Algoritmo 1, foi escrito na linguagem C utilizando-se para a leitura de problemas-teste o resolvedor de código aberto COIN-OR.

O código foi compilado no GCC, versão 4.6.3. Os experimentos foram realizados em um computador 3,4 GHz Intel Core i7-3770R com 16 GB de RAM executando em um sistema operacional openSUSE 12.3 Linux.

Para os experimentos foram utilizados 32 problemas binários da biblioteca MIPLIB 2010 [16]. A biblioteca MIPLIB foi criada em 1992 e obteve sua última atualização em 2010. Esta biblioteca foi montada a partir da coletânia de problemas reais obtidos da indústria ou da comunidade acadêmica. Os problemas-teste contidos nessa biblioteca têm sido largamente usados para comparar o desempenho de resolvedores de programação inteira.

O algoritmo HGVPRLB foi testado com relação à sua capacidade de encontrar uma solução viável de qualidade variando-se o tempo de processamento. De forma a poder compará-lo com outros métodos da literatura, que especificam os resultados alcançados em 60 e 300 segundos, foram utilizados esses dois valores de tempo do parâmetro *tempoLimite* como referência de comparação.

Após uma bateria preliminar de testes os parâmetros do algoritmo HGVPRLB foram fixados nos seguintes valores: $\beta = 0,3$, $\gamma = 0,01$ e $\theta = 0,3$. O parâmetro t_vnd assumiu como valor a metade do tempo total de execução, isto é, t_vnd = *tempoLimite*/2. Assim, quando o critério de parada era 60 segundos, então t_vnd assumia o valor 30 segundos, e quando o tempo total de execução do algoritmo era 300 segundos, t_vnd = 150 segundos. O valor *max_iter* inicia com 10 quando o temço limite é 60 segundos e inicia com 20 quando o tempo limite é 300 segundos, e a cada iteração de busca local ele é multiplicado por 4.

Os resultados dos experimentos realizados foram comparados com aqueles de dois dos melhores resolvedores de código aberto para programação inteira: CBC e GLPK, e também com o método BPLS de [21]. Em todas as comparações o critério de parada foi o mesmo.

Nas tabelas I e II são apresentados os valores da função objetivo alcançados por cada método de solução em 60 e 300 segundos, respectivamente. A penúltima linha de cada tabela indica o número de soluções viáveis encontradas por

Tabela I
VALORES DAS MELHORES SOLUÇÕES ENCONTRADAS EM 60 SEGUNDOS

Prob.-Teste	CBC	GLPK	BPLS	HGVPRLB	
				Melhor	Média
acc-tight5					
air04	**56137**	57830	x	56138	56549,77
bab5					
bley xl1					
bnatt350					
cov1075	**20**	**20**	x	**20**	20,03
eil33.2	993,61	**934**	x	987,67	988,00
eilB101	1529,89	1374,96	x	**1326,05**	1343,07
ex9					
iis-100-0-cov	**29**	30	x	**29**	**29**
iis-bupa-cov	40	40	x	**36**	36,23
iis-pima-cov	36	36	x	**33**	33,93
m100n500k4r1	**-24**	**-24**	x	**-24**	-23,70
macrophage	392	495	x	**386**	468,17
mine-166-5	**-553252300**	-531532749,40	x	-3459148,24	680296,82
mine-90-10			x	**-13321729,51**	719767,38
mspp16					
n3div36	**135000**		x	165200	172606,67
n3seq24			x	**63000**	35733,33
neos-1109824	**378**	**378**		**378**	317,10
neos-1337307	**-202191**	-201708		-202038	-6734,60
neos18	**16**	22	x	**16**	16,07
neos-849702					
netdiversion					
ns1688347					
opm2-z7-s2	-8239	-9430	x	**-9841**	-1038,60
reblock67	**-34388335**		x	-2423910,13	-449419,31
rmine6	**-456,94**	-455	x	-454,99	-74,61
sp98ic	451003580	520753842,70	x	**450568218,08**	454242165,50
tanglegram1			x		
tanglegram2	515	**443**	x	**443**	**443**
vpphard					
#Sols Viáveis	19	17	20	21	21
#Melh. Sols	11	5		14	2

cada método, enquanto a última linha indica a quantidade de melhores soluções produzidas por cada método no conjunto de problemas-teste. As linhas em branco indicam que o método correspondente não encontrou solução viável no tempo estipulado. Em cada uma dessas tabelas, a coluna "Prob.-teste" indica o problema-teste em análise, as colunas "CBC" e "GLPK" os resolvedores homônimos, a coluna "BPLS" o algoritmo homônimo de [21]. A coluna "HGVPRLB" é subdividida em duas colunas, sendo que na primeira é reportado o melhor resultado encontrado em 30 execuções desse método, enquanto na segunda consta o resultado médio das execuções. Como em [21] não é relatado o valor da função objetivo, e tão somente a existência ou não de solução viável, então em cada célula da coluna relativa a esse método é assinalado com a letra 'x' a existência de uma solução viável, deixando a célula em branco caso o método não consiga alcançar uma solução viável no tempo estipulado. Na penúltima linha de cada tabela é totalizado o número de soluções viáveis obtidas por cada método, enquanto na última linha é contabilizado o número de melhores soluções alcançadas por cada método.

Pela Tabela I observa-se que o algoritmo proposto encontrou o maior número de soluções viáveis, no caso, 21, o que representa uma solução a mais que o BPLS, duas soluções a mais que o método CBC e 4 soluções a mais que o GLPK. Em relação à qualidade das soluções, observa-se que o CBC detém 11 melhores resultados, enquanto o GLPK detém 5 e o algoritmo proposto, 14.

Tabela II
VALORES DAS MELHORES SOLUÇÕES ENCONTRADAS EM 300 SEGUNDOS

Prob.-Teste	CBC	GLPK	BPLS	HGVPRLB	
				Melhor	Média
acc-tight5					
air04	**56137**	56143	x	**56137**	56408,97
bab5	-103368,24			**-105984,96**	-78130,51
bley xl1					
bnatt350					
cov1075	**20**	**20**	x	**20**	**20**
eil33.2	**934**	**934**	x	**934**	934,01
eilB101	1227,71	1310,27	x	**1216,92**	1270,05
ex9					32,40
iis-100-0-cov	**29**	**29**	x	**29**	**29**
iis-bupa-cov	37	39	x	**36**	36,10
iis-pima-cov	34	36	x	**33**	33,87
m100n500k4r1	**-24**	**-24**	x	**-24**	-23,70
macrophage	**378**	481	x	386	393,10
mine-166-5	**-566395707,87**	-531532749,40	x	-3459148,24	400413,19
mine-90-10	-783742624,05		x	**-13321729,51**	-287284,39
mspp16					
n3div36	**131400**	179600	x	140400	164386,67
n3seq24			x	**63000**	35226,67
neos-1109824	**378**	**378**	x	**378**	315,13
neos-1337307	**-202319**	-201708		-202038	-6734,60
neos18	**16**	20	x	**16**	**16**
neos-849702					
netdiversion					
ns1688347					
opm2-z7-s2	-10145	-10205	x	**-10257**	-9996
reblock67	-34630648	**-15541889,16**	x	-2423910,13	-449419,31
rmine6	**-457,01**	-455	x	-454,99	-74,61
sp98ic	451968910	**487333102,70**	x	449213315,20	450379464,59
tanglegram1			x		
tanglegram2	**443**	**443**	x	**443**	**443**
vpphard	**66**				
#Sols Viáveis	22	19	21	22	22
#Melh. Sols	14	8		15	4

Por outro lado, pela Tabela II observa-se que o algoritmo proposto foi capaz de melhorar a qualidade das soluções obtidas em relação à Tabela I, alcançando, assim como o CBC, o maior número de soluções viáveis encontradas, no caso, 22. Esse valor representa duas soluções a mais que o método BPLS, e 3 a mais que o GLPK. Em relação à qualidade das soluções observa-se que o CBC detém 14 melhores resultados, enquanto o GLPK detém 8 e o método proposto, 15.

VI. CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho foi proposto um algoritmo híbrido, denominado HGVPRLB, para resolver problemas de programação linear binária, sendo ele avaliado com relação à capacidade de alcançar uma solução viável de qualidade.

Experimentos computacionais realizados em um conjunto de problemas-teste da biblioteca MIPLIB 2010 mostraram que o algoritmo proposto é superior a outros métodos de solução de problemas binários da literatura. Quando o tempo de execução é limitado a 60 segundos, o algoritmo proposto encontra um maior número de soluções viáveis do que o algoritmo BPLS de [21] e do que dois dos melhores resolvedores de código aberto de programação inteira disponíveis: CBC e GLPK. Além disso, o algoritmo HGVPRLB produz soluções de melhor qualidade em relação a esses métodos.

Por outro lado, quando o tempo de processamento é aumentado, o algoritmo HGVPRLB consegue um número ainda maior de soluções viáveis que os encontrados pelo método BPLS e também pelo resolvedor GLPK, e empata no número de soluções viáveis encontradas pelo resolvedor CBC. Já com relação à qualidade da solução final, observa-se que ela é melhorada, sendo que ao comparar os resultados obtidos com os de outros métodos, o HGVPRLB é o que apresenta o maior número de melhores soluções para o conjunto de problemas-teste.

Esses fatos mostram a superioridade do algoritmo proposto em relação a outros da literatura no que diz respeito tanto com relação à capacidade de encontrar uma solução viável em um tempo restrito quanto em relação à qualidade da solução final produzida.

AGRADECIMENTOS

Os autores agradecem às agências de fomento FAPEMIG, CNPq e CAPES, bem como ao Programa de Pós-graduação em Ciência da Computação da Universidade Federal de Ouro Preto pelo apoio ao desenvolvimento deste trabalho.

REFERÊNCIAS

[1] R. S. Garfinkel and G. L. Nemhauser, *Integer programming*. Wiley New York, 1972, vol. 4.

[2] R. Takahashi and A. Gaspar-Cunha, *Manual de Computação Evolutiva e Metaheurítica*. Imprensa da Universidade de Coimbra, 2012, vol. 1, ch. Introdução, pp. 1–21.

[3] IBM, *CPLEX - IBM CPLEX Optimizer*, http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/, 2015.

[4] C.-O. Foundation, *CBC - COIN-OR Branch-and-Cut MIP Solver*, https://projects.coin-or.org/Cbc, 2015.

[5] SYMPHONY, *SYMPHONY*, https://projects.coin-or.org/SYMPHONY, 2015.

[6] MINTO, *MINTO - Mixed INTeger Optimizer*, http://coral.ie.lehigh.edu/minto/, 2015.

[7] SCIP, *Solving Constraint Integer Programs*, http://scip.zib.de/, 2015.

[8] GLPK, *GLPK - GNU Project*, http://www.gnu.org/software/glpk/, 2015.

[9] lp solve, *lp solve reference guide*, http://lpsolve.sourceforge.net/5.5/, 2015.

[10] T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua, "Localsolver 1. x: a black-box local-search solver for 0-1 programming," *4OR*, vol. 9, no. 3, pp. 299–316, 2011.

[11] M. Fischetti and A. Lodi, "Local branching," *Mathematical programming*, vol. 98, no. 1-3, pp. 23–47, 2003.

[12] A. M. Geoffrion, *Lagrangean relaxation for integer programming*. Springer, 1974.

[13] K. R. Apt, "The essence of constraint propagation," *Theoretical computer science*, vol. 221, no. 1, pp. 179–210, 1999.

[14] T. A. Feo and M. G. Resende, "Greedy randomized adaptive search procedures," *Journal of global optimization*, vol. 6, no. 2, pp. 109–133, 1995.

[15] N. Mladenović and P. Hansen, "Variable neighborhood search," *Computers & Operations Research*, vol. 24, no. 11, pp. 1097–1100, 1997.

[16] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz *et al.*, "Miplib 2010," *Mathematical Programming Computation*, vol. 3, no. 2, pp. 103–163, 2011.

[17] J. Fourier, "Second extrait," *Oeuvres*, pp. 325–328, 1890.

[18] G. B. Dantzig, *Activity Analysis of Production and Allocation*. New York: John Wiley & Sons, 1951, ch. Maximization of a Linear Function of Variables Subject to Linear Inequalities, pp. 339–347.

[19] T. Sandholm and R. Shields, "Nogood learning for mixed integer programming," in *Workshop on Hybrid Methods and Branching Rules in Combinatorial Optimization*, Montréal, 2006, disponível em http://users.encs.concordia.ca/ chvatal/sandholm2.pdf.

[20] T. M. Gomes, H. G. Santos, and M. J. F. Souza, "A pre-processing aware rins based mip heuristic," *Lecture Notes in Computer Science*, vol. 7919, pp. 1–11, 2013, in: BLESA, M. J.; BLUM, C.; FESTA, P.; ROLI, A.; SAMPELS, M.. (Org.). Proceedings of the 8th International Workshop on Hybrid Metaheuristics.

[21] S. S. Brito, H. G. Santos, and B. H. M. Santos, "A local search approach for binary programming: Feasibility search," *Lecture Notes in Computer Science*, vol. 8457, pp. 45–55, 2014, proceedings of the Hybrid Metaheuristics 2014.