Pesquisa Operacional para o Desenvolvimento
Vol. 12, e12005, p. 1-13, 2020

ISSN 1984-3534

TUTORIAL CONVIDADO
UMA INTRODUÇÃO À CONFIGURAÇÃO AUTOMÁTICA DE ALGORITMOS[1]

Lorena Kerollen Botelho Tavares[a], Mayron César O. Moreira[a*]

[a] Departamento de Ciência da Computação
Universidade Federal de Lavras - UFLA, Lavras-MG, Brasil

Recebido 10/03/2020, aceito 14/09/2020

RESUMO

Algoritmos para a resolução de problemas de otimização são divididos em componentes. A combinação de cada elemento gera versões distintas de um método. Tal característica traz uma dificuldade a pesquisadores, visto que planejar a versão apropriada de um algoritmo implica em um elevado esforço humano com testes manuais, além de um provável viés por favorecer algumas combinações em detrimento de outras. Este tutorial propõe uma introdução ao projeto de algoritmos de configuração automática, que preconizam soluções mais robustas, generalização de componentes de código e flexibilidade aos mesmos para serem incorporados por outros métodos. Através do problema do grupo maximamente diverso (MDGP, do inglês *maximally diverse group problem*), descrevemos uma gramática baseada na metaheurística Busca Local Iterada (ILS, do inglês *iterated local search*). Para tanto, utilizamos o *software* livre *Irace* para a calibração dos parâmetros e escolha da melhor versão da ILS no conjunto de instâncias teste adotado. Assim, esperamos que com este tutorial, pesquisadores interessados em configuração automática de algoritmos possam utilizar essa ferramenta em diferentes problemas de otimização combinatória.

Palavras-chave: Configuração automática, Algoritmos, ILS, Irace.

ABSTRACT

Algorithms to solve optimization problems are divided into components. The combination of each element generates distinct versions of a method. Such a characteristic brings difficulties to researchers since the planning of an appropriate version of an algorithm implies a higher level of human effort with manual tests, and bias to favor some combinations against others. This tutorial proposes an introduction to the design of automatic configuration algorithms that preconizes more robust solutions, a generalization of code components, and their flexibility to be incorporated in other methods. Through the maximally diverse group problem, we describe a grammar based on the metaheuristic Iterated Local Search (ILS). For this purpose, we use Irace software to parameter tuning and to choose the best version of ILS for the set of test instances adopted. We expect that with this tutorial, interested researchers in the automatic configuration of algorithms can use this tool in different combinatorial optimization problems.

Keywords: Automatic configuration, Algorithms, ILS, Irace.

* Autor para correspondência. E-mail: mayron.moreira@ufla.br
DOI: 10.4322/PODes.2020.005

[1] *Todos os autores assumem a responsabilidade pelo conteúdo do artigo.*

1. Introdução

Algoritmos utilizados para a resolução de problemas de otimização combinatória possuem algumas características similares. Podemos citar, por exemplo, procedimentos de geração de vizinhança baseados em remoção e inserção de indivíduos em uma solução, comuns em adaptações de Busca Local Iterada (ILS, do inglês *iterated local search*), Busca Tabu e Busca em Vizinhança Variável (VNS, do inglês *variable neighborhood search*). De fato, ao considerarmos a implementação de um algoritmo através de uma abordagem *bottom-up*, podemos separar cada um de seus componentes em pequenos blocos e propor diferentes combinações desses, gerando versões distintas de um mesmo método.

Souza e Ritt (2018) chamam a atenção para a necessidade de estudos que considerem configuração automática de algoritmos (AAC, do inglês: *automatic algorithm configuration*). Tal metodologia possui vantagens, como as mostradas a seguir:

- redução do esforço humano no tempo de buscar ou propor heurísticas promissoras;
- redução de algum viés proveniente de testes manuais, levando a uma maior robustez nas soluções propostas;
- generalização de componentes de código, flexíveis a serem incorporados em outros métodos.

A configuração automática de algoritmos tem apresentado êxito para problemas como *flowshop* permutacional (Brum e Ritt, 2018; Pagnozzi e Stützle, 2019), designação de avaliações em contextos educacionais (Souza e Ritt, 2018), *bin packing* (Mascia et al., 2014), problema da mochila multi-objetivo (Bezerra et al., 2012) e problema de satisfazibilidade *booleana* (KhudaBukhsh et al., 2016). A base da AAC está na representação de um algoritmo através de uma gramática que explora as possíveis combinações de seus componentes. Em especial, este tutorial apresenta uma abordagem de AAC para um algoritmo ILS (Lourenço et al., 2019), uma metaheurística robusta e frequentemente utilizada na resolução de problemas combinatórios.

Para a configuração automática, consideramos o pacote estado da arte *Irace*, desenvolvido para a calibração de parâmetros de algoritmos López-Ibáñez et al. (2016). Um estudo que aborda implementações do ILS no contexto da AAC pode visto em Hutter et al. (2009). Considerando outras metaheurísticas que utilizam gramática para configuração automática de algoritmos, podemos citar uma proposta para o Algoritmo de Têmpera Simulada (SA, do inglês *simulated annealing*) (Franzin e Stützle, 2019), Busca Tabu e VND (Pagnozzi e Stützle, 2019), Colônia de Formigas (Bezerra et al., 2012) e Heurísticas Gulosas Iterativas (IG, do inglês *iterated greedy*) (Mascia et al., 2014). Outras abordagens baseadas em evolução de gramáticas através de programação genética podem ser vistas em (Lourenço et al., 2016; Moreira e Ritt, 2019).

Este tutorial é destinado a pesquisadores que queiram iniciar seus trabalhos no contexto do projeto de algoritmos de configuração automática. Para tanto, fornecemos alicerces para auxiliar a desenvolver as etapas de uma configuração automática, desde a gramática à interpretação dos resultados retornados pelo *Irace*. Na continuação desse documento, apresentamos na Seção 2 uma breve descrição da metaheurística ILS. A Seção 3 ilustra o desenvolvimento de uma gramática aplicada ao problema do grupo maximamente diverso (MDGP, do inglês *maximally diverse group problem*) (Gallego et al., 2013). A Seção 4 enumera passo a passo as etapas de configuração do *Irace* para a ILS aplicada ao MGDP. Exemplos de execução e testes são mostrados na Seção 5. Considerações finais a respeito da AAC são apresentadas na Seção 6.

2. Busca Local Iterativa

A Busca Local Iterada (ILS) foi introduzida por Lourenço et al. (2003) e se baseia na geração de uma sequência de soluções obtida a partir de uma heurística acoplada ao algoritmo. A ILS tenta evitar reinícios aleatórios ao longo da busca, aplicando perturbações em mínimos locais encontrados por meio de um procedimento de busca local. Além disso, tal algoritmo tem como uma de suas vantagens a ausência de uma grande quantidade de parâmetros a serem calibrados. A

ILS possui quatro componentes principais, listados a seguir:

- **Solução inicial**: consiste em uma solução (comumente) factível, que define o ponto de partida do caminho no espaço de busca do problema estudado. Em geral, os algoritmos dessa natureza utilizam da estratégia gulosa para a construção de uma solução.
- **Perturbação**: com o objetivo de escapar de mínimos locais, uma perturbação insere uma determinada quantidade de movimentos que modificam a estrutura de uma solução. Determinar a intensidade da perturbação consiste em um parâmetro a ser calibrado, pois caso haja poucas alterações efetuadas, as novas soluções tendem a ficar próximas do mínimo local atual. Por outro lado, modificações extensas podem levar a uma busca com características próximas a uma busca randômica.
- **Busca local**: visa a geração de um conjunto de soluções a partir de uma determinada vizinhança. Tal etapa pode ser composta por uma única ou um conjunto de vizinhanças. Ao fim, o algoritmo de busca local adotado retorna o mínimo local obtido.
- **Critério de aceitação**: etapa aplicada após a execução do algoritmo de busca local. Determina uma metodologia para aceitar ou não um mínimo local, a fim de dar continuidade na busca. Em suma, dada uma solução atual s e a melhor solução vizinha s^*, dizemos que s^* é aceita: (i) se possuir melhor valor de função objetivo que s; ou (ii) sempre, independente de seu valor de função; ou (iii) caso tenha um valor de função pior que s, tenha probabilidade de aceitação similar ao que é feito com o critério de escolha do *Simulated Annealing* (Nikolaev e Jacobson, 2010).

O Algoritmo 1 apresenta um exemplo de pseudocódigo da metaheurística ILS. Note que `melhor(`x, y`)` é uma função que retorna a melhor solução entre x e y, de acordo com a função objetivo do problema. As funções `SolInicial()`, `Perturbacao(`x`)`, `BuscaLocal(`x`)` e `Aceitacao(`x, y`)` correspondem a cada um dos componentes da ILS já descritos anteriormente. O algoritmo deve parar caso o seu tempo de execução ou número de iterações tenha chegado ao limite pré-estabelecido. No entanto, critérios específicos de cada problema podem ser também aplicados. A solução s_b é a melhor solução encontrada durante toda a busca.

Algoritmo 1: `Framework` padrão da ILS.

```
1  s_0 ← SolInicial();
2  s* ← BuscaLocal(s_0);
3  s_b ← melhor(s_0, s*);
4  repita
5  |  s' ← Perturbacao(s*);
6  |  s'* ← BuscaLocal(s');
7  |  s_b ← melhor(s_b,melhor(s', s'*));
8  |  s* ← Aceitacao(s*, s'*);
9  até alcançar o critério de parada;
10 retorna s_b;
```

Alguns exemplos resolvidos com êxito por adaptações da ILS podem ser vistos para Problemas de Roteamento de Veículos (Allahyari et al., 2015; Coelho et al., 2016; Kramer et al., 2019), Escalonamento de Tarefas (Dong et al., 2009; Subramanian et al., 2014; Meignan e Knust, 2019) e variantes do Problema de Localização de Facilidades (Derbel et al., 2010, 2012; Nguyen et al., 2012). Outros contextos aos quais a ILS é aplicada podem ser vistos em (Lourenço et al., 2019).

3. Estudo de Caso: Problema de Agrupamento Maximamente Diverso

O Problema de Agrupamento Maximamente Diverso (MDGP, do inglês *maximally diverse grouping problem*) consiste em particionar determinado conjunto de elementos em subconjuntos de tamanhos iguais ou diferentes, com o objetivo de maximizar a diversidade dos elementos dentro

de cada subgrupo. Possíveis aplicações do problema são atribuições de estudantes para classes, montagem de grupos de estudo em universidades, e montagem de grupos de trabalho em projetos (Johnes, 2015).

Para definirmos formalmente o MDGP, considere $G = (V, E)$ um grafo completo não-direcionado e ponderado, em que $V = \{1, ..., n\}$ é o conjunto de vértices (elementos). Cada aresta $(u, v) \in E$ possui um peso $d_{uv} \geq 0$, que corresponde ao grau de diversidade entre dois vértices u e v. Assim, quanto maior o valor de d_{uv}, maior a heterogeneidade entre os elementos u e v. Tomemos m subconjuntos disjuntos (grupos), com quantidade vértices pertencendo ao intervalo $[l_g, u_g]$, $g = 1, ..., m$. Definimos x_{ig} como uma variável binária igual a 1 se o vértice i pertence ao grupo g. Com isso, podemos escrever o modelo quadrático do MDGP:

$$\text{Maximizar} \sum_{g=1}^{m} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_{ig} x_{ig} \tag{1}$$

Sujeito a:

$$\sum_{g=1}^{m} x_{ig} = 1, \quad i = 1, ..., n \tag{2}$$

$$l_g \leq \sum_{i=1}^{n} x_{ig} \leq b_g, \quad g = 1, ..., m \tag{3}$$

$$x_{ig} \in \{0, 1\}, \quad i = 1, ..., n; g = 1, ..., m \tag{4}$$

A função objetivo (1) maximiza a soma da diversidade de cada grupo. As restrições (2) garantem que cada elemento seja designado a exatamente um grupo. Já as restrições (3) estabelecem um limite mínimo e máximo da quantidade de elementos em cada grupo. O domínio das variáveis de decisão está definido em (4).

3.1. Definição de uma Gramática

A gramática que utilizamos para a configuração automática da ILS desenvolvida é a apresentada abaixo. A ideia geral é determinar quais regras serão chamadas de acordo com a regra inicial, para dessa forma termos as chamadas dos componentes finais, que são os símbolos terminais da gramática. Cada símbolo não-terminal estará entre os caracteres "<>", enquanto os símbolos terminais, que são as funções de cada componente da ILS, não possuem tal notação. Exemplificando: na quarta da linha da gramática, temos `insertion <ls_insertion>`, onde vemos o terminal `insertion` seguido do não terminal `<ls_insertion>`. Vale observar também que o `insertion` da sexta linha é o mesmo `insertion` da linha quatro, com a diferença de que na quarta linha existe, além do próprio terminal, a chamada do não terminal `<ls_insertion>`. O mesmo acontece com o `swap` da quarta e da quinta linha da gramática. Ainda a respeito do símbolo "<>", a diferente dele com o símbolo "()" é que este último é chamada de função em código, enquanto que o primeiro é chamada de regra dentro da gramática. O caractere "," significa operação "e" dentro de chamadas de funções que possuem mais de um parâmetro, indicando por exemplo, que todas as regras dentro da função `ils()` são chamadas a cada iteração da gramática. Enquanto o caractere "|" significa operação "ou", nesse caso, na segunda linha da gramática, por exemplo, apenas uma das três opções de soluções iniciais (lh, gc, wj) é chamada por iteração da gramática. Nossa gramática foi construída de forma que no mínimo uma e no máximo duas buscas locais distintas sejam utilizadas em sequência. As explicações de cada linha da gramática encontram-se logo abaixo a mesma. Por simplicidade, consideramos que a ILS gerada teria uma única vizinhança, e não múltiplas, como descrito na Seção 2.

```
<start> ::= ils(<ini_sol>, <perturbation>, <local_search>,
            <acceptance>)
<ini_sol> ::= lh | gc | wj
<perturbation> ::= <weak_per> | <strong_per>
<local_search> ::= insertion <ls_insertion> | swap <ls_swap>
<ls_insertion> ::= swap | three_chain | none
<ls_swap> ::= insertion | three_chain | none
<weak_per> ::= lhw(<local_search_per>)
<strong_per> ::= lhs(<local_search_per>)
<local_search_per> ::= insertion_per <ls_insertion_per>
                     | swap_per <ls_swap_per>
<ls_insertion_per> ::= swap_per | three_chain | none
<ls_swap_per> ::= insertion_per | three_chain | none
<acceptance> ::= accept_always | accept_better
                | accept_better_equal
```

Apresentamos a seguir a explicação de cada componente da gramática.

• <start>: regra inicial, a qual chamamos a função ils(). Esse procedimento é responsável por determinar quais, entre os parâmetros necessários, serão escolhidos para criar nossa configuração. Por conseguinte, esta é a regra que define a ILS.

• <ini_sol>: primeiro parâmetro da ils(). Esta regra é constituída apenas de símbolos terminais que definem a solução inicial. As possibilidades implementadas são a lh (acrônimos dos autores Lai e Hao) (Lai e Hao, 2016), e os algoritmos gc (*greedy construction*) e wj (referentes aos autores Weitz-Jelassi) (Weitz e Jelassi, 1992). Os últimos dois algoritmos, gc e wj, podem ser encontrados em (Gallego et al., 2013).

• <perturbation>: determina qual tipo de perturbação que a configuração vai utilizar. Como possibilidades, optamos por uma perturbação fraca (<weak_per>), e por uma perturbação forte (<strong_per>).

• <weak_per> e <strong_per>: utilizam as funções lhw(<local_search_per>) e a lhs(<local_search_per>), respectivamente. Ambos os algoritmos estão presentes em (Lai e Hao, 2016). A diferença entre as duas abordagens consiste no fato de que enquanto na <weak_per> há a garantia de que a melhor solução encontrada no método (a partir da solução corrente) seja retornada, fazendo com que a solução corrente seja modificada, mas sem muita piora do resultado, na <strong_per> é sempre a última solução gerada a retornada, isso faz com que a solução corrente seja modificada mais bruscamente.

• <local_search> e <local_search_per>: regras que determinam a primeira busca local a ser realizada. Para a regra <local_search>, o algoritmo substitui a solução atual com a nova solução gerada somente se esta for melhor que a atual. Já para <local_search_per>, a busca local é realizada por alguma das duas perturbações (<weak_per> ou <strong_per>), por esse motivo, a nova solução gerada é aceita mesmo que seja inferior à solução atual. Para ambas as buscas locais, o conjunto de vizinhanças é formado por movimentos de inserção de um vértice do seu grupo atual a um novo grupo (insertion), e de troca entre dois vértices de grupos distintos (swap). Os símbolos chamados por cada uma dessas regras garantem que não teremos buscas locais repetidas numa mesma configuração.

• <ls_insertion>, <ls_swap>, <ls_insertion_per> e <ls_swap_per>: regras que determinam a segunda busca local a ser executada. Nessas regras, para não haver a possibilidade de repetir um mesmo movimento de busca local, em cada uma delas, definimos apenas os movimentos que ainda não foram escolhidos. Dessa forma, na <ls_insertion>, por exemplo, as possibilidades de símbolos terminais são swap, three_chain e none, pois o movimento já realizado na solução, nesse caso, foi o insertion.

As duas primeiras (<ls_insertion> e <ls_swap>) são aplicadas à busca local que mantém a solução atual em caso de não melhoria. Já as outras duas restantes são utilizadas na busca local que aceita piora de solução . Além disso, acrescentamos mais um procedimento de melhoria, denominado three_chain. Tal método consiste em uma busca com três movimentos de troca (swap) em sequência. O símbolo terminal none é definido para o caso onde não temos uma segunda busca local. Quando isso acontece, o programa considera apenas a primeira busca como válida.

- <acceptance>: regra que determina qual será o critério de aceitação da ILS. Neste tutorial, definimos o <accept_always>, que sempre aceita a solução modificada, o <accept_better>, que aceita apenas quando a solução encontrada é melhor que a anterior, e o <accept_better_equal>, que aceita quando a solução encontrada é melhor ou igual a anterior.

A implementação se encontra disponível em https://bitbucket.org/quionee/ils_tutorial.

4. Configurações no Irace

O pacote *Irace* (López-Ibáñez et al., 2016) utiliza instâncias de treinamento e configurações geradas de acordo com as possibilidades da gramática criada. O *software* realiza uma corrida iterada, que determina as melhores configurações para aquele determinado grupo de instâncias. A cada passo da corrida, as configurações são avaliadas de acordo com os resultados obtidos para as instâncias. Assim, o *Irace* é capaz de determinar se as configurações avaliadas são aptas a continuarem na corrida ou não. Ao final de cada corrida, as configurações remanescentes são ranqueadas, e uma outra corrida se inicia. Esse processo continua até que algum dos critérios de parada seja atingido (número de experimentos, tempo computacional, entre outros). Após as corridas serem finalizadas, as melhores configurações (de acordo com o ranqueamento) são apresentadas para cada conjunto de instâncias.

É importante ressaltar que algumas escolhas, como quais configurações farão parte da corrida, são decididas através de testes estatísticos, de acordo com a gramática a ser utilizada. Outras decisões, como a quantidade de corridas que serão realizadas, são estabelecidas durante o processo de configuração do cenário do ambiente em que o *Irace* executará.

A partir de agora, nesta seção, apresentamos uma série de instruções para a configuração do *Irace*. Adotamos a execução dos comandos em um terminal de sistemas GNU/Linux, por se tratar de um sistema operacional de distribuição gratuita e utilizado por uma parcela considerável de pesquisadores.

4.1. Requisitos do Sistema

Para executar o *Irace*, é necessário ter o R (versão $\geq$ 3.2.0) instalado (R Core Team, 2017). As linhas de comando dos executáveis irace e parallel-irace requerem o *GNU Bash*.

4.2. Instalação Local

Abrimos o terminal na pasta onde instalaremos o *Irace*, e executamos os seguintes comandos para criar a pasta onde armazenaremos os pacotes adicionados ao R:

```
$ export R_LIBS_USER="<R_LIBS_USER>"
$ mkdir $R_LIBS_USER
```

4.3. Instalação do *Irace* com R

Dentro do console R, realizamos a instalação do pacote *Irace*. Os comandos utilizados são os listados a seguir:

```
> install.packages("irace")
> library("irace")
> irace.cmdline("--help")
```

Com o primeiro comando, talvez seja necessário selecionar um espelho próximo a sua localização. No Brasil, atualmente existem as seguintes opções: *Brazil (BA) [https], Brazil (PR) [https], Brazil (RJ) [https], Brazil (SP 1) [https] e Brazil (SP 2) [https]*. O segundo comando testa se a instalação foi efetuada com êxito. Por fim, a terceira linha exibe o diretório de instalação e informações sobre o *Irace*.

4.4. Preparação do Cenário

Nessa etapa, criamos um repositório para a configuração do cenário de testes, e copiamos os *templates* do repositório onde o pacote *Irace* foi instalado para a pasta *tuning*. Abaixo, a variável IRACE_HOME é o diretório onde o *Irace* foi instalado.

```
$ mkdir ~/tuning
$ cd ~/tuning
$ cp IRACE_HOME/<R_LIBS_USER>/irace/templates/*.tmpl .
```

Em seguida, retiramos o sufixo `.tmpl` dos arquivos de configuração para podermos começar as alterações necessárias, a fim de executar o *Irace* com nossa gramática. Como exemplo, tomando os arquivos "parameters.txt.tmpl" e "target-evaluator.tmpl", basta executar os seguintes comandos:

```
$ mv parameters.txt.tmpl parameters.txt
$ mv target-evaluator.tmpl target-evaluator
```

Ainda no diretório *tuning*, criamos uma pasta *Instances*, onde colocaremos as instâncias de treinamento que serão utilizadas durante a calibração, e uma pasta *mdgp-arena*, onde as execuções do *target-runner* serão efetuadas. Este último arquivo, *target-runner*, é um *script* responsável por executar nosso programa para cada configuração candidata durante a corrida iterada do Irace. Ele retorna um valor de custo, que é o valor que deve ser minimizado ou maximizado dependendo da necessidade.

```
$ mkdir Instances
$ mkdir mdgp-arena
```

Alguns dos arquivos de configuração que não foram necessários nesse tutorial foram excluídos. Entre eles, podemos citar: (i) o arquivo `forbidden.txt`, utilizado para definir combinações proibidas de parâmetros; (ii) o arquivo `instances-list.txt`, que tem a função de definir grupos de instâncias específicos a serem utilizados da pasta *Instances*); (iii) o arquivo `configurations.txt`, que define configurações iniciais a serem utilizadas pelo *Irace*; e (iv) o *target_evaluator*, que é utilizado quando precisamos adiar a avaliação das configurações candidatas executadas pelo *target_runner*.

Seguindo a gramática definida na seção anterior, modificamos o arquivo *parameters.txt* para os parâmetros especificados abaixo. Como nossa regra inicial chama a função *ils*, são os parâmetros dela que adicionamos no arquivo. Os parâmetros podem ser números reais, números inteiros, categóricos e ordinais. Em nosso caso, utilizamos apenas parâmetros categóricos. Por esse motivo, são todos *strings* sem aspas, separados por vírgula. Dessa forma, a regra <ini_sol>, por exemplo, no arquivo de configurações é definida por:

```
1. ini_sol "--inisol " c <lh, gc, wj>
```

Algumas das regras dependem de outra para serem executadas. As regras <ls_insertion>, <ls_swap>, <ls_insertion_per> e <ls_swap_per> são condicionais e possuem definição similar. No arquivo de configurações, a regra <ls_insertion> é denotada por:

```
1. ls_insertion "--lsin " c (swap, threeChain, none) |
   local_search == "insertion"
```

O caracter "|" separa os parâmetros (do lado esquerdo) da condição (do lado direito). No caso acima, esse parâmetro será utilizado apenas quando o parâmetro *local_search* escolhido for o *insertion*.

4.5. Alteração dos Arquivos de Configuração

No arquivo `scenario.txt`, há diversas formas de configuração. Neste tutorial, adotamos as configurações descritas abaixo, por conveniência. Para tanto, basta abrir o arquivo de configurações e definir os diretórios necessários a seguir, realizando as devidas alterações.

```
parameterFile = "./parameters.txt"
execDir = "./mdgp-arena"
trainInstancesDir = "./Instances"
targetRunner = "./target-runner"
maxExperiments = 5000
```

A configuração `maxExperiments` indica a quantidade de vezes que o `target-runner` será chamado, ou seja, corresponde ao número máximo de experimentos a serem realizados.

No arquivo `target-runner`, as alterações necessárias são as seguintes:

- Na linha

  ```
  EXE=~/bin/program
  ```

  que determina o executável do programa, trocamos por:

  ```
  EXE=~/bin/mdgp
  ```

- Na linha

  ```
  if [ -s "\$\{STDOUT\}" ]; then
  ```

  retiramos o *-s*, pois não é necessário no nosso caso, obtendo:

  ```
  if [ "\$\{STDOUT\}" ]; then
  ```

- Por fim, na linha

  ```
  echo "$COST"
  ```

  multiplicamos por -1 a função objetivo considerada pelo *Irace* (*default*: minimização) para maximização, dado que maximizamos a diversidade dos elementos dos grupos criados:

  ```
  echo "-$COST"
  ```

É necessário também que criemos uma pasta *bin* no mesmo diretório onde criamos a pasta *tuning*. Nessa nova pasta, copiamos o arquivo executável do nosso código. Na sequência, basta acrescentar as instâncias de treinamento na pasta *Instances*. Assim, para executar o *Irace* na pasta *tuning*, o comando necessário no terminal é:

```
$ IRACE_HOME/<R_LIBS_USER>/irace/bin/irace
```

No console R, após o comando:

```
> library("irace")
```

basta executar o comando:

```
> irace.cmdline()
```

5. Execução e Testes

As instâncias de treinamento, conjuntos *RanInt* e *RanReal*, utilizadas pelo *Irace* foram geradas aleatoriamente pelos autores seguindo as definições disponíveis em `http://grafo.etsii.urjc.es/optsicom/mdgp/`. Os dados gerados para este trabalho podem ser acessados em `https://bitbucket.org/quionee/ils_tutorial`. Testamos os problemas do conjunto *RanInt* e *RanReal*, com grupos de tamanhos diferentes, e com quantidade de vértices iguais a 120 e 240. O código foi implementado em C++ e compilado pela versão 4.2.1 do GNU Compiler Collection (GCC). A execução do *Irace* foi em um computador Intel Core i7, 7^a geração, 16GB de memória RAM, com sistema operacional Linux Ubuntu 18.4.

Alguns parâmetros foram predefinidos por questões de simplicidade na gramática. Não obstante, é importante frisar que os mesmos podem ser calibrados pelo *Irace*. Tais parâmetros fixados são o tempo limite de execução, correspondendo a 10 segundos para instâncias de tamanho 120, e 60 segundos para instâncias de tamanho 240; e a quantidade de iterações em ambos os algoritmos de perturbação, dada pela quantidade de vértices do grafo dividido pela quantidade de grupos desejados (n/m).

Nas tabelas 1 a 4, apresentamos os melhores resultados obtidos pelas execuções do *Irace*, para os conjuntos de instâncias de teste adotados. Cada uma das colunas é um parâmetro definido pelo *Irace* de acordo com a gramática. A Tabela 5 apresenta o melhor resultado obtido pelas execuções do *Irace* para todos os conjuntos. De acordo com os resultados, para instâncias do tipo *RanInt*, a solução inicial mais recorrente nas configurações geradas é a `lh`, que aplica buscas locais durante sua construção. Já para instâncias do tipo *RanReal*, tabelas 3 e 4, as outras duas soluções iniciais, `wj` e `gc`, ocorreram com maior frequência, (`lh` aparece apenas uma vez na tabela 3), mesmo tendo uma construção inicial inferior a `lh`. A primeira perturbação escolhida nas configurações apresentadas é a perturbação forte, que é o tipo de alteração mais brusca na estrutura da solução atual, indicando uma necessidade recorrente do método em escapar de mínimos locais. Como explicado na seção 3.1, na nossa gramática existe a possibilidade de serem realizadas duas perturbações em sequência, apesar de não ter sido o caso para as instâncias de teste utilizadas. Com relação às buscas locais, vemos um domínio dos movimentos de `swap` e `three_chain`. Em todos os cenários testados, aceitar uma nova solução sempre, independente de seu valor de função, gerou resultados de melhor qualidade.

Ao analisar a Tabela 5, vemos que a melhor configuração considerando todas as instâncias é diferente apenas na "Solução Inicial" e na "LS 1", quando comparada ao conjunto *RanReal* de 240 vértices (Tabela 4). Isso nos mostra que para a amostra de testes considerada, o algoritmo gerado pela escolha dos atributos "*lh, strong, swap, three chain, swap, none, accept always*" se mostra o mais robusto para todos os casos.

Tabela 1: Resultados para o grupo de instâncias *RanInt* (120 vértices).

Solução Inicial	Perturbação	LS 1	LS 2	LS Perturbação 1	LS Pertubação 2	Critério de Aceitação
lh	strong	swap	three chain	swap	none	accept always

Fonte: Elaborada pelos autores.

Tabela 2: Resultados para o grupo de instâncias *RanInt* (240 vértices).

Solução Inicial	Perturbação	LS 1	LS 2	LS Perturbação 1	LS Pertubação 2	Critério de Aceitação
lh	strong	swap	three chain	swap	none	accept always
lh	strong	insertion	three chain	insertion	none	accept always

Fonte: Elaborada pelos autores.

Tabela 3: Resultados para o grupo de instâncias *RanReal* (120 vértices).

Solução Inicial	Perturbação	LS 1	LS 2	LS Perturbação 1	LS Pertubação 2	Critério de Aceitação
lh	strong	swap	three chain	swap	none	accept always
wj	strong	insertion	three chain	insertion	none	accept better equal

Fonte: Elaborada pelos autores.

Tabela 4: Resultados para o grupo de instâncias *RanReal* (240 vértices).

Solução Inicial	Perturbação	LS 1	LS 2	LS Perturbação 1	LS Pertubação 2	Critério de Aceitação
wj	strong	insertion	three chain	swap	none	accept always
gc	strong	insertion	three chain	swap	none	accept always

Fonte: Elaborada pelos autores.

Tabela 5: Resultados para os grupos de instâncias *RanInt* (120 vértices), *RanInt* (240 vértices), *RanReal* (120 vértices) *RanReal* (240 vértices).

Solução Inicial	Perturbação	LS 1	LS 2	LS Perturbação 1	LS Pertubação 2	Critério de Aceitação
lh	strong	swap	three chain	swap	none	accept always

Fonte: Elaborada pelos autores.

6. Conclusões

A configuração automática de algoritmos vem sendo utilizada a fim de evitar um trabalho manual de configuração, e impedir que um tipo específico de delineamento de algoritmos seja utilizado para instâncias que possuem características diferentes entre si. O pacote *Irace* é uma ferramenta que possibilita a configuração automática de maneira eficaz. Após estabelecer as regras da gramática, determinando quais os parâmetros que serão calibrados pelo *Irace*, basta realizar as alterações necessárias nos arquivos que determinam como o *Irace* irá funcionar.

Essa abordagem traz flexibilidade na proposição de métodos de solução, mas deve ser implantada com alguns cuidados. Primeiramente, o tempo que o *Irace* leva para encontrar uma configuração adequada pode ser elevado, se o número máximo de experimentos não for escolhido adequadamente. Na prática, para uma boa calibração, aconselha-se valores variando de 1.000 a 10.000. A quantidade de instâncias de treinamento é outro gargalo do método, e uma escolha que não prioriza a diversidade dos problemas pode levar a resultados tendenciosos, priorizando alguns componentes da gramática em detrimento de outros. Na prática, deve-se priorizar problemas testes que englobem todas as características das instâncias, que abranjam ao menos 10% da quantidade final testada. Por fim, salienta-se a importância de uma programação estruturada e parametrizada dos algoritmos, a fim de deixar o código mais flexível a possíveis alterações na gramática.

Agradecimentos. Os autores agradecem o apoio do CNPq (projeto 420348/2016-6), da FAPEMIG (projeto TEC-APQ-02694-16), do Programa Institucional de Bolsas de Iniciação Científica da UFLA (PIBIC/UFLA) e da Fundação de Desenvolvimento Científico e Cultural (FUNDECC/UFLA).

Referências

Allahyari, S., Salari, M. e Vigo, D. A hybrid metaheuristic algorithm for the multi-depot covering tour vehicle routing problem. *European Journal of Operational Research*, v. 242, n. 3, p. 756–768, 2015.

Bezerra, L. C. T., López-Ibáñez, M. e Stützle, T. Automatic generation of multi-objective ACO algorithms for the bi-objective knapsack. In: *International Conference on Swarm Intelligence*. Springer, Berlin, Heidelberg, 2012. p. 37–48.

Brum, A. e Ritt, M. Automatic design of heuristics for minimizing the makespan in permutation flow shops. In: *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2018. p. 1–8.

Coelho, V. N., Grasas, A., Ramalhinho, H., Coelho, I. M., Souza, M. J. F. e Cruz, R. C. An ILS-based algorithm to solve a large-scale real heterogeneous fleet VRP with multi-trips and docking constraints. *European Journal of Operational Research*, v. 250, n. 2, p. 367–376, 2016.

Derbel, H., Jarboui, B., Hanafi, S. e Chabchoub, H. An iterated local search for solving a location-routing problem. *Electronic Notes in Discrete Mathematics*, v. 36, p. 875–882, 2010.

Derbel, H., Jarboui, B., Hanafi, S. e Chabchoub, H. Genetic algorithm with iterated local search for solving a location-routing problem. *Expert Systems with Applications*, v. 39, n. 3, p. 2865–2871, 2012.

Dong, X., Huang, H. e Chen, P. An iterated local search algorithm for the permutation flowshop problem with total flowtime criterion. *Computers & Operations Research*, v. 36, n. 5, p. 1664–1669, 2009.

Franzin, A. e Stützle, T. Revisiting simulated annealing: A component-based analysis. *Computers & Operations Research*, v. 104, p. 191–206, 2019.

Gallego, M., Laguna, M., Martí, R. e Duarte, A. Tabu search with strategic oscillation for the maximally diverse grouping problem. *Journal of the Operational Research Society*, v. 64, n. 5, p. 724–734, 2013.

Hutter, F., Hoos, H. H., Leyton-Brown, K. e Stützle, T. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, v. 36, p. 267–306, 2009.

Johnes, J. Operational research in education. *European Journal of Operational Research*, v. 243, n. 3, p. 683–696, 2015.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H. e Leyton-Brown, K. SATenstein: Automatically building local search SAT solvers from components. *Artificial Intelligence*, v. 232, p. 20–42, 2016.

Kramer, R., Cordeau, J.-F. e Iori, M. Rich vehicle routing with auxiliary depots and anticipated deliveries: An application to pharmaceutical distribution. *Transportation Research Part E: Logistics and Transportation Review*, v. 129, p. 162–174, 2019.

Lai, X. e Hao, J.-K. Iterated maxima search for the maximally diverse grouping problem. *European Journal of Operational Research*, v. 254, n. 3, p. 780–800, 2016.

López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L. P., Birattari, M. e Stützle, T. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, v. 3, p. 43–58, 2016.

Lourenço, H. R., Martin, O. C. e Stützle, T. Iterated local search. In: Glover, F. e Kochenberger, G.A. (eds.) *Handbook of Metaheuristics*, p. 320–353. Springer, 2003.

Lourenço, H. R., Martin, O. C. e Stützle, T. Iterated local search: Framework and applications. In: Gendreau, M. e Potvin, J.-Y. (eds.) *Handbook of Metaheuristics*, p. 129–168. Springer, 2019.

Lourenço, N., Pereira, F. B. e Costa, E. Unveiling the properties of structured grammatical evolution. *Genetic Programming and Evolvable Machines*, v. 17, n. 3, p. 251–289, 2016.

Mascia, F., López-Ibáñez, M., Dubois-Lacoste, J. e Stützle, T. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & Operations Research*, v. 51, p. 190–199, 2014.

Meignan, D. e Knust, S. A neutrality-based iterated local search for shift scheduling optimization and interactive reoptimization. *European Journal of Operational Research*, v. 279, n. 2, p. 320–334, 2019.

Moreira, J. P. G. e Ritt, M. Evolving task priority rules for heterogeneous assembly line balancing. In: *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2019. p. 1423–1430.

Nguyen, V.-P., Prins, C. e Prodhon, C. A multi-start iterated local search with tabu list and path relinking for the two-echelon location-routing problem. *Engineering Applications of Artificial Intelligence*, v. 25, n. 1, p. 56–71, 2012.

Nikolaev, A. G. e Jacobson, S. H. Simulated annealing. In: Gendreau, M. e Potvin, J.-Y. (eds.) *Handbook of Metaheuristics*, p. 1–39. Springer, 2010.

Pagnozzi, F. e Stützle, T. Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. *European Journal of Operational Research*, v. 276, n. 2, p. 409–421, 2019.

R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. 2017. Disponível em: https://www.r-project.org/. Acesso em: 17/02/2020.

Souza, M. e Ritt, M. An automatically designed recombination heuristic for the test-assignment problem. In: *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2018. p. 1–8.

Subramanian, A., Battarra, M. e Potts, C. N. An iterated local search heuristic for the single machine total weighted tardiness scheduling problem with sequence-dependent setup times. *International Journal of Production Research*, v. 52, n. 9, p. 2729–2742, 2014.

Weitz, R. R. e Jelassi, M. T. Assigning students to groups: a multi-criteria decision support system approach. *Decision Sciences*, v. 23, n. 3, p. 746–757, 1992.