Comparative Study of AMPL, Pyomo and JuMP Optimization Modeling Languages on a Network Linear Programming Problem Example

Andrzej Karbowski, Krzysztof Wyskiel
Warsaw University of Technology, Faculty of Electronics and Information Technology, Nowowiejska 15/19, 00-665 Warszawa

Abstract: The purpose of this work is a comparative study of three languages (environments) of optimization modeling: AMPL, Pyomo and JuMP. The comparison will be based on three implementations of the shortest path problem formulated as a linear programming problem. The codes for individual models and differences between them will be presented and discussed. Various aspects will be taken into account, such as: simplicity and intuitiveness of implementation, availability of specific data structures for a LP network problems, etc.

Keywords: optimization, modeling languages, programming, shortest path problem, network problems, linear programming

1. Introduction

In the mid-1980s, along with the development of personal computers, the continuous increase in available computing power and subsequent generations of programming languages, the AML (Algebraic Modeling Languages) class of languages arised. It facilitated the process of creating optimization problem models and solving them [9]. AMPL is among the pioneers of this type of software [8], together with other languages, e.g., GAMS or AIMMS.

Today, in addition to specialized optimization modeling tools, there are also libraries and packages for general programming languages, e.g., Pyomo for Python or JuMP for Julia, playing the role of optimization modeling languages.

The aim of this work is a comparative study of three popular optimization modeling languages and their capabilities: AMPL, Pyomo and JuMP. The study consists of implementing in these three languages and solving in their environments a shortest path in a directed graph problem formulated in linear programming (LP) terms. The assessment of the listed environments was made taking into account various criteria, such as: availability of specific constructs for given problem classes (e.g., arcs in graph/network problems), ways to define constraints and objective functions, declarations and operations on input data sets, as well as convenience and effectiveness of implementation, expressed, for example, through the size of the code.

Autor korespondujący:
Andrzej Karbowski, A.Karbowski@ia.pw.edu.pl

Artykuł recenzowany
nadesłany 16.06.2021 r., przyjęty do druku 08.08.2021 r.

2. The shortest path in a directed graph problem

The problem presented in this section is one of the best known and widely described (discrete) optimization problems in the literature. There are many types of graph problems; from them looking for the shortest path between two indicated nodes in a directed graph was selected, because it can be easily formulated as a (continuous) linear programming problem, that is in the most standard way.

The mathematical model of the shortest path problem is as follows [7]:

$$\min_{x} \sum_{(i,j)\in A} c_{i,j} x_{i,j}$$

subject to

$$\forall i \in N: \quad \sum_{(i,j)\in A} x_{i,j} - \sum_{(j,i)\in A} x_{j,i} = \begin{cases} 1, & \text{if } i = s \\ -1, & \text{if } i = t \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$$x_{i,j} \geq 0, \qquad \forall (i,j) \in A$$

A graph is represented here as a set N of nodes and a set A of edges. Each edge is represented by a pair of nodes $(i, j) \in A$, $i, j \in N$ (beginning and end of the link). A transition cost (distance) c_{ij} is assigned to each edge. The decision variable x defines a path in the graph in such a way, that each x_{ij} equals 1 if the edge (i, j) belongs to the path and 0 otherwise. The start node of the path is marked as s and the terminal node as t.

3. Optimization modeling languages

This section will briefly discuss the basic features of subsequent optimization modeling environments, the installation process, and how to prepare them for work.

3.1. AMPL

AMPL (A Mathematical Programming Language) is the oldest optimization programming language among those discussed here, which has been developed since 1985 [8]. The AMPL package is commercial software, but there is also a free variant for academic purposes. This program has a fully functional, integrated development environment `amplide`, which includes a file explorer, a console for various language commands, related to loading, solving or displaying the model, and a text editor, in which we can create and edit the models and input data files.

This language is created from the beginning and does not depend on any external libraries, thanks to which it has a simple and clear syntax. It offers constructs for parameters, sets, variables, objective functions and constraints. It also supports more complex constructs for sets, that are useful, e.g., in graph problems. The model itself is saved in files with the .mod extension. In addition, AMPL has .dat files in which we can save input data for the model, according to the imposed syntax.

And finally, there are also .run files, that we can optionally use to write scripts involving, e.g., decomposed or even parallel optimization [10]. Usually we place in them instructions necessary to solve the model, i.e. loading a file with the model, a file with input data, choosing a solver, an optional list of parameters transferred to the solver and a command to call it.

3.2. Pyomo

Pyomo (Python Optimization Modeling Objects) is a collection of packages for the general purpose programming language Python [4, 5]. It provides tools and constructs needed to build models, declare variables, sets, solver interfaces, etc. The first version of Pyomo appeared in 2008; since then it has been constantly developed and maintained. The recommended by the authors method of exploitation is to use Pyomo together with one of the available scientific Python distributions. For this purpose, we can use for example Anaconda platform, available for download at https://www.anaconda.com/download/. The package includes also Spyder, an IDE for writing scripts and programs in Python and testing them in the embedded console.

As mentioned at the beginning, Pyomo is a collection of Python libraries. There are some constraints and dependencies. The syntax for functions and constructs offered must be compatible with the native language, which also forces a certain style of code writing, e.g., the use of appropriate indentation in conditional statement blocks or loops. On the other hand, we can also use Python methods. Modeling elements include parameters, sets, variables, objective functions, constraints, as well as expressions, which we can use e.g., to divide long mathematical operations into smaller elements to increase readability. We save the model in standard Python files with the .py extension. We have also .tab files with one or more parameters useful for long arrays. In addition, we have the ability to read data from csv, Json, xml, yaml, Excel and some relational databases. For the last three of them, however, additional Python libraries, that are not a part of Pyomo, are necessary.

3.3. JuMP

JuMP (Julia for Mathematical Programming) is a library for the fairly young language Julia (similarly to Pyomo for Python) [2, 3]. It is the youngest environment to create optimization models among those described here. The first version of Julia appeared in 2012, while the JuMP prototype was released in 2015 [2]. There are several versions of Julia's distribution. The recommended package is JuliaPRO, which we can download from https://juliacomputing.com/products/juliapro. It contains many different libraries including JuMP and CLP solver, so we can start work immediately after installation. It is available in a personal (free) and licensed for enterprises version. The package also includes the Juno IDE based on the Atom text editor.

Due to the fact that JuMP is a library for the Julia language, there are some dependencies and constraints associated with it. The syntax of the functions and constructs offered by the package must be compatible with the native language. This also applies to our style of writing program code, e.g., terminating blocks of conditional statements and loops with the keyword end. However, this also allows us to use Julia's native methods, including vector and matrix operations, which can often be useful. JuMP offers constructs for variables, constraints, objective functions and Affine expressions, which consist of three fields: coefficient vector, variable vector and constant.

Unlike the other languages discussed here, we don't have defined constructs for parameters and sets in JuMP. However, due to the rich set of mathematical functions offered by Julia, this is generally not a problem. JuMP also lacks an interface for loading data from files, which requires to use native language methods (as in Pyomo). On the other hand, it gives us the freedom to define and adopt a layout convenient for us in the files with model input data.

3.4. Available solvers and their use

All these languages have appropriate interfaces for communication with various solvers. The list of available solvers is quite large and includes both commercial (such as Cplex or Gurobi) and free (such as Ipopt or Cbc). In JuMP, we need an additional interface written in Julia to communicate with the solver. This is due to the fact that, unlike the other languages discussed here, JuMP does not generate files with the model in an appropriate format (e.g., .nl in AMPL), which are then transferred to the solver, but communicates with it directly in memory.

The way the environment is prepared varies slightly in each case. In AMPL to use a given solver we write the following command:

```
option solver "name";
```

where for "name" we substitute the name of the solver's executable file.

In Pyomo, we create the object for communication with the solver as follows:

```
solverFactory("name")
```

where, similarly to AMPL, we write the name of the solver's executable file.

JuMP can boast about the simplest solution. In Julia's console, it is enough to type `Pkg.add("solver name")`. In the case of a free solver, its source files will be downloaded along with the appropriate Julia interface, compiled and prepared for use. In the case of commercial solvers, we must have a working installation with an active license (in this case only the interface is downloaded). Then we import the appropriate package with the solver interface in the program code by the `using` command and we pass the constructor for this interface to the method responsible for the initialization of the model object, e.g.,

```
Model(solver = name())
```

where for "name" we substitute the name of the constructor method for the given interface.

4. The shortest path in a directed graph – implementation

This section will describe three implementations of the shortest path problem in a directed graph in all environments discussed in the paper. Each subsection will deal with the program code in a given modeling language. The obtained results will be described and compared in a separate section. In every case, the Cbc solver was used to solve this problem.

The naming of variables in different environments is consistent as far as possible. Where there are any additional variables that are not present in the others, they will be described in the subsection on a specific implementation.

Common variable names:
- `nodes` – set of graph nodes,
- `arcs` – set of graph edges,
- `x` – decision variable designating the edges used in the path,
- `cost` – vector containing the cost of transition (length) of the graph edge,
- `source` – starting node,
- `target` – end node.

4.1. AMPL

The implementation of this problem in AMPL looks like this:

```
set NODES;
set ARCS within {NODES cross NODES};

param source symbolic in NODES;
param target symbolic in NODES;
param COST {ARCS} >= 0;

var x {ARCS} >= 0;

minimize distance:
  sum{(i,j) in ARCS} COST[i,j] * x[i,j];

subject to cons1 {i in NODES}:
  sum{(i,j) in ARCS} x[i,j] - sum{(j,i) in ARCS} x[j,i] =
    if i=source then 1
    else if i=target then -1
    else 0;

data GraphData.dat;
option solver cbc;

solve;
display distance, x;
```

Listing 1. The shortest path in a directed graph – implementation in AMPL
Listing 1. Najkrótsza scieżka w grafie skierowanym – implementacja w AMPLu

All commands or constructs must end with a semicolon. Basic elements such as parameters are easily defined using the appropriate keyword and name. In AMPL we have the ability to create sets using the `set` keyword, which can be used, among others, for indexing. We also have a dedicated construct for graph edges. The `within` keyword defines the range of values that elements of this particular set can take. Next, in braces we specify `NODES cross NODES`, which means that each item in the set `ARCS` is described by two values, each of which falls in the set `NODES`.

Parameters are defined using the keyword `param`. We can also impose that they can take text values from a specific set using `symbolic in`. Array parameters are defined by specifying the appropriate index expression of the form `{i in NODES }`, `{(i, j) in ARCS }` after the name in braces, or a set, as in the case of the parameter `COST`. Values of all parameters indexed or not (except for those that use the expression `symbolic`) can also be limited from below or above by inequality signs, as seen in the above code.

Both sets and parameters can be given values in the model file (which is usually inadvisable) or in a dedicated file which will be described in a moment.

We can declare variables for our model using the keyword `var`. The rules for indexing and limiting from the bottom/top are the same as for parameters.

The objective function is given according to the following format:

`minimize` | `maximize` any_name: *expression*

and similarly for constraints:

`subject to` any_name optionally_indices: *expression*

where we can also specify optionally indices while creating an indexed constraint. We build expressions for the objective function and constraints using simple notation close to natural, mathematical syntax. The `sum` function allows us to sum vector elements, matrices or various expressions. We provide indices for the sum in a similar way as described earlier. As we can see, in the constraint we can also easily use the expression `if-then--else` to define the values for the sum difference, depending on the current node in the index, so we don't have to build several separate constraints, and the whole expression is very similar to that of the mathematical model. It should be remembered that `"="` is an equality operator, and `":="` is absent in this model.

In order to load data into the model, the `data` command is used in conjunction with the name of the data file, which can be given in quotation marks or not. AMPL uses files with the .dat extension in which we define parameter and set values. For the above problem it looks like this:

```
set NODES := 1 2 3 4 5 6 7 8 9 10;

param source := 1;
param target := 6;

param: ARCS: COST :=
1, 8 37
1, 10 10
...;
```

Listing 2. Sample .dat file in AMPL
Listing 2. Przykładowy plik .dat w AMPLu

Finally, to solve the model, we need to provide the appropriate solver using the `option solver` command, simply giving its name. Then we solve the model with the `solve` command. After successful solution, using the `display` function, we can specify any model elements whose values we want to check, separated by commas. An interesting option is `omit_zero_rows`, which allows us to display vector, matrix, etc. variables while skipping lines with the value zero.

4.2. Pyomo

The implementation of this problem in Pyomo looks like this:

```
from pyomo.environ import *
from pyomo.opt import SolverFactory
m = AbstractModel()

m.nodes = Set()
m.arcs = Set(within=m.nodes*m.nodes)

m.source = Param(within=m.nodes)
m.target = Param(within=m.nodes)
m.cost = Param(m.arcs)

m.x = Var(m.arcs, within=NonNegativeReals)

def distance_rule(m):
  return sum(m.cost[i,j] * m.x[i,j] for (i,j) in m.arcs)
m.distance = Objective(rule=distance_rule, sense=minimize)

def cons1_rule(m, node):
  if node == m.source:
    rhs = 1
  elif node == m.target:
    rhs = -1
  else:
    rhs = 0
  return sum(m.x[a] for a in m.arcs if a[0] == node) - \
   sum(m.x[a] for a in m.arcs if a[1] == node) == rhs
m.constraint1 = Constraint(m.nodes, rule=cons1_rule)

opt = SolverFactory('cbc')
instance = m.create_instance("GraphData.dat")
results = opt.solve(instance)
instance.display()
```

Listing 3. Shortest path in a directed graph – implementation in Pyomo
Listing 3. Najkrótsza ścieżka w grafie skierowanym – implementacja w Pyomo

Pyomo, as mentioned earlier, is a set of libraries for the high-level programming language Python. There are some restrictions, rules and writing style requirements imposed by the native language. Commands and expressions do not end with any special semicolon character, their end is determined by the end of line in which they are located. However, if an expression is too long and we want to move it to the next line, we can use the "\" (backslash) character. In Python, indents (tabs) in the code are extremely important and required for the proper operation of each program. Therefore, when writing method definitions, conditional statements or loops, we must ensure appropriate spacing in accordance with the language specification. On the one hand, this ensures consistency and readability of the code, which is an important aspect, but on the other hand, it limits the programmer in some way.

To use Pyomo in our code, we need to import the appropriate libraries, what is shown at the beginning of the above code. We import all classes from a given package using the "star". Because objectivity is one of Python's paradigms, the model in Pyomo is an example of an object to which subsequent elements are closely related. We call it to life using the `AbstractModel()` method, which creates an empty model to which elements added later are not fully constructed, even giving them initial values (the construction is carried out in two stages).

The principle of adding variables, parameters and other elements is the same – using the model object, we enter the name we choose after the dot and assign the object with a specific element to it by calling the appropriate method and, if required, providing the appropriate parameters.

The `Set()` method allows us to create sets. We also have a dedicated construct for graph arcs. We add to the previously mentioned method the keyword (named) argument `within` with the value `m.nodes * m.nodes`, which means that every element of the set `arcs` will be described by two numbers, each of which must fit in the set `nodes` (similar to AMPL). We add parameters to the model using the `Param()` method. By specifying the `within` argument, we can (similarly to sets) request that the value of this element belong to the indicated set or to other parameter. We create an indexed parameter by simply specifying the file after which the element is to be indexed (as in the case of `cost`).

We create variables using the `Var()` method. Indexing rules are the same as for parameters. To specify the lower/upper limit, we can also use one of several identifiers of predefined constants for the `within` argument. In the example above, `NonNegativeReals` simply means greater than or equal to zero. We use `Objective()` and `Constraint()` for the objective function and constraints. In both cases, there is the `rule` argument that takes the name of a specific, previously defined method. We create the one using the keyword `def`, the name and a list of arguments given after the comma in brackets (`distance_rule` and `constraint1_rule`). In the objective function, we additionally specify `minimize` or `maximize` in the `sense` argument if we want to search for the minimum or maximum of a function, respectively.

The first argument of a `_rule` type method is always the model object (which we created at the beginning with the `AbstractModel()` method). The next, optional, argument is an indexed component (e.g., set), or more precisely, a specific element of this set. In the body of the method we define, we must use the keyword `return`, and then provide a mathematical expression that defines a constraint or an objective function.

Having all components of a model we may solve it. To do this, we "build" it by calling the `create_instance` method, in which we give the name of the data file as an argument. Then we get a new object (hereinafter referred to as the instance). Pyomo uses, among others, .dat files that have the same, AMPL equivalent, syntax (Listing 2), which allows us to use the same file in both environments. However, there is a slight difference. We cannot separate (e.g., for reading convenience) elements of sets and parameters with commas. This file will not be loaded by Pyomo. The next step is to create an object that is a "shell" for communication with a solver using the `SolverFactory()` method, in which we give the name of the solver as an argument. All that remains is to call the `solve` method on the `opt` object, giving the instance we created earlier. This method returns only basic data regarding the problem solved by the solver (among others, the number of variables, constraints and status). To view the values of the objective function, constraints, etc., we need to call the `display()` method on the instance object.

4.3. JuMP

The implementation of this problem in JuMP looks like this:

```
using JuMP
using Cbc

immutable arc
  startNode::Int32
  endNode::Int32
end

data = readdlm("GraphData.txt", ' ', Int32)
numOfEdges = size(data)[1] - 1
N = data[1,3]

nodes = [i for i in 1:N]
arcs = [arc(data[i+1,1], data[i+1,2]) for i in 1:numOfEdges]
cost = Dict(arcs[i] => data[i+1,3] for i in 1:numOfEdges)

source = data[1,1]
target = data[1,2]

m = Model(solver = CbcSolver())
@variable(m, x[arcs] >= 0)

@objective(m, Min, sum(cost[k] * x[k] for k in arcs) )

for i in nodes
  if i == source
    rhs = 1
  elseif i == target
    rhs = -1
  else
    rhs = 0
  end
  @constraint(m, (sum(x[a] for a in arcs if a.startNode == i) -
    sum(x[a] for a in arcs if a.endNode == i) ) == rhs )
end

status = solve(m)
print(status)
print("Objective value: $(getobjectivevalue(m)) \n")
print("Arcs: $(getvalue(x)) \n")
```

Listing 4. The shortest path in a directed graph – implementation in JuMP
Listing 4. Najkrótsza ścieżka w grafie skierowanym - implementacja w JuMP

JuMP, as mentioned at the beginning of the paper, is a set of libraries for the high-level programming language Julia. As in the case of Pyomo, there are some restrictions, rules and writing style requirements imposed by the native language. Commands and expressions do not end with any special semicolon character, their end is designated by the end of the line in which they appear. If any expression is too long and we want to partially move it to the next line, we just do it, it does not require adding specific characters or keywords. However, we can use the "\" character. The recommended way is to enclose such an expression in parentheses. Unlike Python, the indentation in the code we write does not affect its operation and only serves for our convenience and readability.

To use JuMP in our program, we must import the library containing it using the keyword `using`. The same applies to the package containing the solver interface that we want to use. One of Julia's paradigms is objectivity, hence the model in JuMP is presented as an object, which we then supplement with further elements that are closely related to it (not all, however, about which in a moment). We create our main object using the `Model()` method. As in Python, there are so-called keyword arguments. What is characteristic for JuMP, when creating a model, we must specify the method initiating the Julia interface for a given solver in the `solver` argument.

Unlike the other modeling languages described here, JuMP has no constructs for sets or parameters. For this we need to use the standard constructs offered by Julia (which are, fortunately, simple and comfortable), which somewhat spoils the idea of the model as an object keeping in itself all the components belonging to it. Therefore, if we would like to use some constructs for indexing, we must immediately initialize them with appropriate values.

JuMP does not offer methods for loading data from files – here we must again use the functions offered by the native language. The simplest solution was the `readdlm()` method, in which we give as arguments the file name, character (or string) separating subsequent numbers (or words) and the third optional parameter – the type of data contained in the file. Entering the last parameter (`Int32` – 32-bit integer) was necessary, because by default the loaded numbers were saved in the program memory as floating point numbers (while there should be integers here). As in Python, we declare variables/objects by providing a name. The downloaded data was saved in the `data` variable. The `size(data)` method returns a two-element vector with, in order, the number of rows and the number of columns. Specifying index 1 in square brackets, we extract the number of rows (indexing of tables in Julia begins with one) and reducing by 1 we get the number of edges (`numOfEdges`). We read the number of nodes N from the loaded matrix. The set (array) `nodes` is created by a simple expression with a `for` loop, giving `i for i in 1: N` in square brackets, which generates subsequent values of `i` in the range `1: N`.

There are no graph constructs in JuMP, either. However, we can easily deal with this by creating the arc structure (the keyword for the construct is `immutable` or `type`, but the first of them is recommended, due to the speed of filling the arrays). Inside, we only have the start node `startNode` and the end node `endNode`. Structure, function, conditional statement, etc. definitions must end in Julia with the keyword `end`. Next, we create a set of edges `arcs` by reading from the data object pairs of nodes from subsequent rows of data. The vector `cost` is created as a dictionary using the method `Dict()`, in which we give the label (here a single edge) as an argument, operator =>, the value and finally an analogous expression with a loop `for` as before.

Variables can be created by the `@variable()` method. The first argument is always the model object, while the second argument is the variable expression. Specifying `x[arcs]` creates a variable indexed with the elements of the vector `arcs`. In addition, we can specify the lower limit (as in the above code), upper limit (in the same way) or both. In the last case it would be e.g., `0 <= x[arcs] <= 1`. For this the `@objective()` method is responsible for the objective function, in which we enter the model object, `Min` or `Max` and a mathematical expression. In the function `sum()` we give what we add (`m.x[a]`), after which indices (`for a in m.arcs`) and optional condition for the index using the keyword `if` (similarly to Pyomo).

We create constraints using the `@constraint()` method, where we give the model object and a mathematical expression (analogous rules as before for indexing, sums, etc.).

Finally, there remains the solution to the problem by calling the `solve()` method, to which we pass the object with the model. After finishing, only a short status is returned, e.g., "Optimal", if an optimal solution has been found. The values of the objective function, decision variables, etc. are saved in the model object. Using `print()`, we can print specific items in the console. To get the value of the objective function from the model, we use `getobjectivevalue()` (entered in the model argument), and for variables, `getvalue()` (here we enter only the variable name). In defining string objects in Julia, we can use the expression `$()` to "inject" some numerical value from the indicated object, or even the result of a mathematical operation.

5. Tests

We used the GTGraph program to generate test graphs [6, 1]. According to the description of the mathematical model, the generated graph is a set of edges, where each edge is described by three consecutive values: the starting node, the end node and the cost of the transition (distance). Values for all three edge components are positive integers. The graph used to test the implemented models is below. The graph in graphic form is in the Fig. 1.

1 8 37	4 2 21	7 3 29
1 10 10	4 10 36	8 7 33
2 4 17	5 3 27	8 4 31
2 10 12	5 2 40	9 6 10
3 1 15	5 8 40	9 5 36
3 10 37	6 2 37	9 3 22
3 2 33	6 4 25	9 2 35
3 6 36	6 5 13	9 7 18
3 9 31	7 9 10	10 8 13
4 5 10	7 6 11	10 4 14

For all implementations of the models discussed in this section, the results were the same. They are presented in the Table 1. There are many more (90 exactly) possible pairs of nodes between which paths can be drawn, but among them there will be many which have the lengths of one or two edges.

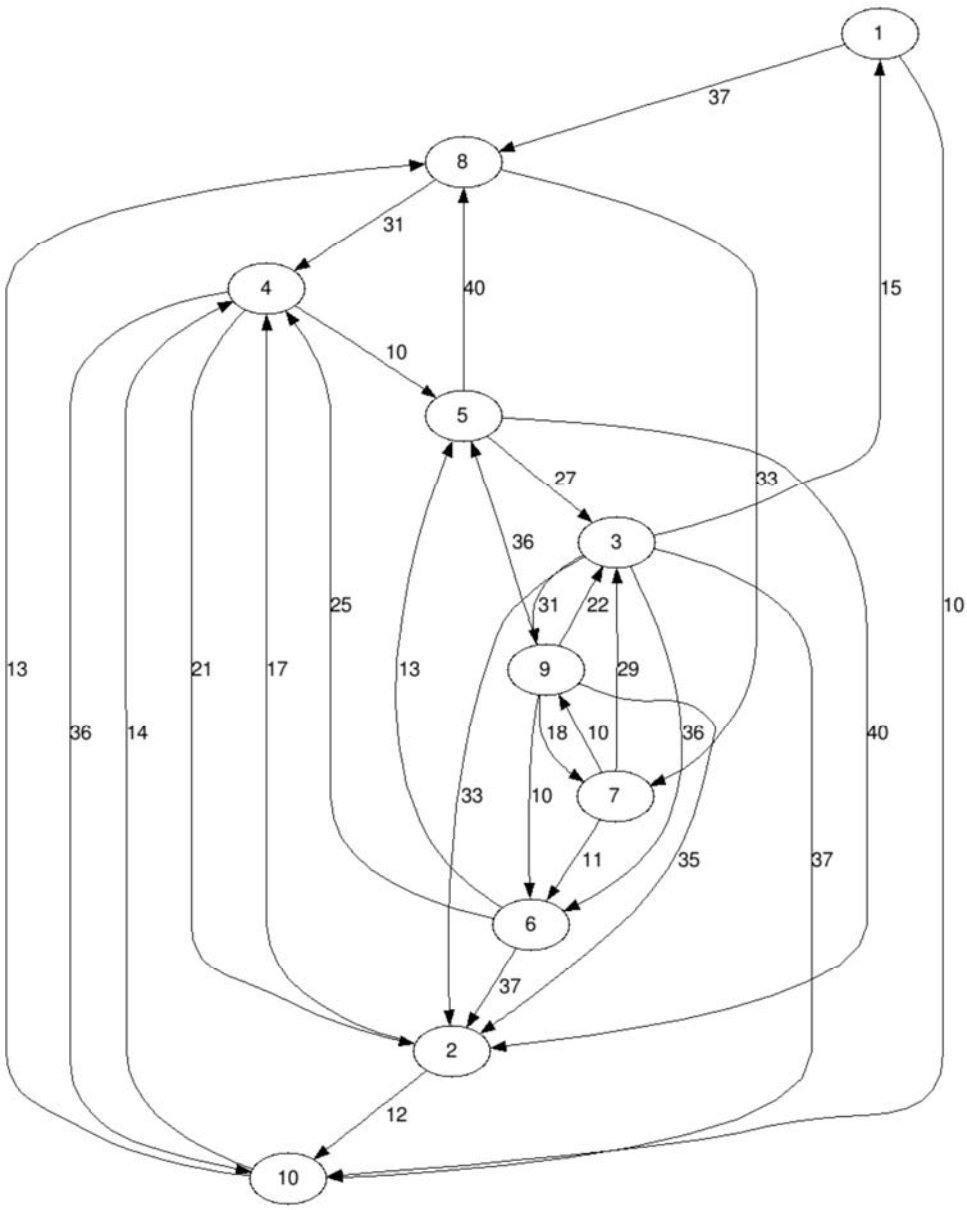

Fig. 1. Test graph
Rys. 1. Graf testowy

Table 1. Results for the shortest path problem in the directed graph
Tabela 1. Wyniki dla zadania najkrótszej ścieżki w grafie skierowanym

Start node, Terminal node	The optimal path	Path length
(1, 6)	1, 10 ,8, 7, 6	67
(4, 8)	4, 2, 10, 8	46
(2, 9)	2, 10, 8, 7, 9	68
(10, 9)	10, 8, 7, 9	56
(8, 10)	8, 4, 2, 10	64
(10, 1)	10, 4, 5, 3, 1	66

6. Conclusions

The AMPL, Pyomo and JuMP environments presented in this paper proved to be effective tools for optimization modeling. We must remember that Pyomo is embedded in Python and JuMP in Julia and there are some limitations and code style requirements to write codes compatible with the native languages. AMPL, as a language dedicated to optimization modeling, does not have such a problem, thanks to which the constructs in AMPL are simpler, which translates into easier and more convenient implementation of models. It also provides the shortest code of all the environments discussed here. In terms of the number of characters and lines we need to write, the implementation in JuMP is the worst. Pyomo is between the two. Despite the lack of constructs for parameters and sets in JuMP and Pyomo one can successfully use the standard functionalities offered by Julia and Python languages. The creation of mathematical expressions was comfortable and quite intuitive in all the environments presented. JuMP and, more specifically, Julia, offer built-in operations on vectors and arrays (including even vector methods), which are missing in AMPL, and in Python are offered by external libraries.

People who do not have everyday contact with object-oriented programming languages or programming at all will certainly find their way in the field of optimization modeling using AMPL. In contrast, users familiar with object-oriented programming, should not have big problems using Pyomo and JuMP.

Based on this work, we can conclude that AMPL, although it is a comfortable environment with many possibilities and wins in many places, is not the best in all aspects.

References

1. GTgraph Program. www.cse.psu.edu/~kxm85/software/GTgraph.
2. JuMP documentation. www.juliaopt.org/JuMP.jl/0.18/index.html.
3. JuMPa GitHub Repository. https://github.com/JuliaOpt/JuMP.jl.
4. Pyomo documentation. https://pyomo.readthedocs.io/en/latest/.
5. Pyomo GitHub Repository. https://github.com/Pyomo/pyomo.
6. Bader D.A., Madduri K., *GTgraph: A Synthetic Graph Generator Suite*. College of Computing, Georgia Institute of Technology, 2006.
7. Bertsekas D.P., *Linear Network Optimization*. MIT Press, 1991.
8. Fourer R., Gay D.M., Kernighan B.W., *AMPL A Modeling Language for Mathematical Programming*, Second Edition. Duxbury, Thomson, 2003.
9. River Logic, Inc., *Optimization Modeling: Everything You Need to Know*, www.riverlogic.com/blog/optimization-modeling-everything-you-need-to-know, 2021.
10. Olszak A., Karbowski A., *Parampl: A Simple Tool for Parallel and Distributed Execution of AMPL Programs*. "IEEE Access", Vol. 6, 2018, 49282–49291, DOI: 10.1109/ACCESS.2018.2868222.

Studium porównawcze języków modelowania optymalizacyjnego AMPL, Pyomo i JuMP na przykładzie liniowego zadania programowania sieciowego

Streszczenie: Celem pracy jest zbadanie i porównanie możliwości trzech języków (środowisk) modelowania optymalizacyjnego: AMPL, Pyomo i JuMP. Porównanie zostanie oparte na trzech implementacjach zadania najkrótszej ścieżki sformułowanego jako zadanie programowania liniowego. Przedstawione i omówione zostaną kody poszczególnych modeli oraz różnice między nimi. Pod uwagę będą brane różne aspekty, takie jak: prostota i intuicyjność implementacji, dostępność określonych struktur danych dla problemów z siecią LP itp.

Słowa kluczowe: optymalizacja, języki modelowania, zadanie najkrótszej ścieżki, zadania sieciowe, programowanie liniowe, zadania grafowe

Andrzej Karbowski, PhD, DSc
A.Karbowski@ia.pw.edu.pl
ORCID: 0000-0002-8162-1575

Andrzej Karbowski received PhD (1990) and habilitation (2012) in Automatic Control and Robotics from Warsaw University of Technology, Faculty of Electronics and Information Technology. Currently he is an assistant professor at the Institute of Control and Computation Engineering of Warsaw University of Technology and at NASK National Research Institute in Warsaw.
He is the editor and the co-author of two books (on parallel and distributed computing), the author and the co-author of two e-books (on grid computing and optimal control synthesis), the editor of a special issue of the „Energies" journal („Mixed-Integer Linear and Nonlinear Programming Methods for Energy Aware Traffic Control in Stationary Networks and Clouds") and the author over 150 journal and conference papers. His research interests concentrate on optimal control, MIP and MINLP methods, energy-aware data networks management, cybersecurity, decomposition and parallel implementation of optimization algorithms on multicore computers, clusters and clouds.

Krzysztof Wyskiel, BSc
k.wyskiel@protonmail.com
ORCID: 0000-0001-6851-9755

Krzysztof Wyskiel is an MSc student of Computer Science at the Faculty of Electronics and Information Technology of the Warsaw University of Technology.