Solver de Busca Local para Problemas de Otimização Combinatória

Thiago A. de Queiroz
Luiz Henrique Cherri
Franklina M. B. Toledo

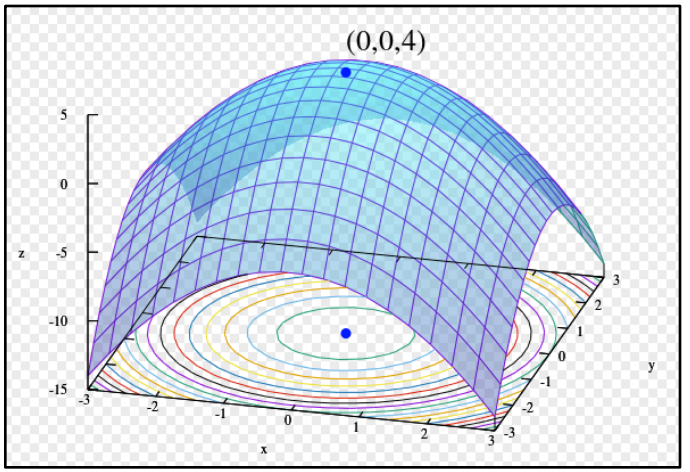
Sumário

- Otimização Combinatória;
- Busca Local;
- LocalSolver;
- Aplicações.

Otimização

- Usa métodos para encontrar as melhores alternativas conforme objetivos determinados.
- Nas ciências e engenharia consiste no estudo de problemas em que se busca minimizar ou maximizar uma função por meio da escolha de valores para as suas variáveis.
- Problemas de otimização são geralmente escritos na forma:

$$\begin{aligned} \text{Maximizar} \quad & f(\mathbf{x}) \\ \text{Sujeito a:} \quad & h_i(\mathbf{x}) = 0, \qquad i = 1, 2, \ldots, m \\ & g_j(\mathbf{x}) \leq 0, \qquad j = 1, 2, \ldots r \\ & \mathbf{x} \in S. \end{aligned}$$



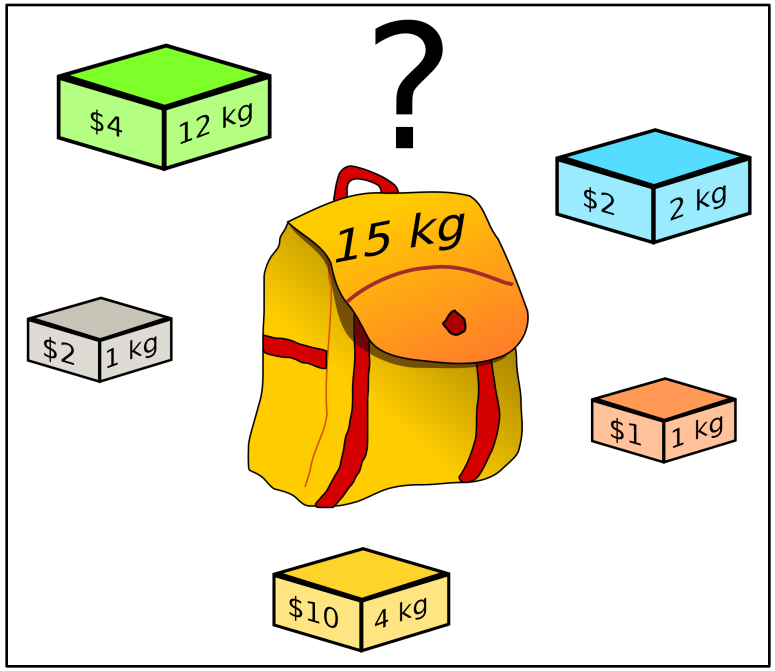
Fonte: https://en.wikipedia.org/wiki/Mathematical_optimization

Otimização

- As variáveis em *x* podem ser contínuas ou discretas, ao passo que o domínio dado em *S* pode ter finitas ou infinitas soluções.
- As soluções:
  - Viáveis: satisfazem as restrições;
  - Ótima: é viável e atende ao critério de otimização (mínima ou máxima).
- Problemas de Otimização Combinatória (OC) possuem o domínio formado por um conjunto discreto de soluções viáveis.

Otimização Combinatória

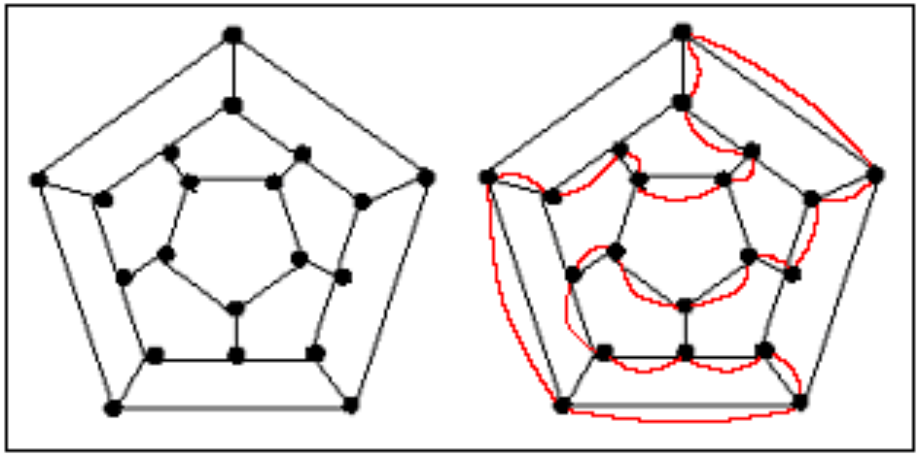
- [Problema da Mochila] Dada uma mochila capacitada e ***n*** objetos com valor e peso, determinar um subconjunto de objetos de máximo valor que respeita a capacidade da mochila.

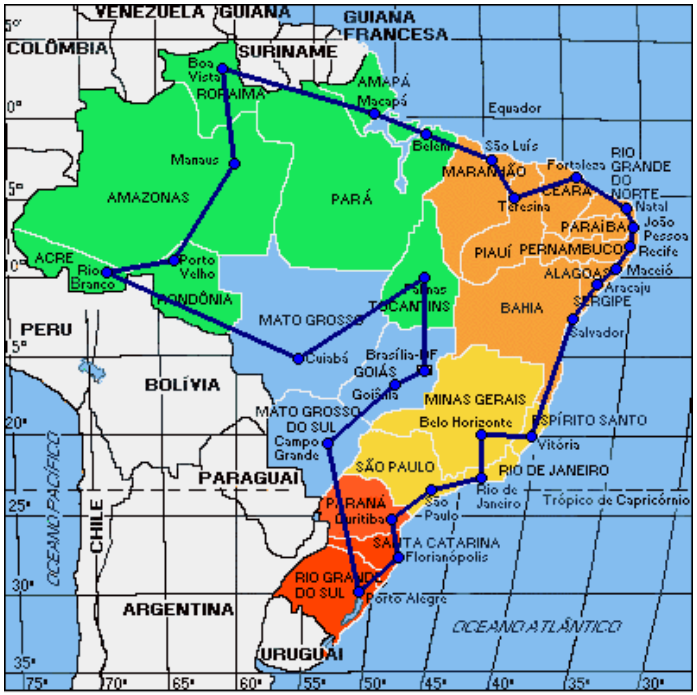


Fonte: https://en.wikipedia.org/wiki/Knapsack_problem

Otimização Combinatória

- [Problema do Caixeiro Viajante] Dado um conjunto de cidades, determinar a menor rota para percorrê-las visitando passar uma única vez em cada cidade e retornar a cidade de origem.





Fonte: http://www.siscorp.com.br/siscorpnews/sexta_edicao/assunto_ma rco.htm

Otimização Combinatória

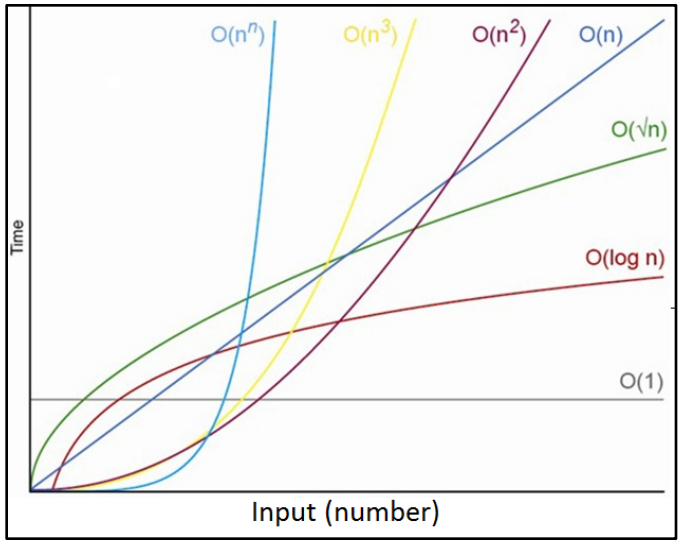
- Outros exemplos:
  - Alocar tarefas para pessoas;
  - Determinar quadro de horários em organizações;
  - Determinar a quantidade de itens a ser produzida em máquinas em um dado horizonte de tempo;
  - Localizar antenas em uma dada região geográfica;
  - Cortar placas para obter objetos menores;
  - Escalonar tarefas em uma ou várias máquinas;
  - Etc.

Métodos e Complexidade

- O grau de dificuldade com que se resolve o problema expressa a sua complexidade computacional:
  - A complexidade mede a qualidade dos métodos de resolução desenvolvidos.
- A medida mais comum é a complexidade de tempo, no caso pessimista, em função do tamanho da entrada ***n***. Um método é:
  - **Eficiente**: complexidade polinomial em função de *n*;
  - **Não eficiente**: complexidade não polinomial, geralmente exponencial (ou fatorial) em função de *n*.

Métodos e Complexidade

- Métodos para resolver problemas de otimização:
  - Exatos: retornam a solução ótima e geralmente trabalham sobre busca em árvores (enumera e poda);
  - Algoritmos aproximados: garantem uma solução que respeita uma certa distância máxima da ótima;
  - Heurísticas: fornecem uma solução (viável) sem qualquer garantia da sua qualidade.



Fonte: http://codedreaming.com/algorithm-time-complexity/

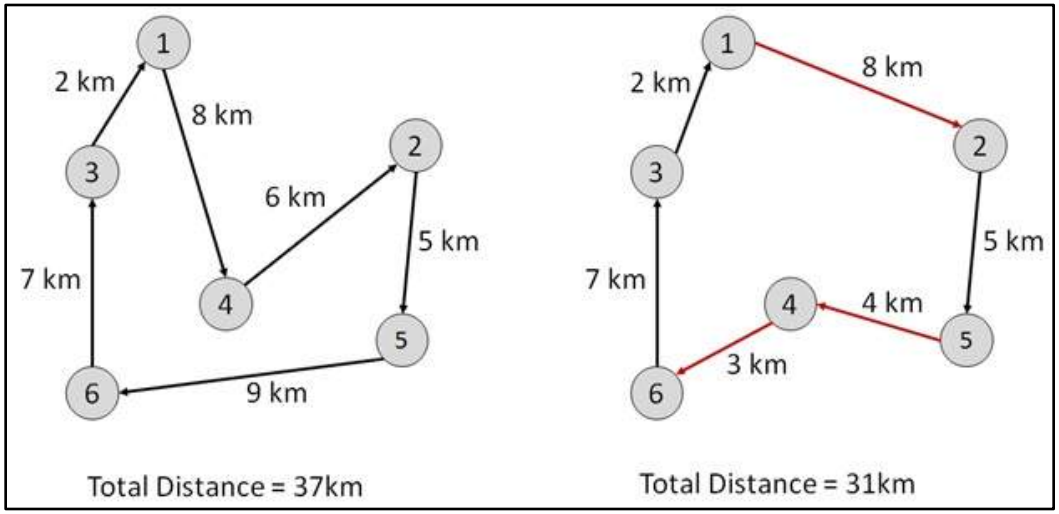
- Exatos: ***em geral*** possuem complexidade de tempo exponencial;
- Algoritmos Aproximados e Heurísticas: com complexidade de tempo polinomial;

Heurísticas

- Resolve problemas por meio de um enfoque intuitivo, em geral, racional, no qual a estrutura do problema é interpretada e explorada inteligentemente para se obter uma solução razoável.
- Alguns motivos para o uso de heurísticas:
  - Método exato não disponível;
  - Não vale a pena o esforço computacional para obter uma solução ótima;
  - Necessidade de um conjunto de soluções com características distintas, pois o tomador de decisões escolherá conforme a situação real.

Tipos de Heurísticas

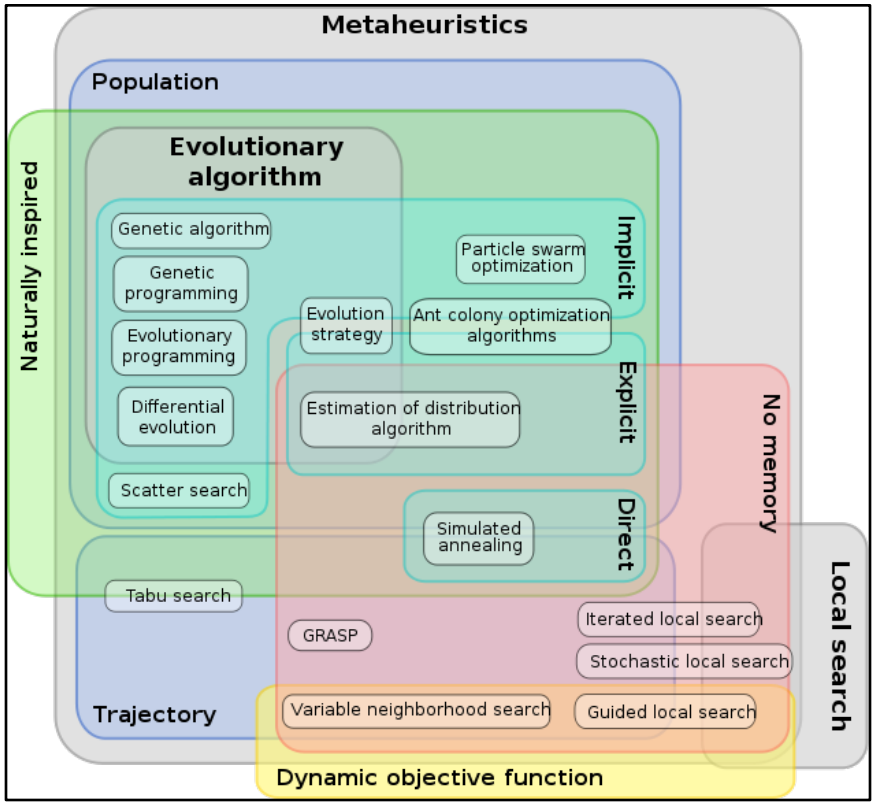
- Construtiva: a solução é construída elemento a elemento.
  - Problema da Mochila: ir inserindo o elemento de maior valor desde que respeitando a capacidade da mochila.
- Busca Local: inicia com uma solução completa e vai melhorando a solução corrente por meio movimentos locais (move-se para soluções vizinhas).



Fonte: http://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_popular_search_algorithms.htm

Tipos de Heurísticas

- Meta-heurística: heurísticas genéricas, com escolhas aleatórias e/ou determinísticas e uso de informações passadas, que independem do problema.
- Podem ser:
  - Baseada em trajetória ou com uso de população;
  - Busca local ou global;
  - Bioinspirada;
  - Objetivo estático ou dinâmico;
  - Armazena e usa informações anteriores.

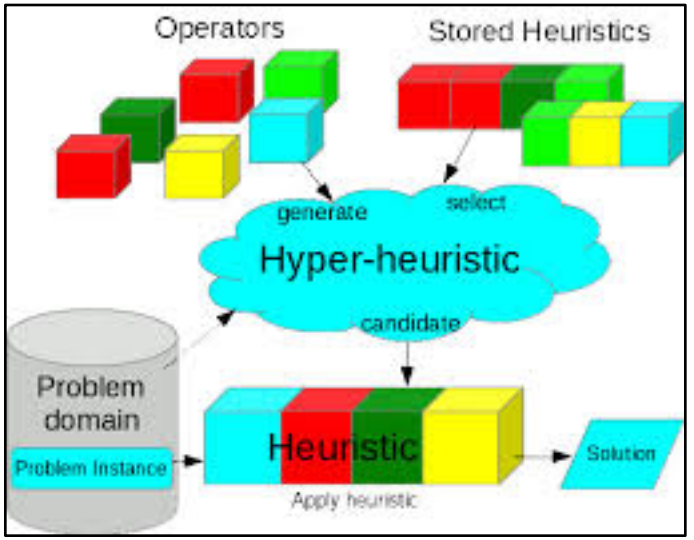


Fonte: https://en.wikipedia.org/wiki/Metaheuristic

Tipos de Heurísticas

- Hibrída: combinação de duas ou mais heurísticas diferentes e que pode envolver outros métodos de otimização.
- Hiper-heurística: totalmente independente do problema, gerencia e aplica heurísticas específicas em cada etapa da otimização.

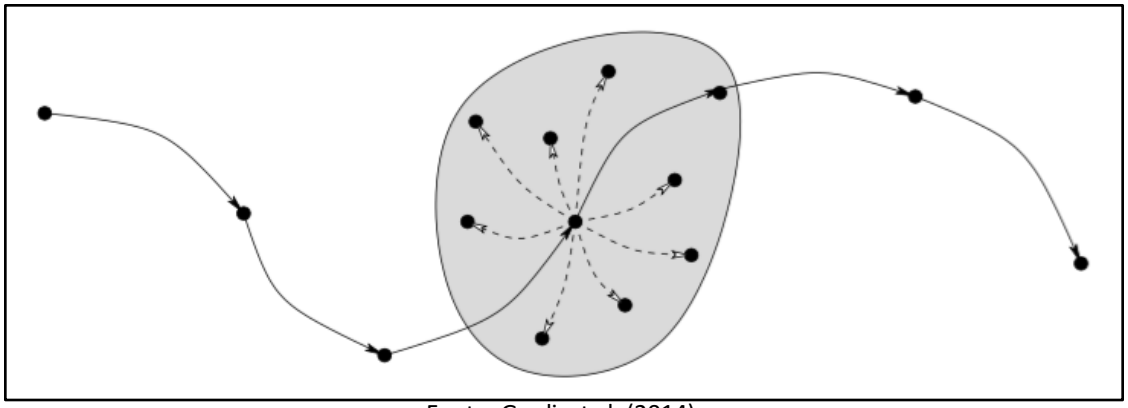
Fonte: http://www.seage.org/web/data/malek10hyper.pdf



Fonte: http://researcharchive.vuw.ac.nz/xmlui/bitstream/handle/10063/4242/thesis.pdf?sequence=2

Busca Local

- Não é apenas uma heurística, mas um paradigma de otimização.
- A partir de movimentos (*moves*) modifica a solução corrente (suas variáveis) na busca de uma nova solução (melhor).
- Explora a vizinhança da solução corrente, o que permite avaliar a nova solução rapidamente.
- Busca local presente no:
  - Método de ordenação por bolha (*bubble sort*);
  - Algoritmo Simplex.



Fonte: Gardi et al. (2014)

Busca Local

- Considera um espaço de busca S, uma vizinhança N e uma função de custo F.
- Diferem em:
  - Como explorar a(s) vizinhança(s) (SelectMove);
  - Quando executar/aceitar um movimento (AcceptableMove);
  - Quando finalizar a busca (StopSearch)

```
procedure LocalSearch(S, N, F)
    s_0 ← GenerateInitialSolution()
    while ¬StopSearch(s_i, i) do
        m ← SelectMove(s_i, F, N)
        if AcceptableMove(m, s_i, F) then
            s_{i+1} ← s_i ⊕ m
        else
            s_{i+1} ← s_i
        end if
        i ← i + 1
    end while
end procedure
```

Fonte: http://www.slideshare.net/lucadigaspero/habilitationskolloquium

Solvers

- Empresas e indústrias geralmente buscam um *software* simples de usar e que fornece boas soluções rapidamente.
- ***Solvers*** são *softwares* de programação matemática, neste caso otimização matemática que:
  - Contêm métodos de propósito geral, o que exige a escrita do problema na forma de um ”modelo”;
  - Geralmente usados como ponto de partida para a resolução de problemas práticos.
- Para problemas de otimização há, por exemplo, *solvers* baseados em métodos de:
  - Busca em árvore: IBM CPLEX, Gurobi, FICO Xpress, COIN;
  - Heurísticas/meta-heurísticas: LocalSolver, MIDACO.

LocalSolver

- Solver de programação matemática que integra:
  - Busca local;
  - Busca em vizinhança adaptativa (migra de pequenas para grandes);
  - Usa técnicas de propagação de restrições;
  - Usa técnicas de programação linear, inteira e não-linear.
- O projeto iniciou em 2007 como uma ”caixa-preta”, versão 1.x, para problemas de otimização com restrições e objetivo (não) lineares.

LocalSolver: Conceitos Gerais

- Não aplica reduções ou decomposições no problema visando obter uma solução rapidamente:
  - tratado conforme foi modelado pelo usuário;
  - espaço de busca explorado é próximo do original ou até uma extensão dele.
- Variáveis contínuas e discretas são tratadas em conjunto pelos movimentos.
- Preza pela construção de vizinhanças ”ricas” que visam:
  - assegurar uma exploração totalmente aleatória do espaço de busca;
  - acelerar a avaliação das vizinhanças por meio de cálculos incrementais.

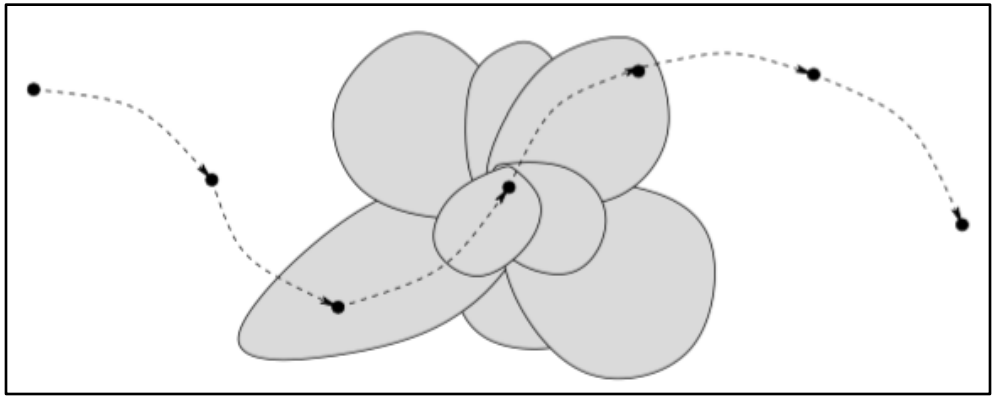
LocalSolver: Conceitos Gerais

- Espaço de busca é modelado para assegurar movimentos com alta probabilidade de sucesso (melhorar a solução):
  - A principal causa de falha é a presença de muitas restrições no modelo de otimização.
- Problemas que possuem (muitas) restrições apertadas requerem vizinhanças largas na busca de soluções viáveis:
  - Vizinhanças largas são a melhor maneira de diversificar a busca e garantir convergência para soluções de qualidade.
- LocalSolver começa explorando vizinhanças pequenas e vai incrementando a busca em vizinhanças maiores:
  - Vizinhanças pequenas são, geralmente, exploradas em tempo constante, o que permite obter muitas soluções diferentes.

LocalSolver: Conceitos Gerais

- As vizinhanças pequenas surgem a partir de uma grande variedade de movimentos simples.
  - Uma vizinhança é formada pelo conjunto de soluções obtidas a partir de todos os movimentos de um dado tipo.
- Aumenta-se o tamanho da vizinhança a partir da execução de movimentos mais complexos.
  - Aplicar diferentes movimentos simples consecutivamente, buscando combinar vizinhanças com diferentes estruturas.
  - Usar métodos de busca em árvore ou outros algoritmos eficientes, porém requer bastante tempo computacional.
- A vizinhança é explorada no esquema de *first-improvement* com aleatoriedade.
  - Quando uma solução melhor é encontrada, ela torna-se a solução corrente cuja vizinhança será explorada.

LocalSolver: Conceitos Gerais



Fonte: Gardi et al. (2014)

- Um conceito primordial é o de ***incrementalidade*** e está associado a avaliação de soluções vizinhas.
  - A parte da solução que não modifica na nova solução é chamada de **invariante**.
  - Avaliar o custo da nova solução deve ser feito mais rapidamente a partir da determinação de invariantes.
  - Usar e explorar invariantes permite acelerar a convergência por várias ordens de magnitude (10 ou até 100x).

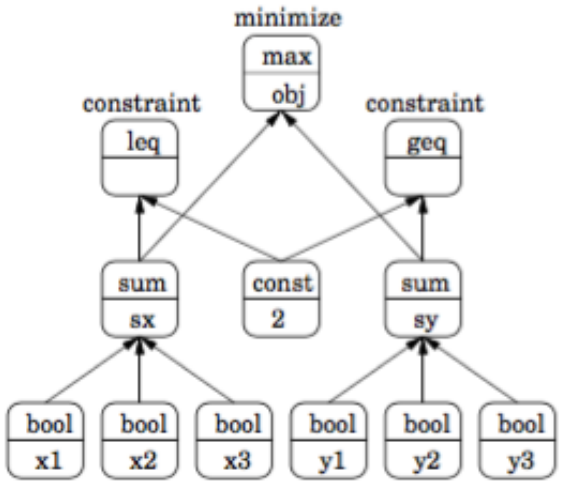
LocalSolver: Modelos

- O formalismo suportado pelo LocalSolver é ligeiramente similar ao usado em ***solvers*** de programação inteira, porém enriquecido com operadores matemáticos comuns.
  - As expressões podem ser combinadas por meio de operadores aritméticos, lógicos, relacionais ou condicionais para derivar novas expressões.
- O modelo para o LocalSolver deve ser escrito assumindo uma formulação ”elástica”.
  - As variáveis devem ser combinadas para restringir apenas o essencial.
  - Assegurar que soluções viáveis para o problema sejam obtidas facilmente.
  - Restrições difíceis (*hard constraints*) devem ser consideradas como objetivos.

LocalSolver: Modelos

- O modelo é transformado em um grafo direcionado acíclico:
  - as variáveis de decisão estão nas raízes;
  - as folhas contêm as restrições e objetivos avaliados;
  - os nós internos representam variáveis intermediárias e também estão relacionados com as invariantes.

```
x1 <- bool();
x2 <- bool();
x3 <- bool();
y1 <- bool();
y2 <- bool();
y3 <- bool();
sx <- sum(x1, x2, x3);
sy <- sum(y1, y2, y3);
constraint leq(sx, 2);
constraint geq(sy, 2);
obj <- max(sx, sy);
minimize obj;
```



Fonte: Gardi et al. (2014)

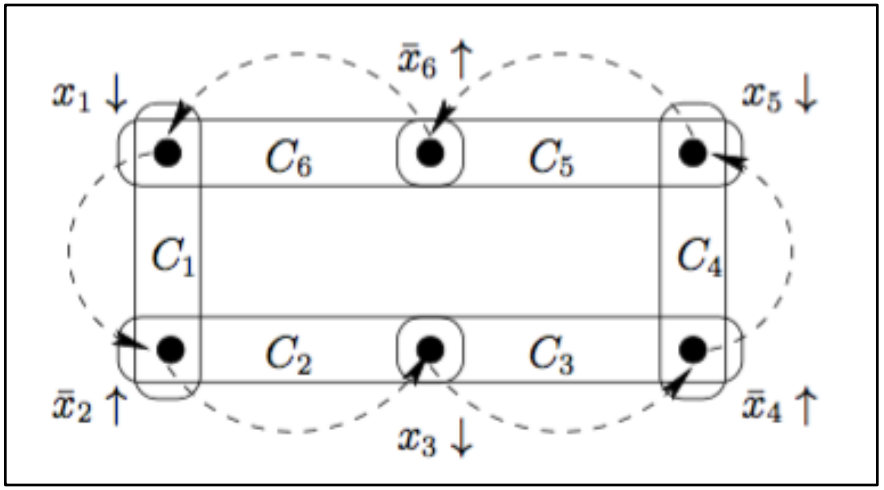
LocalSolver: Movimentos

- Alguns movimentos partem do princípio de trocas no valor de *k*-variáveis de decisão.
  - Trocar o valor das variáveis em uma restrição ao mesmo tempo que a viabilidade da restrição é mantida.
- Os movimentos modificam o valor das variáveis e propagam tais modificações sobre o grafo.
- Os movimentos são escolhidos de forma aleatória a partir de uma distribuição não uniforme.
  - Corrigida a partir de um *feedback* constante sobre as melhoras e pioras no valor da solução.
  - Movimentos que sempre geram soluções inviáveis ou ruins acabam sendo descartados.

LocalSolver: Movimentos

- Conta também com movimentos para explorar pequenas vizinhanças complexas.
  - Obtidos a partir da composição de movimentos mais simples.
  - Ideia: manter a viabilidade da solução ou pelo menos de algumas restrições.
- Corrige-se o valor de outras variáveis para que as restrições relacionadas ainda continuem satisfeitas.

- Exemplo de movimento cíclico:
  - x_1, x_3 e x_5 com valor 1;
  - x_2, x_4 e x_6 com valor 0;
- Restrições:
  - $x_1 + x_2$ 1;
  - $x_1 + x_6$ 1;
  - …



Fonte: Gardi et al. (2014)

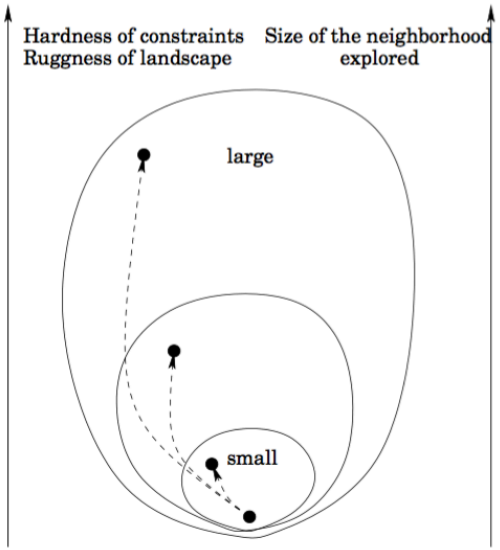
LocalSolver: Invariantes

- O poder do LocalSolver está na eficiência de seus movimentos e na avaliação da solução resultante.
- A propagação das mudanças na nova solução é feita por uma busca em largura no grafo.
  - A propagação é reduzida e ocorre nos nós que foram impactados.
- Por exemplo, na expressão **z <– a < b**, sendo verdadeira, então o nó que a contém não é impactado se:
  - **a** diminui de valor; ou,
  - **b** aumenta de valor.
- Então, os nós são marcados conforme o grafo vai sendo percorrido.

LocalSolver: Estratégia Global

- Usa-se busca em vizinhança como sendo a estratégia global de busca, ao invés de busca em árvore.
  - A busca em árvore é utilizada para representar uma vizinhança grande e que envolve soluções promissoras.
  - Nesta busca em árvore usam-se heurísticas para fixar/relaxar variáveis ao invés de resolver relaxações de modelos ou propagações.
- A vizinhança adapta-se dinamicamente durante a busca:

- Pequenas vizinhanças ajudam a convergir para uma solução ou obter várias soluções diferentes muito rapidamente;
- Vizinhanças maiores permitem escapar de ótimos locais ou de regiões dominadas por restrições difíceis.



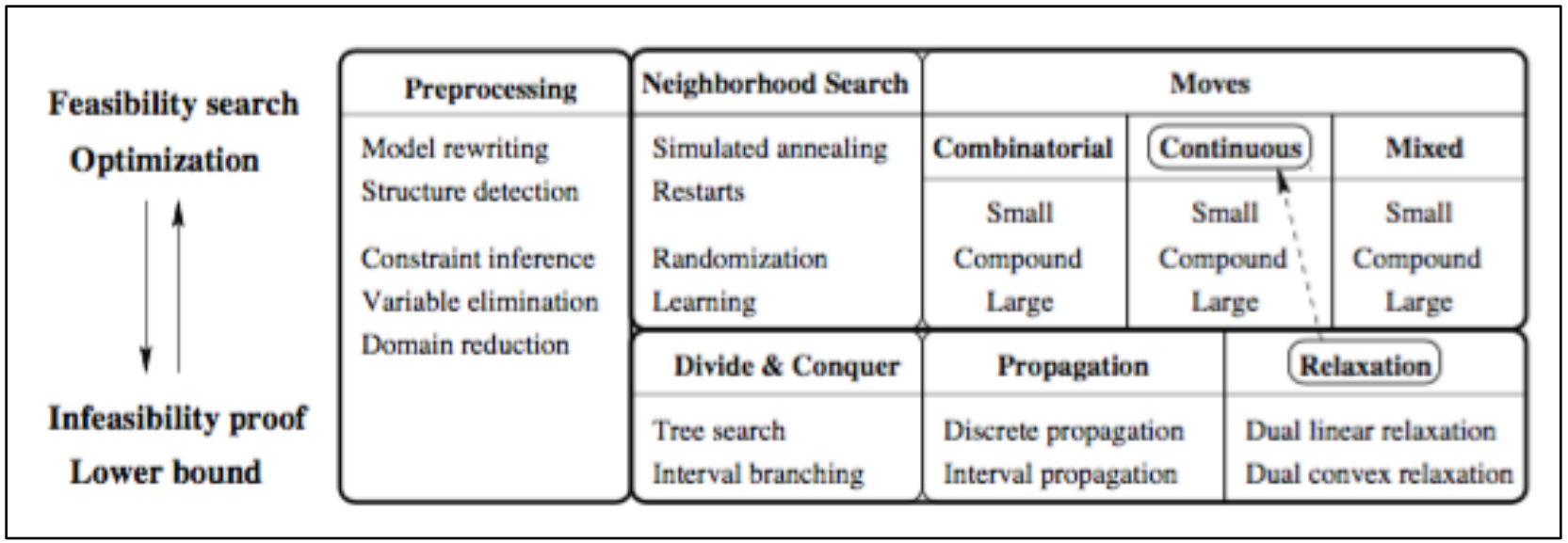
Fonte: Gardi et al. (2014)

LocalSolver: Estratégia Global

- Usa o recozimento simulado com limitantes para escapar de ótimos locais em pequenas vizinhanças.
  - A busca pode reiniciar a partir de soluções anteriores.
- Ainda combina busca em múltiplas vizinhanças e o mecanismo de aprendizagem de hiper-heurísticas.
- Limitantes inferiores são obtidos a partir de relaxações, técnicas de inferência e propagação de restrições, todas embutidas em um esquema de divisão-e-conquista.
  - Usa relaxação dual ao invés de primal.
  - Identifica estruturas discretas favoráveis para melhorias.
  - Aplica propagação em intervalos e filtros no domínio.

LocalSolver: Arquitetura

- Unifica heurísticas e abordagens de otimização exata de duas formas:
  - abordagem de busca em vizinhança generalizada (múltipla, variável e adaptativa);
  - separa a busca e a otimização (busca no espaço primal) da computação de limitantes (busca no espaço dual).



Fonte: Gardi et al. (2014)

LocalSolver: Linguagem

- Tem uma linguagem própria, denominada LSP, ou APIs para Python, C++, Java e C#. A LSP é:
  - Fortemente ***tipada*** e estruturada em blocos;
  - Programa é feito por uma sequência de funções, os módulos, limitados por { };
  - As expressões terminam com ; e são *case-sensitive*;
  - Comentários são dados por // ou /* Comentário */
  - Identificadores formado por caracteres alfanuméricos começando por letra ou _ seguido de letra ou dígito;
  - Palavras reservadas:

```
true        false       nil         nan         inf
function    local       return      this        use
while       do          break       continue
for         in          if          else
minimize    maximize    constraint
try         throw       catch
is          typeof
```

LocalSolver: LSP

- Literais:
  - Strings na forma *”texto aqui”;*
  - Inteiros não consideram 0 a esquerda: 0, 1, 5, 10 ou 120002 ou …
  - Ponto flutuante: 12.45 ou .45 ou 34e-12 ou …
- Tipos:
  - Para representar nulo: nil
  - Inteiros, número de pontos flutuantes, booleanos ou strings
  - Mapas (arrays): *a = {-5, 4, ”foo”}* começando do índice 0 ou não, tipo dicionário
  - LSExpressions: criadas e manipuladas com o operador <-
  - Funções podem ser passadas para outras funções, atribuídas para variáveis ou retornada por outras funções. Permite definir variáveis locais.
  - Módulo é uma coleção de funções e variáveis globais servindo para agrupar funções. Um arquivo .lsp é um módulo e pode ser usada por outros módulos.

LocalSolver: LSP

- Variáveis:
  - Refere-se ao valor e não tem um tipo
  - Pode receber valores de diferentes tipos
  - Sempre atribuídas por referência e podem ser locais ou globais. Para variáveis locais pode-se usar a palavra reservada *local*
  - A atribuição de valores é feita usando o operador =
- Expressões:
  - Aritméticas: envolve os operadores +, -, *, /, %
  - Operadores aplicados em números, strings e LSExpressions

```
a = 10;
b = a + 5.0;       // a is equals to 15
c <- a * 3.0;      // a is converted to an LSExpression
                   // and a new LSExpression of type PRODUCT is added to the model.
d = "foo" + 42     // 42 is converted to a string, then the concatenation is performed.

// This arithmetic operation will throw an exception:
// the modulo operation is not overloaded for strings.
e = "foo" % 2;
```

LocalSolver: LSP

- Expressões:
  - Relacionais: envolvem os operadores ==, !=, <, >, >=, <=
  - Operadores aplicados em números, strings e LSExpressions

```
a = {1,8};
b = {1,8};

println(8 < 9.2);           // print 1
println(a == b);            // throw an exception since "==" is not defined on maps
println("abc" <= "abcde");  // print 1
println("abc" == "aBc");    // print 0 since comparison is case-sensitive
```

  - Lógicas: envolvem os operadores !, &&, ||
  - Operadores aplicados em booleanos e LSExpressions

  - Condicional ternário é da forma: *cond ? True : false*
  - Operadores *typeof* e *is* para identificar o tipo dos argumentos

LocalSolver: LSP

- Atribuições:
  - Simples: usa o = para atribuir valor

```
a = true;      // a = 1
b = 9;         // b = 9
c = a + b;     // c = 10
c = a * b;     // c = 9
c = (a == b); // c = 0
c = a < b;     // c = 1
```

  - Composto, que combina um operador aritmético com o =

```
a += 2;      // a = a + 2
a /= 2;      // a = a/2
a *= b + c; // a = a*(b + c)
```

  - Expressões do LocalSolver

```
x <- 1;      // 1 is transformed to a constant LS expression and is added to the mathematical
y <- bool(); // y is a new decision variable and is added to the mathematical model
z <- x+y;    // z is an LS expression and is added to the mathematical model

// This assignment will throw an exception: strings cannot be casted to LSExpression.
z <- "a string";
```

LocalSolver: LSP

- Atribuições:
  - Iteradas

```
for[i in 0..9][j in i+1..9][k in j+2..9]
    a[i][j][k] = i + j + k;
a[i in 0..9][j in i+1..9][k in j+2..9] = i + j + k; // compact
```

- Condicional

```
if (0) c = "ok";
if (true) c = "ok";
if (2) c = "error"; // ERROR: invalid condition
```

- Laços

```
// Non compact
for[i in 0..9]
    for [j in i+1..9 : j % 2 == 0]
        for [k in j+2..9]
            a[i][j][k] = i + j + k;

// Compact
for[i in 0..9][j in i+1..9 : j % 2 == 0][k in j+2..9]
    a[i][j][k] = i + j + k;
```

LocalSolver: LSP

- Laços do tipo *while(expression) {}*
- Laços do tipo *do {} while (expression);*
- Pode-se empregar o *continue* (volta ao teste do laço) e o *break* (interrompe o laço);
- Caracterização do modelo matemático a ser resolvido:
  - Adicionar restrições: *constraint expression;*
  - Adicionar um objetivo a ser minimizado: *minimize expression;*
  - Adicionar um objetivo a ser maximizado: *maximize expression;*
- Funções do sistema ou definidas pelo usuário

```
function f(a) {
    a[2] = 8; // applies to the original map
    a = {2,3,4}; // no impact on the map since it just assigns locally a new map to the name "
}

function g() {
    a[0] = 0;
    f(a);
    print(a); // will print [  0 => 0  2 => 8 ]
}
```

```
function f(a) {
    if (a) return 2;
    else return "zz";
}
```

LocalSolver: LSP

- Funções do sistema ou definidas pelo usuário:

```
x <- sum[i in 1..10](w[i] * y[i]);  // declare x as a sum of 10 terms
v = min[j in 1..5][k in 1..9](m[j](k)); // retrieve the smallest element of a matrix
v = prod[v in a : v != 0](a[j]);  // compute the product of non-zero elements of an map a
```

- Linha de comando:

```
localsolver file.lsp [arguments]
```

  - Exemplos:

```
localsolver test.lsp x=12 y=abc z="a string with whitespaces" t=true
```

```
localsolver test.lsp a=z,12,8:text,key:41
```

LocalSolver: LSP

- Bibliotecas trazem várias funções para trabalhar com arquivos, strings, estrutura de dados complexas, etc.
  - Consistem de uma coleção de módulos importados pela palavra reservada *use*
- Funções pré-definidas:
  - Imprimir: *println(args)*
  - Decisão booleana: bool()
  - Decisão de ponto flutuante: float(lb, ub)
  - Decisão inteira: int(lb, ub)
  - Cria expressão com a soma, diferença, produto, divisão, módulo, mínimo, máximo, igualdade, desigualdade, condição ternária, absoluto, distância, potência, seno…
    - sum(args), sub(a, b), prod(args), div(a, b), mod(a, b), min(args), max(args), eq(a, b), neq(a, b), iff(cond, trueE, falseE), abs(a), dist(a, b), pow(a, b), sin(a)…
- Variáveis globais:
  - lsTimeLimit, lsIterationLimit, lsNbThreads, lsSeed, lsTimeBetweenDisplays, lsAnnealingLevel, lsVerbosity;

LocalSolver: Instalação

- Ir em www.localsolver.com -> Software -> Download
  - Escolher a versão 6.5 (lançada recentemente) adequada para o seu sistema operacional;
  - Free test version: não precisa de registro, porém está limitada para modelos com até 100 decisões;
  - Free trial licenses: precisa de registro para obter uma licença válida por 30 dias;
  - Free academic licenses: para estudantes/professores, sendo válida por 1 mês e podendo ser renovada.
- Seguir o procedimento de instalação como usual (duplo clique, aceitar termos e continuar….);
- Abrir um editor de texto simples (ou uma IDE) e ”por a mão na massa”!

LocalSolver: Exemplo*

```
/********** toy.lsp **********/

/* Declares the optimization model. */
function model() {

    // 0-1 decisions
    x_0 <- bool(); x_1 <- bool(); x_2 <- bool(); x_3 <- bool();
    x_4 <- bool(); x_5 <- bool(); x_6 <- bool(); x_7 <- bool();

    // weight constraint
    knapsackWeight <- 10*x_0 + 60*x_1 + 30*x_2 + 40*x_3 + 30*x_4 + 20*x_5 + 20*x_6 + 2*x_7;
    constraint knapsackWeight <= 102;

    // maximize value
    knapsackValue <- 1*x_0 + 10*x_1 + 15*x_2 + 40*x_3 + 60*x_4 + 90*x_5 + 100*x_6 + 15*x_7;
    maximize knapsackValue;
}

/* Parameterizes the solver. */
function param() {
    lsTimeLimit = 1;
}
```

- Para executar no terminal (linha de comando):

```
localsolver/examples/toy$ localsolver toy.lsp lsTimeLimit=2
```


*Os exemplos podem ser obtidos em: http://www.localsolver.com/documentation/exampletour/index.html

Exemplo: Problema da Mochila

```
/********** knapsack.lsp **********/

use io;

/* Reads instance data. */
function input() {
    local usage = "Usage: localsolver knapsack.lsp "
        + "inFileName=inputFile [solFileName=outputFile] [lsTimeLimit=timeLimit]";

    if (inFileName == nil) throw usage;

    local inFile = io.openRead(inFileName);
    nbItems = inFile.readInt();
    weights[i in 0..nbItems-1] = inFile.readInt();
    prices[i in 0..nbItems-1] = inFile.readInt();
    knapsackBound = inFile.readInt();
}

/* Declares the optimization model. */
function model() {
    // 0-1 decisions
    x[i in 0..nbItems-1] <- bool();

    // weight constraint
    knapsackWeight <- sum[i in 0..nbItems-1](weights[i] * x[i]);
    constraint knapsackWeight <= knapsackBound;

    // maximize value
    knapsackValue <- sum[i in 0..nbItems-1](prices[i] * x[i]);
    maximize knapsackValue;
}
```

Exemplo: Problema da Mochila

```
/* Parameterizes the solver. */
function param() {
    if (lsTimeLimit == nil) lsTimeLimit = 20;
}

/* Writes the solution in a file */
function output() {
    if(solFileName == nil) return;
    local solFile = io.openWrite(solFileName);
    solFile.println(knapsackValue.value);
    for [i in 0..nbItems-1 : x[i].value == 1]
        solFile.print(i + " ");
    solFile.println();
}
```

- Para executar no terminal (linha de comando):

```
~/examples/knapsack$ localsolver knapsack.lsp inFileName=instances/toy.in lsTimeLimit=1
```

Exemplo: Caixeiro Viajante

```
/********** tsp.lsp **********/

use io;

/* Reads instance data. */
function input() {
    local usage = "Usage: localsolver tsp.lsp "
        + "inFileName=inputFile [lsTimeLimit=timeLimit]";

    if (inFileName == nil) throw usage;
    local inFile = io.openRead(inFileName);

    // The input files follow the TSPLib "explicit" format.
    while (true) {
        local str = inFile.readln();
        if (str.startsWith("DIMENSION:")) {
            local dim = str.trim().split(":")[1];
            nbCities = dim.toInt();
        } else if (str.startsWith("EDGE_WEIGHT_SECTION")) {
            break;
        }
    }

    // Distance from i to j
    distanceWeight[0..nbCities - 1][0..nbCities - 1] = inFile.readInt();
}
```

Exemplo: Caixeiro Viajante

```
/* Declares the optimization model. */
function model() {
    // A list variable: cities[i] is the index of the ith city in the tour
    cities <- list(nbCities);

    // All cities must be visited
    constraint count(cities) == nbCities;

    // Distance of the arc reaching the ith city
    distance[0] <- distanceWeight[cities[nbCities - 1]][cities[0]];
    distance[i in 1..nbCities - 1] <- distanceWeight[cities[i - 1]][cities[i]];

    // Minimize the total distance
    obj <- sum[i in 0..nbCities - 1](distance[i]);

    minimize obj;
}

/* Parameterizes the solver. */
function param() {
    if (lsTimeLimit == nil) lsTimeLimit = 5;
}
```

Exemplo: Caixeiro Viajante

```
/* Writes the solution in a file */
function output() {
    if(solFileName == nil) return;
    local solFile = io.openWrite(solFileName);
    solFile.println(obj.value);
    for [c in cities.value]
        solFile.print(c, " ");
    solFile.println();
}
```

- Para executar no terminal (linha de comando):

```
$ localsolver tsp.lsp inFileName=instances/br17.atsp [lsTimeLimit=]
  [solFileName=]
```

Sudoku

- Codifique um algoritmo no LocalSolver para resolver o tabuleiro ao lado;

- **Regras**:
- Completar todas as 81 células usando números de 1 a 9;
- Não pode repetir números na mesma linha;
- Não pode repetir números na mesma coluna;
- Não pode repetir números na grade 3x3.

	1	2	3	4	5	6	7	8	9
A	7	3			2	8			
B	8								9
C			9			6			
D						7	8		
E			1				2		
F			6	5					
G				4			9		
H	6	9							4
I				1	9			5	6

Sudoku

- A solução deve ser algo como mostrado na figura abaixo;

	1	2	3	4	5	6	7	8	9
A	7	3	5	9	2	8	6	4	1
B	8	6	2	3	4	1	5	7	9
C	1	4	9	7	5	6	3	8	2
D	9	5	4	2	1	7	8	6	3
E	3	7	1	6	8	4	2	9	5
F	2	8	6	5	3	9	4	1	7
G	5	1	7	4	6	2	9	3	8
H	6	9	3	8	7	5	1	2	4
I	4	2	8	1	9	3	7	5	6

REFERÊNCIAS

- Arenales, M., Armentano, V., Morabito, R. e Yanasse, H. *Pesquisa Operacional*. Rio de Janeiro: Campus, 2007.
- Benoist, T., Estellon, B., Gardi, F., Megel, R., e Nouioua, K. Localsolver 1.x: a black-box local-search solver for 0-1 programming. *4OR*, 9(3):299–316, 2011.
- Gardi, F., Benoist, T., Darlay, J., Estellon, B., e Megel, R. *Mathematical Programming Solver Based on Local Search*. John Wiley & Sons, Inc. ISBN 9781118966464, 2014.
- Garey, M. R. e Johnson, D. S. *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*. San Francisco: Freeman, 1979.