Primeiros passos para resolver problemas de MINLP com Muriqui

Wendel Melo
Faculdade de Computação
Universidade Federal de Uberlândia
wendelmelo@ufu.br

Marcia Fampa
Instituto de Matemática e COPPE
Universidade Federal do Rio de Janeiro
fampa@cos.ufrj.br

Fernanda Raupp
Laboratório Nacional de Computação Científica
fernanda@lncc.br

RESUMO

Muriqui Optimizer é um software para a resolução de problemas de Programação Não Linear Inteira Mista (MINLP). Muriqui inclui um executável para ser utilizado com os sistemas AMPL e/ou GAMS e uma poderosa biblioteca C++ para a integração em softwares diversos. O principal diferencial de Muriqui é que o mesmo traz a possibilidade de uso dos principais algoritmos de MINLP da literatura, com uma série de parâmetros customizáveis. Este texto traz instruções iniciais para obtenção, instalação e uso de Muriqui .

1 O que é o Muriqui Optimizer

Muriqui Optimizer é um resolvedor de problemas convexos de Programação Não Linear Inteira Mista (MINLP). Algebricamente, Muriqui Optimizer resolve problemas no formato:

$$\begin{array}{lllll} (P) & z := \min\limits_{x} & f(x) + \frac{1}{2}x'Q_0x + c'x + d, & & \\ & \text{sujeito a:} \quad l_{c_i} \leq & g_i(x) + \frac{1}{2}x'Q_ix + a_ix & \leq u_{c_i}, & i = 1, \ldots, m, \\ & l_x \leq & x & \leq u_x, & \\ & & x_j \in \mathbb{Z}, \text{ para } j \in \mathcal{I}, & & \end{array} \tag{1}$$

Observe que x denota um vetor com as variáveis de decisão. Os vetores l_x e u_x trazem limites inferiores e superiores para x, e $\mathcal{I}$ é o conjunto de índices de variáveis restritas a valores inteiros. Com relação à função objetivo, $f(x)$ representa um termo não linear, Q_0 é uma matriz simétrica utilizada para descrever termos quadráticos, c é um vetor coluna usado para escrever termos lineares e d é um termo constante. Para cada restrição i, $g_i(x)$

representa um termo não linear, Q_i é uma matriz simétrica utilizada para descrever termos quadráticos, a_i é um vetor linha usado para descrever termos lineares e l_{c_i} e u_{c_i} são limites inferior e superior, respectivamente.

Obviamente, termos lineares e quadráticos podem vir a ser representados nas funções $f(x)$ e $g(x)$ em vez de seus respectivos termos matriciais. No entanto, a possível separação de termos lineares, quadráticos e não-lineares genéricos visa possibilitar um melhor aproveitamento das características do problema (P). Ademais, alguns solvers de Programação Não Linear (PNL) podem tomar proveito dessa decomposição das expressões para resolverem relaxações do problema (P) mais rapidamente.

Apontamos que alguns dos elementos de P podem estar "ausentes". Por exemplo, algumas componentes dos vetores l_c ou l_x podem ser definidos como $-\infty$, algumas componentes dos vetores u_c ou u_x podem ser definidos como ∞, algumas das funções $g_i(x)$ ou $f(x)$ podem ser nulas, bem como matrizes Q_i e vetores c e a_i.

Muriqui tem sido desenvolvido para a resolução de problemas convexos. No entanto, problemas não convexos também podem ser abordados, sem a garantia de obtenção de solução ótima. Idealmente, as funções $f(x)$ e $g_i(x)$ devem ser duplamente diferenciáveis e permitirem avaliações em qualquer ponto x que satisfaça as restrições de caixa (1), mas pode ser possível, em alguns casos, contornar esse último requisito com aproximações para as derivadas.

Muriqui traz consigo a ideia de que uma configuração genérica pode não ser a melhor escolha em uma aplicação específica. Assim, uma característica notável de Muriqui é o grande leque de opções customizáveis disponíveis ao utilizador. Desse modo, usuários tem uma ampla gama de escolhas de aplicação que vão desde a escolha do algoritmo a ser aplicado, dos parâmetros desses algoritmos e dos solvers utilizados na resolução de subproblemas. Sob essa perspectiva, usuários são incentivados a transpassar os ajustes default e testar diferentes configurações de modo a encontrar a melhor opção para cada situação específica.

2 Quem somos nós

Somos um grupo de pesquisadores interessados na área de MINLP. Como parte de nosso trabalho, desenvolvemos o nosso próprio solver Muriqui, cujo primeiro objetivo é dar suporte à pesquisas na área. Ficaríamos feliz se nosso trabalho de algum modo servisse a técnicos e pesquisadores de MINLP. Estamos desde já interessados em receber qualquer relato de uso, seja bom ou ruim.

3 Licença e condições de uso

Por hora, Muriqui é um resolvedor MINLP totalmente livre e aberto, e pode ser utilizado por meio da licensa MIT. Nossos únicos pedidos são para citar nossos artigos [12] nos documentos que relatem seu uso, e um pouco (talvez muita) paciência em sua utilização, especialmente com as versões iniciais que podem conter um número de bugs. Agradecemos de antemão qualquer colaboração no sentido de reportagem de erros e bugs aos desenvolvedores.

É importante frisar que, embora Muriqui seja um software livre, ele não é auto-contido. Isso significa que é necessária a instalação de softwares e bibliotecas de terceiros para o seu correto funcionamento (consulte a Seção 4). Cada um desses softwares pode ter suas próprias políticas de uso e licença, sendo alguns deles proprietários. Para o uso adequado de Muriqui, estes softwares devem ser instalados sob responsabilidade do usuário.

4 Softwares necessários

Inicialmente, se você pretende instalar Muriqui por meio do código fonte, você precisará de um compilador C++ 2011. Em nossos testes, temos obtido sucesso com distintas versões dos compiladores GCC e ICPC em sistemas Linux. De modo a aproveitar ao máximo as funcionalidades oferecidas, é recomendável a instalação da AMPL Solver Library (ASL), pelo menos um solver suportado de Programação Linear Inteira Mista (MILP) e pelo menos um solver suportado de Programação Não Linear (NLP). A seguir, algumas instruções quanto a essas instalações:

4.1 AMPL Solver Library

A biblioteca AMPL Solver Library (ASL) é necessária para integração de Muriqui com o sistema AMPL. A partir de sua instalação, é possível gerar um executável de Muriqui capaz de ler e processar modelos no formato nl (modelo compilado oriundos de AMPL). No entanto, se seu objetivo é utilizar Muriqui por meio de sua API em C++, você não precisa instalar essa biblioteca. Todavia, recomendamos fortemente o uso do executável construído sobre ASL sempre que for possível, uma vez que a linguagem AMPL facilita enormemente a escrita e manipulação de modelos de otimização, além de dispensar a codificação de derivadas. Se ainda assim você tiver que optar pela utilização da API, por exemplo para a integração em outro projeto de software, saiba de antemão que será preciso codificar (de forma correta) a derivada de primeira e segunda ordem de cada termo não linear $f(x)$ e $g_i(x)$, o que pode ser muito propenso a erros e vir a se tornar um verdadeiro inferno.

4.1.1 Instalação de ASL

A ASL pode ser baixada no endereço `http://www.netlib.org/ampl/solvers.tgz`. Após descompactar, entre no diretório gerado e rode o script de configuração com o comando

```
> configure
```

De modo a habilitar avaliações de função em paralelo, vá para o diretório gerado e rode o comando:

```
make arith.h
```

e acrescente a seguinte linha no arquivo `arith.h`

```
#define ALLOW_OPENMP
```

Acrescentar a linha acima é suficiente para habilitar a ASL em modo multithreading. Todavia, alternativamente, pode-se definir por conta própria as funções que deverão garantir a exclusão mútua em ASL (consulte [8] para mais detalhes).

Por fim, rode o comando "make".

Vale mencionar que nunca é demais realizar uma compilação otimizada para fins de ganho de performance. Portanto, se você puder, adicione flags de otimização para o compilador no arquivo de makefile como -O3"ou -march=native". Se você não souber do que estou falando ou não puder adicionar as flags por algum motivo, apenas ignore esse comentário:

```
> make
```

Se tudo ocorrer bem e você não tiver alterado nenhuma opção default, deverá ser gerado um arquivo chamado amplsolver.a. Recomendamos que você renomeie este arquivo para libamplsolver.a de modo estar em conformidade com o padrão de nomes de bibliotecas.

4.2 Solvers de MILP

Para a execução de alguns algoritmos de MILP, é necessária a resolução de subproblemas de Programação Linear Inteira Mista (MILP). Como resolver este tipo de problema está fora do escopo de Muriqui, é necessária a instalação de pelo menos um dos seguintes solvers suportados de MILP:

- Cplex (mais recomendado);
- Gurobi (recomendado);
- Xpress (meio termo recomendado);
- Mosek (pouco recomendado);
- Cbc (pouco recomendado);
- GLPK (quase nada recomendado).

Na maioria de nossos testes, temos utilizado Cplex para resolver os subproblemas de MILP gerados por Muriqui. Por essa razão, as chances de haver algum bug de interface com Cplex são menores. Essa é a principal razão para que este seja o solver mais recomendado, além do fato de uma licença de uso para pesquisa acadêmica ser de fácil obtenção. Todavia, se você acreditar que Gurobi ou Xpress podem funcionar melhor no seu caso, você está incentivado a experimentá-los. Se obter uma licença de software comercial está fora do seu alcance ou desejo, você ainda tem, como última alternativa, a opção de usar GLPK, que é um solver livre. Entretanto, GLPK é praticamente não recomendável pelo fato do mesmo não ser thread safe, o que pode atrapalhar a execução em alguns contextos. O dia em que alguém tiver notícia de que GLPK se tornou thread safe, por favor me avise que, com prazer, subirei a classificação dele aqui quanto a sua recomendação.

Após escolher um dos solvers suportados, realize a instalação segundo instruções dos seus desenvolvedores (procure os links em algum buscador de internet). Depois da instalação, você deve ser capaz de gerar um programa executável em C usando a API do solver escolhido para prosseguir com os requisitos necessários a instalação de Muriqui (procure compilar e rodar os exemplos de código em C/C++ distribuídos junto com o software escolhido).

4.3 Solvers de NLP

Para a execução de alguns algoritmos de MILP, é necessária a resolução de subproblemas de Programação Não Linear (NLP). Como resolver este tipo de problema também está fora do escopo de Muriqui, é necessária a instalação de pelo menos um dos seguintes solvers suportados:

- Mosek (se escolher este, escolha também ao menos um dos outros abaixo);
- Ipopt;
- Knitro;

Observamos que temos uma leve preferência por Mosek, pelo fato do mesmo ser capaz de lidar mais facilmente com os termos separáveis do problema (P), e por ser de fácil obtenção de licença acadêmica. Todavia, você está convidado a experimentar os outros solvers, especialmente se você tiver o feliz acesso a uma licença do Knitro. Para quem não quer ou não pode utilizar um software comercial, a opção indicada é o solver livre Ipopt,

que pode, em alguns casos, até funcionar melhor que os demais. É importante frisar que mesmo que você opte por Mosek, ainda assim recomendamos a instalação de Ipopt, uma vez que Mosek simplesmente se recusa a abordar problemas não convexos (mesmo para tentar encontrar ótimos locais). Assim, se algum dia você tratar um problema de MINLP não convexo, o Ipopt ou Knitro são as escolhas adequadas.

Após escolher um ou mais dos solvers suportados, realize a instalação segundo instruções dos seus desenvolvedores (procure os links em algum buscador de internet). Depois da instalação, você deve ser capaz de gerar um programa executável em C usando a API do solver escolhido para prosseguir com os requisitos necessários a instalação de Muriqui (procure compilar e rodar os exemplos de código em C/C++ distribuídos junto com o software escolhido).

5 Instalando Muriqui Optimizer

5.1 Downloading Muriqui Optimizer

O código fonte de Muriqui pode ser baixado em `http://wendelmelo.net/software`. Antes de proceder, é necessária a instalação dos softwares requeridos para o funcionamento apropriado de Muriqui conforme explanado na Seção 4.

5.2 Compilando Muriqui Optimizer

Após baixar e descompactar Muriqui no diretório de sua preferência, referido aqui como $MURIQUIDIR, o passo seguinte consiste em configurar a compilação de modo a utilizar as opções e bibliotecas escolhidas. Essa configuração é feita editando-se os arquivos WAXM_config.h e make.inc, presentes em $MURIQUIDIR, de forma integrada e consistente, como explanado a seguir:

5.2.1 Editando o arquivo WAXM_config.h

O arquivo WAXM_config.h traz definições para a correta compilação de Muriqui. As flags indicam a presença de bibliotecas e funções específicas a serem adotadas. Por exemplo, para compilar Muriqui com integração ao sistema AMPL por meio da biblioteca ASL, após instalar ASL seguindo os passos da Seção 4.1, edite a linha:

```
#define WAXM_HAVE_ASL                    0
```

para:

```
#define WAXM_HAVE_ASL                    1
```

Não se esqueça também de editar as linhas correspondentes aos solvers de MILP e NLP adotados. Assim, por exemplo, se o utilizador desejar compilar Muriqui com Cplex, Gurobi, Mosek and Ipopt deve editar as linhas:

```
#define WAXM_HAVE_CPLEX                  0
#define WAXM_HAVE_GUROBI                 0
#define WAXM_HAVE_MOSEK                  0
#define WAXM_HAVE_IPOPT                  0
```

para:

```
#define WAXM_HAVE_CPLEX                    1
#define WAXM_HAVE_GUROBI                   1
#define WAXM_HAVE_MOSEK                    1
#define WAXM_HAVE_IPOPT                    1
```

Note que é possível compilar Muriqui com diversos solvers de MILP e NLP simultaneamente, dando assim liberdade ao utilizador por escolher dentre esses softwares no momento da aplicação de algum algoritmo implementado em Muriqui.

Outras flags marcantes são: `WAXM_HAVE_CLOCK_GETTIME` que indica a presença da função C clock_gettime (geralmente presente em sistemas Unix); `WAXM_HAVE_POSIX` indica a presença de bibliotecas descritas no padrão Posix (também presente em sistemas Unix).

5.2.2 Editando o arquivo make.inc

O arquivo make.inc traz a definição de elementos da compilação de Muriqui, como por exemplo o compilador a ser adotado, flags de compilação, e a localização de arquivos de inclusão e bibliotecas requeridos. Ressaltamos que é preciso ter especial atenção com a edição deste arquivo de modo que o mesmo fique em conformidade com WAXM_config.h. Por exemplo, se WAXM_config.h contiver a definição `#define WAXM_HAVE_CPLEX  1`, isto significa que Muriqui está configurado para ser compilado com Cplex. Portanto, o arquivo make.inc deve trazer a definição da variável CPLEXINC com as definições de inclusão e CPLEXLIB com as definições de linkagem de biblioteca. Por default, makeinc traz linhas comentadas com a definição dessas variáveis, de modo que é possível descomentar as respectivas linhas e fazer as alterações necessárias de modo a possibilitar a correta compilação. Por exemplo, supondo que cplex está instalado no diretório $CPLEXDIR, uma possível definição para estas variáveis poderia ser:

```
CPLEXINC = -I${CPLEXDIR}/cplex/include/ilcplex/
CPLEXLIB = -L${CPLEXDIR}/cplex/lib/x86-64_linux/static_pic/ -lcplex -lpthread -pthread
```

Note que as definições acima podem variar dependendo das configurações do sistema em questão. Você deve realizar definições semelhantes para cada biblioteca de software definida com o valor 1 em WAXM_config.h, incluindo a ASL. Em nosso exemplo na Seção 5.2.1, configuramos Muriqui para ser compilado também com Gurobi, Mosek e Ipopt. Assim seria necessário também definir em make.inc as variáveis $GUROBIINC, $GUROBILIB, $MOSEKINC, $MOSEKLIB, $IPOPTINC e $IPOPTLIB (você pode descomentar as respectivas linhas e realizar as edições necessárias). Na maioria dos casos, para a integração de ASL na compilação de Muriqui, deveria ser suficiente seguir os passos descritos na Seção 4.1, alterar a definição de `WAXM_HAVE_ASL` para 1 (Seção 5.2.1) e alterar a linha em make.inc com a definição da variável $ASLDIR com o valor do diretório onde se encontra o arquivo libamplsolver.a.

5.2.3 Habilitando a integração com sistema GAMS

Para habilitar a integração com o sistema GAMS, é necessário seguir os seguintes passos:

- Baixar e instalar o sistema GAMS;
- Adicionar o diretório de instalação de GAMS no PATH do sistema e no PATH de bibliotecas dinâmicas (LD_LIBRARY_PATH);
- No arquivo WAXM_config.h, editar a linha:

```
#define WAXM_HAVE_GAMS 0
```

para:

```
#define WAXM_HAVE_GAMS 1
```

- Editar o arquivo make.inc, descomentando as linhas referentes a compilação com GAMS e alterando a linha de definição da variável GAMSDIR para que contenha o diretório de instalação de GAMS, por exemplo:

```
#GAMS variables to compilation.
GAMSDIR = /opt/gams/
GAMSINC = -I$(GAMSDIR)/apifiles/C/api/
GAMSLIB = $(GAMSDIR)/apifiles/C/api/gmomcc.c
```

Após a realização dos passos anteriores, a compilação de Muriqui habilitará integração com GAMS. Todavia, para utilizar Muriqui dentro do sistema GAMS, será necessário ainda configurá-lo seguindo os passos na Seção 6.3.

5.3 Rodando o Makefile

Tendo realizado a correta configuração da compilação descrita nas subseções anteriores, estamos prontos para rodar o programa make em $MURIQUIDIR.

```
> make
```

Se tudo ocorrer bem, deverão ser gerados o executável muriqui e a biblioteca libmuriqui.{a, lib}. O executável muriqui é construído de modo a ser utilizado de forma integrada ao sistema AMPL ou para ler arquivos .nl (modelos AMPL compilados), de modo que só faz sentido utilizá-lo se a biblioteca ASL tiver sido incorporada em sua compilação. A biblioteca libmuriqui traz as definições e rotinas disponíveis na API de Muriqui. Para se certificar de que a compilação ocorreu com sucesso, você pode rodar os exemplos distribuídos com o código fonte de Muriqui (Seção 6).

Se sua compilação não tiver sido bem sucedida, você pode tentar editar as definições de variáveis em make.inc ou WAXM_config.h. Sinta-se a vontade também para pedir ajuda na lista de discussão de Muriqui. Tentaremos auxiliá-lo nessa tarefa na medida do possível.

6 Rodando exemplos

6.1 Via sistema AMPL

A pasta $MURIQUIDIR/examples traz exemplos de uso de Muriqui via sistema AMPL, por meio de arquivos com extensão .mod. Você pode navegar até este diretório e escolher um dos exemplos para examinar, por exemplo, $MURIQUIDIR/examples/t1gross/t1gross.mod:

```
var x1 >= 0 <= 2;  # Continuous variables
var x2 >= 0 <= 2;  # Continuous variables
var x6 >= 0 <= 1; # Continuous variable
var y1 binary;  # Integer variable
var y2 binary;  # Integer variable
var y3 binary;  # Integer variable
```

```
param U;

minimize fcObj: 5*y1 +6*y2 +8*y3 +10*x1 -7*x6 -18 * log(x2 +1) - 19.2*log(x1 -x2 +1) +10;

subject to g1: 0.8*log(x2 + 1) + 0.96*log(x1 - x2 + 1) - 0.8*x6 >= 0;
subject to g2: x2 - x1 <= 0;
subject to g3: x2 - U*y1 <= 0;
subject to g4: x1 - x2 - U*y2 <= 0;
subject to g5: log(x2 + 1) + 1.2*log(x1 - x2 + 1) - x6 - U*y3 >= -2;
subject to g6: y1 + y2 <= 1;

data;
param U := 2;


#in this example, we solve this problem by several different algorithms.
#Note, you will probably need solve your models just once choosing one of the
#available algorithms.

option solver "muriqui";

#solving model by lp-nlp BB algorithm
options muriqui_alg_choice "str MRQ_LP_NLP_BB_OA_BASED_ALG";
#options muriqui_options "str in_milp_solver MRQ_GUROBI
#                         str in_nlp_solver MRQ_NLP_MOSEK
#                         dbl in_max_cpu_time 600
#                         int in_max_iterations 1000000000";
solve;

#solving model by outer approximation algorithm
options muriqui_alg_choice "str MRQ_OA_ALG";
solve;

#solving model by extended cutting plane algorithm
options muriqui_alg_choice "str MRQ_ECP_ALG";
solve;

#solving model by extended supporting hyperplane algorithm
options muriqui_alg_choice "str MRQ_ESH_ALG";
solve;

#solving model by bonmin hybrid algorithm
options muriqui_alg_choice "str MRQ_BONMIN_HYBRID_ALG";
solve;

#solving model by branch-and-bound algorithm (hybriding with outer approximation)
options muriqui_alg_choice "str MRQ_BB_ALG";
solve;


#solving model by pure branch-and-bound algorithm
options muriqui_alg_choice "str MRQ_BB_ALG";
options muriqui_options "int in_use_outer_app 0
                         int in_use_outer_app_as_heuristic  0";
solve;
```

O arquivo t1gross.mod especifica um modelo de MINLP. Após a definição do modelo,

especificamos o uso de Muriqui com a linha:

```
option solver "muriqui";
```

. Para que AMPL encontre o executável de Muriqui corretamente, é necessário incluir $MURIQUIDIR no path do sistema. O próximo passo é especificar um dos algoritmos disponíveis em Muriqui para resolver o modelo. Isso pode ser feito ajustando a opção *muriqui_ alg_ choice* para "str <algorithm>", onde <algorithm> deve ser alguma constante válida referente a algum dos algoritmos de Muriqui (veja a seção 7.3). Por exemplo, para usar Outer Approximation, usamos a linha

```
options muriqui_alg_choice "str MRQ_OA_ALG";
```

Através da opção *muriqui_ options* pode-se passar parâmetros para o algoritmo. Os parâmetros são passados através de uma string composta por trios do tipo "<type> <param name> <param value>" onde <type> pode ser *int* (integer number), *dbl* (real number) ou *str* (string). De forma semelhante, pode-se passar parâmetros para os solver de MILP e NLP adotado através das opções *muriqui_ milp_ options* e *muriqui_ nlp_ options*, respectivamente. O trecho a seguir de código AMPL especifica parâmetros a serem passados aos solver de MILP (Cplex) e NLP (Ipopt) utilizados por Muriqui:

```
#seting muriqui algorithm options, setting CLPEX as MILP solver
#and IPOPT as in_nlp_solver
option muriqui_options "str in_milp_solver MRQ_CPLEX
                        str in_nlp_solver MRQ_IPOPT ";

#seting options to milp solver (CPLEX)
option muriqui_milp_options "int CPX_PARAM_THREADS 1
                             dbl CPX_PARAM_TILIM 100";

#seting options to nilp solver (IPOPT)
option muriqui_nlp_options  "int max_cpu_time 5000
                             str linear_solver  pardiso";
```

.

6.2 Via arquivos nl AMPL

O arquivo t4gross.nl presente em $MURIQUIDIR traz a compilação de um modelo AMPL. Para rodá-lo, basta chamar o executável muriqui compilado com ASL passando o nome do arquivo como parâmetro.:

```
> ./muriqui t4gross.nl

------------------------------------------------------------------------------
Muriqui MINLP solver, version 0.7.00 compiled at Dec  2 2017 20:40:39.
By Wendel Melo, Computer Scientist, Federal University of Uberlandia, Brazil
collaborators: Marcia Fampa (UFRJ, Brazil), Fernanda Raupp (LNCC, Brazil)

if you use, please cite:
W Melo, M Fampa & F Raupp. Integrating nonlinear branch-and-bound and outer
approximation for convex Mixed Integer Nonlinear Programming. Journal of Global
Optimization, v. 60, p. 373-389, 2014.
------------------------------------------------------------------------------
```

```
muriqui: parameter 1: t4gross.nl
muriqui: input file: t4gross.nl
muriqui: Reading user model:
muriqui: Maximization problem addressed as minimization
muriqui: Done
muriqui: Reading user parameters:
muriqui: Trying reading algorithm choice file muriqui_algorithm.opt . Not found!
muriqui: Done
muriqui: Trying read input parameter file muriqui_params.opt
muriqui: Done
muriqui: preprocessing...

Starting Branch-and-Bound algorithm

muriqui: milp solver: cplex, nlp solver: mosek
iter lb  ub  gap nodes at open

1  -13.5901 -5.32374 8.26636 2 0
muriqui: Appliyng Outer Approximation on the root
cpu time: 1.03897
cpu time: 1.04079
------------------------------------------------------------------------------
Problem: t4gross.nl Algorithm: 1004 Return code: 0 obj function: -8.0641361104
time: 0.70 cpu time: 1.04 iters: 21
------------------------------------------------------------------------------
An optimal solution was found! Obj function: 8.064136. CPU Time: 1.040791
```

Note que os resultados de cada execução podem sofrer pequenas variações dependendo do sistema e da configuração em questão.

6.3 Via GAMS

Muriqui também pode receber problemas de MINLP oriundos do sistema GAMS. Para isso, primeiramente é necessário compilar Muriqui habilitando integração com GAMS, conforme descrito na Seção 5.2.3. Após essa tarefa, é necessário configurar GAMS para que reconheça Muriqui como resolvedor MINLP. Para isso, é necessário realizar os seguintes passos (UNIX):

1. Copiar os arquivos gmsmq_us.run e gmsmq_ux.out do diretório de Muriqui para o diretório de instalação de GAMS e torná-los executáveis;
2. Editar o arquivo gmscmpun.txt (no diretório de instalação de GAMS) acrescentando as seguintes linhas:

   ```
   MURIQUI 2111 5 mq 1 0 1 MIP QCP MIQCP RMIQCP NLP RMINLP MINLP
   gmsmq_us.run
   gmsmq_ux.out
   ```

Os passos acima podem ser ligeiramente diferentes em sistemas não UNIX. Entre em contato com o suporte GAMS para eventuais dúvidas ou problemas. Tendo realizado os passos acima com sucesso, é possível resolver um modelo de MINLP usando GAMS setando sua variável MINLP com o valor `muriqui`, por exemplo:

```
* GAMS model file ex1223.gms from MINPLib
```

```
Variables  x1,x2,x3,x4,x5,x6,x7,b8,b9,b10,b11,objvar;
Positive Variables  x1,x2,x3,x4,x5,x6,x7;
Binary Variables  b8,b9,b10,b11;
Equations  e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13,e14;

e1..    x1 + x2 + x3 + b8 + b9 + b10 =L= 5;
e2.. sqr(x6) + sqr(x1) + sqr(x2) + sqr(x3) =L= 5.5;
e3..    x1 + b8 =L= 1.2;
e4..    x2 + b9 =L= 1.8;
e5..    x3 + b10 =L= 2.5;
e6..    x1 + b11 =L= 1.2;
e7.. sqr(x5) + sqr(x2) =L= 1.64;
e8.. sqr(x6) + sqr(x3) =L= 4.25;
e9.. sqr(x5) + sqr(x3) =L= 4.64;
e10..    x4 - b8 =E= 0;
e11..    x5 - b9 =E= 0;
e12..    x6 - b10 =E= 0;
e13..    x7 - b11 =E= 0;
e14.. -(sqr((-1) + x4) + sqr((-2) + x5) + sqr((-1) + x6) - log(1 + x7)
+ sqr((-1) + x1) + sqr((-2) + x2) + sqr((-3) + x3)) + objvar =E= 0;

* set non-default bounds
x1.up = 10; x2.up = 10; x3.up = 10; x4.up = 1;
x5.up = 1;  x6.up = 1;  x7.up = 1;

Model m / all /;

* HERE, WE DEFINE MURIQUI LIKE THE MINLP SOLVER
option minlp = muriqui;

$if NOT '%gams.u1%' == '' $include '%gams.u1%'

$if not set MINLP $set MINLP MINLP
Solve m using %MINLP% minimizing objvar;
```

O ajuste de parâmetros de Muriqui pode ser realizados através dos seguintes arquivos, presentes no diretório corrente na execução do interpretador GAMS:

- muriqui_algorithm.opt: Define o algoritmo a ser utilizado por Muriqui;
- muriqui_params.opt: Define valores de parâmetros de algoritmos de Muriqui;
- muriqui_milp_params.opt: Define valores de parâmetros para o solver de MILP usado por Muriqui;
- muriqui_nlp_params.opt: Define valores de parâmetros para o solver de NLP usado por Muriqui;

Para mais detalhes sobre o uso destes arquivos, consulte a Seção 10.

6.4 Via API

A pasta $MURIQUIDIR/examples traz exemplos de uso da API de Muriqui em C++. Você pode navegar até um deles, por exemplo, $MURIQUIDIR/examples/t1gross e rodar o make a partir daí. Se tudo ocorrer bem, deverá ser gerado o executável t1gross, que resolve um exemplo por diversos algoritmos de MINLP com implementações presentes em Muriqui:

```
> make
...
> ./t1gross
```

7 Algoritmos implementados

Até o presente momento, os seguintes algoritmos MINLP estão implementados em Muriqui e disponíveis para a utilização:

7.1 Exatos

- LP based Branch-and-Bound (LP-BB) [13]
- LP/NLP based Branch-and-Bound (LP/NLP-BB) [14];
- Hybrid Outer Approximation Branch-and-Bound (HOABB) [10];
- Outer Approximation (OA) [6, 7];
- Extended Cutting Plane (ECP) [15];
- Extended Supporting Hyperplane (ESH) [9];
- Bonmin Hybrid Branch-and-Bound (BHBB) [2].

Observamos que a aplicação de um algoritmo de Branch-And-Bound puro também pode ser realizada através da escolha do algoritmo HOABB e desabilitando os subprocedimentos de aproximação externa.

7.2 Heurísticas

- Feasibility Pump (FP) [4];
- Diving heuristic [4];
- Outer Approximation based Feasibility Pump (OAFP) [3];
- Relaxation Enforced Neighborhood Search (RENS) [1];
- Integrality Gap Minimization Heuristic 1 (IGMA1), [11];
- Integrality Gap Minimization Heuristic 2 (IGMA2), [11].

7.3 Constantes e classes de Algoritmos

A Tabela 1 exibe a constante usada em Muriqui para designar cada algoritmo, com a respectiva classe na API em C++ que a implementa.

Tabela 1: Constantes e classes C++ de algoritmos de Muriqui

Sigla	**Tipo**	**Constante Muriqui**	**Classe na API**
LP-BB	Exact	MRQ_LP_BB_ECP_BASED_ALG	MRQ_LPBBExtCutPlan
LP/NLP-BB	Exato	MRQ_LP_NLP_BB_OA_BASED_ALG	MRQ_LPNLPBBOuterApp
HOABB	Exato	MRQ_BB_ALG	MRQ_BranchAndBound
OA	Exato	MRQ_OA_ALG	MRQ_OuterApp
ECP	Exato	MRQ_ECP_ALG	MRQ_ExtCutPlan
ESH	Exato	MRQ_ESH_ALG	MRQ_ExtSupHypPlane
BHBB	Exato	MRQ_BONMIN_HYBRID_ALG	MRQ_BonminHybrid
FP	Heurística	MRQ_FP_HEUR_ALG	MRQ_FeasibilityPump
Diving	Heurística	MRQ_DIVE_HEUR_ALG	MRQ_Diving
OAFP	Heurística	MRQ_OA_FP_HEUR_ALG	MRQ_OAFeasibilityPump
IGMA1	Exato/Heurística	MRQ_IGMA1_ALG	MRQ_IGMA1
IGMA2	Heurística	MRQ_IGMA2_ALG	MRQ_IGMA2
RENS	Heurística	MRQ_RENS_HEUR_ALG	MRQ_RENS
CR	Relax contínua	MRQ_CONT_RELAX_ALG	MRQ_ContinuousRelax

8 Qual algoritmo eu devo utilizar?

Uma vez que Muriqui disponibiliza uma série de algoritmos de MINLP a serem aplicados, uma dúvida natural diz respeito a como escolher o algoritmo a ser aplicado. Embora o utilizador seja incentivado a realizar testes diversos de modo a determinar a melhor abordagem para a sua aplicação específica, sabemos que isto pode ser impraticável em algumas situações. A escolha adequada pelo melhor algoritmo para cada situação pode levar em conta uma série de fatores. Por exemplo, se o utilizador deseja apenas a obtenção de uma solução viável de modo rápido, deve preferir a aplicação de uma heurística de viabilidade (por exmeplo IGMA2, se o problema for binário, OAFP se o problema for convexo ou FP para um problema mais genérico).

Caso seja necessário resolver um determinado problema até a o otimalidade, uma série de informações podem ser levadas em conta. Temos observado, por exemplo, que algoritmos de aproximação linear costumam se sair bem em, problemas convexos. Assim, se você souber de ante-mão que seu problema está nessa categoria, algum algoritmo dessa classe seria a escolha mais natural. Caso contrário, a melhor opção pode ser HOABB. Discutimos a seguir algumas características dos algoritmos e de sua implementações que podem auxiliar a tomada de decisão do usuário:

8.1 Algoritmos de aproximação linear

8.1.1 LP based Branch-and-Bound (LP-BB)

Em nossos testes [13], o algoritmo LP-BB [13] tem se mostrado o melhor desempenho prático sobre problemas convexos. Este algoritmo pode apenas ser aplicado se a API de Cplex ou Gurobi estiverem disponíveis no ambiente de execução e um destes estiver ajustado como o solver MILP. Este algoritmo é o default de Muriqui e pode fazer uso de todos os processadores disponíveis no hardware.

8.1.2 LP/NLP based Branch-and-Bound (LP/NLP-BB)

Em nossos testes [12], o algoritmo LP/NLP-BB [14] tem mostrado bom desempenho prático sobre problemas convexos. Este algoritmo pode apenas ser aplicado se a API de Cplex ou

Gurobi estiverem disponíveis no ambiente de execução e um destes estiver ajustado como o solver MILP. Este pode fazer uso de todos os processadores disponíveis no hardware.

8.1.3 Outer Approximation (OA)

O algoritmo OA [6, 7] pode ser uma boa opção em situações gerais onde o utilizador deseja resolver um problema convexo em um hardware com múltiplos processadores disponíveis. No entanto, o algoritmo OA resolve, a cada iteração, um ou dois problemas de NLP, que são subproblemas construídos a partir de (P). Se, por alguma razão, você desconfiar que a resolução da relaxação contínua ou de algum dos subproblemas de (P) exige esforço demasiado, pode ser uma boa ideia testar algoritmos como ECP ou ESH.

8.1.4 Extended Cutting Plane (ECP)

O algoritmo ECP [15] se caracteriza pela não resolução de nenhum subproblema de NLP ao longo de sua execução. Este algoritmo possui ainda a amigável qualidade de ser um método totalmente primeira ordem, o que significa que não faz uso de informação proveniente de segundas derivadas ou mesmo de aproximações para estas. Se você suspeitar que seu problema (convexo) de MINLP possui derivadas de segunda ordem demasiadamente caras de serem calculadas, ou se os pacotes de NLP apresentarem dificuldades para solucionar a sua relaxação contínua, este pode ser o algoritmo mais indicado. Se você estiver usando Muriqui por meio da API, onde é necessário codificar o cálculo das derivadas, você não precisará construir callbacks para o cálculo das derivadas de segunda ordem se optar por este algoritmo. Assim como OA, ECP também está habilitado a utilizar todos os processadores disponíveis no hardware.

8.1.5 Extended Supporting Hyperplane (ESH

O algoritmo ESH [9] pode ser visto como uma boa alternativa ao algoritmo ECP. No entanto, a implementação corrente em Muriqui precisa resolver um subproblema de NLP no início do algoritmo, o que pode exigir o cálculo de derivadas de segunda ordem ou de suas aproximações. Daí para frente, ESH prossegue como um método de primeira ordem também. Testes bem preliminares mostraram certa superioridade de ESH em comparação com ECP sobre problemas com menor percentual de restrições lineares. Assim como OA e ECP, ESH também está habilitado a utilizar todos os processadores disponíveis no hardware.

8.1.6 Bonmin Hybrid Branch-and-Bound (BHBB)

Este algoritmo é o Branch-and-Bound híbrido implementado no solver Bonmin [2]. Se baseia na aplicação de LP/NLP-BB com o uso de OA e a resolução de relaxações contínuas NLP de subproblemas ao longo da árvore de enumeração. Em nossos testes, nossa implementação deste algoritmo não conseguiu superar o desempenho de LP/NLP-BB puro. No entanto, se você souber de antemão que a implementação de Bonmin apresenta boa performance em seu problema (convexo) de MINLP, escolher este algoritmo pode talvez ser uma ideia (de toda forma, não deixe de testar LP/NLP-BB puro apenas por desencargo de consciência). Como este algoritmo está fundamentado na execução das iterações de LP/NLP-BB, sua implementação em Muriqui apresenta as mesmas restrições de uso que este último (veja a Subseção 8.1.2).

8.2 Hybrid Outer Approximation Branch-and-Bound (HOABB)

O algoritmo HOABB [10] parece ser a mais indicada para o tratamento de problemas não convexos, uma vez que algoritmos de aproximação linear são fortemente baseados em convexidade, e não raro, costumam declarar inviabilidade em problemas não convexos viáveis. É preciso ressaltar, no entanto, que este algoritmo também não garante otimalidade na resolução de problemas não convexos, podendo ser utilizado apenas na qualidade de heurística. Este algoritmo é baseado na aplicação de Branch-And-Bound não linear com a aplicação recorrente de OA. Por meio do ajuste de seus parâmetros, essa aplicação de OA pode vir a ser desabilitada, tornando assim o algoritmo como um Branch-And-Bound não linear puro. A dificuldade de resolução de relaxações de NLP faz com que, em muitos dos casos observados, os algoritmos de aproximação linear apresentem desempenho superior a HOABB. No entanto, se você se deparar com uma situação onde algoritmos de aproximação linear apresentem performance ruim, adotar HOABB pode talvez vir a ser uma boa escolha. Este algoritmo pode fazer uso de todos os processadores disponíveis no hardware.

9 Entendendo a API C++

9.1 Avaliando os termos não lineares

Para usar a API de Muriqui para solucionar um modelo, é necessário codificar procedimentos (métodos de classe) para os cálculos envolvendo termos não lineares e suas derivadas de primeira e segunda ordem (avaliações de função). Dependendo do contexto de uso, alguns procedimentos de cálculo de derivada podem ser dispensados. Por exemplo, o algoritmo ECP não faz uso de derivada de segunda ordem. Também é possível usar o algoritmo de Branch-And-Bound puro de Muriqui com algum solver de NLP, como, por exemplo, Ipopt, que faça aproximação das derivadas. Nesse caso, os procedimentos de cálculo das respectivas derivadas sendo aproximadas também poderiam ser omitidos. Todavia, para um melhor aproveitamento das funcionalidades de Muriqui como um todo, é altamente recomendável codificar os procedimentos de cálculo de derivada.

O subdiretório $MURIQUIDIR/examples traz exemplos de uso de Muriqui Optimizer. O primeiro exemplo, t1gross, foi extraído de [6], e sua formulação é dada por:

$$
\begin{aligned}
(E_1) \quad \min_{x} \quad & 5x_3 + 6x_4 + 8x_5 + 10x_0 - 7x_2 - 18\ln(x_1+1) && (2)\\
& -19.2\ln(x_0 - x_1 + 1) + 10, \\
\text{s. t} \quad & 0.8\ln(x_1+1) + 0.96\ln(x_0 - x_1 + 1) - 0.8x_2 \geq 0, && (3)\\
& x_1 - x_0 \leq 0, && (4)\\
& x_1 - Ux_3 \leq 0, && (5)\\
& x_0 - x_1 - Ux_4 \leq 0, && (6)\\
& \ln(x_1+1) + 1.2\ln(x_0 - x_1 + 1) - x_2 - Ux_5 \geq -2, && (7)\\
& x_3 + x_4 \leq 1, && (8)\\
& 0 \leq x_0 \leq 2, && (9)\\
& 0 \leq x_1 \leq 2, && (10)\\
& 0 \leq x_2 \leq 1, && (11)\\
& x_3, x_4, x_5 \in \{0,\, 1\} && (12)\\
& U = 2. && (13)
\end{aligned}
$$

O arquivo $MURIQUIDIR/examples/t1gross/t1gross.hpp traz uma codificação para o cálculo das derivadas do problema (E_1). O primeiro passo, consiste em declarar uma classe especificamente para os cálculos dos termos não lineares do problema e de suas derivadas. Esta classe deve ser derivada da classe MRQ_NonLinearEval, cuja declaração pode ser disponibilizada a partir da inclusão da biblioteca "muriqui.hpp", presente no diretório $MURIQUIDIR/includes:

```
#include <cmath>
#include "muriqui.hpp"

using namespace minlpproblem;
using namespace muriqui;

//we have to define a class to perform nonlinear function evaluations
class MyEval : public MRQ_NonLinearEval
{
public:

   //that method is called before other functions evaluations in a algorithm
   virtual int initialize(const int nthreads, const int n, const int m, const
        int nzNLJac, const int nzNLLagHess) override;

   //to evaluate objective function nonlinear part
   virtual int eval_nl_obj_part(const int threadnumber, const int n, const
       bool newx, const double *x, double &value) override;

   //to evaluate nonlinear constraints part
   virtual int eval_nl_constrs_part(const int threadnumber, const int n,
       const int m, const bool newx, const bool *constrEval, const double *x,
       double *values) override;

   //to evaluate nonlinear objective gradient part
   virtual int eval_grad_nl_obj_part(const int threadnumber, const int n,
       const bool newx, const double* x, double* grad) override;

   //to evaluate nonlinear jacobian part
   virtual int eval_grad_nl_constrs_part(const int threadnumber, const int n,
        const int m, const int nz, const bool newx, const bool *constrEval,
       const double *x, MIP_SparseMatrix& jacobian) override;

   //to evaluate nonlinear hessian of lagrangian (only lower triangle)
   // lagrangian:  objFactor*f(x,y) + lambda*g(x,y)
   virtual int eval_hessian_nl_lagran_part(const int threadnumber, const int
       n, const int m, const int nz, const bool newx, const double *x, const
       double objFactor, const double *lambda, MIP_SparseMatrix& hessian)
       override;

   //that method is called after all functions evaluations in a algorithm
   virtual void finalize(const int nthreads, const int n, const int m, const
       int nzNLJac, const int nzNLLagHess) override;

   virtual ~MyEval(){};
};
```

De início, após a inclusão da bibliotca muriqui.hpp, podemos notar o uso de dois namespaces distintos. Além do namespace muriqui há também o uso do namespace minlpproblem, que é um subpacote de Muriqui responsável por gerenciar a leitura, a representação e a escrita de problemas de MINLP. Observe que nossa classe responsável pelos cálculos dos termos não lineares e suas derivadas foi denominada MyEval. O primeiro método da classe é o método initialize, cuja definição, neste exemplo, é dada por:

```
int MyEval::initialize(const int nthreads, const int n, const int m, const
    int nzNLJac, const int nzNLLagHess)
{
   return 0;
}
```

O método *initialize* é chamado uma vez a cada execução de algoritmo com o método *run*, antes que qualquer avaliação de função seja realizada. Assim, quaisquer operações de inicialização necessárias aos cálculos, como definição de parâmetros internos e alocação de estruturas de memória auxiliares, pode ser realizada nesse método. Em contrapartida, cada algoritmo ao ser finalizado chamará o método *finalize* onde procedimentos de encerramento, como liberação de memória alocada, podem ser feitos. Chamamos atenção para um cuidado especial que deve ser tomado se sua classe precisar fazer alocação de memória para realizar as avaliações de função: alguns dos algoritmos de Muriqui são multithreads, o que significa que múltiplas chamadas aos procedimentos de avaliação podem ser feitas simultaneamente. Portanto, é preciso garantir que seus procedimentos funcionarão corretamente sobre as estruturas de memória alocada, sem que uma thread indevidamente interfira no cálculo de outra. Uma opção para evitar problemas desse tipo, especialmente nos primeiros testes, é restringir o número de threads a 1. Isto pode ser feito por meio do ajuste do parâmetro de entrada *in_number_of_threads* em qualquer classe que represente algum algoritmo.

Os parâmetros que o método *initialize* recebe são: *nthreads*: número de threads simultâneas de execução a serem criadas pelo algoritmo para avaliação de funções; n: número total de variáveis de decisão presentes no problema de MINLP; m: número total de restrições presentes no problema de MINLP; *nzNLJac*: número declarado de não-zeros presentes no Jacobiano dos termos não lineares; *nzNLLagHess*: número declarado de não-zeros presentes no triângulo inferior da hessiana do Lagrangeano. Como em nosso exemplo não é necessário nenhum procedimento de inicialização, apenas retornamos o valor 0 para indicar a Muriqui o sucesso na execução do método. Sempre que algum método da classe em questão retornar um valor diferente de zero, Muriqui assumirá que ocorreu um erro na execução do respectivo procedimento, podendo então o algoritmo em execução ser abortado, caso o valor requisitado seja crucial para o seu prosseguimento.

O próximo método da classe MyEval é *eval_nl_obj_part*, que é o método responsável por calcular o valor da parte não linear da função objetivo $f(x)$ sobre uma determinada solução x. Observe que, aqui, a varável *value* é um parâmetro de saída, e deve conter, ao final da execução deste método, o valor da calculado função objetivo na solução x informada:

```
//to evaluate objective function nonlinear part
int MyEval::eval_nl_obj_part(const int threadnumber, const int n, const bool
    newx, const double *x, double &value)
{
   value = 5*x[3] + 6*x[4] + 8*x[5] + 10*x[0] - 7*x[2] - 18*log(x[1]+1)
                   -19.2*log(x[0] -x[1] + 1) + 10;
   return 0;
}
```

Neste método, parâmetro de entrada *threadnumber* especifica qual a thread que está solicitando o cálculo (lembre-se de que para algoritmos multithreads, mais de uma thread pode solicitar cálculos simultaneamente). O parâmetro *newx* indica se a solução x sendo correntemente passada ao método foi a mesma passada no último cálculo envolvendo avaliações de função para a thread em questão. É importante ressaltar aqui, que, apenas os termos não lineares precisam efetivamente ser calculados nesse método. Embora tenhamos codificado toda a função objetivo nesse exemplo, poderíamos ter especificado os termos lineares e constantes da função objetivo na própria definição do problema de MINLP, deixando para

este método apenas o cálculo dos logaritmos, conforme foi feito no exemplo t2gross ($MURIQUIDIR/examples/t2gross). Veja este último exemplo para obter mais detalhes de como realizar esta separação de termos na função objetivo e restrições dos problemas. Em geral, realizar esta separação de termos torna a avaliação de função mais simples e pode aumentar a eficiência de Muriqui, pois o mesmo pode se aproveitar da decomposição do problema em seus procedimentos. Embora tenhamos feito esse primeiro exemplo sem essa separação para demonstrar o uso geral da API de modo mais simplificado, recomendamos a separação dos termos lineares e quadráticos do problema para uma melhor eficiência conforme realizado no exemplo t2gross.

O método seguinte da classe MyEval é o método *eval_nl_constrs_part*, que é o responsável por avaliar os termos presentes nas restrições não lineares ($g_i(x)$) em uma determinada solução x. Observe que aqui, *values* é um parâmetro de saída que contém um array com os valores de termos não lineares:

```
//to evaluate nonlinear constraints part
int MyEval::eval_nl_constrs_part(const int threadnumber, const int n, const
    int m, const bool newx, const bool *constrEval, const double *x, double *
    values)
{
   const double U = 2;
   const double *y = &x[3];

   //linear and quadratic constraints must not be evaluated

   if( !constrEval || constrEval[0] )
      values[0] = 0.8*log(x[1] + 1) + 0.96*log(x[0] - x[1] + 1) - 0.8*x[2];

   if( !constrEval || constrEval[4] )
      values[4] = log(x[1] + 1) + 1.2*log(x[0] - x[1] + 1) -x[2] - U*y[2];

   return 0;
}
```

. Nesse método, o vetor de booleanos *constrEval* indica quais restrições precisam ser ter seus termos não lineares efetivamente calculados. Caso este ponteiro esteja nulo, significa que **todas** as restrições com termos não lineares devem ser avaliadas na solução x passada como argumento. Chamamos a atenção para o fato de que restrições lineares e quadráticas não devem ser avaliadas nesse método! Qualquer termo linear ou quadrático pode ser especificado na própria definição do problema de MINLP e, portanto, não deveria ser calculado, a princípio pelos métodos da classe MyEval. Por essa razão, apenas os valores para primeira e a quinta restrição são calculadas pelo método *eval_nl_constrs_part*, pois estas são as únicas que contém termos não lineares. Ressaltamos novamente que os termos lineares e quadráticos presentes nessas restrições não precisariam estar codificados nesse método, e que só os deixamos aqui para questões didáticas. Termos lineares ou quadráticos declarados na definição do problema de MINLP são automaticamente calculados por Muriqui quando alguma avaliação de função é necessária.

O próximo elemento é o método *eval_grad_nl_obj_part*, que calcula o vetor gradiente referente aos termos não lineares da função objetivo ($\nabla f(x)$) em uma dada solução x:

```
//to evaluate nonlinear objective gradient part
int MyEval::eval_grad_nl_obj_part(const int threadnumber, const int n, const
    bool newx, const double *x, double *grad)
{
   grad[0] = 10 - 19.2/(x[0] -x[1] + 1);
   grad[1] = -18/(x[1]+1) + 19.2/(x[0] -x[1] + 1);
   grad[2] = -7;
```

```
    grad[3] = 5;
    grad[4] = 6;
    grad[5] = 8;

    return 0;
}
```

, onde o parâmetro de saída *grad* é um array cheio (não esparso) de n posições, com uma posição para cada derivada parcial das n variáveis de decisão. É preciso preencher todas as n posições do vetor, mesmo que alguma variável sequer apareça na função objetivo (nesse caso, sua respectiva posição será preenchida com zero). Novamente, apenas derivadas dos termos não lineares poderiam ser consideradas aqui se os termos lineares e quadráticos da função objetivo forem especificados na definição do problema de MINLP (que não é o caso desse exemplo).

A seguir, vem o método *eval_grad_nl_constrs_part*, responsável pelo cálculo do Jacobiano (derivadas de primeira ordem) dos termos não lineares das restrições ($\nabla g_i(x)$):

```
//to evaluate nonlinear jacobian part
int MyEval::eval_grad_nl_constrs_part(const int threadnumber, const int n,
    const int m, const int nz, const bool newx, const bool *constrEval, const
    double *x, MIP_SparseMatrix& jacobian)
{
    const double U = 2.0;

    if( !constrEval || constrEval[0] )
    {
        jacobian.setElement(0, 0, 0.96/(x[0] - x[1] + 1)); //line 0, column 0
        jacobian.setElement(0, 1, ( 0.8/(x[1] + 1) - 0.96/(x[0] - x[1] + 1) ));
            //line 0, column 1
        jacobian.setElement(0, 2, -0.8); //line 0, column 2
    }

    if( !constrEval || constrEval[4] )
    {
        jacobian.setElement(4, 0, 1.2/( x[0] - x[1] + 1 )); //line 4, column 0
        jacobian.setElement(4, 1, ( 1/(x[1] + 1) - 1.2/(x[0] - x[1] + 1) )); //
            line 4, column 1
        jacobian.setElement(4, 2, -1); //line 4, column 2
        jacobian.setElement(4, 5, -U); //line 4, column 5
    }

    return 0;
}
```

. Neste método, *nz* traz o número de elementos não nulos no jacobiano não linear especificados na definição do problema. O parâmetro de saída *jacobian* representa uma matriz esparsa para os armazenamento dos elementos não nulos no jacobianos. As respectivas posições dessa matriz que conterão os valores não nulos **devem** ser especificadas na definição do problema de MINLP. Desse modo, este método deve calcular valores de jacobiano para **todas** as posições (das restrições requisitadas em *constrEval*) definidas como não nulas na especificação do problema de MINLP. Mais uma vez, observe que apenas as restrições não lineares foram consideradas.

Na sequência, aparece o método *eval_hessian_nl_lagran_part*, cuja finalidade é realizar a avaliação da matriz hessiana do Lagrangeano (apenas o triângulo inferior). O Lagrangeano aqui é definido como:

$$L(x, \lambda) = \alpha f(x) + \sum_{i=1}^{m} \lambda_i g_i(x)$$

, e, portanto, sua hessiana será definida por:

$$\nabla^2 L(x,\lambda) = \alpha \nabla^2 f(x) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x)$$

. Desse modo, o nosso método *eval_hessian_nl_lagran_part* é definido como:

```
//to evaluate nonlinear hessian of lagrangian (only lower triangle)
// lagrangian:  objFactor*f(x,y) + lambda*g(x,y)
int MyEval::eval_hessian_nl_lagran_part(const int threadnumber, const int n,
    const int m, const int nz, const bool newx, const double *x, const double
    objFactor, const double *lambda, MIP_SparseMatrix& hessian)
{
   double aux;
   hessian.setAllSparseMatrix(0.0); //set all sparse matrix to 0.0


   //first we set the objective part
   if( objFactor != 0.0 )
   {
      aux = objFactor * 19.2/pow(x[0] -x[1] + 1, 2);

      hessian.setElement(0, 0, aux ); //setting value to position 0,0
      hessian.setElement(1, 0, -aux ); //setting value to position 1,0
      hessian.setElement(1, 1, objFactor * ( 18.0/pow( x[1] + 1, 2) + 19.2/
          pow(x[0] - x[1] + 1, 2) ) ); //setting value to position 1,1
   }


   //now, the constraints

   aux = lambda[0]*( -0.96/pow( x[0] - x[1] + 1, 2) ) + lambda[4]*( -1.2/pow(
       x[0] - x[1] + 1, 2) );

   hessian.addToElement(0, 0, aux); //adding value to position 0,0
   hessian.addToElement(1, 0, -aux); //adding value to position 1,0


   aux = lambda[0]*( -0.8/pow( x[1] + 1, 2) - 0.96/pow( x[0] - x[1] + 1, 2) )
       + lambda[4]*( -1/pow(x[1] + 1, 2) - 1.2/pow(x[0] - x[1] + 1, 2) );

   hessian.addToElement(1, 1, aux); //adding value to position 1,1

   return 0;
}
```

Aqui, o parâmetro de entrada *objFactor* representa o fator α, e o parâmetro de saída *hessian* representa uma matriz esparsa com o triângulo inferior da hessiana do Lagrangeano. Como esta matriz acumula um somatório de outras matrizes, foi oportuno zerar todas as suas posições antes dos cálculos começarem através do método *setAllSparseMatrix*:

```
void MIP_SparseMatrix::setAllSparseMatrix(double value);
```

Para ajustar devidamente as posições da matriz, usamos os métodos da matriz esparsa hessian *setElement* e *addToElement*, cujo protótipo pode ser conferido a seguir:

```
int MIP_SparseMatrix::setElement(unsigned int rowIndex, unsigned int colIndex
    , double value);
int MIP_SparseMatrix::addToElement(unsigned int rowIndex, unsigned int
    colIndex, double value);
```

Por fim, o método *finalize* é chamado após o algoritmo em execução finalizar todos os seus cálculos e nenhuma avaliação de função ser mais necessária. Esse método é útil para encerrar possíveis inicializações feitas no método *initialize*. Uma vez que, em nosso exemplo, não precisamos fazer qualquer operação ou alocação de memória no método *initialize*, também não necessitamos de qualquer operação no método *finalize*. Na realidade, neste exemplo, este método nem precisaria ser especificado. Note que este método não retorna nenhum valor.

```
void MyEval::finalize(const int nthreads, const int n, const int m, const int
     nzNLJac, const int nzNLLagHess)
{
}
```

9.2 Verificando a avaliação dos termos não lineares

É muito comum ocorrerem erros durante a codificação das avaliações dos termos não lineares, o que pode fazer as funções serem avaliadas incorretamente e comprometer a execução dos algoritmos que delas dependem. Por essa razão, Muriqui incorpora um verificador de derivadas simples baseado em diferenças finitas. Este verificador compara as derivadas codificadas com valores obtidos por diferenças finitas, alertando sobre eventuais inconsistências observadas. O arquivo $MURIQUIDIR/examples/t1gross/t1gross.cpp traz um exemplo de uso desse verificador, chamado através dos métodos *checkFisrtDerivatives* e *checkSecondDerivatives*, cujas assinaturas são dadas por:

```
int MRQ_MINLPProb::checkFisrtDerivatives(bool checkObj, bool checkConstr,
    const double *x, const double step, const double *steps, const double
    tolerance, bool &answer);

int MRQ_MINLPProb::checkSecondDerivatives(bool checkObj, bool checkConstr,
    const double* x, const double objFactor, const double* lambda, const
    double step, const double* steps, const double tolerance, bool& answer);
```

presentes na classe MRQ_MINLPProblem, responsável por gerenciar a representação do problema de MINLP.

9.3 Especificando características do problema

O namespace minlpproblem traz a definição da classe MRQ_MINLPProblem (MIP_MINLPProblem), cuja finalidade é gerenciar a representação do problema de MINLP. Esta classe traz diversos métodos que permitem especificar o problema sendo abordado como um todo. Os exemplos $MURIQUIDIR/examples/t1gross/t1gross.cpp e $MURIQUIDIR/examples/t2gross/t2gross.cpp são bons pontos de partida para compreender o funcionamento dos métodos presentes na classe.

9.4 Criando seu próprio Makefile para compilação

Uma vez que Muriqui pode fazer uso de diversas bibliotecas de terceiros, pode não ser trivial compilar uma nova aplicação que incorpore a sua API. Nossa recomendação é que se faça um Makefile para a compilação de novos projetos com base nos arquivos Makefile e make.inc presentes nos exemplos. Dessa forma, apenas mudando poucas linhas no arquivo Makefile dos exemplos, deve ser possível construir Makefiles adequados a novas aplicações. Especial atenção deve ser dada a definição da variável MYMURIQUIDIR no arquivo Makefile, que deve apontar para o diretório $MURIQUIDIR.

9.5 Ajustando parâmetros dos solvers suportados

Para a adequada execução dos algoritmos implementados, Muriqui trabalha com solvers terceiros para a resolução de subproblemas de MILP e NLP, por exemplo. Em algumas situações específicas, pode ser necessário ajustar parâmetros específicos desses solvers. Para permitir um fácil ajuste desses parâmetros, Muriqui implementa uma classe denominada MRQ_GeneralSolverParams, que armazena um número arbitrário de parâmetros dos tipos *inteiro*, *double* e *string*. Em geral, a execução dos algoritmos de MINLP implementados em Muriqui pode receber dois objetos da classe MRQ_GeneralSolverParams através do método *run*. O primeiro costuma especificar parâmetros a serem passados ao solver de MILP, ao passo que o segundo costuma especificar parâmetros ao solver de NLP.

Por exemplo, vamos supor que estamos interessados em executar o algoritmo Aproximação Externa (OA) usando Cplex e Ipopt como solvers de MILP e NLP, respectivamente. O trecho de código abaixo serve para armazenar parâmetros a serem passados ao Cplex:

```
MRQ_GeneralSolverParams milpsParams;

milpsParam.storeDoubleParameter("CPX_PARAM_EPGAP", 1e-3); //ajusta
    tolerância de gap de integralidade relativo de cplex para 1e-3
milpsParam.storeIntegerParameter("CPX_PARAM_SCRIND", 0);  //desabilita
    mensagens de cplex com impressão de resultados
```

Já o trecho a seguir armazena parâmetros para Ipopt

```
MRQ_GeneralSolverParams nlpsParams;

nlpsParams.storeIntegerParameter("max_iter", 1000); //ajusta o número máximo
    de iterações de ipopt para 1000
nlpsParams.storeIntegerParameter("print_level", 4); //ajusta o nível de
    impressão de ipopt para 4
nlpsParams.storeStringParameter("linear_solver", "ma27"); //ajusta solver
    linear usado por ipopt para ma27
```

Em seguida, pode-se chamar o método *run* para executar um algoritmo passando ponteiros para os objetos MRQ_GeneralSolverParams responsáveis por armazenar os parâmetros a serem ajustados:

```
MRQ_OuterApp oa;
oa.run(problem, &milpsParams, &nlpsParams);
```

Note que, em alguns casos, objetos MRQ_GeneralSolverParams podem vir a ser ignorados. Por exemplo, o algoritmo ECP não faz uso de solver NLP, e assim, o mesmo ignora o objeto com os parâmetros para esse tipo de solver. Já a heurística Diving, por outro lado, não utiliza solver MILP, e portanto, não faz uso do objeto com parâmetros para tal.

9.6 Organização das classes de Algoritmos na API C++

Muriqui define uma classe geral para representação de algoritmos, denominada MRQ_Algorithm. Os algoritmos são implementados em Muriqui em classes derivadas desta. Isto significa que definições feitas para a classe MRQ_Algorithm, como parâmetros e métodos gerais, são válidas para todas as classes que implementam algoritmos. Outras (sub)classes gerais abaixo de MRQ_Algorithm são MRQ_LinearApproxAlgorithm, que generaliza definições de algoritmos de aproximação linear, e MRQ_Heuristic que traz definições gerais para heurísticas. A Figura 1 apresenta um organograma das classes de algoritmos de Muriqui. As definições realizadas nas classes superiores são válidas para as classes inferiores em um determinado fluxo do organograma. Isto é especialmente útil para entender o funcionamento dos métodos

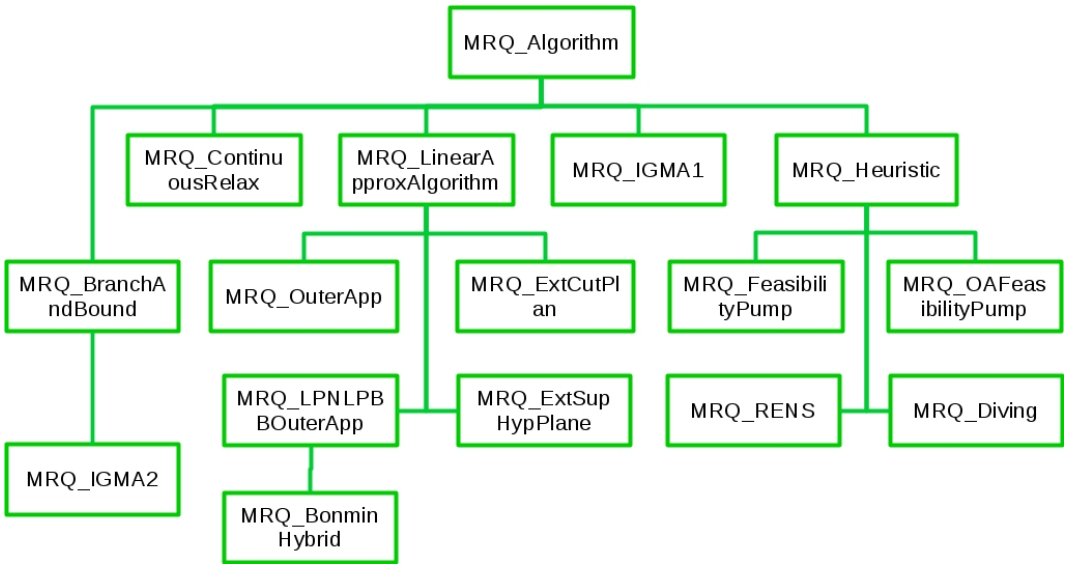

Figura 1: Organograma de classes de algoritmos em Muriqui

e parâmetros de entrada e saída gerais utilizados por Muriqui. Por fim, aproveitamos para apontar a existência da classe MRQ_ContinuousRelax cuja finalidade é permitir a resolução da relaxação contínua de um problema de MINLP por meio de algum dos *solvers* de NLP suportados. Se o respectivo solver fornecer suporte a definição de variáveis inteiras, como é o caso do Knitro, é possível usar esta classe para resolver efetivamente o problema de MINLP por meio desse solver usando a API de Muriqui como intermediária.

10 Parâmetros de entrada

Os parâmetros de entrada controlam aspectos do comportamento dos algoritmos. De um modo geral, são divididos em 3 categorias:

- Parâmetros ajustados com números inteiros (*int*) (para algumas finalidades práticas, aqui também se incluem os parâmetros booleanos, que passam então a ser tratados como inteiros binários, isto é restritos aos valores 0 e 1);
- Parâmetros ajustados com números reais (*dbl*);
- Parâmetros ajustados com constantes enumeradas (por dentro da API) ou cadeias de texto (*str*) (por fora da API).

Ressaltamos que para que o ajuste ocorra de forma correta, é necessário saber o tipo do parâmetro a ser ajustado. O ajuste de parâmetros pelo sistema AMPL está exemplificado na Seção 6.1, onde é necessário descrever uma tripla, especificando o tipo do parâmetro(*int*, *dbl* ou *str*), seu nome, e o valor a ser ajustado. Por exemplo, o comando:

```
options muriqui_options "   str in_nlp_solver MRQ_NLP_MOSEK
                            dbl in_max_cpu_time 600.0
                            int in_max_iterations 1000";
```

estabelece que:

1. o parâmetro *in_ nlp_ solver* (do tipo *str*) deve ser ajustado com o valor MRQ_NLP_MOSEK, o que fará com que Mosek seja utilizado como solver NLP;

2. o parâmetro *in_max_cpu_time* (do tipo *double*) deve ser ajustado para 600.0, o que fará com que o algoritmo em questão só possa ser processado por, no máximo 600.0 segundos;

3. o parâmetro *in_max_iterations* (do tipo *int*) deve ser ajustado para 1000, o que fará com que o algoritmo em questão só possa executar, no máximo, 1000 iterações;

Para as execuções de Muriqui como programa executável pode-se especificar parâmetros através de um arquivo especial chamado muriqui_params.opt. Assim, esses mesmos parâmetros poderiam ser ajustados definindo esse arquivo no diretório de execução com o conteúdo:

```
str   in_nlp_solver       MRQ_NLP_MOSEK
dbl   in_max_cpu_time     600.0
int   in_max_iterations   1000
```

A escolha do algoritmo, que é feita antes dos ajustes de parâmetro, pode ser feita através da especificação de um arquivo especial chamado muriqui_algorithm.opt, por exemplo com o conteúdo:

```
MRQ_OA_ALG
```

Que especifica que o algoritmo a ser utilizado é o Outer Approximation. Aqui, qualquer constante de algoritmo da Tabela 1 poderia ter sido utilizada. Note que, apenas nesse caso, em particular, não há definição de tipo nem de nome do parâmetro, apenas o seu valor.

É oportuno observar que os nomes de parâmetros de entrada sempre são prefixados por *in_*, ao passo que os nomes dos parâmetros de saída sempre são prefixados por *out_*.

Muriqui também permite o ajuste de parâmetros dos solvers de MILP e NLP utilizados nos algoritmos. Para isso, pode-se especificar, respectivamente os arquivos muriqui_milp_params.opt e muriqui_nlp_params.opt com os parâmetros desejados seguindo o mesmo esquema do arquivo muriqui_params.opt. Note que o conjunto de parâmetros disponíveis para ajuste varia de acordo com cada solver, e é preciso consultar as respectivas documentações para maiores detalhes. Por exemplo, para o Cplex, poderíamos definir um arquivo muriqui_milp_params.opt com as linhas

```
dbl   CPX_PARAM_EPGAP     1e-5
int   CPX_PARAM_NODELIM   9999999
```

Ao passo que, para Ipopt, poderíamos definir um arquivo muriqui_nlp_params.opt da seguinte forma:

```
str   linear_solver     ma27
int   print_level       2
dbl   constr_viol_tol   1e-6
int   max_iter          5000
```

Por comodidade, linhas dos arquivos de definição de parâmetros cujo primeiro caracter for # serão tratadas como linhas de comentário, isto é, serão ignoradas por Muriqui. Ressaltamos que, no máximo um parâmetro deve ser setado por linha nesses arquivos.

O ajuste de parâmetros de entrada por meio da API pode ser realizado de duas formas distintas. A primeira delas é acessando o parâmetro diretamente como atributo de alguma das classes de algoritmo (veja os exemplos em $MURIQUIDIR/examples), como no trecho a seguir:

```
MRQ_OuterApp oa;
oa.in_nlp_solver = MRQ_NLP_MOSEK;
oa.in_max_cpu_time = 600.0;
oa.in_max_iterations = 1000;
```

A segunda forma consiste no uso dos métodos *setDoubleParameter*, *setIntegerParameter* e *setStringParameter*, que, ajustam parâmetros do tipo *double*, *int* e *str*, respectivamente, conforme a assinatura abaixo:

```
int MRQ_Algorithm::setDoubleParameter(const char *name, double value);

int MRQ_Algorithm::setIntegerParameter(const char *name, long int value);

int MRQ_Algorithm::setStringParameter(const char *name, const char *value);
```

. Por estarem definidos na classe MRQ_Algorithm, esses métodos se aplicam a qualquer algoritmo implementado em Muriqui, como no exemplo a seguir:

```
MRQ_OuterApp oa;
oa.setStringParameter(  "in_nlp_solver",      "MRQ_NLP_MOSEK");
oa.setDoubleParameter(  "in_max_cpu_time",    600.0);
oa.setIntegerParameter( "in_max_iterations", 1000);
```

10.1 Parâmetros definidos na classe MRQ_Algorithm

Os parâmetros definidos na classe MRQ_Algorithm se aplicam aos algoritmos implementados por Muriqui em geral. Parâmetros específicos de cada abordagem são definidos nas classes derivadas.

10.1.1 Parâmetros *double* (dbl)

- **in_absolute_convergence_tol**: tolerância de convergência absoluta. Se a diferença entre o limite inferior e superior se tornar menor ou igual a este valor, o algoritmo para declarando otimalidade se alguma solução viável for conhecida, ou declara inviabilidade, caso contrário. Valor default: $1.0e-3$;

- **in_absolute_feasibility_tol**: tolerância de viabilidade absoluta, utilizada para determinar se algumas soluções satisfazem as restrições gerais de (P). Valor default: $1.0e-6$;

- **in_integer_tol**: tolerância de integralidade. Uma solução é considerada como inteira se o maior gap de integralidade dentre as variáveis inteiras é menor ou igual a este valor. Valor default: $1.0e-4$;

- **in_lower_bound**: limite inferior inicial para a execução dos algoritmos. Em algumas situações, já se conhece a priori um limite inferior válido para o valor da função objetivo ótimo de (P). Este valor pode ser passado aos algoritmos por meio deste parâmetro, como tentativa de acelerar a detecção de otimalidade. Valor default: -MRQ_INFINITY;

- **in_max_time**: tempo máximo real (no relógio, em segundos) de execução do algoritmo, desde o momento de seu início. Este parâmetro é indiferente ao número de processadores do hardware e ao tempo de execução do algoritmo nos processadores propriamente dito, cabendo o ajuste desse último ao parâmetro *in_max_cpu_time*. Valor default: ∞;

- **in_max_cpu_time**: tempo máximo de execução (em segundos) em processadores do algoritmo. Esse parâmetro leva em conta o tempo de execução propriamente dita em todos os processadores do hardware. Valor default: ∞;

- **in_relative_convergence_tol**: tolerância de convergência relativa. Se a diferença relativa entre o melhor limite inferior e superior for menor ou igual este valor, o algoritmo para declarando otimalidade se alguma solução viável for conhecida, ou declarando inviabilidade, caso contrário.

- **in_relative_feasibility_tol**: tolerância de viabilidade relativa, utilizada para determinar se algumas soluções satisfazem as restrições gerais de (P). Valor default: $1.0e-6$;

- **in_upper_bound**: limite superior inicial para a execução dos algoritmos. Em algumas situações, já se conhece a priori um limite superior para o valor da função objetivo ótimo de (P). Este valor pode ser passado aos algoritmos por meio deste parâmetro, como tentativa de acelerar a detecção de otimalidade. Valor default: MRQ_INFINITY;

10.1.2 Parâmetros *inteiros* (int)

- **in_assume_convexity**: flag que especifica se o problema de entrada deve ser tratado como convexo. Correntemente, este parâmetro não possui finalidade prática. Valor default: 1 (*true*);

- **in_call_end_of_iteration_callback**: flag que especifica se a callback de final de iteração deve ser chamada. Se estiver setada como 1 (*true*), esta callback permite ao usuário acompanhar a execução do algoritmo a cada iteração. Este parâmetro só se aplica ao uso pela API e callback de acompanhamento precisa ser provida através do parâmetro *in_user_callbacks*. Valor default: 0 (*false*);

- **in_call_update_best_sol_callback**: flag que especifica se a callback de atualização de solução deve ser chamada. Se estiver setada como 1 (*true*), esta callback permite ao usuário acompanhar a evolução da melhor solução encontrada pelo algoritmo. Este parâmetro só se aplica ao uso pela API e a callback de acompanhamento precisa ser provida através do parâmetro *in_user_callbacks*. Valor default: 0 (*false*);

- **in_max_iterations**: número máximo de iterações do algoritmo. Valor default: ULONG_MAX (maior inteiro longo sem sinal representável no sistema corrente);

- **in_number_of_threads**: número de threads de execução utilizado pelo algoritmo. Na prática, este número especifica a quantidade de processadores a serem utilizados simultaneamente na resolução do problema, quando aplicável. Note que alguns algoritmos não permitem execução em paralelo. O ajuste deste parâmetro também afeta o desempenho dos solvers suportados, uma vez que este valor é também repassado aos mesmos. Quando este parâmetro é ajustado em 0, Muriqui assume como número de threads de execução o mesmo número de processadores existentes no hardware. Não recomendamos ajustar esse valor com um número superior ao número de processadores existentes no hardware. Valor default: 0;

- **in_preprocess_lin_constr**: flag que especifica se deve ser realizado pré-processamento sobre restrições lineares presentes no problema abordado. Valor default: 1 (*true*);

- **in_preprocess_obj_function**: flag que especifica se deve ser realizado pré-processamento sobre a função objetivo do problema abordado. Valor default: 1 (*true*);

- **in_preprocess_quad_constrs**: flag que especifica se deve ser realizado pré-processamento sobre restrições quadráticas presentes no problema abordado. Valor default: 1 (*true*);

- **in_print_level**: nível de impressão de informações do algoritmo ao longo da execução. Quanto maior o nível, maior a quantidade de informações exibidas. Valor default: 3;

- **in_print_parameters_values**: flag que especifica se o valor de todos os parâmetros do algoritmo corrente deve ser impresso antes da execução do mesmo. É útil para se certificar do correto ajuste dos parâmetros. Valor default: 0 (*false*);

- **in_printing_frequency**: frequência de impressão de resumo de iteração. A cada *in_printing_frequency* iterações, Muriqui imprime informação referente a iteração corrente. Valor default: 1 (o valor default deste parâmetro pode ser diferente em classes derivadas);

- **in_set_special_nlp_solver_params**: flag que especifica se parâmetros especiais do solver NLP devem ser ajustados. Em geral, estes parâmetros especiais são um subconjunto de parâmetros do solver ajustados para tentar acelerar a convergência do mesmo. No entanto, esse ajuste automático pode trazer problemas numéricos em alguns casos. Caso sua aplicação esteja apresentando problemas na resolução dos subproblemas de NLP, experimente desabilitar essa flag. Valor default: 0 (*false*) (o valor default deste parâmetro pode ser diferente em classes derivadas, especialmente algoritmos relacionados a Branch-And-Bound);

- **in_store_history_solutions**: flag que especifica se um histórico de soluções deve ser construído com as soluções viáveis encontradas ao longo das iterações. Esta funcionalidade é útil quando se deseja conferir, ao final de execução, todas as soluções viáveis encontradas pelo algoritmo. Caso esta flag seja ativada, estas soluções serão armazenadas no parâmetro de saída *out_sol_hist*. Valor default: 0 (*false*);

- **in_use_dynamic_constraint_set**: flag que especifica se o Conjunto de Restrição Dinâmico (DCS) deve ser adotado. Esta funcionalidade só é implementada no algoritmo de Branch-And-Bound e permite desconsiderar algumas restrições de (P) se determinadas variáveis binárias estiverem com valor inteiro, de forma semelhante ao uso de Big-M. Para o correto funcionamento, é preciso fornecer informação específica do problema através dos métodos *setDCS0Array* e *setDCS1Array*. Valor default: 0 (*false*);

- **in_use_initial_solution**: flag que especifica se a solução inicial fornecida pelo usuário deve ser utilizada pelo algoritmo, por exemplo, como ponto inicial para o solver NLP. Valor default: 0 (*false*);

10.1.3 Parâmetros *string* (str)/enumerativos

- **in_milp_solver**: determina o solver MILP adotado pelo algoritmo, quando aplicável. Certifique-se de Muriqui foi corretamente compilado para usar o solver escolhido (veja a Seção 4). As opções para este parâmetro são listadas na tabela 2. Valor Default: MRQ_CPLEX.

- **in_nlp_solver**: determina o solver NLP adotado pelo algoritmo, quando aplicável. Certifique-se de Muriqui foi corretamente compilado para usar o solver escolhido (veja a Seção 4). As opções para este parâmetro são listadas na tabela 3. Valor default: MRQ_NLP_KNITRO.

Tabela 2: Constantes de solvers MILP usados em Muriqui

Solver	**Constante**	**Ordem de Prioridade**
Cbc	MRQ_CBC	6
Cplex	MRQ_CPLEX	1
Glpk	MRQ_GLPK	7
Gurobi	MRQ_GUROBI	2
Knitro	MRQ_MILP_KNITRO	4
Mosek	MRQ_MILP_MOSEK	5
Xpress	MRQ_XPRESS	3

Tabela 3: Constantes de solvers NLP usados em Muriqui

Solver	**Constante**	**Ordem de Prioridade**
Ipopt	MRQ_IPOPT	2
Knitro	MRQ_NLP_KNITRO	1
Mosek	MRQ_NLP_MOSEK	3

10.2 Parâmetros definidos na classe MRQ_LinearApproxAlgorithm

A classe MRQ_LinearApproxAlgorithm é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui se aplicam a algoritmos de aproximação linear em geral.

10.2.1 Parâmetros *double* (dbl)

- **in_eps_to_active_constr_to_linearisation**: tolerância (absoluta e relativa) usada considerar restrição ativa. Este valor será usado se o parâmetro *in_constr_linearisation_strategy* estiver definido como *MRQ_CLS_ONLY_INFEAS_AND_ACTIVE_MASTER_SOL_ALSO*. Valor default: $1.0e-4$.

10.2.2 Parâmetros *inteiros* (int)

- **in_measure_nlp_time**: flag que especifica medição o tempo de despendido com a resolução de problemas NLP no algoritmo. O tempo medido (em segundos) estará disponível, ao final da execução, nos parâmetros de saída *out_clock_time_of_nlp_solving* (tempo de relógio) e *out_cpu_time_of_nlp_solving* (tempo de processamento). Valor default: 0 (*false*);

- **in_set_obj_lower_bound_on_master_problem**: flag que determina se o limite inferior corrente deve ser explicitamente passado ao problema mestre. Valor default: 0 (*false*);

- **in_set_quadratics_in_master_problem**: flag que especifica te função objetivo e restrições quadráticas (se houver) serão diretamente adicionadas ao problema mestre em sua forma original quadrática. Caso esta flag seja habilitada, será preciso utilizar um solver MILP que resolva problemas quadráticos inteiros mistos. Valor default: 0 (*false*);

10.2.3 Parâmetros *string* (str)/enumerativos

- **in_constr_linearisation_strategy**: especifica estratégia de linearização das restrições. Possíveis valores para este parâmetro são:
  - MRQ_CLS_ALL_CONSTRS: lineariza todas as restrições em cada ponto de linearização;
  - MRQ_CLS_ONLY_INFEAS_AND_ACTIVE: lineariza apenas restrições violadas e ativas na solução do problema mestre;

  Valor default: MRQ_CLS_ALL_CONSTRS;

- **in_obj_linearisation_strategy**: determina estratégia de linearização da função objetivo. Possíveis valores para este parâmetro são:
  - MRQ_OLS_ALL_POINTS: lineariza função objetivo em todos os pontos de linearização;
  - MRQ_OLS_NON_OBJ_CUT_POINTS: lineariza função objetivo em pontos não dominados (cortados). Esta opção ainda está em fase de experimentação.

  Valor default: MRQ_OLS_ALL_POINTS;

- **in_quad_app_master_strategy**: determina estratégia de uso de aproximação quadrática no problema mestre. Este parâmetro está em fase experimental. Possíveis valores para este parâmetro são:
  - MRQ_QAMS_NO_QUAD_APP: não faz uso de aproximação quadrática;
  - MRQ_QAMS_ON_BEST_POINT: aproximação quadrática construída sobre a melhor solução;
  - MRQ_QAMS_ON_LAST_POINT: aproximação quadrática construída sobre ponto obtido na iteração anterior;

  Valor default: MRQ_QAMS_NO_QUAD_APP.

10.3 Parâmetros definidos na classe MRQ_Heuristic

A classe MRQ_Heuristic é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui se aplicam a algoritmos heurísticos em geral.

10.3.1 Parâmetros *inteiros* (int)

- **in_solve_nlp_as_local_search_at_end**: flag que especifica se um problema NLP de busca local deve ser resolvido caso o algoritmo encontre alguma solução viável. Valor default: 1 (*true*);

- **in_seed_to_random_numbers**: semente para geração de números pseudo-aleatórios, usado caso o parâmetro *in_use_random_seed_to_random_numbers* esteja com o valor *false*. Valor default: 1986;

- **in_use_random_seed_to_random_numbers**: flag que determina se uma semente "aleatoria" será utilizada para geração de números pseudo-aleatórios. Na prática, essa semente "aleatoria" é gerada a partir da hora corrente do sistema. Valor default: 0 (*false*).

10.4 Parâmetros definidos na classe MRQ_ExtCutPlan (ECP)

A classe MRQ_ExtCutPlan é derivada de MRQ_LinearApproxAlgorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Extended Cutting Plane.

10.4.1 Parâmetros *inteiros* (int)

- **in_refine_final_solution_using_nlp**: flag que ativa refinamento da solução encontrada por meio da resolução de um problema de programação não linear onde as variáveis inteiras são fixas nos valores correspondentes a melhor solução encontrada. O objetivo desse refinamento é reduzir possíveis erros numéricos presentes na melhor solução encontrada. Valor default: 1 (*true*).

10.5 Parâmetros definidos na classe MRQ_OuterApp (OA)

A classe MRQ_OuterApp é derivada de MRQ_LinearApproxAlgorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Outer Approximation.

10.5.1 Parâmetros *inteiros* (int)

- **in_binarie_cut_when_nlp_infeasible**: flag que determina a adoção de corte binário quando o problema NLP obtido ao se fixar as variáveis inteiras se mostrar inviável. Esta estratégia só pode ser adotada em problemas binários. Nesse caso, o problema NLP de viabilidade não será resolvido. Valor default: 0 (*false*);
- **in_round_first_nlp_relaxation_solution**: flag que especifica o uso de arredondamento sobre a solução do problema NLP de relaxação contínua para uma possível linearização. Valor default: 0 (*false*);
- **in_use_first_nlp_relaxation**: flag que determina se a solução da relaxação contínua será utilizada como ponto de linearização. Este parâmetro só pode ser desabilitado caso o usuário tenha fornecido um ou mais pontos alternativos de linearização através do método *addPointsToLinearisation*. Valor default: 1 (*true*);

10.6 Parâmetros definidos na classe MRQ_LPNLPBBOuterApp (LP/NLP-BB)

A classe MRQ_LPNLPBBOuterApp é derivada de MRQ_LinearApproxAlgorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo LP/NLP based Branch-and-Bound.

10.6.1 Parâmetros *inteiros* (int)

- **in_binarie_cut_when_nlp_infeasible**: flag que determina a adoção de corte binário quando o problema NLP obtido ao se fixar as variáveis inteiras se mostrar inviável. Esta estratégia só pode ser adotada em problemas binários. Nesse caso, o problema NLP de viabilidade não será resolvido. Valor default: 0 (*false*);
- **in_use_first_nlp_relaxation**: flag que determina se a solução da relaxação contínua será utilizada como ponto de linearização. Este parâmetro só pode ser desabilitado caso o usuário tenha fornecido um ou mais pontos alternativos de linearização através do método *addPointsToLinearisation*. Valor default: 1 (*true*);

- **in_linearize_obj_in_nl_feas_solutions**: flag que determina se uma possível função objetivo não linear deve ser linearizada em soluções oriundas da resolução de problemas de viabilidade. Valor default: Valor default: 1 (*true*).

10.7 Parâmetros definidos na classe MRQ_BonminHybrid (BHBB)

A classe MRQ_BonminHybrid é derivada de MRQ_LPNLPBBOuterApp, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Bonmin Hybrid Branch-and-Bound.

10.7.1 Parâmetros *double* (dbl)

- **in_outer_app_max_cpu_time**: tempo máximo de processamento, em segundos, para execução inicial do algoritmo Aproximação Externa. Outras opções relacionadas a Aproximação Externa podem ser ajustadas por meio do parâmetro *in_outer_app*. Valor default: 30.0;

- **in_outer_app_max_time**: tempo máximo de relógio, em segundos, para execução inicial do algoritmo Aproximação Externa. Outras opções relacionadas a Aproximação Externa podem ser ajustadas por meio do parâmetro *in_outer_app*. Valor default: ∞;

10.7.2 Parâmetros *inteiros* (int)

- **in_out_app_max_iterations**: número máximo de iterações a cada aplicação do algoritmo OA. Valor Default: ULONG_MAX (maior inteiro longo sem sinal representável no sistema corrente);

- **in_nlp_relaxation_solving_frequence**: frequência de resolução de relaxação NLP ao longo da árvore de BB para MILP. Valor Default: 10.

10.8 Parâmetros definidos na classe MRQ_ExtSupHypPlane (ESH)

A classe MRQ_ExtSupHypPlane é derivada de MRQ_LinearApproxAlgorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Extended Supporting Hyperplane.

10.8.1 Parâmetros *double* (dbl)

- **in_absolute_tol_to_check_previous_sol**: tolerância absoluta para habilitar a verificação de repetição de solução entre as iterações. Devido a problemas numéricos, em alguns casos, a solução da iteração anterior pode se repetir. Este parâmetro controla quando duas soluções devem ser testadas componente a componente se o valor da função objetivo de ambas estiver suficientemente próximo. Valor default: $1.0e-6$;

- **in_cont_relax_absolute_convergence_tol**: tolerância de convergência absoluta na resolução da relaxação contínua de (P). Este algoritmo resolve a relaxação contínua explicitamente por meio do método de hiperplano de suporte, e essa tolerância é utilizada como critério de parada dessa resolução. Valor default: $1.0e-3$;

- **in_cont_relax_relative_convergence_tol**: tolerância de convergência relativa na resolução da relaxação contínua de (P). Este algoritmo resolve a relaxação contínua explicitamente por meio do método de hiperplano de suporte, e essa tolerância é utilizada como critério de parada dessa resolução. Valor default: $1.0e-3$;

- **in_delta_to_inc_eps_to_active_constraints_to_linearization**: valor para incremento de tolerância de restrições ativas se repetição de solução for detectada. Valor default: 0.05;

- **in_eps_to_enforce_interior_sol_on_cont_relax_sol**: valor épsilon para ser usado como distancia relativa e absoluta para forçar obtenção de solução interior na resolução da relaxação contínua. Este parâmetro se é utilizado se o valor do parâmetro *in_starting_point_strategy* for MRQ_ESHP_SPS_CLOSED_TO_CONT_RELAX_SOL. Valor default: 0.025;

- **in_eps_to_interior_sol**: valor épsilon usado para considerar uma solução como interior. Valor default: 0.01;

- **in_eps_to_line_search**: tolerância utilizada para valor zero no procedimento de busca linear. Valor default: $1.0e-6$;

- **in_relative_tol_to_check_previous_sol**: tolerância relativa para habilitar a verificação de repetição de solução entre as iterações. Devido a problemas numéricos, em alguns casos, a solução da iteração anterior pode se repetir. Este parâmetro controla quando duas soluções devem ser testadas componente a componente se o valor da função objetivo de ambas estiver suficientemente próximo. Valor default: $1.0e-6$;

10.8.2 Parâmetros *inteiros* (int)

- **in_linearize_on_interior_sol**: flag que especifica se a solução interior deve ser utilizada também como ponto de linearização. Valor default: 0 (*false*);

- **in_max_lp_subiters**: número máximo de subiterações usando LP (para a resolução da relaxação contínua de (P)). Valor default: 500;

- **in_max_lp_subiter_to_improve_obj_app**: número máximo de subiterações usando LP (para a resolução da relaxação contínua de (P)) sem melhora de valor objetivo. Valor default: 50;

- **in_try_solve_interior_problem_if_cont_relax_fail**: flag que determina se o problema de obtenção de solução interior deve ser resolvido caso a solução interior não tenha podido ser obtida a partir da relaxação contínua de (P) modificada. Este parâmetro só é utilizado se o valor do parâmetro *in_starting_point_strategy* for MRQ_ESHP_SPS_CLOSED_TO_CONT_RELAX_SOL. Valor default: 1 (*true*).

10.8.3 Parâmetros *string* (str)/enumerativos

- **in_interior_point_strategy**: estratégia de obtenção de solução interior. Os possíveis valores para este parâmetro são:

  - MRQ_ESHP_IPS_MOST_INTERIOR: busca a solução mais interior;
  - MRQ_ESHP_IPS_CLOSED_TO_CONT_RELAX_SOL: busca uma solução que esteja próxima a da relaxação contínua de (P).

Valor default: MRQ_ESHP_IPS_MOST_INTERIOR;

- **in_lp_constr_linearisation_strategy**: estratégia de linearização de restrições nas subiterações baseadas em LP. Os valores possíveis para esse parâmetro são os mesmos para o parâmetro *in_constr_linearisation_strategy* da classe MRQ_LinearApproxAlgorithm. Valor default: MRQ_CLS_ONLY_INFEAS_AND_ACTIVE.

10.9 Parâmetros definidos na classe MRQ_BranchAndBound (HOABB)

A classe MRQ_BranchAndBound é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Hybrid Outer Approximation Branch-and-Bound.

10.9.1 Parâmetros *double* (dbl)

- **in_feas_heuristic_max_time**: tempo máximo (relógio), em segundos, para cada aplicação de heurística de viabilidade. Valor default: 30.0;

- **in_igma2_factor_to_max_dist_constr**: fator (entre 0 e 1) para determinação de vizinhança da heurística IGMA2. Valor default: 0.05;

- **in_igma2_factor_to_max_dist_constr_on_bin_vars**: estabelece fator (entre 0 e 1) para determinação de vizinhança com base em variáveis binárias da heurística IGMA2. Este parâmetro só é utilizado se o parâmetro *in_igma2_set_max_dist_constr_on_bin_vars* estiver com o valor 1 (*true*) Valor default: 0.05;

- **in_outer_app_subprob_time**: tempo máximo (relógio), em segundos, para cada aplicação de Outer Approximation em subproblemas na árvore de BB. Valor default: 10.0;

- **in_outer_app_time**: tempo máximo (relógio), em segundos, para cada aplicação de Outer Approximation no problema (P). Valor default: 10.0;

- **in_pseudo_cost_mu**: fator (entre 0 e 1) para cálculo de pseudo-custos. Quando a estratégia de branching está baseada em pseudo-custos, este fator faz um balanceamento para os cálculos do pseudo-custo de cada variável. O valor 0 privilegia variáveis com maior aumento de limite inferior em ambos os lados da ramificação, simultaneamente, ao passo que o valor 1 privilegia a variáveis com maior aumento de limite inferior em apenas um dos lados da ramificação. Valor default: 0.2.

10.9.2 Parâmetros *inteiros* (int)

- **in_call_after_solving_relax_callback**: flag que habilita a chamada a callback do usuário after_solving_relax, que pode ser especificada através do parâmetro *in_user_callbacks*. Esta callback é chamada após da resolução de cada relaxação em cada nó de BB e permite modificar o comportamento padrão do Branch-And-Bound, por exemplo habilitando podas adicionais (podas do usuário), executando heurísticas especializadas formuladas pelo usuário, ou mesmo modificando a solução apontada como ótima para as relaxações. Valor default: 0 (*false*);

- **in_call_before_solving_relax_callback**: flag que habilita a chamada a callback do usuário before_solving_relax, que pode ser especificada através do parâmetro *in_user_callbacks*. Esta callback é chamada antes da resolução de cada relaxação

em cada nó de BB e permite modificar o comportamento padrão do Branch-And-Bound, por exemplo habilitando podas adicionais (podas do usuário) ou modificando a relaxação a ser resolvida. Valor default: 0 (*false*);

- **in_call_new_best_solution_callback**: flag que habilita a chamada a callback do usuário new_best_solution, que pode ser especificada através do parâmetro *in_user_callbacks*. Esta callback é chamada após cada atualização da melhor solução conhecida pelo algoritmo, e permite, por exemplo, modificar a solução sendo atualizada ou executar procedimentos de busca local próprios do usuário sobre essa solução. Valor default: 0 (*false*);

- **in_consider_relax_infeas_if_solver_fail**: esta flag traz o seguinte comportamento ao BB: se a mesma estiver habilitada e o solver NLP falhar em resolver alguma relaxação, a mesma será considerada como inviável. Se a flag estiver desabilitada e essa mesma falha ocorrer, o algoritmo encerrará sua execução retornando um erro. Valor default: 1 (*true*);

- **in_count_total_prunes**: flag que habilita a contagem do número de podas realizadas pelo algoritmo, discriminando podas por inviabilidade, limite, otimalidade e podas do usuário (realizadas por meio de callbacks). Caso esta flag esteja ativa, a contagem de podas fica disponível no parâmetro de saída *out_prune_counter*. Valor default: 0 (*false*);

- **in_feas_heuristic_frequency**: frequência de aplicação de heurísticas de viabilidade. As heurísticas de viabilidade habilitadas são aplicadas de forma alternada a cada *in_int_feas_heuristic_frequency* iterações, podendo esta aplicação ser interrompida após a obtenção da primeira solução viável se o parâmetro *in_int_feas_heurs_strategy* estiver com o valor MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL. Valor default: 50000;

- **in_igma2_frequency**: frequência de aplicação da heurística IGMA2. IGMA2, se habilitada, é aplicada a cada *in_igma2_frequency* iterações, podendo esta aplicação ser interrompida após a obtenção da primeira solução viável se o parâmetro *in_igma2_strategy* estiver com o valor MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL. Valor default: 1;

- **in_igma2_set_max_dist_constr_on_bin_vars**: flag que habilita a vizinhança sobre variáveis binárias da heurística IGMA2. Valor default: 0 (*false*);

- **in_igma2_set_special_gap_min_solver_params**: flag que habilita o ajuste de parâmetros especiais para o solver usado na resolução dos problemas de minimização de gap de integralidade de IGMA2. Este ajuste visa tentar acelerar a execução do solver. Valor default: 1 (*true*);

- **in_igma2_solve_local_search_problem_even_on_non_int_sol**: flag que habilita a resolução do problema de busca local de IGMA2 mesmo sobre soluções não inteiras. Valor default: 1 (*true*);

- **in_lists_reorganization_frequency**: frequência de reorganização da estrutura de dados de armazenamento da lista de nós em aberto. A reorganização dessa estrutura ocorre a cada *in_lists_reorganization_frequency* iterações. Valor default: 10000;

- **in_number_of_branching_vars**: número máximo de variáveis adotadas no processo de ramificação. A cada ramificação, podem ser escolhidas até *in_number_of_branching_vars*

para particionar o espaço. Observe todavia, que se este parâmetro for ajustado com um valor $\bar{p}$, serão geradas $2^{\bar{p}}$ subpartições da partição corrente. Valor default: 1;

- **in_number_of_node_sublists**: número de sublistas na estrutura de dados que armazena a lista de nós em aberto. Valor default: 100;

- **in_max_tree_level_to_count_prunes_by_level**: este parâmetro define contagem de podas na árvore de BB por nível. Serão contadas podas realizadas até o nível determinado por este parâmetro. Após a execução do algoritmo, a contagem de podas por nível se torna disponível no array parâmetro de saída *out_prune_counter_by_level*, onde a i-ésima posição do array se refere a contagem do i-ésimo nível. Valor default: 0;

- **in_outer_app_frequence**: frequência de aplicação de Outer Approximation. O algoritmo Outer Approximation, se habilitado, será aplicado no problema (P) a cada *in_outer_app_frequence* iterações. Valor default: 10000;

- **in_outer_app_subprob_frequence**:frequência de aplicação de Outer Approximation como heurística em subproblemas. O algoritmo Outer Approximation será aplicado em subproblemas, se habilitado, a cada *in_outer_app_subprob_frequence* iterações. Valor default: 100;

- **in_rounding_heuristic_call_iter_frequence**: frequência de aplicação da heurística de arredondamento. Esta heurística, se habilitada, é aplicada a cada *in_rounding_heuristic_call_* iterações. Valor default: 10000;

- **in_seed_to_random_numbers**: semente para geração de números pseudo-aleatórios. Valor default: 1986;

- **in_stop_multibranch_after_first_bound_prune**: esta flag, se ativa, é responsável por desabilitar a ramificação sobre múltiplas variáveis após a primeira poda por limite. Valor default: 1 (*true*);

- **in_use_dual_obj_to_bound_prunning**: esta flag, se ativa, faz com o valor objetivo dual seja usado para poda por limite. Caso não esteja ativa, o valor primal é utilizado. Valor default: 0 (*false*);

- **in_use_feas_heuristic_diving**: flag que habilita o uso da heurística Diving. Valor default: 1 (*true*);

- **in_use_feas_heuristic_fp**: flag que habilita o uso da heurística Feasibility Pump. Valor default: 1 (*true*);

- **in_use_feas_heuristic_oafp**: flag que habilita o uso da heurística OA Feasibility Pump. Valor default: 1 (*true*);

- **in_use_feas_heuristic_rens**: flag que habilita o uso da heurística RENS. Valor default: 1 (*true*);

- **in_use_outer_app**: flag que habilita o uso do algoritmo Outer Approximation no BB. Se esta flag estiver desativada, juntamente com a flag *in_use_outer_app_as_heuristic*, o algoritmo se torna um Branch-And-Bound puro. Valor default: 1 (*true*);

- **in_use_outer_app_as_heuristic**: flag que habilita o uso do algoritmo Outer Approximation como heurística em subproblemas. Se esta flag estiver desativada, juntamente com a flag *in_use_outer_app*, o algoritmo trabalha como um Branch-And-Bound puro. Valor default: 1 (*true*).

10.9.3 Parâmetros *string* (str)/enumerativos

- **in_branching_strategy**: determina estratégia de branching. Define as variáveis selecionadas para ramificação. Possíveis valores para esse parâmetro são:
  - MRQ_BB_BS_HIGHEST_INT_GAP: escolhe variáveis com maior gap de integralidade;
  - MRQ_BB_BS_BIN_FIRST_HIGHEST_INT_GAP: escolhe variáveis com maior gap de integralidade com prioridade para variáveis binárias. isto significa que variáveis não binárias apenas são selecionadas quando todas as variáveis binárias possuem valor inteiro na solução da relaxação corrente;
  - MRQ_BB_BS_STBRANCH_PSEUDO_COSTS: escolhe variáveis usando estratégia baseada em strong branching e pseudocustos segundo [5];
  - MRQ_BB_BS_VAR_PRIORITIES: escolhe variáveis segundo prioridades definidas pelo usuário antes da execução do algoritmo (ainda não implementado);
  - MRQ_BB_BS_USER_INDEX_CHOICE: chama callback definida pelo usuário para determinar, dinamicamente, a escolha das variáveis para realização da ramificação. Esta callback deve ser provida através do parâmetro *in_user_callbacks*;
  - MRQ_BB_BS_USER_NODE_GENERATION: chama callback definida pelo usuário para particionar, dinamicamente, o espaço. Aqui, os novos nós da enumeração de BB são gerados diretamente pelo usuário. Esta callback deve ser provida através do parâmetro *in_user_callbacks*;

  Valor default: MRQ_BB_BS_STBRANCH_PSEUDO_COSTS;

- **in_exp_strategy**: estabelece estratégia de exploração. Possíveis valores para esse parâmetro são listados na Tabela 4: A estratégia profundidade/melhor limite usa

Tabela 4: Constantes de estratégias de exploração de Branch-and-Bound

Estratégia	**Constante**
profundidade	MRQ_BB_ES_DEPTH
largura	MRQ_BB_ES_WIDTH
melhor limite	MRQ_BB_ES_BEST_LIMIT
profundidade/melhor limite	MRQ_BB_ES_DEPTH_BEST_LIMIT

  profundidade até a primeira solução viável para (P) ser encontrada, passando então à estratégia de melhor limite. Valor default: MRQ_BB_ES_BEST_LIMIT;

- **in_int_feas_heurs_strategy**: especifica estratégia de uso de heurísticas de viabilidade. A heurística IGMA2 não é afetada por esse parâmetro, tendo seu uso regulado pelo parâmetro *in_igma2_strategy*. Possíveis valores para esse parâmetro são:
  - MRQ_BB_IHS_NO_HEURISTICS: não usa heurísticas de viabilidade;
  - MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL: usa heurísticas de viabilidade até a obtenção da primeira solução viável para (P);

– MRQ_BB_IHS_ALWAYS: usa heurístcias de viabilidade periodicamente até o final da execução do algoritmo.

Valor default: MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL;

- **in_igma2_gap_min_solver**: determina solver NLP para uso na heurística de viabilidade IGMA2. Os valores possíveis para esse parâmetro são fornecidos na Tabela 3. Valor default: MRQ_NLP_KNITRO;

- **in_igma2_strategy**: especifica estratégia para execução da heurística de viabilidade IGMA2. Os valores possíveis são os mesmos do parâmetro *in_int_feas_heurs_strategy*. Valor default: MRQ_BB_IHS_UNTIL_FIRST_FEAS_SOL;

- **in_parent_sol_storing_strategy**: estabelece a estratégia de armazenamento da solução do pai de cada nó da árvore de enumeração. Armazenar a solução do pai de cada nó permite o uso de técnicas de warm start ao custo de aumentar a demanda por memória. Se a estratégia de pseudo custos estiver sendo usada como estratégia de branching, o algoritmo precisará armazenar ao menos a solução primal do pai de cada nó. Os valores possíveis para este parâmetro são:

  – MRQ_BB_PSSS_NO_STORING: não armazena solução do pai;
  – MRQ_BB_PSSS_ONLY_PRIMAL: armazena apenas a solução primal do pai. Esta opção já permite o uso de warm start na resolução de cada relaxação, todavia, espera-se um melhor resultado com o uso simultâneo da solução primal e dual;
  – MRQ_BB_PSSS_PRIMAL_AND_DUAL: armazena apenas a solução dual do pai.

  Valor default: MRQ_BB_PSSS_NO_STORING;

- **in_rounding_heuristic_strategy**: define a estratégia de uso de heurística de arredondamento. Os possíveis valores para este parâmetro são:

  – MRQ_RS_NO_ROUNDING: não utiliza arredondamento;
  – MRQ_RS_NEAREST_INTEGER: arredonda cada variável inteira para o valor inteiro mais próximo;
  – MRQ_RS_PROBABILISTIC: arredonda cada variável inteira de modo aleatório-probabilístico, onde a parte fracionária do valor da variável dará a probabilidade da mesma ser arredondada para cima. Por exemplo, se uma variável inteira adquirir valor 3.25 na solução da relaxação corrente, então esta variável terá 25% de chance de ser arredondada para cima e 75% de chance de ser arredondada para baixo;

  Valor default: MRQ_RS_PROBABILISTIC.

10.10 Parâmetros definidos na classe MRQ_ContinuousRelax (CR)

A classe MRQ_ContinuousRelax é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para a resolução de relaxação contínua do problema (P). Note que esta classe não se refere a um algoritmo de MINLP.

10.10.1 Parâmetros *inteiros* (int)

- **in_set_integer_vars_as_integers**: flag que especifica se as variáves inteiras de (P) devem ser consideradas como inteiras pelo solver NLP, caso o mesmo suporte essa operação. Através deste parâmetro, é possível usar solvers, como Knitro, por exemplo, para a resolução completa de (P) através da API de Muriqui. Valor default: 0 (*false*);

10.11 Parâmetros definidos na classe MRQ_IGMA1 (IGMA1)

A classe MRQ_IGMA1 é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Integrality Gap Minimization Heuristic 1.

10.11.1 Parâmetros *double* (dbl)

- **in_absolute_obj_cut_eps**: épsilon absoluto para restrição de corte objetivo nos problemas de minimização de gap. Valor default: $1.0e-4$;
- **in_absolute_obj_tol_to_active_obj_cut**: tolerância absoluta para verificação de corte de nível objetivo ativo. Valor default: $1.0e-3$;
- **in_factor_to_increase_abs_obj_cut_eps**: fator de incremento do épsilon absoluto para restrição de corte objetivo nos problemas de minimização de gap. Este fator é utilizado nos casos em que o corte objetivo, devido a problemas numéricos, não foi capaz de cortar a melhor solução conhecida da região viável. Valor default: 1.5;
- **in_factor_to_increase_abs_obj_cut_eps_on_zero**: fator de incremento do épsilon absoluto para restrição de corte objetivo nos problemas de minimização de gap. Este fator é utilizado nos casos em que o corte objetivo, devido a problemas numéricos, não foi capaz de cortar a melhor solução conhecida da região viável e o valor objetivo é zero. Valor default: 100.0;
- **in_factor_to_increase_rel_obj_cut_eps**: fator de incremento do épsilon relativo para restrição de corte objetivo nos problemas de minimização de gap. Este fator é utilizado nos casos em que o corte objetivo, devido a problemas numéricos, não foi capaz de cortar a melhor solução conhecida da região viável. Valor default: 1.5;
- **in_factor_to_min_gap_obj_by_avg_gap**: fator utilizado para multiplicar a média dos gaps na função objetivo dos problemas de minimização do gap. Este parâmetro só tem efeito se o parâmetro *in_gap_min_obj_strategy* estiver ajustado com o valor MRQ_IGMA1_GMOS_BY_GAP_AVERAGE. Valor default: 10.0;
- **in_lower_bound_to_random_sol**: limitante inferior artificial para determinação de solução inicial pseudo-aleatória para os problemas de minimização de gap de integralidade. Valor default: -100.0;
- **in_relative_obj_cut_eps**: épsilon relativo para restrição de corte objetivo nos problemas de minimização de gap de integralidade. Valor default: $1.0e-3$;
- **in_relative_obj_tol_to_active_obj_cut**: tolerância relativa para verificação de corte de nível objetivo relativo. Valor default: $1.0e-3$;
- **in_upper_bound_to_random_sol**: limitante superior artificial para determinação de solução inicial pseudo-aleatória para os problemas de minimização de gap de integralidade. Valor default: 100.0;

10.11.2 Parâmetros *inteiros* (int)

- **in_adopt_obj_cut**: flag que habilita a adoção do corte de nível objetivo. Este parâmetro só é utilizado quando o algoritmo está em modo heurístico. Valor default: 1 (*true*);

- **in_consider_relax_infeas_if_solver_fail**: quando esta flag está ativa, problemas onde o solver NLP falha em fornecer alguma resposta são considerados inviáveis. Se esta mesma situação ocorrer com esta flag inativa, o algoritmo será interrompido e um código de erro será retornado. Valor default: 1 (*true*);

- **in_heuristic_mode**: flag que habilita execução em modo heurístico. Valor default: 0 (*false*);

- **in_lists_reorganization_frequency**: frequência de reorganização da estrutura de dados de armazenamento da lista de nós em aberto. A reorganização dessa estrutura ocorre a cada *in_lists_reorganization_frequency* iterações, se habilitada. Valor default: 10000;

- **in_max_improvments_of_best_sol**: número máximo de melhoras da melhor solução. Este parâmetro define um critério de parada adicional que é especialmente útil para execuções em modo heurístico. Valor default: UINT_MAX;

- **in_max_nonimprovment_integer_sols**: número máximo de soluções inteiras obtidas sem melhora da melhor solução. Este parâmetro só é utilizado quando o corte de nível objetivo está desabilitado (parâmetro *in_adopt_obj_cut*). Valor default: 10;

- **in_reorganize_lists**: flag que habilita a reorganização periódica da lista de nós em aberto. Valor default: 1 (*true*);

- **in_seed_to_random_numbers**: semente para geração de números pseudo-aleatórios. Valor default: 1986;

- **in_set_special_gap_min_solver_params**: flag que habilita o ajuste de parâmetros especiais para o solver usado na resolução dos problemas de minimização de gap de integralidade. Este ajuste visa tentar acelerar a execução do solver. Valor default: 1 (*true*);

- **in_use_general_feas_heuristics**: flag que habilita o uso de heurísticas de viabilidade. Valor default: 1 (*true*);

- **in_use_random_initial_sols**: flag que habilita o uso de soluções iniciais pseudo-aleatórias para a resolução dos problemas de minimização de gap de integralidade. Valor default: 1 (*true*);

10.11.3 Parâmetros *string* (str)/enumerativos

- **in_exp_strategy**: define a estratégia de exploração na árvore de BB. Os possíveis valores para este parâmetro são apresentados na Tabela 4. Valor default: MRQ_BB_ES_BEST_LIMIT;

- **in_gap_min_obj_strategy**: especifica a estratégia de construção da função objetivo dos problemas de minimização de gap. Os valores possíveis para este parâmetro são:

  - MRQ_IGMA1_GMOS_SAME_WEIGHT: Todas as variáveis binárias recebem o mesmo peso na função objetivo;
  - MRQ_IGMA1_GMOS_BY_GAP_AVERAGE: O peso de cada variável é determinado em função da média do gap de integralidade nas iterações anteriores.

- **in_gap_min_solver**: estabelece o solver NLP usado para a resolução dos problemas de minimização de gap. Os valores admissíveis para esse parâmetro são apresentados na Tabela 3. Valor default: MRQ_NLP_KNITRO.

10.12 Parâmetros definidos na classe MRQ_IGMA2 (IGMA2)

A classe MRQ_IGMA2 é derivada de MRQ_BranchAndBound, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Integrality Gap Minimization Heuristic 2.

10.12.1 Parâmetros *double* (dbl)

- **in_factor_to_max_dist_constr**: fator (entre 0 e 1) para determinação de vizinhança. Valor default: 0.05;

- **in_factor_to_max_dist_constr_on_bin_vars**: fator (entre 0 e 1) para determinação de vizinhança com base em variáveis binárias. Este parâmetro só é utilizado se o parâmetro *in_set_max_dist_constr_on_bin_vars* estiver com o valor 1 (*true*). Valor default: 0.05;

- **in_percentual_to_rectangular_neighborhood**: percentual das caixas de variáveis para construção de vizinhança retangular. Este parâmetro só é utilizado se o parâmetro *in_neighborhood_strategy* estiver com o valor MRQ_IGMA2_NS_RECTANGULAR. Valor default: 0.05;

10.12.2 Parâmetros *inteiros* (int)

- **in_set_max_dist_constr_on_bin_vars**: flag que habilita construção de restrição de vizinhança apenas sobre variáveis binárias. Valor default: 0 (*false*);

- **in_set_special_gap_min_solver_params**: flag que habilita o ajuste de parâmetros especiais para o solver usado na resolução dos problemas de minimização de gap de integralidade. Este ajuste visa tentar acelerar a execução do solver. Valor default: 1 (*true*);

- **in_solve_local_search_problem_even_on_non_int_sol**: flag que habilita a resolução do problema de busca local mesmo sobre soluções não inteiras. Valor default: 1 (*true*);

10.12.3 Parâmetros *string* (str)/enumerativos

- **in_gap_min_solver**: estabelece o solver NLP usado para a resolução dos problemas de minimização de gap. Os valores admissíveis para esse parâmetro são apresentados na Tabela 3. Valor default: MRQ_NLP_KNITRO;

- **in_neighborhood_strategy**: estratégia de formulação de vizinhança no problema de minimização de gap de integralidade. Os valores possíveis para este parâmetro são:

– MRQ_IGMA2_NS_RECTANGULAR: vizinhança (hiper) retangular em torno da solução corrente;
– MRQ_IGMA2_NS_SPHERIC: vizinhança (hiper) esférica em torno da solução corrente;

10.13 Parâmetros definidos na classe MRQ_FeasibilityPump (FP)

A classe MRQ_FeasibilityPump é derivada de MRQ_Heuristic, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Feasibility Pump.

10.13.1 Parâmetros *double* (dbl)

- **in_lower_bound_to_pi**: limitante inferior para o parâmetro π, usado para chacoalhar (flip) a solução quando ciclo é detectado. Valor default: -0.3;
- **in_upper_bound_to_pi**: limitante superior para o parâmetro π, usado para chacoalhar (flip) a solução quando ciclo é detectado. Valor default: 0.7;

10.13.2 Parâmetros *inteiros* (int)

- **in_last_iters_considered_to_cycles**: número de iterações anteriores consideradas na verificação de ciclagem de soluções. Valor default: 5;
- **in_max_cycle_subiters**: número máximo de tentativas de quebra de ciclos consecutivas. Se for detectado um número de tentativas de quebra de ciclos consecutivas superior a *in_max_cycle_subiters*, o algoritmo pára retornando um erro. Valor default: 20;
- **in_set_linear_obj_term_on_bin_vars_at_nlp**: flag que habilita o uso de termo linear no problema NLP de viabilidade para as variáveis binárias. Este termo linear calcula a distância entre a solução do problema e a solução de entrada. Caso esta flag esteja desativada, as variáveis binárias terão termos para o cálculo da distância como as demais variáveis, que pode envolver elementos mais pesados no problema NLP. Valor default: 1 (*true*);
- **in_set_norm1_on_nlp**: flag que habilita uso da norma 1 no lugar da norma 2 no problema NLP de viabilidade. Valor default: 1 (*true*).

10.14 Parâmetros definidos na classe MRQ_OAFeasibilityPump (OAFP)

A classe MRQ_OAFeasibilityPump é derivada de MRQ_LinearApproxAlgorithm, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Outer Approximation FeasibilityPump.

10.14.1 Parâmetros *inteiros* (int)

- **in_set_linear_obj_term_on_bin_vars_at_nlp**: flag que habilita o uso de termo linear no problema NLP de viabilidade para as variáveis binárias. Este termo linear calcula a distância entre a solução do problema e a solução de entrada. Caso esta flag esteja desativada, as variáveis binárias terão termos para o cálculo da distância como as demais variáveis, que pode envolver elementos mais pesados no problema NLP. Valor default: 1 (*true*);

- **in_set_norm1_on_nlp**: flag que habilita uso da norma 1 no lugar da norma 2 no problema NLP de viabilidade. Valor default: 1 (*true*);

- **in_solve_nlp_as_local_search_at_end**: flag que especifica se um problema NLP de busca local deve ser resolvido caso o algoritmo encontre alguma solução viável. Valor default: 1 (*true*).

10.15 Parâmetros definidos na classe MRQ_Diving (Diving)

A classe MRQ_Diving é derivada de MRQ_Heuristic, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Diving.

10.15.1 Parâmetros *double* (dbl)

- **in_percentual_of_add_var_fixing**: percentual de variáveis adicionais sendo fixadas na simulação de ramificação. Valor default: 0.2.

10.15.2 Parâmetros *inteiros* (int)

- **in_consider_relax_infeas_if_solver_fail**: quando esta flag está ativa, problemas onde o solver NLP falha em fornecer alguma resposta são considerados inviáveis. Se esta mesma situação ocorrer com esta flag inativa, o algoritmo será interrompido e um código de erro será retornado. Valor default: 1 (*true*).

10.15.3 Parâmetros *string* (str)/enumerativos

- **in_dive_selec_strategy**: estratégia de seleção de variáveis para simulação de ramificação. Os valores possíveis para esse parâmetro são:

  - MRQ_DIVE_SS_FRACTIONAL: escolha pela variável com maior gap de integralidade;
  - MRQ_DIVE_SS_VECTORLENGHT: escolha pelo comprimento do vetor, que procura levar em conta a variação na função objetivo e número de restrições sendo afetadas.

  Valor default: MRQ_DIVE_SS_FRACTIONAL.

10.16 Parâmetros definidos na classe MRQ_RENS (RENS)

A classe MRQ_RENS é derivada de MRQ_Heuristic, herdando assim seus parâmetros e métodos. Os parâmetros definidos aqui são específicos para o algoritmo Relaxation Enforced Neighborhood Search.

10.16.1 Parâmetros *double* (dbl)

- **in_continuous_neighborhood_factor**: fator para definição de vizinhança de variáveis contínuas. Valor default: 0.25;

- **in_integer_neighborhood_factor**: fator para definição de vizinhança de variáveis inteiras. Valor default: 0.25.

10.16.2 Parâmetros *inteiros* (int)

- **in_apply_only_heuristics_on_subproblems**: flag que restringe a aplicação somente de heurísticas de viabilidade no subproblema de MINLP. Valor default: 0 (*false*).

10.16.3 Parâmetros *string* (str)/enumerativos

- **in_algorithm_to_solve_subproblem**: especifica algoritmo para resolução de subproblema de MINLP. Os valores possíveis para este parâmetro são os descritos na Tabela 1 (com exceção de MRQ_RENS, é claro).

11 Parâmetros de saída (saída dos algoritmos)

Os parâmetros de saída contemplam a resposta fornecida pela aplicação de um algoritmo (status de execução, solução, valor objetivo) a um problema de MINLP, bem como dados referentes a execução (número de iterações, tempo de cpu, etc). Se por um lado os parâmetros de entrada tem o prefixo *in_* em seu nome, por outro lado, os nomes dos parâmetros de saída são prefixados por *out_*. Os parâmetros de saída são ajustados pela aplicação de algum dos algoritmos de Muriqui (método run na API C++).

11.1 Parâmetros de saída definidos na classe MRQ_Algorithm

Os parâmetros de saída definidos na classe MRQ_Algorithm se aplicam aos algoritmos implementados por Muriqui em geral. Parâmetros específicos de cada abordagem são definidos nas classes derivadas. Os parâmetros de saída definidos nessa classe são:

- **out_feasible_solution**: flag ajustada para o valor *true* se a execução do algoritmo foi capaz de encontrar uma solução viável para (P) e *false*, caso contrário;
- **out_number_of_feas_sols**: número de soluções viáveis encontradas na execução do algoritmo;
- **out_number_of_iterations**: número geral de iterações executadas pelo algoritmo;
- **out_number_of_threads**: número de threads de execução utilizadas diretamente pelo algoritmo. É válido destacar que é possível que um algoritmo, como por exemplo OA ou ECP, utilize uma única thread de execução direta, mas, o processo de resolução de subproblemas de MILP utilize mais de uma thread. O número total de threads utilizado de forma direta ou indireta pode ser controlado através do parâmetro de entrada *in_number_of_threads*.
- **out_return_code**: código de retorno do algoritmo. Os valores possíveis para esse parâmetro enumerativo, juntamente com seus respectivos significados são:
  - MRQ_OPTIMAL_SOLUTION: solução ótima encontrada;
  - MRQ_INFEASIBLE_PROBLEM: problema inviável;
  - MRQ_UNBOUNDED_PROBLEM: problema ilimitado;
  - MRQ_ALG_NOT_APPLICABLE: algoritmo não aplicável;
  - MRQ_BAD_DEFINITIONS: definições inconsistentes no problema ou parâmetros de entrada;

- MRQ_BAD_PARAMETER_VALUES: valores inadequados para um ou mais parâmetros;
- MRQ_INDEX_FAULT: uso de índice inválido para alguma estrutura de dados;
- MRQ_MEMORY_ERROR: erro de memória;
- MRQ_UNDEFINED_ERROR: erro indefinido;
- MRQ_MAX_TIME_STOP: parada por limite máximo de tempo (relógio, ou cpu) alcançado;
- MRQ_MAX_ITERATIONS_STOP: parada por limite máximo de iterações alcançado;
- MRQ_CALLBACK_FUNCTION_ERROR: erro retornado por alguma função callback do usuário usada para avaliação das funções não lineares;
- MRQ_STOP_REQUIRED_BY_USER: função callback de acompanhamento de algoritmo solicitou a parada do mesmo;
- MRQ_MILP_SOLVER_ERROR: erro na execução de solver MILP;
- MRQ_NLP_SOLVER_ERROR: erro na execução de solver NLP;
- MRQ_LIBRARY_NOT_AVAILABLE: biblioteca não disponível. Esse erro ocorre quando algum algoritmo está configurado para usar uma biblioteca que não estava disponível na compilação de Muriqui. Por exemplo, se Muriqui for compilado apenas usando Cplex como solver MILP, esse erro será obtido se usuário tentar executar qualquer algoritmo que demande a resolução de poblemas de MILP e um solver diferente de Cplex for especificado;
- MRQ_NAME_ERROR: erro de nome. Usado, por exemplo, em funções que realizam ajuste de parâmetro por nome e valor;
- MRQ_VALUE_ERROR: erro de valor. Usado, por exemplo, em funções que realizam ajuste de parâmetro por nome e valor;
- MRQ_MINLP_SOLVER_ERROR: erro na execução de solver MINLP. Até o presente momento, este código de retorno está sem uso;
- MRQ_INITIAL_SOLUTION_ERROR: erro de solução inicial;
- MRQ_HEURISTIC_FAIL: código de retorno utilizado por algoritmos heurísticos para designar falha no processo de busca por solução viável para (P);
- MRQ_LACK_OF_PROGRESS_ERROR: erro de falta de progresso;
- MRQ_NLP_NO_FEASIBLE_SOLUTION: não foi possível obter solução viável para uma relaxação contínua de (P);
- MRQ_NONIMPLEMENTED_ERROR: funcionalidade ainda não implementada;
- MRQ_HEURISTIC_SUCCESS: código de retorno utilizado por algoritmos heurísticos para designar sucesso no processo de busca por solução viável para (P);
- MRQ_CONT_RELAX_OPTIMAL_SOLUTION: solução ótima na resolução de relaxação contínua de (P).

Para obtenção de uma string a partir do código de retorno é útil utilizar a função

```
std::string MRQ_getStatus(int returnCode);
```

presente na API;

- **out_algorithm**: código do algoritmo utilizado para a resolução do problema. Os possíveis valores para este parâmetro são os descritos na Tabela 1;
- **out_clock_time**: tempo de relógio decorrido durante a execução do algoritmo;
- **out_cpu_time**: tempo de processamento total utilizado pelo algoritmo em todas as threads de execução;
- **out_cpu_time_to_fisrt_feas_sol**: tempo de cpu para obtenção da primeira solução viável;
- **out_lower_bound**: melhor limite inferior explicitamente calculado pelo algoritmo;
- **out_upper_bound**: melhor limite superior obtido;
- **out_obj_opt_at_continuous_relax**: valor objetivo ótimo da relaxação contínua de $(\overline{P})$;
- **out_sol_hist**: histórico de soluções. Se o parâmetro de entada *in_store_history_solutions* estiver ajustado em *true*, este objeto conterá um histórico com todas as soluções viáveis de melhora encontradas pelo algoritmo ao longo de sua execução;
- **out_best_sol**: array com a melhor solução encontrada. A dimensão do array é o número de variáveis de (P). O valor deste parâmetro só é considerado válido se o parâmetro de saída *out_feasible_solution* estiver com o valor *true*, e por isso recomendamos o teste sobre esse último parâmetro antes da tentativa de acesso a out_best_sol. Note que este parâmetro se remete a um ponteiro que pode estar nulo, caso ocorra erro de alocação de memória, ou o método run não tenha sido chamado pela primeira vez;
- **out_best_obj**: valor da função objetivo na melhor solução encontrada. O valor deste parâmetro só é considerado válido se o parâmetro de saída *out_feasible_solution* estiver com o valor *true*.

11.2 Parâmetros de saída definidos na classe MRQ_LinearApproxAlgorithm

A classe MRQ_LinearApproxAlgorithm é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros apresentados aqui se aplicam a algoritmos de aproximação linear em geral. Os parâmetros de saída definidos nessa classe são:

- **out_number_of_nlp_probs_solved**: número total de problemas de NLP resolvidos;
- **out_number_of_milp_solver_iters**: número total de iterações do solver MILP;
- **out_clock_time_of_nlp_solving**: tempo de relógio total transcorrido nas resoluções de problemas de NLP;
- **out_cpu_time_of_nlp_solving**: tempo de processamento total transcorrido nas resoluções de problemas de NLP;

11.3 Parâmetros de saída definidos na classe MRQ_Heuristic

A classe MRQ_Heuristic é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros apresentados aqui se aplicam a algoritmos heurísticos em geral. Os parâmetros de saída definidos nessa classe são:

- **out_seed_to_random_numbers**: semente utilizada pelo algoritmo para a geração de número aleatórios;

11.4 Parâmetros de saída definidos na classe MRQ_ExtSupHypPlane

A classe MRQ_ExtSupHypPlane é derivada de MRQ_LinearApproxAlgorithm, herdando assim seus parâmetros e métodos.

- **out_number_of_lp_iterations**: número total de iterações na fase LP;

11.5 Parâmetros de saída definidos na classe MRQ_BranchAndBound

A classe MRQ_BranchAndBound é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros de saída definidos nessa classe são:

- **out_nlp_failure_in_some_relaxation**: flag que assume o valor *true* se, em alguma relaxação contínua de alguma partição, o solver NLP falhou em fornecer solução ótima. Caso contrário, essa flag, assume o valor *false*;

- **out_seed_to_random_numbers**: semente utilizada pelo algoritmo para a geração de número aleatórios;

- **out_user_callback_error_code**: caso o algoritmo tenha parado devido a erro retornado por função callback de usuário, esse parâmetro conterá o valor retornado pela respectiva função responsável pela parada;

- **out_number_of_open_nodes**: número de nós em aberto no momento em que o algoritmo parou. Útil quando o algoritmo pára devido a algum critério diferente da otimalidade como, por exemplo, tempo máximo de processamento, número máximo de iterações, erro de função callback ou parada solicitada por callback e usuário;

- **out_best_sol_iter**: iteração onde a melhor solução foi encontrada;

- **out_prune_counter**: contador geral de podas realizadas pelo algoritmo. Este objeto contém contadores para quatro tipos de podas: otimalidade, limite, inviabilidade e podas do usuário (realizadas por meio de função callback). Para que a contagem de podas efetivamente ocorra, é preciso que o parâmetro de entrada *in_count_total_prunes* esteja ajustado com o valor *true* antes da execução do algoritmo;

- **out_prune_counter_by_level**: contador de podas por nível realizadas pelo algoritmo. Note que este parâmetro é um array onde a i-ésima posição contém a contagem de podas no i-ésimo nível. Para cada nível, há um objeto com contadores para quatro tipos de podas: otimalidade, limite, inviabilidade e podas do usuário (realizadas por meio de função callback). É válido ressaltar que as podas são contadas apenas nos primeiros *in_max_tree_level_to_count_prunes_by_level* níveis, onde *in_max_tree_level_to_count_prunes_by_level* é um parâmetro de entrada definido antes da execução do algoritmo.

11.6 Parâmetros de saída definidos na classe MRQ_ContinuousRelax

A classe MRQ_ContinuousRelax é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros de saída definidos nessa classe são:

- **out_nlp_feasible_solution**: flag que assume o valor *true* se a execução do algoritmo encontrou uma solução viável para a relaxação contínua o problema de entrada, e *false* caso contrário;
- **out_original_solver_return_code**: código de retorno originalmente retornado pelo solver de NLP adotado. Este valor varia conforme a escolha do solver NLP;
- **out_constraint_values**: valores das restrições do problema na solução retornada pelo algoritmo. Para o correto uso desse parâmetro de saída, é necessário verificar o parâmetro *out_nlp_feasible_solution*;
- **out_dual_sol**: valores das soluções duais do problema referente a solução retornada pelo algoritmo. Para o correto uso desse parâmetro de saída, é necessário verificar o parâmetro *out_nlp_feasible_solution*;

11.7 Parâmetros de saída definidos na classe MRQ_IGMA1

A classe MRQ_IGMA1 é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros de saída definidos nessa classe são:

- **out_best_sol_iter**: iteração onde a melhor solução foi encontrada;
- **out_number_of_open_nodes**: número de nós em aberto no momento em que o algoritmo parou. Útil quando o algoritmo pára devido a algum critério diferente da otimalidade como, por exemplo, tempo máximo de processamento, número máximo de iterações, erro de função callback ou parada solicitada por callback e usuário;
- **out_number_of_inner_iterations**: número de iterações do laço interno de IGMA1;
- **out_number_of_prunes_by_obj_cut_active**: número de podas realizadas por corte de nível objetivo ativo (apenas no modo heurístico através do parâmetro dde entrada *in_heuristic_mode*);

11.8 Parâmetros de saída definidos na classe MRQ_IGMA2

A classe MRQ_IGMA2 é derivada de MRQ_Algorithm, herdando assim seus parâmetros e métodos. Os parâmetros de saída definidos nessa classe são:

- **out_feas_sol_on_gap_min_problem**: flag que assumirá o valor *true* se o algoritmo retornar solução viável e a mesma for obtida através a partir de solução viável retornada pelo problema de minimização de gap de integralidade. Caso contrário, esta flag assumirá o valor *false*;
- **out_feas_sol_found_by_igma2_procedure**: flag que assumirá o valor *true* se o algoritmo encontrou solução viável por meio dos procedimentos específicos de IGMA2. Caso contrário, esta flag assumirá o valor *false*. É válido ressaltar que uma solução viável pode ser obtida por este algoritmo por procedimentos não específicos de IGMA2, como arredondamento, ou obtenção de solução inteira por meio de resolução de relaxação.

11.9 Parâmetros de saída definidos na classe MRQ_RENS

A classe MRQ_RENS é derivada de MRQ_Heuristic, herdando assim seus parâmetros e métodos. Os parâmetros de saída definidos nessa classe são:

- **out_algorithm_to_solve_subproblem**: código de algoritmo utilizado para a resolução do subproblema MINLP de RENS. Os valores possíveis para este parâmetro são os mesmos do parâmetro *out_ algorithm* da classe MRQ_Algorithm (Seção 11.1).

Referências

[1] Timo Berthold. Rens. *Mathematical Programming Computation*, 6(1):33–54, Mar 2014.

[2] Pierre Bonami, Lorenz T. Biegler, Andrew R. Conn, Gérard Cornuéjols, Ignacio E. Grossmann, Carl D. Laird, Jon Lee, Andrea Lodi, François Margot, and Nicolas Sawaya. An algorithmic framework for convex mixed integer nonlinear programs. *Discrete Optimization*, 5(2):186–204, May 2008.

[3] Pierre Bonami, Gérard Cornuéjols, Andrea Lodi, and François Margot. A feasibility pump for mixed integer nonlinear programs. *Mathematical Programming*, 119:331–352, 2009. 10.1007/s10107-008-0212-2.

[4] Pierre Bonami and João Gonçalves. Heuristics for convex mixed integer nonlinear programs. *Computational Optimization and Applications*, pages 1–19, 2008. 10.1007/s10589-010-9350-6.

[5] Pierre Bonami, Jon Lee, Sven Leyffer, and Andreas Wächter. On branching rules for convex mixed-integer nonlinear optimization. *J. Exp. Algorithmics*, 18:2.6:2.1–2.6:2.31, November 2013.

[6] Marco Duran and Ignacio Grossmann. An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming*, 36:307–339, 1986. 10.1007/BF02592064.

[7] Roger Fletcher and Sven Leyffer. Solving mixed integer nonlinear programs by outer approximation. *Mathematical Programming*, 66:327–349, 1994. 10.1007/BF01581153.

[8] David M. Gay. Hooking your solver to ampl. Technical report, 1997.

[9] Jan Kronqvist, Andreas Lundell, and Tapio Westerlund. The extended supporting hyperplane algorithm for convex mixed-integer nonlinear programming. *Journal of Global Optimization*, 64(2):249–272, 2016.

[10] Wendel Melo, Marcia Fampa, and Fernanda Raupp. Integrating nonlinear branch-and-bound and outer approximation for convex mixed integer nonlinear programming. *Journal of Global Optimization*, 60(2):373–389, 2014.

[11] Wendel Melo, Marcia Fampa, and Fernanda Raupp. Integrality gap minimization heuristics for binary mixed integer nonlinear programming. *Journal of Global Optimization*, 71(3):593–612, Jul 2018.

[12] Wendel Melo, Marcia Fampa, and Fernanda Raupp. An overview of minlp algorithms and their implementation in muriqui optimizer. *Annals of Operations Research*, May 2018.

[13] Wendel Melo, Marcia Fampa, and Fernanda Raupp. Two linear approximation algorithms for convex mixed integer nonlinear programming. *Annals of Operations Research*, 2020.

[14] Ignacio Quesada and Ignacio E. Grossmann. An lp/nlp based branch and bound algorithm for convex minlp optimization problems. *Computers & Chemical Engineering*, 16(10-11):937 – 947, 1992. An International Journal of Computer Applications in Chemical Engineering.

[15] Tapio Westerlund and Frank Pettersson. An extended cutting plane method for solving convex minlp problems. *Computers & Chemical Engineering*, 19, Supplement 1(0):131 – 136, 1995. European Symposium on Computer Aided Process Engineering.