Pesquisa Operacional para o Desenvolvimento
Vol. 9, n. 2, p. 48-61, 2017

ISSN 1984-3534

TUTORIAL CONVIDADO

JULIA E JuMP: NOVAS FERRAMENTAS PARA PROGRAMAÇÃO MATEMÁTICA[1]

Pedro Belin Castellucci[a]*

[a]Instituto de Ciências Matemática e de Computação
Universidade de São Paulo - USP, São Carlos-SP, Brasil

RESUMO

Existem diversas linguagens de programação para aplicações gerais ou científicas. No entanto, particularmente em aplicações científicas, é comum se encontrar linguagens adequadas para certas tarefas (desenvolvimento de algoritmos, por exemplo) e inadequadas para outras (geração de gráficos para visualização de resultados). Julia foi criada como uma alternativa que combinasse as vantagens de diversas linguagens. Neste tutorial, Julia é apresentada como uma opção para o desenvolvimento de soluções no contexto de Programação Matemática, com foco em Programação Linear Inteira Mista, juntamente com seu pacote JuMP. O pacote JuMP fornece uma linguagem de modelagem independente do *software* de otimização utilizado, inclusive permitindo a implementação de *callbacks*. Aqui, são apresentados guias de instalação e de desenvolvimento, além de exemplos com funções usualmente utilizadas e *callbacks*.

Palavras-chave: Programação Linear Inteira Mista, Linguagens de programação, Linguagens de modelagem.

ABSTRACT

There are many programming languages for developing general purpose or scientific applications. However, particularly for scientific applications, it is common to find languages suitable for certain tasks (e. g. algorithm development) and unsuitable for others (e. g. plotting results). Julia was created as an option, which combines the advantages of different languages. In this tutorial, Julia is presented as an alternative for developing solutions in the context of Mathematical Programming, focusing on Mixed Integer Linear Programming, using the JuMP package. JuMP is a solver independent modeling language, which allows the implementation of callbacks. Here, we present installation and development guidelines, examples of common functions and callbacks.

Keywords: Mixed Integer Linear Programming, Programming languages, Modeling languages.

*Autor para correspondência. E-mail: pbc@icmc.usp.br

DOI: 10.4322/PODes.2017.005

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

1. Por que Julia e JuMP?

As linguagens de programação usualmente utilizadas no contexto de otimização costumam ter desvantagens em relação a produtividade ou desempenho. Como exemplos encontrados na literatura de otimização estão as linguagens C/C++ e Python (ou Java). Ambas estão em extremos opostos nos quesitos desempenho e produtividade. C/C++ é uma linguagem conhecida, dentro do contexto de otimização, por possibilitar a implementação de algoritmos de rápida execução, importantes para o estabelecimento de referências do estado-da-arte. No entanto, é uma linguagem difícil de ser dominada e que exige conhecimentos específicos de computação como hierarquia, gerência de memória, arquitetura e sistemas operacionais, levando a uma baixa produtividade em relação a outras linguagens. Do lado oposto, tem-se Python que possui diversas caixas de ferramentas com funções de alto nível e expressividade, possibilitando a implementação de algoritmos sofisticados com um número relativamente pequeno de linhas de código, embora possua um desempenho inferior ao do C/C++ em algumas ordens de grandeza (julialang.org/benchmarks).

Julia é uma linguagem de propósito geral e código aberto que visa um bom desempenho e produtividade. Julia foi criada por usuários de Matlab, Lisp, Python, Ruby, Perl, R e C/C++, identificando que essas linguagens eram poderosas para algumas tarefas importantes de computação científica, mas muito deficiente para outras (Bezanson et al., 2012). Embora reconhecedores de sua ambição, os criadores pretendem, com Julia, ter uma linguagem capaz de agregar as vantagens de diversas linguagens de programação como a produtividade do Python e a eficiência do C. Criada em 2012 por um grupo de quatro pesquisadores, o repositório da linguagem (github.com/JuliaLang/julia) conta com mais de 500 colaboradores e com diversos recursos online (lista de e-mails, canal do IRC, GitHub, blogs, canais de vídeos e redes sociais). Além disso, existem grupos locais de usuários em diversas cidades do mundo como Chicago (E.U.A.), Ottawa (Canadá), Londres (Inglaterra), Beijing (China), Sydney (Austrália), São Paulo e Rio de Janeiro (Brasil) e conferências internacionais (vide *site* do projeto julialang.org).

Embora, até a redação deste artigo, não haja uma versão 1.0 de Julia, já existem publicações científicas e disciplinas acadêmicas ministradas utilizando a linguagem. Devido a sua alta produtividade e há existência de diversos pacotes científicos disponíveis é possível encontrar artigos científicos na área genética, computação paralela, álgebra linear, estatística, aprendizado de máquina entre outras; e, também, disciplinas como otimização linear e discreta, análise de risco, neurociência computacional, econometria, *big data*, processamento de sinais e biologia molecular. Para listas extensas o leitor é referido, novamente, ao *site* oficial do projeto.

Aqui, o foco é a aplicabilidade de Julia para Programação Matemática, em especial para Programação Linear Inteira Mista. Particularmente, é explorada a JuMP (*Julia for Mathematical Programming*). JuMP é uma linguagem de modelagem embutida no Julia que atualmente suporta mais de 10 resolvedores de Programação Linear, Linear Inteira Mista, Cônica de Segunda Ordem, Semidefinida e Não-linear (a lista completa e atualizada pode ser encontrada em juliaopt.org/JuMP.jl/0.17/. Uma das principais características da JuMP está a independência do resolvedor a ser utilizado, isto é, um modelo pode ser implementado em JuMP independente do resolvedor que será utilizado para sua solução.

Este texto, está organizado de forma a apresentar um guia para instalação da linguagem Julia, do pacote JuMP e sugestões para o ambiente de desenvolvimento (Seção 2). Em seguida, na Seção 3, é apresentado um exemplo simples para introdução da sintaxe de definição de variáveis, restrições e função objetivo. Além disso, é descrita uma implementação parcial do problema do caixeiro viajante. A implementação é finalizada e são exemplificados o uso de diferentes *callbacks* na Seção 4. Por fim, são tecidos alguns comentários finais e listados alguns materiais complementares (Seção 5).

2. Guia para Instalação e Sugestões

Nesta seção, são apresentadas algumas linhas guia para a instalação de Julia (Subseção 2.1) e JuMP (Subseção 2.2) e sugestões para ambientes para programação e ferramentas

(Subseção 2.3) que já foram utilizadas pelo autor. Para instruções detalhadas, o leitor é referido aos respectivos manuais de Julia (julialang.org/downloads), de JuMP (juliaopt.org/JuMP.jl/0.17/) e das ferramentas de desenvolvimento.

2.1. Julia

Julia pode ser instalada em ambientes Linux, MacOS e Windows. Para a instalação em ambientes Linux, por exemplo, basta adicionar os repositórios da Julia (ppa:staticfloat/juliareleases e ppa:staticfloat/julia-deps) ao gerenciador de pacotes utilizado e instalá-lo como um pacote padrão. Para MacOS e Windows o pacote dmg e o arquivo executável (.exe) estão disponíveis em (julialang.org/downloads).

A instalação básica do Julia, conta com uma aplicação de linha de comando interativa, semelhante ao Python e Matlab. Essa opção é interessante para a visualização rápida de operações simples e instalação de pacotes. Uma alternativa para um desenvolvimento mais sofisticado é a utilização de editores de texto ou ambientes de desenvolvimento. Algumas opções são apresentadas na Subseção 2.3.

2.2. JuMP

Para instalar a JuMP, basta executar o comando *Pkg.add*(*"JuMP"*) na linha de comando interativa da Julia. Esse é o procedimento padrão para instalar qualquer pacote da Julia. Para utilização do JuMP é necessário instalar também o pacote correspondente. Para o Gurobi, por exemplo, o comando é *Pkg.add*(*"Gurobi"*). A adição do pacote correspondente do Julia, não implica, necessariamente, na instalação do resolvedor no sistema, portanto, ainda pode ser necessária a instalação resolvedor correspondente. O Cbc é um exemplo de resolvedor que não necessita de processos adicionais para funcionar – necessita apenas do comando (*Pkg.add*(*"Cbc"*)). A instalação de resolvedores está além do escopo deste documento, mas pode ser encontrada nos respectivos manuais.

2.3. Ferramentas para Desenvolvimento

Além da linha de comando interativa e de editores de texto convencionais, é possível utilizar a distribução JuliaPro (juliacomputing.com/products/juliapro.html). Trata-se de uma distribuição do Julia que conta com compilador, *debugger*, ambiente de desenvolvimento, diversos pacotes pré-instalados (inclusive o JuMP) entre outras facilidades.

Outra opção, é a ferramenta online JuliaBox (juliabox.com), que permite o desenvolvimento e execução de código. Para o uso das funcionalidades padrão da Julia, que não inclui os resolvedores, não é necessária nenhum tipo de instalação. Ainda, é possível instalar pacotes como o JuMP, na conta do usuário. Um dos casos de uso interessantes para o JuliaBox é a realização de demonstrações e cursos, pois a ferramenta já oferece um ambiente funcional básico para o usuário, evitando possíveis problemas no processo de instalação. Também, o JuliaBox é interessante para criar e compartilhar Jupyter Notebooks.

O Jupyter Notebook é uma ferramenta do projeto Jupyter (jupyter.org) que permite a integração de texto, imagens, equações matemáticas e códigos-fonte de diversas linguagens além de Julia. Em um Jupyter Notebook, é possível executar e visualizar os resultados de algoritmos, permitindo, inclusive, sua alteração. Com isso, tem-se um certo grau de interatividade do leitor com o material. Uma versão implementada em Jupyter Notebook deste material está disponível em (Castellucci, 2017).

Tanto o JuliaBox quanto o Jupyter Notebook são ferramentas interessantes, no contexto de Programação Matemática, para disponibilização de materiais didáticos para tutoriais, palestras, cursos e até mesmo livros (vide github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks). Para o desenvolvimento de aplicações mais elaboradas, alternativas como o ambiente Atom ou Visual Studio Code devem ser consideradas. O Atom (atom.io) é um editor de texto multiplataforma e, com a instalação do *plugin* Juno (junolab.org),

é uma alternativa para ambiente de desenvolvimento para Julia. Outra opção é o Visual Studio Code (code.visualstudio.com/download) com o respectivo *plugin* para desenvolvimento em Julia, também multiplataforma. Ambos possuem funcionalidades com o realce de sintaxe, autocompletamento de código, ferramentas de *debug* e integração com gerenciadores de versões.

3. Primeiros Passos

Nesta seção, são apresentados dois exemplos de utilização do pacote JuMP. O primeiro, Subseção 3.1, introduz a sintaxe de definição de modelo, função objetivo, variáveis e restrições. O segundo (Subseção 3.2) é uma implementação parcial do clássico problema do caixeiro viajante, que utiliza funções para a construção de modelos com porte tipicamente encontrados em aplicações de Otimização Linear Inteira. A implementação é concluída na Seção 4. São utilizados os pacotes JuMP, Cbc, Gurobi, Plotly e PlotRecipes. Para instalação desses pacotes, basta executar os comandos a seguir em terminal interativo do Julia ou do Jupyter Notebook.

```
Pkg.add("JuMP")
Pkg.add("Cbc")
Pkg.add("Gurobi")
# Pkg.add("CPLEX") # Adiciona o CPLEX

# Pacotes para visualizar resultados
Pkg.add("Plotly")
Pkg.add("PlotRecipes")
```

O leitor é convidado a executar todos códigos apresentados. Para isso, pode-se utilizar o próprio Jupyter Notebook. Uma opção é utilizar o código-fonte *JuliaJumpTutorial.jl* disponível em (Castellucci, 2017). Para executá-lo basta utilizar o comando:

julia JuliaJumpTutorial.jl

em um terminal em um computador com uma instalação de Julia e do Gurobi. Para utilizar outro resolvedor, além do Gurobi, é necessária a alteração de trechos do código. As alterações para utilização do CPLEX são indicadas ao longo do documento. Além disso, destaca-se que os códigos apresentados foram testados com a versão 0.5 do Julia e algumas alterações podem ser necessárias caso seja utilizada outra versão.

3.1. Um Exemplo Simples

Considere o problema de otimização (1)-(5).

$$\text{Max } x+2y \quad (1)$$

$$\text{s.a } 2x+3y \leq 25 \quad (2)$$

$$3x+2y \leq 37 \quad (3)$$

$$0 \leq x \leq 15 \quad (4)$$

$$y \in Z^{+} \quad (5)$$

Para implementar o modelo utilizando JuMP, o primeiro passo é incluir os pacotes necessários e criar o objeto que armazena o modelo, com o resolvedor a ser utilizado. Nesse exemplo, é utilizado o Cbc, na Subseção 3.2 é utilizado o Gurobi.

```
using JuMP
```

```
using Cbc
# Criando o modelo com o resolvedor Cbc
model = Model(solver=CbcSolver());
```

Em seguida, são definidas as variáveis com os respectivos domínios. Também, é necessário especificar o modelo ao qual as variáveis estão associadas. Então, podem ser definidas a função objetivo e as restrições, que também devem estar associadas a um modelo.

```
# 0 <= x <= 15
@variable(model, x, lowerbound=0, upperbound=15)

# y é variável inteira não negativa
@variable(model, y, lowerbound=0, Int)

# Problema de maximização da função x + 2y
@objective(model, Max, x + 2y)

# Restrições
@constraint(model, 2x + 3y <= 25)
@constraint(model, 3x + 2y <= 37)

println(model)
```

A solução do modelo é obtida com o método *solve*. Esse método retorna o estado da solução encontrada, então, o código a seguir verifica se trata-se da solução ótima e a imprime, caso contrário, exibe uma mensagem com informações sobre o processo de solução.

```
status = solve(model)

if status == :Optimal
        println("Solução ótima encontrada!")
        println("x = $ (getvalue(x)), y = $ (getvalue(y))")
        println("F.O. $ (getobjectivevalue(model))")
else
        println("Solução ótima não foi encontrada.")
end
```

Note que o exemplo é simples e não representa, em questão de número de variáveis e restrições, os problemas de interesse acadêmico ou prático da área de Pesquisa Operacional. Na Subseção 3.2, é apresentada uma implementação parcial, que é finalizada na Seção 4, do problema do caixeiro viajante. Nessa implementação, são apresentadas as sintaxes e funções que permitem modelar problemas cujo porte é correspondente ao estado-da-arte em pacotes computacionais baseados em *branch* e *cut* para a solução de problemas lineares inteiros mistos.

3.2. O Problema do Caixeiro Viajante

Não está no escopo deste trabalho discutir a importância, métodos de solução nem diferentes modelos para o problema do caixeiro viajante. Aqui, é apresentando um modelo para o problema, que é utilizado como exemplo para explorar o pacote computacional JuMP. Para uma referência sobre o problema o leitor é referido ao livro de Applegate et al. (2011).

Sejam V um conjunto de pontos a serem visitados por um viajante e c_{ij} o custo de se deslocar do ponto i para o ponto j, $i, j \in V$, $i \neq j$. O problema do caixeiro viajante consiste em encontrar uma rota de menor custo que visite todos os pontos exatamente uma vez. Uma formulação para o problema pode ser definida da seguinte forma. Sejam x_{ij} variáveis binárias

indicando se o viajante utiliza o caminho (i, j), $i, j \in V, i \neq j$ como parte de sua rota. Com isso, tem-se o modelo (6)-(10).

$$\text{Min} \quad \sum_{i \in V} \sum_{\substack{j \in V \\ j \neq i}} c_{ij} x_{ij} \tag{6}$$

$$\text{s.a.} \quad \sum_{\substack{i \in V \\ i \neq j}} x_{ij} = 1 \qquad j \in V \tag{7}$$

$$\sum_{\substack{j \in V \\ j \neq i}} x_{ij} = 1 \qquad i \in V \tag{8}$$

$$\sum_{\substack{(i,j) \in S \\ j \neq i}} x_{ij} \leq |S| - 1 \qquad S \subset V \tag{9}$$

$$x_{ij} \in \{0,1\} \qquad i, j \in V, i \neq j \tag{10}$$

A função objetivo (6) minimiza o custo do percurso, enquanto que as restrições (7) garantem que cada vértice é alcançado, o viajante sai de todo vértice (de acordo com as restrições (8)) e não há subciclos no percurso devido às restrições (9).

Antes de implementar o modelo utilizando JuMP e Julia, considere o problema exemplo definido pelo código a seguir. O problema é definido pela posição cartesiana de cada ponto a ser visitado, então o código calcula os valores c_{ij}, $i, j \in V, i \neq j$, utilizando a distância euclidiana entre i e j. A Figura 1 ilustra a posição relativa dos vértices a serem visitados.

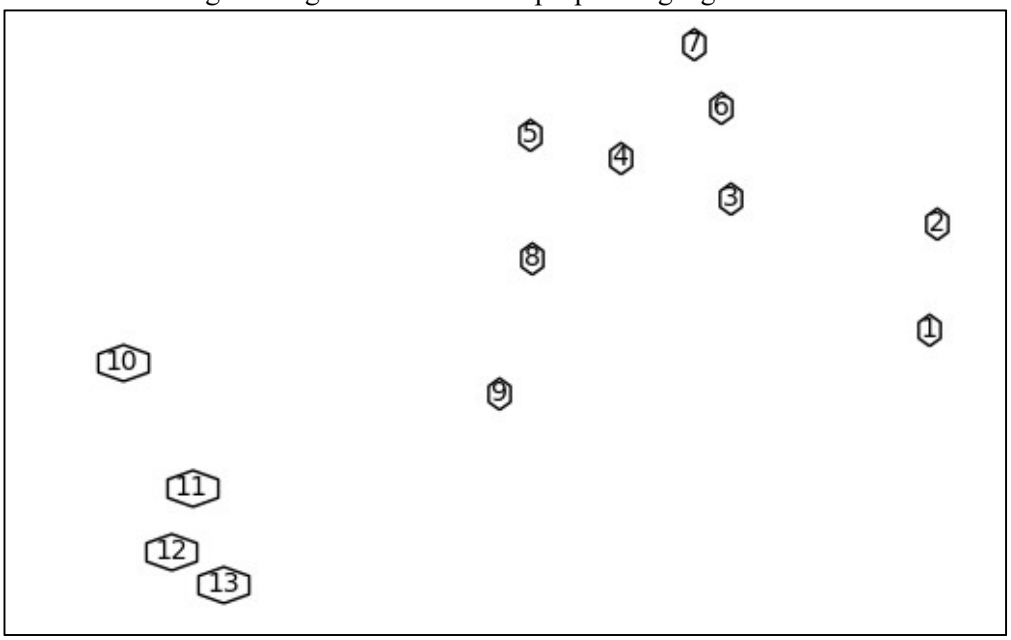
```
# Posições de cada ponto a ser visitado
citiesDict = Dict{Int, Any}()
citiesDict[1] = (523, 418)
citiesDict[2] = (527, 566)
citiesDict[3] = (435, 603)
citiesDict[4] = (386, 660)
citiesDict[5] = (346, 692)
citiesDict[6] = (431, 730)
citiesDict[7] = (419, 818)
citiesDict[8] = (347, 520)
citiesDict[9] = (332, 330)
citiesDict[10] = (165, 374)
citiesDict[11] = (196, 198)
citiesDict[12] = (187, 108)
citiesDict[13] = (210, 63)

nCities = length(citiesDict)
# Matriz para amazenar as distâncias entre os pontos
 c = zeros(nCities, nCities)
```

Pode-se utilizar a própria Julia para construção de uma visualização do exemplo. O trecho de código a seguir depende de bibliotecas para construção de gráficos. A instalação dessas bibliotecas está além do escopo deste documento, por isso, as linhas de código correspondentes à plotagem estão comentadas.

```
# Pkg.add("PyPlot") # # Pacote para plotagem
# using PlotRecipes # Pacote usado para plotar a figura
```

```
# Backend utilizado para plotagem
# pyplot() # Melhor visualização, mas mais dependências
# plotly() # Mais simples

# Definição da posição de cada vértice para o desenho
posX, posY = [], []

for i in sort(collect(keys(citiesDict)))
        posI = citiesDict[i]
        for j in sort(collect(keys(citiesDict)))
                posJ = citiesDict[j]
                c[i,j]=((posI[1]-posJ[1])^2+(posI[2]-posJ[2])^2)^0.5
        end
        append!(posX, posI[1])
        append!(posY, posI[2])
end

# Configuração do desenho
# graphplot(1:nCities, 1:nCities, names=1:nCities,
        # x=posX, y=posY, fontsize=10, m=:white, l=:black)
```

Figura 1. Exemplo a ser resolvido. Os vértices 1, ..., 15 indicam pontos a serem visitados. A figura foi gerada utilizando a própria linguagem Julia.



Fonte: Elaborada pelo autor.

O primeiro passo para implementação do modelo é a inclusão dos pacotes computacionais que serão utilizados. Para esse exemplo, foram utilizados o pacote JuMP e o Gurobi. Em seguida, define-se o objeto modelo, no qual podem-se definir parâmetros para solução: no caso, foram limitados tempo de solução, número de processadores, desabilitadas as heurísticas do Gurobi – é implementada uma heurística na Seção 4.3 – e suprimida a impressão de informações sobre o procedimento de solução. Com o modelo, definem-se as variáveis, a função objetivo e as restrições do problema, respectivamente. Note o uso da sintaxe para a definição de variáveis matriciais e a função *sum* como atalho para a definição de somatórios.

```
using JuMP
```

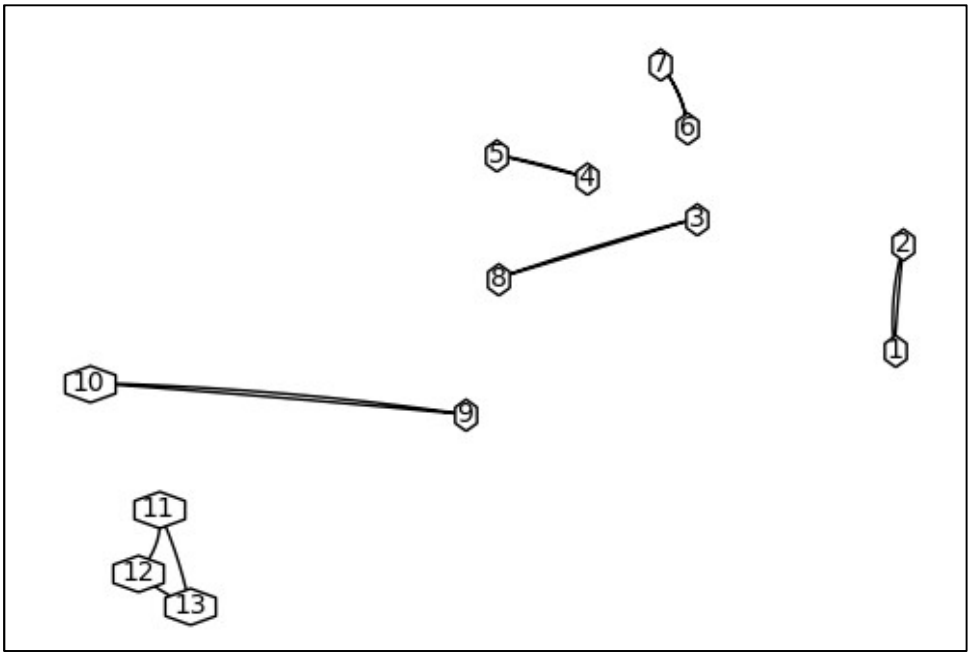
```
using Gurobi
# using CPLEX

# Definição do modelo e parâmetros para a solução
model=Model(solver=GurobiSolver(TimeLimit=20, Threads=1,
                Heuristics=0.0, OutputFlag=0))

# Para o CPLEX pode-se usar
# model = Model(solver=CplexSolver(CPX_PARAM_TILIM=20,
#                 CPX_PARAM_THREADS=1, CPX_PARAM_HEURFREQ=0,
#                 CPX_PARAM_SCRIND=0))

# Definição da variável matricial xij
# Note que a variável nCities foi definida anteriormente
@variable(model,x[i=1:nCities,j=1:nCities;i !=j],Bin)

# Função objetivo
@objective(model, Min, sum(c[i, j] * x[i, j]
        for i in 1:nCities, j in 1:nCities if i != j))

# Restrições para garantir a saída de todo vértice
for i in 1:nCities
        @constraint(model,sum(x[i,j] for j in 1 : nCities if i != j ) == 1)
end

# Restrições para garantir a chegada em cada vértice
for j in 1:nCities
        @constraint(model,sum(x[i,j] for i in 1:nCities if i != j) == 1)
end
```

A solução é realizada pela chamada do método *solve* e pode-se utilizar a própria Julia para visualização (vide Figura 2).

```
status = solve(model)

if status == :Optimal
        edgeOrigin = []
        edgeDest = []

        for i in keys(citiesDict)
                for j in keys(citiesDict)
                        if i != j && getvalue(x[i, j]) > 0.99
                                append!(edgeOrigin, i)
                                append!(edgeDest, j)
                        end
                end
        end

        display(
                graphplot(
                        edgeOrigin, edgeDest, names=1:nCities,
                        x=posX, y=posY, fontsize=10,
                        m=:white, l=:black
```

```
                    )
            )
else
            println("Solução ótima não encontrada!")
end
```

Figura 2. Solução parcial do exemplo, sem a restrições de subciclo (9). Essa figura foi gerada utilizando Julia.



Fonte: Elaborada pelo autor.

Devido à ausência das restrições de eliminação de subciclo, a solução não corresponde a uma rota desejada. Uma desvantagem das restrições de eliminação de subciclos (9) é o seu número exponencial em relação ao número de pontos a serem visitados. Por isso, enumerar todas essas restrições *a priori* é factível apenas para exemplos com poucos pontos e uma abordagem mais eficiente é adicionar restrições apenas quando são necessárias. Na Seção 4.1, é utilizada a *callback lazy constraints* da JuMP para inclusão das restrições de eliminação de subciclo de forma eficiente.

4. Callbacks

Callbacks são funções que permitem ao usuário personalizar o procedimento do resolvedor. Em geral, são utilizadas para explorar características específicas dos problemas ou obter informações detalhadas do andamento sobre o processo de solução. No momento da escrita deste artigo, JuMP dispunha de três tipos de *callbacks* para customização, que são apresentadas nas Subseções 4.1 – 4.3, além de *callbacks* informativas, que possuem o uso relativamente simples. Informações detalhadas sobre essa funcionalidade podem ser encontradas na documentação oficial da JuMP (juliaopt.org/JuMP.jl/0.17/). Ainda, destaca-se que o comportamento específico de cada uma das *callbacks* pode depender do resolvedor utilizado e recomenda-se consultar seu respectivo manual. Inclusive, o número de resolvedores que suportam seu uso é limitado (Gurobi, CPLEX, GLPK e SCIP eram os suportados no momento da escrita deste artigo).

4.1. *Lazy Constraints*

Uma das utilidades da *callback lazy constraints* é em casos em que o conjunto completo de restrições é muito numeroso para ser explicitado. Para utilizar essa *callback*, é necessário implementar uma função que é chamada sempre que o resolvedor encontrar uma solução inteira, permitindo, então, a adição de restrições ao modelo. Como exemplo, segue uma implementação de função que identifica o menor subciclo presente na solução relaxada do caixeiro viajante e, caso ele exista, adiciona uma restrição que o elimina. Note que é possível utilizar outros pacotes de Julia dentro de uma *callback*, como foi feito no exemplo com o pacote *Graphs*, permitindo uma manipulação simples do grafo da solução.

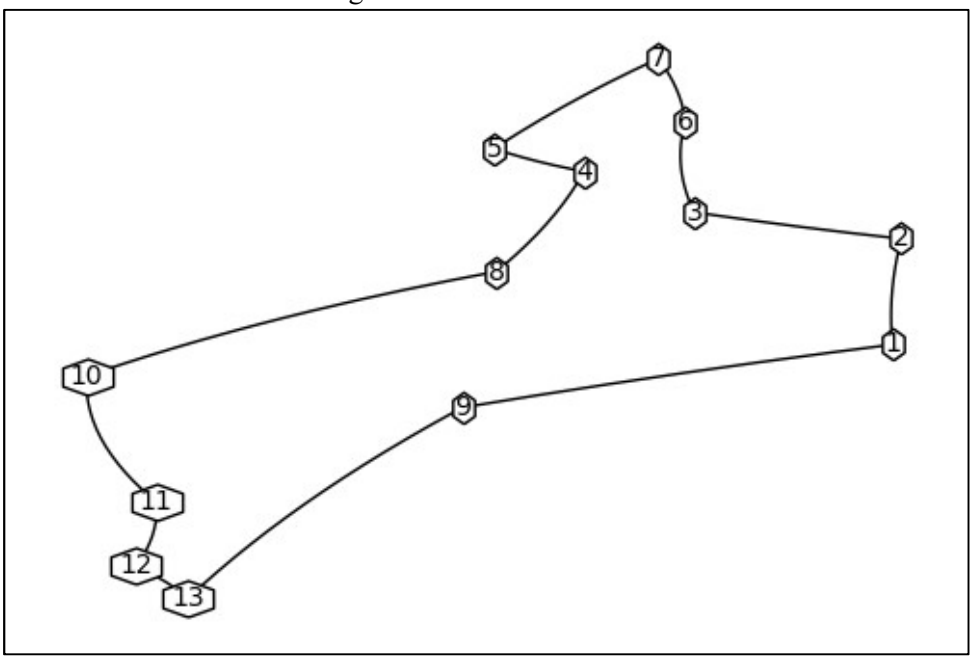
```
using Graphs

function lazyConstraintsCallback(cb)
        # Criando um grafo
        g = simple_graph(nCities, is_directed=false)

        for i in 1:nCities, j in 1:nCities
                if i != j
                        if getvalue(x[i, j]) > 0.01
                                add_edge!(g, i, j) # Arestas
                        end
                end
        end

        # Encontrando os componentes conexos do grafo
        cc = connected_components(g)

        if length(cc) > 1
                # Caso só haja uma componente conexa não há subciclo e
                # não se adiciona nenhuma restrição
                minTour = sort(cc, by=length)[1]
                subtourLhs = AffExpr()
                # Encontrando arestas do subciclo
                for i in minTour
                        for j in minTour
                                if i != j && getvalue(x[i, j]) > 0.01
                                        subtourLhs += x[i, j]
                                end
                        end
                end

                # Adicionando a restrição
                @lazyconstraint(cb,subtourLhs<=length(minTour)-1)
        end
end # Function
```

Com a função de inclusão de restrições de eliminação de subciclo, o único passo necessário, além da implementação do modelo, é informar ao resolvedor a função a ser considerada como *callback*. Isso deve ser feito antes da chamada do método *solve*. Como exemplificado a seguir para obtenção da solução ilustrada na Figura 3.

```
# [... Código com a implementação do modelo ...]
```

```
addlazycallback(model, lazyConstraintsCallback)

solve(model)

edgeOrigin = []
edgeDest = []

for i in keys(citiesDict)
        for j in keys(citiesDict)
                if i != j && getvalue(x[i, j]) > 0.01
                        append!(edgeOrigin, i)
                        append!(edgeDest, j)
                end
        end
end

graphplot(edgeOrigin, edgeDest, names=1:nCities,
        x=posX, y=posY, fontsize=10, m=:white, l=:black)
```

Figura 3. Exemplo de solução completa, utilizando a *callback lazy constraints*. Essa figura foi gerada utilizando Julia.



Fonte: Elaborada pelo autor.

No caso do Gurobi, é possível visualizar o número de restrições que foram durante o processo de resolução. As informações disponíveis podem variar de acordo com o pacote computacional utilizado e podem ser consultadas no respectivo manual.

4.2. *User Cuts*

A *callback user cuts* pode ser interessante caso o usuário conheça desigualdades válidas (cortes) para o seu problema e seja custoso adicioná-las *a priori*. Essa *callback* é chamada sempre que o resolvedor encontra uma solução ótima para a relaxação linear do problema. Embora seja chamada em situação diferente, a sua implementação é semelhante ao caso de *lazy constraints*.

```
using Graphs

function userCutsCallback(cb)

        # Criando um grafo
        g = simple_graph(nCities, is_directed=false)

        for i in 1:nCities, j in 1:nCities
                if i != j
                        if getvalue(x[i, j]) > 0.01
                                add_edge!(g, i, j) # Arestas
                        end
                end
        end
        # Encontrando os componentes conexos do grafo
        cc = connected_components(g)

        minTour = sort(cc, by=length)[1]
        subtourLhs = AffExpr()

        # Encontrando arestas do subciclo
        countEdges = 0
        for i in minTour
                for j in minTour
                        if i != j && getvalue(x[i, j]) > 0.01
                                subtourLhs += x[i, j]
                                countEdges += 1

                        end
                end
        end

        # Adicionando a restrição
        if length(cc) > 1
                @usercut(cb, subtourLhs <= length(minTour)-1)
        elseif countEdges > length(vertices(g))
                @usercut(cb, subtourLhs==length(vertices(g)))
        end
end # Function
```

É importante destacar que as restrições adicionadas utilizando *user cuts* não devem alterar o conjunto de soluções factíveis do problema original, elas têm a função, apenas, de melhorar o limitante dual do problema. Portanto, para o exemplo do caixeiro viajante aqui apresentado, é necessário utilizar as duas *callbacks*.

```
addcutcallback(model, userCutsCallback)
addlazycallback(model, lazyConstraintsCallback)

solve(model)
```

4.3. *User Heuristics*

A *callback user heuristics* permite que sejam submetidas soluções heurísticas para o resolvedor. O comportamento específico dessa *callback* depende do resolvedor que está sendo

utilizado e é necessário consultar o respectivo manual. O CPLEX, por exemplo, não permite a submissão de duas soluções heurísticas em uma mesma chamada da função de *callback* enquanto o Gurobi permite. Para o exemplo do caixeiro viajante, considere uma heurística que inclui na rota arestas correspondentes as variáveis com valores no intervalo (0,1; 1]. Isto é, atribui 1 a variáveis cuja a solução da relaxação é maior do que 0,1. Esse procedimento constrói uma solução candidata, que é submetida ao Gurobi, caso ela seja factível, ela é incluída nas soluções encontradas até o momento, caso contrário ela é desconsiderada. Esse procedimento pode ser implementado da seguinte maneira.

```
function userHeuristicsCallback(cb)
        for i in 1:nCities, j in 1:nCities
                if i != j
                        if getvalue(x[i, j]) > 0.1
                                # Atribuindo um valor à variável
                                setsolutionvalue(cb, x[i, j], 1)
                        end
                end
        end

        # Submetendo solução
        addsolution(cb)
end
```

Para o exemplo, foram atribuídos valores a todas as variáveis do problema. Isso não é uma exigência, pode-se submeter uma solução parcial, atribuindo valores à apenas algumas variáveis. Novamente, o tratamento dos valores restantes depende do resolvedor. Para utilizar a função implementada como *callback* procede-se de maneira semelhante aos casos anteriores, podendo-se, inclusive, utilizar outras *callback* simultaneamente.

```
addcutcallback(model, userCutsCallback)
addlazycallback(model, lazyConstraintsCallback)
addheuristiccallback(model, userHeuristicsCallback)

solve(model)
```

5. Conclusões e Recursos Adicionais

Julia e JuMP são ferramentas computacionais recentes que se aproveitam das vantagens de diferentes linguagens de programação e pacotes computacionais para melhorar o compromisso entre produtividade e eficiência computacional em relação às alternativas usuais. Embora, tanto Julia quanto JuMP, ainda não possuam uma versão 1.0 a comunidade científica tem mostrado interesse no uso e desenvolvimento de tais ferramentas (vide, por exemplo, julialang.org/teaching, julialang.org/publications e Castellucci, 2017). Este tutorial tenta chamar atenção da comunidade científica e demais interessados, principalmente dos programadores matemáticos, para a linguagem Julia e mais especificamente para a biblioteca JuMP.

Além das referências já mencionadas ao longo do texto, o leitor é referido ao livro de Kwon (2016). Ao canal *www.youtube.com/user/JuliaLanguage*, onde podem ser encontradas palestras das conferências anuais de Julia (JuliaCon, realizada desde 2014) com conteúdos básicos e avançados da própria linguagem e de suas bibliotecas.

Agradecimentos. O autor agradece à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) pelo suporte financeiro (2015/14020-9 e 2013/07375-0).

Referências

Applegate, D. L., Bixby, R. E., Chvátal, V. e Cook, W. J. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2011.

Bezanson, J., Karpinski, S., Shah, V. e Edelman, A. *Why we Created Julia*. 2012. Disponível em: <julialang.org/blog/2012/02/why-we-created-julia>. Acesso em: 19/06/2017.

Castellucci, P. B. 2017. Disponível em: <github.com/pedrocastellucci/Mathematical ProgrammingCourse>. Acesso em: 01/06/2017.

Kwon, C. *Julia Programming for Operations Research: A Primer on Computing*. 2016. Disponível em: <juliabook.chkwon.net/>. Acesso em: 19/06/2017.