IBIMA Publishing
Communications of International Proceedings
https://ibimapublishing.com/p-articles/44SE/2024/4440824/
Vol. 2024 (24), Article ID 4440824

Educational Tool for ILP and BIP Problems*

Gabriela PAWŁOWSKA and Joanna WIŚNIEWSKA

Institute of Information Systems, Faculty of Cybernetics

Military University of Technology, Gen. S. Kaliskiego 2, 00-908 Warsaw, Poland

Correspondence should be addressed to: Gabriela PAWŁOWSKA, Gabriela.Pawlowska@student.wat.edu.pl

* Presented at the 44th IBIMA International Conference, 27-28 November 2024 Granada, Spain

Abstract

We focus here on the program which is a support in learning of solving Integer Linear Programming (ILP) and Binary Integer Programming (BIP) problems. There are many programs solving optimization problems using the simplex algorithm and they also offer the option of solving tasks with the result as integer numbers but our program is specialized for such problems and shows the subsequent stages of solving the tasks with the use of the branch and bound method. In this article, the mentioned method is described. Then, we shortly characterize the environment in which the program was implemented and deliver some details about the implementation. Finally, there is a description and examples of program's functionalities. This tool can generate arbitrary problem but also the user may introduce own tasks. Program uses standard Python libraries to solve the problem according to the branch and bound method – because this approach generates subproblems, as long as it will not find the integer result, the program gathers all intermediary solutions to be accessible for the user. At the end of program's operating, the problem, its solution, and subsequent stages are saved to a file.

Keywords: optimization, the branch and bound method, ILP problems, BIP problems

Introduction

Problems of optimization affect many areas of science. We can find them in economy, medicine, computer science, and also as an element in operational research which are significant in military operations. In Military University of Technology, the operational research and optimization theory are very important subjects, especially in the Faculty of Cybernetics where computer science and cryptology are lectured.

The emphasis is put on students to learn how to solve linear programming problems (Sierksma (1996), Kochenderfer and Wheeler (2019)) with the use of simplex algorithm, but other issues, like non-linear programming and multi-criteria optimization, are also important. In the case of linear programming, we also want sometimes to solve problems that should obtain integer values as a result – it is important in tasks like selection of technological process (Malea and Nitu (2020)) or resource allocation (Menshikh et al (2018)). We can observe that some problems require solution as non-negative integers what we call Integer Linear Programming (ILP) or, in the special case, binary values may be expected what is determined as Binary Integer Programming (BIP).

In this article, we want to present a program, written in the Python programming language, which solves ILP and BIP tasks with the use of Branch and Bound Method (Land and Doig (1960), Clausen (1999)). The program shows

Cite this Article as: Gabriela PAWŁOWSKA and Joanna WIŚNIEWSKA, Vol. 2024 (24) "Educational Tool for ILP and BIP Problems" Communications of International Proceedings, Vol. 2024 (24), Article ID 4440824, https://doi.org/10.5171/2024.4440824

the subsequent steps of the whole process of solving the problem, what is especially important for students learning the mentioned method.

Branch and Bound Method

The Branch and Bound method is widely used in solving optimization problems and is considered one of the more advanced tools in optimization theory. Due to its flexibility and efficiency, it has found applications in many fields, from graph theory to operations research. Below, the key theoretical aspects of this method are presented to highlight its essence and applications.

The first time the Branch and Bound method was proposed was in 1960 by A. H. Land and A. G. Doig in a paper titled "An Automatic Method of Solving Discrete Programming Problems" (see: Land and Doig (1960)). To better define the significance of the Branch and Bound method, we will present several definitions whose terms will be used in describing our topic.

The first term that needs to be discussed is the concept of an optimization problem. This term refers to a problem whose goal is to maximize or minimize a specific objective function while considering given constraints. Optimization problems aim to determine the optimal solution that meets all the conditions of the problem, thereby leading to the best possible outcome according to the chosen criterion.

In reference to the previous paragraph, we will now discuss the definition of optimization models. This model can be represented by equations, inequalities, or other mathematical relationships. It is defined as an abstract representation of a given problem, allowing the formulation of various variables, parameters, objective functions, or constraints. The goal of such a model is to find the optimal solution for the objective function, meaning its maximum or minimum value, while taking the assumed constraints into account.

Breaking down the above definitions, we can ask what the previously mentioned terms — objective functions and constraints, related to the definition of a mathematical model — are. Starting with the objective function, also called the criterion, it is an expression that defines the main goal of solving the optimization problem, describing what solution is the best concerning the problem. Constraints applied in optimization tasks, on the other hand, refer to narrowing down the set of solutions, i.e. they are setting a condition that must be met for the mathematical model and define the range of permissible results for the task.

Another important concept in this discussion is the definition of ILP and BIP problems. The previous paragraphs refer to linear integer programming problems, where the results of such tasks are real numbers. In the case of ILP tasks, values of solution will only take integer values. Moreover, for the PLB problems, the results will only take two values: 0 or 1. In all cases, the goal is to arrive at the optimal solution.

The main aim of the method is to visualize all possible paths in solving the optimization problem using a state-space tree. Such representation allows the solver to identify potential routes that may lead to a solution. The algorithm begins at the first node, calculating the solution of the mathematical model for that node, then makes assumptions regarding individual variables, splitting them into two cases, and continues this process iteratively. This method also includes categorizing nodes as promising or less promising, where the latter are less likely to be part of the solution.

Evaluating some nodes allows for easy categorization, enabling decisions on whether to continue the analysis or move on to the next unexplored nodes. Without the Branch and Bound method in state-space trees, this process would be very time-consuming due to the exponential growth of the problem size. This algorithm significantly speeds up the analysis by optimizing the search process.

Finally, we would like to present the key steps that need to be followed to correctly carry out the process of solving an optimization problem using the Branch and Bound method:

- Solve the basic problem – this involves formulating and solving the mathematical model that represents the problem.
- Divide the original problem – break down the original problem into smaller subproblems of the same type, based on the non-integer variables obtained in the previous solution, and create a decision tree.

- Solve smaller subproblems – solve each of the smaller subproblems in the decision tree by adding new constraints imposed by the decision tree to the basic mathematical model.
- Continue the process – continue this process (returning to the second step) until satisfactory results are obtained.
- Aggregate solutions – finally, combine the solutions of the smaller subproblems to derive the final solution to the original problem, completing the optimization process and enabling decision-making based on the comprehensive analysis.

For the better understanding of the proposed program's results, let us assume that the initial problem to solve is marked as S_0. Of course, the problem S_0 does not have the solution in the form of integer or binary values (which are also integer numbers). Dividing this problem into subproblems relies on choosing a variable x_i ($i = 1,\ldots,n$ and n is the number of variables in a current task) with non-integer value v and making subproblems S_1 and S_2 such that for S_1 $x_i <= floor(v)$ and for S_2 $x_i >= ceil(v)$. If even one variable in subproblem S_1 is not integer, this solution has to be divided into subproblems S_3 and S_4 by this variable. The algorithm ends generation of subproblems for a current checkpoint if this problem has a solution with all integer values or does not have a solution at all.

It should be emphasized that a main advantage of the Branch and Bound method is its flexibility because it can be adapted to different types of optimization problems. Additionally, it systematically explores all potential solutions, ensuring optimality if the method runs to completion. As a disadvantage of this method, we could point out that the search tree can grow exponentially, making it slow for very large or complex problems.

Utilized Technology and Details of Implementation

To implement the program realizing the Branch and Bound method, the Python language (see: Python Software Foundation (2001-2024)) was used, which is a high-level programming language with readable and clear syntax. Python is currently one of the most popular programming languages; it has a very extensive range of available libraries, from which many interesting functions can be imported into your program. Python works on all major operating systems (Windows, macOS, Linux), which means code can be easily transferred between systems. The vast ecosystem of external libraries, which allows working in various areas such as data analysis, artificial intelligence, web development, or solving optimization problems, significantly facilitated the work on the written program. The libraries used are: Pulp by Roy et al (2024), Tabulate by Astanin (2022), PyQt5 (see: PyQt5 (2022)).

Pulp is a library designed for modelling and solving optimization problems. It contains methods for solving issues such as linear, integer, and continuous programming. The presented program utilizes the 'pulp.LpProblem' which enables the definition of mathematical models for optimization tasks. The 'pulp.LpVariable' function was also used, serving to create decision variables, both continuous (Continuous) and binary (Binary). The 'solve' function was implemented to cope with the defined problems, finding the optimal solution according to specified criteria.

Additionally, the program utilized the 'pulp.LpSolverDefault.msg' attribute, which controls the display of solver messages. By default, it is set to True, but in the program, it was set to False to avoid disruptions when displaying results. Another function used is 'pulp.value', which is used to retrieve the values of decision variables and the objective function after solving the optimization problem.

Tabulate is a Python library that simplifies the formatting and presentation of data in a table form. It supports the creation of clear tables based on structures such as lists or dictionaries. This library was used to neatly display results like decision variable values and objective function values in tables using the 'tabulate' function. Tabulate library makes the presentation of obtained the results readable what is specially needed when the steps of the solved subproblems have to be presented.

PyQt5 is a library for creating graphical user interfaces (GUI) in Python. The 'QApplication' class is responsible for managing the application lifecycle and handling events. The base class for all interface elements is 'QWidget', from which components like windows, buttons, and text fields inherit. For example, 'QLabel' is used to display text or images, while 'QLineEdit' allows for the input of single lines of text. Buttons are represented by the 'QPushButton' class, enabling actions to be triggered upon clicking. 'QRadioButton' allows selecting one option from several, and 'QComboBox' provides a dropdown menu for selection. Layouts, such as 'QVBoxLayout' (vertical layout) and 'QHBoxLayout' (horizontal layout), are used for arranging interface elements. 'QMessageBox' is used for displaying messages, while 'QDialog' handles dialog windows requiring user interaction. Additionally, the 'QtCore' module provides definitions and constants for managing GUI element properties.

In the program, we do not implement the simplex algorithm but utilize Python libraries mentioned above. For example, the code solving the tasks with the restriction for problems with no solution is:

```
def solve_continuous(self, initial_problem, var_dict, num_variables, output_file_path, max_iterations=1000):
    objective_values = []
    problems_to_solve = [initial_problem]
    best_integer_solution = None
    all_paths = []
    optimal_solutions = []
    path_counter = 0
    iteration_count = 0
    seen_fractions = {}
    stop_flag = False

    while problems_to_solve and not stop_flag:
        if iteration_count >= max_iterations:
            QMessageBox.information(None, "Unsolvable Problem", "Maximum iterations reached. The problem
                                                                may be unsolvable or is leading to an infinite loop.")
            stop_flag = True
            break

        current_problem = problems_to_solve.pop(0)
        try:
            current_problem.solve()
        except Exception as e:
            QMessageBox.critical(None, "Solver Error", f"An error occurred while solving the problem: {e}")
            stop_flag = True
            break

        objective_value = pulp.value(current_problem.objective)
        if objective_value.is_integer():
            solution_values = [var.varValue for var in current_problem.variables()] + [objective_value]
            objective_values.append(solution_values)
            optimal_solutions.append((solution_values, objective_value))
            if best_integer_solution is None or (
                    (current_problem.sense == pulp.LpMinimize and objective_value < best_integer_solution) or (
                    current_problem.sense == pulp.LpMaximize and objective_value > best_integer_solution)):
                best_integer_solution = objective_value
                path = [var.name for var in current_problem.variables() if var.name in var_dict]
                all_paths.append((path_counter, path))
                path_counter += 1
                self.print_decision_path([], solution_values, num_variables)

        if (best_integer_solution is None) or (
                (current_problem.sense == pulp.LpMinimize and objective_value < best_integer_solution) or (
                current_problem.sense == pulp.LpMaximize and objective_value > best_integer_solution)):
            path = []
            for var in current_problem.variables():
                if var.name in var_dict:
                    var_value = var.varValue
                    if var_value != int(var_value):
                        fraction_key = (var.name, round(var_value, 5))
                        if fraction_key in seen_fractions:
                            QMessageBox.information(None, "Unsolvable Problem",
                                                    "The problem appears to be unsolvable due to repeating fractional values.")
                            stop_flag = True
                            break
                        seen_fractions[fraction_key] = True
                        print(f"Fractional value found for {var.name}: {var_value}")
                        self.generate_continuous_decision_tree(current_problem, var.name, var_value, path,
                                                               path_counter)
                        subproblem1 = current_problem.copy()
                        subproblem1 += var <= int(var_value)
                        subproblem2 = current_problem.copy()
                        subproblem2 += var >= int(var_value) + 1
                        problems_to_solve.extend([subproblem1, subproblem2])

            if path:
                all_paths.append((path_counter, path))
                path_counter += 1
                self.print_decision_path(path,
                                         [var.varValue for var in current_problem.variables()] + [objective_value],
                                         num_variables)

        iteration_count += 1
```

The main goal was to capture and pass the sequence of solved subproblems to the user. We can see the paths' creation which contain the sequence of subproblems and task's results in the last lines of the above code (starting from the command 'all_paths.append'). Then, the path may be presented to the user by the code below:

```
def print_paths(paths, num_variables, file_path):
        detailed_results = "All decision paths:\n"
        with open(file_path, 'w') as file:
            file.write("All decision paths:\n")
            for path_num, path in paths:
                if len(path) != num_variables:
                    line = f"Decision path {path_num}: {path}"
                    file.write(line + '\n')
                    detailed_results += line + '\n'
        self.detailed_results_description.append(detailed_results)

    if not stop_flag:
        print_paths(all_paths, num_variables, output_file_path)

    return objective_values
```

Interface and Functionalities

The program is designed to solve and illustrate the process of solving ILP and BIP problems. When it is started, we can see the view as in Fig. 1.

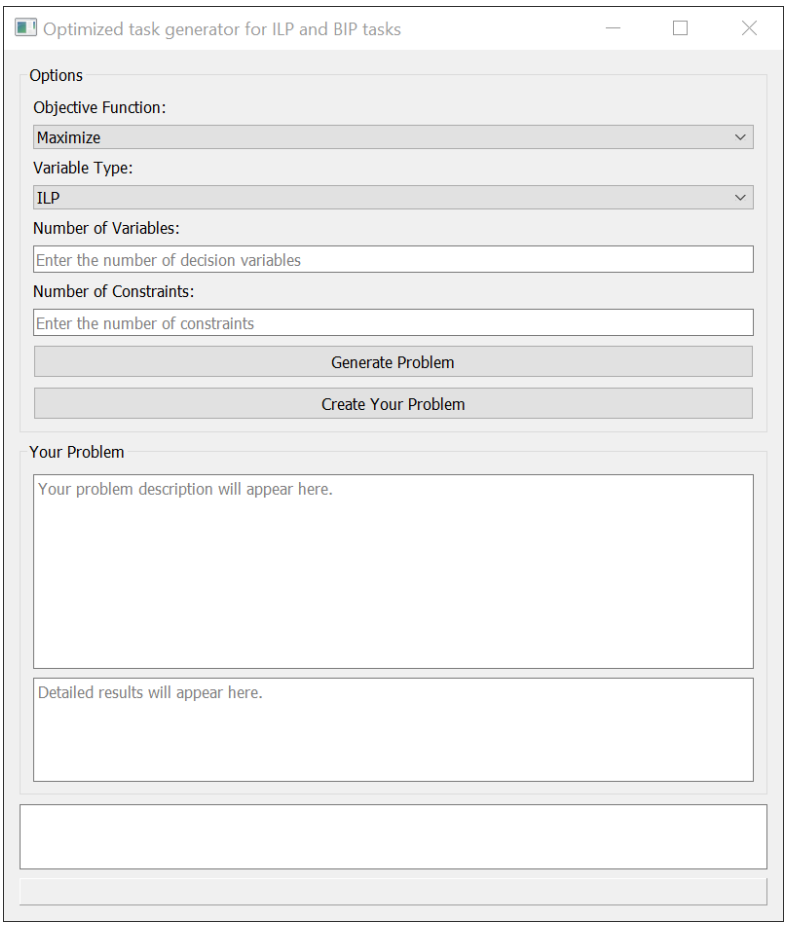

Fig. 1. Initial form ready to enter problem's data

Now, we can describe our problem, which needs to be solved. At first, we have to choose the type of a criterion – it may be maximalization or minimalization. The second step is differentiating between data type, we must point out if problem's variables are integer or binary numbers. Of course, the numbers of variables and constraints also have to be entered in the form. Then, we may decide if the program should generate the task (button "Generate Problem") of defined dimensions or we want to enter our problem (button "Create Your Problem").

Let us to test the program for ILP maximization problem with four variables and three constraints. Moreover, we ask the program to generate the task itself. The result is presented in Fig. 2.

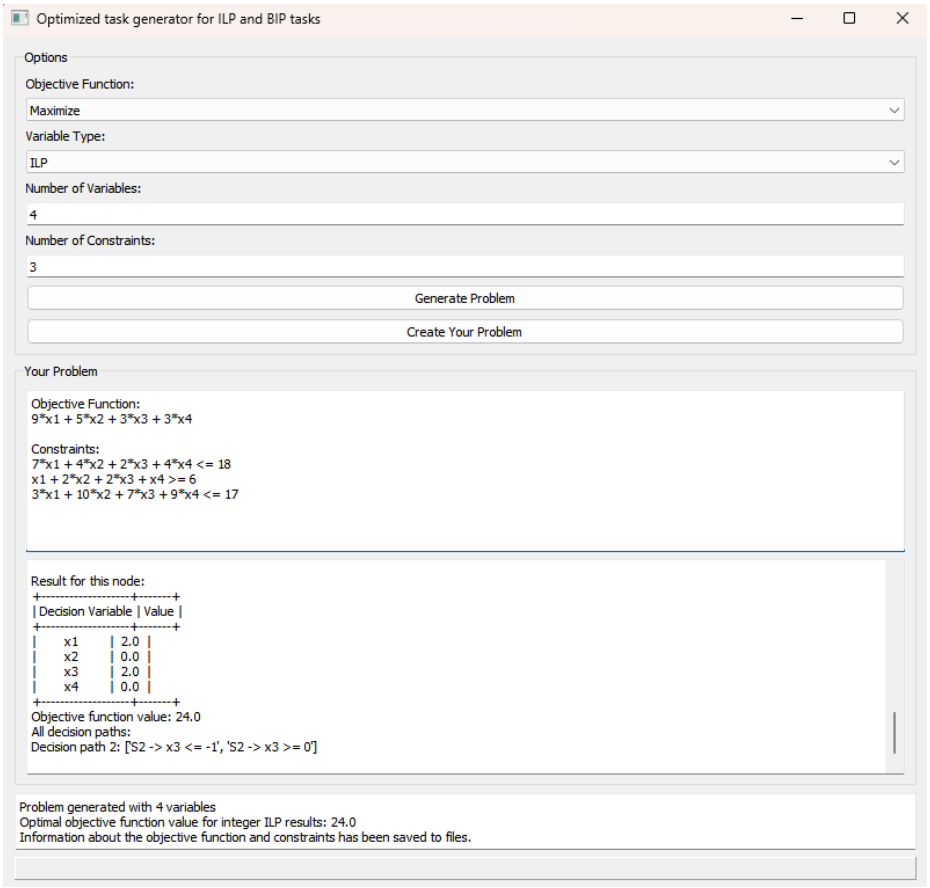

Fig. 2. Generated task and its solution

The three windows on the form serve to display: the generated problem, final solution, and some additional information. The program generates also two text files. In problem_continuous_info.txt we can find the generated mathematical model, and continuous_decision_paths.txt contains an information about the paths of subproblems solved on the way to the solution with integer values (see: Fig. 3).

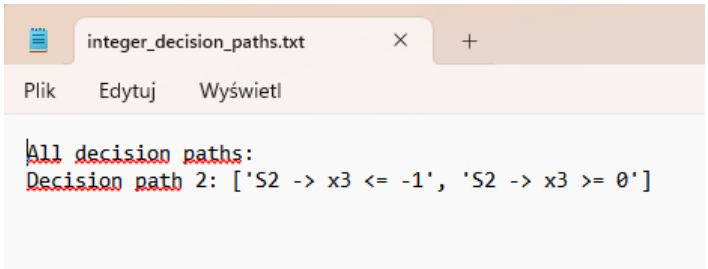

Fig. 3. Decision paths in the file

Now, we can present the second example. Let the problem be the minimization with 4 binary variables and 3 constrains, and we use the option of creating our own problem. After clicking the button “Create Your Problem”, the program opens the form to pass the coefficients for criterion and constrains, what is depict at Fig. 4. We can also set the types of constrains (<= or >=).

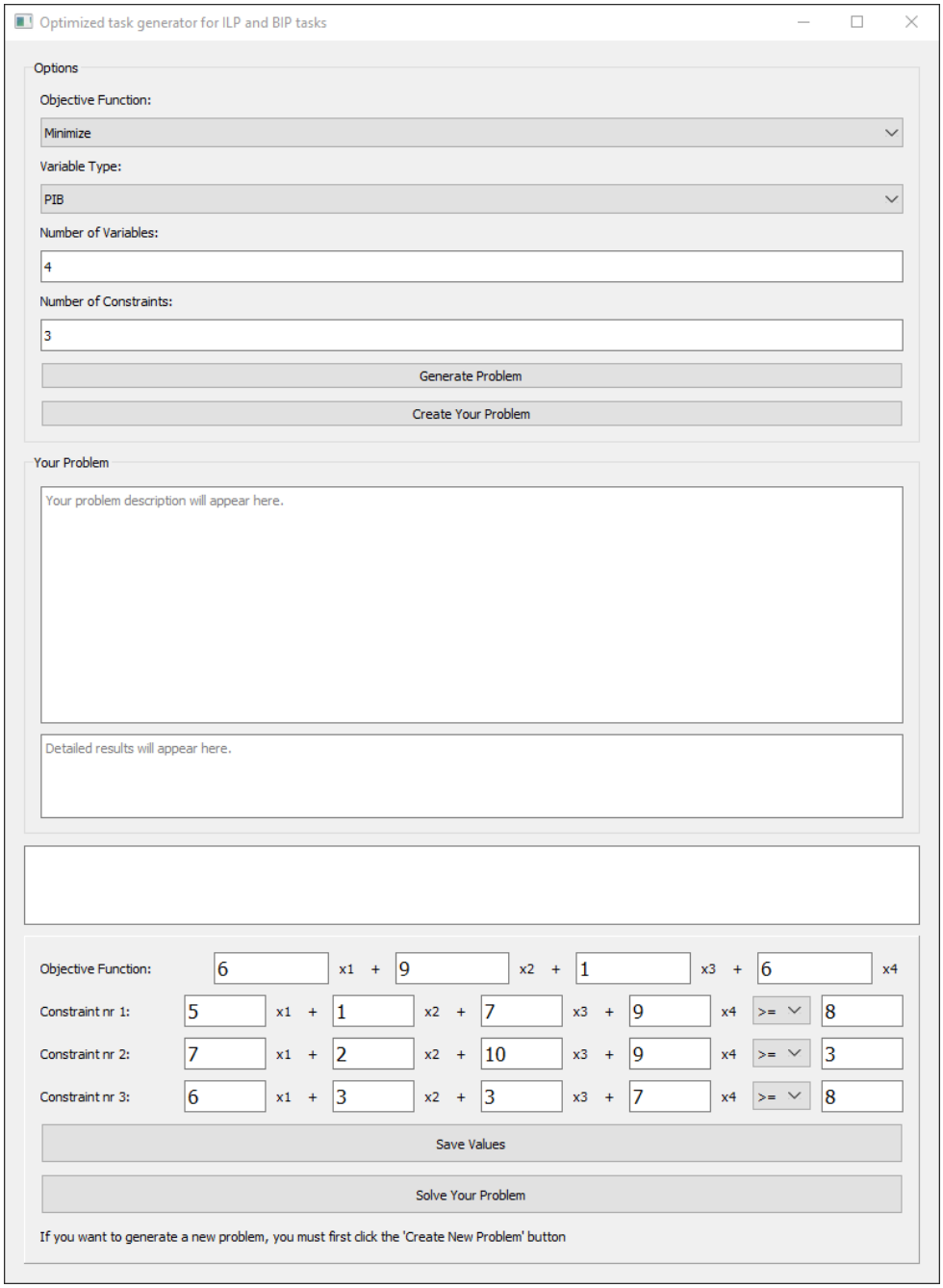

Fig. 4. Generation of own binary problem

Clicking the button “Save Values” creates the file problem_binary_info.txt with our problem – see Fig. 5.

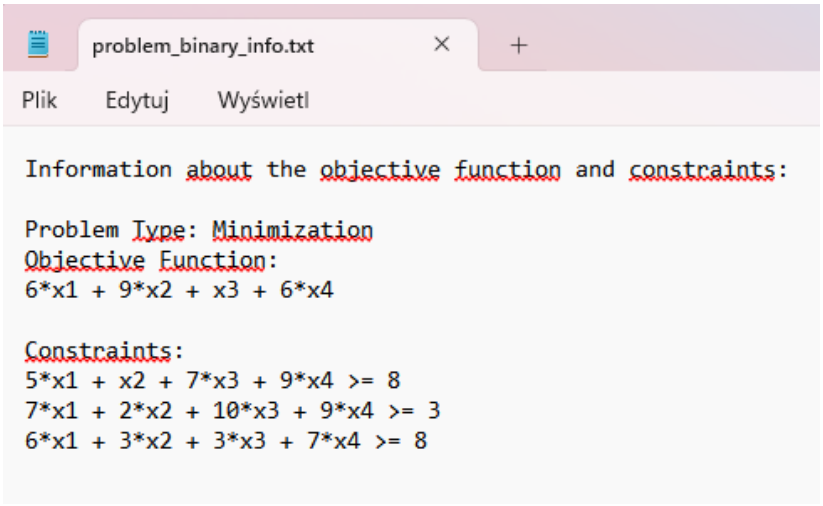

Fig. 5. The binary problem saved into the file

We solve the task by clicking the button "Solve Your Problem". The program returns: "Optimal solution for binary results: 7". The list of computed paths is generated to the file integer_decision_path_user.txt. We can the paths and the problem's solution in Fig. 6.

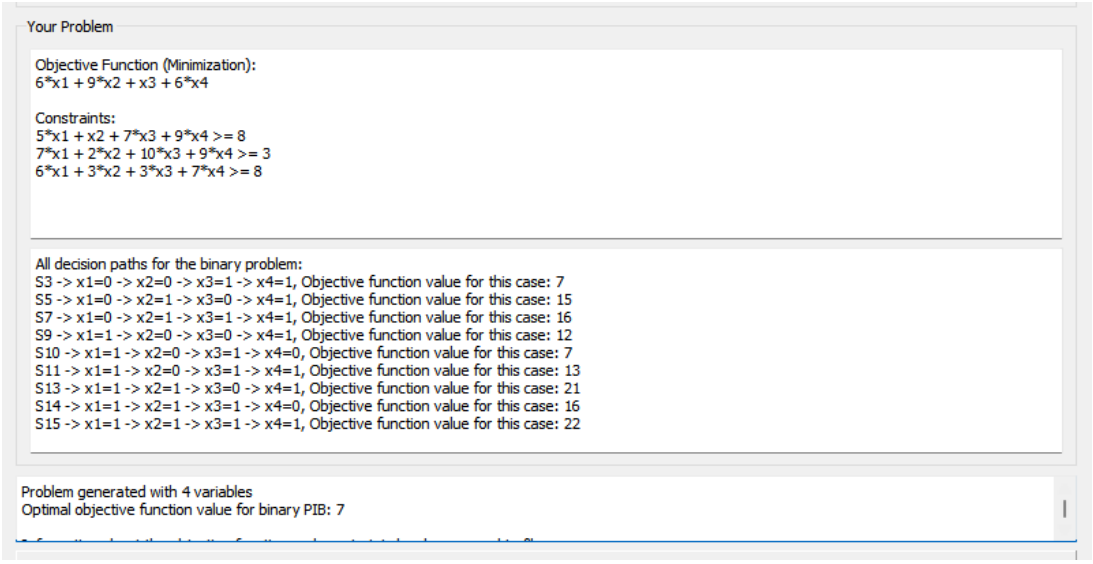

Fig. 6. A part of the program's window with the paths and the solution for the binary problem

Of course, the user (or computer) can generate the task which has no solutions. The program copes with this problem by applying some elements. Firstly, the iteration limit (max_iterations) prevents infinite computations. If this limit is reached, the program stops and informs the user of a possible lack of solution. Secondly, in the case of an error during solving, for instance due to mathematical inconsistencies, the program displays an error message and terminates. Thirdly, the program detects repeating fractional values by tracking variable values. If these values repeat, which suggests the problem is unsolvable, the program halts computations and informs the user. These mechanisms allow the program to safely terminate when there is no solution, avoiding infinite loops and providing helpful messages.

Conclusions

The presented program was firstly prepared as a part of the engineering thesis of Gabriela Pawłowska (Pawłowska (2024)). This first version suffered from minor errors and uncomfortable way to enter user's tasks to the tool. A corrected program was presented at the 21st International Conference of Students and Young Scientists in Poland, September 2024. The program's version described in this article is an improved proposition (compared to both previous versions). The tool has a new interface thanks to the PyQt5 library. The generation of parameters for the

problems is extended by the mechanism avoiding situations when the problems have no solutions. Some other details, like improvement of the information saved in the files, were also added.

References

- Astanin, S. (2022) 'Pretty-print tabular data', [Online], [Retrieved October 23, 2024], https://pypi.org/project/tabulate/
- Clausen, J. (1999) 'Branch and Bound Algorithms - Principles and Examples', [Online], [Retrieved October 23, 2024], https://www.semanticscholar.org/paper/Branch-and-Bound-Algorithms-Principles-and-Examples-Clausen/fffdebefecd6a60a841acc8aafb7f0e89a76f996
- Kochenderfer, M.J. and Wheeler, T.A. (2019) 'Algorithms for Optimization', The MIT Press, Cambridge, Massachusetts, London, England.
- Land, A.H. and Doig, A.G. (1960) 'An Automatic Method of Solving Discrete Programming Problems', Econometrica, Vol. 28, Nr 3, pp. 497-520, The Econometric Society.
- Malea, C.I, and Nitu, E.L. (2020), 'Optimization of the technological process and equipment of complex profiled parts', IOP Conf. Series: Materials Science and Engineering **916**, 012058
- Menshikh,V., Samorokovskiy, A., and Avsentev O. (2018), 'Models of resource allocation optimization when solving the control problems in organizational systems', IOP Conf. Series: Journal of Physics: Conf. Series **973**, 012040
- Pawłowska, G. (2024), "Implementation of the branch and bound method for ILP and BIP problems", engineer thesis in Polish, Military University of Technology.
- PyQt5 (2022), [Online], [Retrieved October 23, 2024], https://python101.readthedocs.io/pl/latest/pyqt
- Python Software Foundation (2001-2024), [Online], [Retrieved October 23, 2024], https://docs.python.org/pl/3/reference/index.html
- Roy, J. S., Mitchell, S. A., and Peschiera, F. (2024) 'Optimization, Linear Programming, Operations Research', [Online], [Retrieved October 23, 2024], https://pypi.org/project/PuLP/
- Sierksma, G. (1996) 'Linear and Integer Programming', Marcel Dekker, Inc.