Implementation of primal Simplex algorithm for optimization

Roger Baiges Trilla & Cai Selvas Sala

March 29, 2024

Universitat Politècnica de Catalunya
Bachelor's degree in Artificial Intelligence
Optimization

Abstract

This project is made for the Optimization subject of the Bachelor's degree in Artificial Intelligence of the Universitat Politècnica de Catalunya (UPC). It offers a practical implementation of the Simplex algorithm, a widely-used method in linear programming for solving optimization problems. Developed with the intent to provide a solution for minimizing a linear objective function under linear constraints, the project utilizes Python along with the *numpy* library for its implementation.

Contents

1 Introduction

The goal of this project is to create an implementation of the primal Simplex algorithm capable of solving linear programming problems in standard form and with an objective function to be minimized. Also, it is a priority to make the implementation easy to use, handling problems of all dimensions, and with tools that provide information of all the process of getting to the solution in order to allow the user to understand all the steps of its resolution.

For this implementation, it has been decided that the programming language used would be Python along with the *numpy* library to make it more efficient.

Regarding the code structure, it has been determined that the solver would be stored in a Python class (`Simplex`) and all the information of the problem would be stored in another Python class (`Problem`). This allows the user to only create one `Simplex` object and solve each problem with only calling a method of the `Simplex` object with the `Problem` instance as a parameter.

Finally, it is worth noting that this primal Simplex algorithm has been made to be able to recognize unbounded or infeasible problems, so it can be also used to determine if a problem has an optimal or not.

2 Python files and implementation

The implementation of the Simplex algorithm is structured around two main classes, `Problem` and `Simplex`, both in separate Python files that will be explained in this section of the document. The execution of the problems is done through the *main.py* file and the validation of the implementation is done in the *comparison_scipy.py* file, both commented in this section as well.

2.1 *problem_class.py*

The Problem class is designed to encapsulate the data specific to the linear programming problem being solved. This includes the objective function coefficients (`c`), the constraints matrix (`A`), and the right-hand side values of the constraints (`b`). All these values of the problem should be given in standard form, posed to be minimized (not maximized), and not be modified during the execution of the Simplex. In each instance of `Problem`, other values like the solution of the problem, the finish status, the number of iterations needed to solve it... are added once the Simplex ends, so there is no need to save the results in other extra variables.

2.1.1 Constructor and initialization

In the initialization, a `data_id` and `problem_id` can be given in order to use the `__read_problem()` method to read the data from an example problem of the *data.txt* file. In case of specifying a `data_id` and a `problem_id`, the user can also set a value to the `dtype` parameter that will define the data type used to read the vectors and matrices of the problem. However, all these parameters are optional and the problem can be manually specified with the parameters `c`, `A` and `b`.

2.1.2 `__read_problem()`

The `__read_problem` method in the `Problem` class loads problem data from a file, using `data_id` and `problem_id` to locate the data.

This method is specific to the file format of the *data.txt* file. Therefore, for other problems it is better to manually specify the `c`, `A` and `b` to each `Problem` instance (it could be both in the initialization or after it).

2.2 *simplex_class.py*

The `Simplex` class is the core component of this project, implementing all the methods for the algorithm to solve linear programming problems (represented in standard form as instances of the `Problem` class).

2.2.1 Constructor and control flags

The constructor of the `Simplex` class initializes multiple control flags: `print_results`, `print_iters`, and `save_results`. These flags allow users to customize the amount of information about the execution that is printed in the output and the saving of the results obtained.

- `print_results`: Controls the printing of final results after solving the problem.
- `print_iters`: Determines whether details of each iteration are printed, showing the state of all vectors, matrices and values at the end of each iteration.
- `save_results`: If `True`, the final results and a summary of all the iterations is saved in a *.txt* file.

Additionally, there is a parameter named `tolerance` that controls the maximum value that `Z` (the objective function) can have in the artificial problem and still consider that it has arrived to an optimal solution (meaning that the original problem is feasible) and the execution should finish. This parameter is 10^{-10} by default, but it might be manually modified by the user depending on the dimensions of the problem.

2.2.2 Public methods

The `solve()` method is the main method of this class and the only one that is public and should be called by the user to solve a problem. Its purpose is to orchestrate the execution of the algorithm for a particular problem (object of the `Problem` class passed as argument).

It mainly calls the `__phase1()` (to organize the generation and solving of the artificial problem) and `__phase2()` (to solve the actual problem) methods, as well as calling other methods to print or save the results of the problems.

2.2.3 Private methods

2.2.3.1 Managing algorithm phases

The simplex algorithm's execution is divided into two distinct phases, each addressed by specific private methods:

- `__phase1`: Creates an artificial problem (done in the `__generate_artificial_problem()` method by adding artificial variables) to solve in order to find the starting Basic Feasible Solution (BFS) of the original problem. This method also saves the results inside the instance of `Problem` of the artificial problem.
- `__phase2`: This method makes the same as `__phase1()` but with the original problem, but without needing to create a new instance of `Problem`, as it was already given when the `solve()` method was called.

Each of this methods call the `__run()` method, which is the actual responsible of solving a problem. However, `__phase1()` calls it passing the artificial as argument and `__phase2()` does it with the original problem.

2.2.3.2 Solving the problem

The main method to solve a problem with the primal Simplex algorithm is `__run()`. It calls the method to initialize all the values (vectors, matrices, ...) and performs a `while` loop that will only stop when the problem is solved or found to be unbounded/infeasible. Each of the iterations of the loop is an iteration of the primal Simplex algorithm, where multiple other methods are called to calculate the reduced costs in order to check if the optimal has been found, find the basic feasible direction and check if the problem is unbounded, select the entering and leaving variables, update the vectors and matrices, update the `Z` value (value of the objective function), etc.

Key to the simplex algorithm's iterative process are the pivoting operations, governed by the `__pivot()` method. This method updates the basis of the problem, facilitating movement towards an optimal solution. Accompanying this are several auxiliary methods:

2.2.3.3 Auxiliary methods

As it has already been mentioned, all the steps of each iteration of the algorithm are mainly handled by individual methods and coordinated by the `__run()` method. From these individual methods (methods that have simple tasks to perform and usually don't call other methods), `__pivot()` is probably the most remarkable. Its purpose is to update all the values, vectors and matrices at the end of each iteration (including the exchange between the entering and the leaving variables), and there is a particular update of matrix that is important to mention...

It is well known that the computational cost of calculating the inverse of a matrix is high, particularly as the size of the matrix increases. This is due to the complex algorithms involved in performing matrix inversion, which typically have a computational complexity of $O(n^3)$ [1] for an $n \times n$ matrix when using traditional methods. Because matrix inversion is a frequent requirement in many algorithms, including those in linear programming, such as our primal Simplex algorithm, it is essential to find more efficient ways to update the inverse.

In this implementation, rather than recalculating the inverse of the basis matrix B from scratch in each iteration, an update method is used. This method exploits the fact that the basis matrix B only changes by one column at each iteration (when one variable enters and another leaves the basis). Consequently, instead of recomputing B^{-1} entirely, a rank-one update is applied to the B^{-1} of the previous iteration, which is much more computationally efficient.

This update of B^{-1} is done in the `__update_B_inv()` method (called by `__pivot()`). Additionally, due to this update of inverse, there is no need to store and update the B matrix because the algorithm only uses B^{-1} to solve the problems, making this approach not only more computationally efficient, but also more memory efficient.

At this point, it would be reasonable to object that the value of the inverse matrix B^{-1} can't be "updated" at the start of the problem because the matrix B is not stored and there is no

previous value of B^{-1}. However, this is not an issue, as for the artificial problem, it is known that the inverse matrix B^{-1} always starts as the identity matrix. Meanwhile, for the original problem, the inverse matrix B^{-1} starts as the final inverse matrix B^{-1} found when solving the artificial problem, corresponding to the first Basic Feasible Solution (BFS) that the primal Simplex algorithm will iterate through.

The only downside of this approach is that some precision is lost while updating the inverse of the matrix. For example, in the artificial problem, instead of finishing with a `Z` value (value of the objective function) of 0, it might finish with a very small number (around 10^{-13}, for instance). As it has been mentioned previously, to address this issue the `Simplex` class includes the parameter `tolerance` in its initialization, which determines the maximum value of `Z` to still consider that the result is the same as it if were 0 (meaning that the original problem is feasible).

Another thing to point out is that the methods used to select the entering and leaving variables in this implementation use the Bland's rule [2] to avoid cycles during the execution (guaranteeing termination with an optimal solution in a finite time). Concretely, this ensures that if the algorithm revisits a previously encountered basis, the same entering and leaving variables will not be selected, thereby avoiding any cycle that would prevent convergence.

Finally, it is worth mentioning that the method `__check_if_feasible()` is always called after the execution of the artificial problem (phase 1) in order to check if the original problem is feasible or not. Furthermore, this method detects if there is degeneracy in the original problem checking if there are artificial variables in the basis of the optimal solution found for the artificial problem (which also implies infeasibility). If the original problem is found to be infeasible, phase 2 is not executed.

2.2.3.4 Other methods

The `Simplex` class also contains other methods to print all the results of the iterations, save the results in *.txt* files, ... Allowing the user to understand how the problem was solved by the algorithm.

2.3 *main.py*

This file contains all the code and calls for the execution of the Simplex algorithm for all the problems needed for this delivery (problems 1, 2, 3 and 4 of the data sets 10 and 48). Due to the dimensions of the vectors and matrices of these problems, the tolerance parameter of the `Simplex` class has been set to 10^{-10}, but it could be modified for other problems in order to guarantee solid results. Also, the parameters `print_iters` and `print_results` are set to `False` in order to not have any output in the stdout channel (because 8 different problems are executed consecutively and it would not be practical to read the results from the output), but the `save_results` parameter is set to `True` in order to save the results in *.txt* files in the */results* directory.

2.4 *comparison_scipy.py*

The goal of this file is to compare the results obtained by this implementation of the primal Simplex algorithm with the implementation that can be found in the *SciPy* library in Python. The comparison is made with all the example problems given in the *data.txt* file, and for all these problems the results are the same in both implementations, which indicates that this implementation is precise in a wide set of problems (including optimal, unbounded and infeasible problems).

3 Results

In this section, the results obtained by this implementation of the primal Simplex algorithm for the problems 1, 2, 3, 4 of the data sets 10 and 48 (all from the *data.txt* file) will be analyzed and commented. All these results can be found in the *.txt* files inside the */results* directory.

The results obtained include the following values:

- ***vb***: The indices of the variables that have a value different than 0 in the final solution (the variables in the basis). It is worth noting that these indices start from 0 (not from 1), so any index i always satisfies $0 \leq i < n$, where n is the total number of variables in the problem.
- ***xb***: The coefficients multiplying the variables of the basis in the final solution.
- ***r***: The reduced costs of all the non-basic variables in the last iteration (all positive if the solution is optimal).
- ***Z***: The value of the objective function in the final solution (minimum if the solution is optimal).

3.1 10.1

In the artificial problem, a Basic Feasible Solution (BFS) has been found after 17 iterations with a value of $Z \approx 0^*$. Therefore, the problem is feasible.

The problem is optimal and its solution is:

$$\begin{aligned} vb &= [11, 9, 10, 8, 4, 19, 15, 18, 3, 16] \\ xb &= [2.73343216, 2.23204975, 0.63602912, 5.14178718, 0.8764317, \\ &\quad 224.46766678, 579.39365316, 150.12947563, 2.88012077, 97.15025427] \\ r &= [1.22603506 \times 10^2, 1.18361579 \times 10^2, 4.62074965 \times 10^{-1}, \\ &\quad 1.37902261 \times 10^2, 3.36200772 \times 10^1, 4.73763195 \times 10^1, \\ &\quad 1.59086332 \times 10^2, 1.38626751 \times 10^{-1}, 4.30009529 \times 10^1, \\ &\quad 2.06711082 \times 10^1] \\ Z &= -520.2228376998106 \end{aligned}$$

This solution has been reached after 14 iterations of the algorithm.

3.2 10.2

In the artificial problem, a Basic Feasible Solution (BFS) has been found after 21 iterations with a value of $Z \approx 0^*$. Therefore, the problem is feasible.

*Value very close to zero, but not 0 due to the inaccuracies explained in 2.2.3.3.

The problem is optimal and the solution found is:

$$
\begin{aligned}
vb &= [10, 8, 5, 18, 12, 11, 3, 1, 2, 17] \\
xb &= [3.21698232 \times 10^{0}, 1.93186213 \times 10^{0}, 8.77131077 \times 10^{-1}, 3.33107639 \times 10^{2}, \\
&\quad 1.31763057 \times 10^{-1}, 1.89795340 \times 10^{0}, 1.18140200 \times 10^{0}, 2.33805588 \times 10^{0}, \\
&\quad 1.45908892 \times 10^{0}, 1.08075693 \times 10^{2}] \\
r &= [4.88309590 \times 10^{1}, 1.51361826 \times 10^{2}, 5.97230027 \times 10^{1}, 4.02335193 \times 10^{1}, \\
&\quad 5.34789882 \times 10^{-1}, 1.16701091 \times 10^{-1}, 5.25151880 \times 10^{-1}, 3.68256437 \times 10^{1}, \\
&\quad 4.98478013 \times 10^{1}, 1.88824086 \times 10^{-2}] \\
Z &= -709.0182772884003
\end{aligned}
$$

This solution has been reached after only 2 iterations of the algorithm.

3.3 10.3

The optimal solution for the phase 1 has a value of $Z \not\approx 0$, indicating that no feasible solution exists for the original problem because not all artificial variables could be removed from the basis (meaning the problem constraints cannot be satisfied without them). Therefore, the problem is infeasible.

3.4 10.4

In the artificial problem, a Basic Feasible Solution (BFS) has been found after 14 iterations with a value of $Z \approx 0^{*}$. Therefore, the problem is feasible.

However, the original problem is unbounded due to the the observation that $d_B \geq [0]$, where d_B represents the direction coefficients for the basic variables. This lack of negative values in d_B means that the descent direction for the objective function is not restricted by any of the problem's constraints; implying that, for at least one variable, the objective function can decrease without limit, thus leading to an unbounded problem.

3.5 48.1

In the artificial problem, a Basic Feasible Solution (BFS) has been found after 19 iterations with a value of $Z \approx 0^{*}$. Therefore, the problem is feasible.

The problem is optimal. The solution found is:

$$
\begin{aligned}
vb &= [8, 9, 17, 13, 16, 2, 10, 3, 6, 11] \\
xb &= [2.60577195, 3.43429796, 273.76791858, 0.57911628, 148.92446464, \\
&\quad 1.08503138, 1.10698349, 1.74568786, 1.75368328, 1.06891412] \\
r &= [1.36142176 \times 10^{2}, 8.89117657 \times 10^{1}, 5.46356125 \times 10^{1}, 9.00554716 \times 10^{-2}, \\
&\quad 7.81794074 \times 10^{-1}, 1.22294647 \times 10^{2}, 5.63032571 \times 10^{1}, 3.19309313 \times 10^{-1}, \\
&\quad 1.98040923 \times 10^{-2}, 5.20844758 \times 10^{-1}] \\
Z &= -880.0192663413169
\end{aligned}
$$

This solution has been found after 10 iterations of the algorithm.

3.6 48.2

In the artificial problem, a Basic Feasible Solution (BFS) has been found after 14 iterations with a value of $Z \approx 0^*$. Therefore, the problem is feasible.

The problem is optimal. The solution found is:

$$
\begin{aligned}
vb &= [7, 13, 14, 4, 10, 2, 0, 11, 15, 16] \\
xb &= [2.15152202 \times 10^{0}, 1.63352822 \times 10^{0}, 4.53709904 \times 10^{2}, 9.74151034 \times 10^{-1}, \\
&\quad 7.44810125 \times 10^{-2}, 1.80668608 \times 10^{-1}, 1.91756726 \times 10^{0}, 1.23933173 \times 10^{0}, \\
&\quad 1.80071753 \times 10^{2}, 2.97186329 \times 10^{2}] \\
r &= [1.11027075 \times 10^{2}, 6.13436525 \times 10^{1}, 1.33242852 \times 10^{2}, 8.96052476 \times 10^{1}, \\
&\quad 1.45917738 \times 10^{1}, 9.06577636 \times 10^{1}, 3.12128627 \times 10^{1}, 1.31830645 \times 10^{-1}, \\
&\quad 6.77179101 \times 10^{-1}, 1.41598841 \times 10^{-1}] \\
Z &= -266.63183625180386
\end{aligned}
$$

This solution has been found after 11 iterations of the algorithm.

3.7 48.3

As observed in the third problem of the data 10, the optimal solution for the phase 1 has a value of $Z \not\approx 0$, indicating that no feasible solution exists for the original problem because not all artificial variables could be removed from the basis (meaning the problem constraints cannot be satisfied without them). Therefore, the problem is infeasible.

3.8 48.4

In the artificial problem, a Basic Feasible Solution (BFS) has been found after 13 iterations with a value of $Z \approx 0^*$. Therefore, the problem is feasible.

However, as seen in the results of the problem 10.4, the original problem is unbounded due to the the observation that $d_B \geq [0]$, where d_B represents the direction coefficients for the basic variables. This lack of negative values in d_B means that the descent direction for the objective function is not restricted by any of the problem's constraints; implying that, for at least one variable, the objective function can decrease without limit, thus leading to an unbounded problem.

4 Conclusions

This implementation of the primal Simplex algorithm has achieved our initial goals, performing as good as the implementation included in the *SciPy* libary of Python. Therefore, it is a good tool to solve linear programming problems, and also to understand how these are solved, as it provides ways of seeing the progress of the execution when solving a problem.

Additionally, the object oriented implementation (`Simplex` and `Problem` classes) allows the user to run multiple problems consecutively and still have the results automatically saved in the same instance of `Problem` (or save the results in *.txt* files too). Furthermore, it encapsulates all the necesary functions inside a single class, making it very easy to import from different documents and to modify the parameters of the algorithm (such as `tolerance`, `print_iters`, ...).

5 References

[1] Wikipedia contributors. Computational complexity of mathematical operations — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Computational_complexity_of_mathematical_operations&oldid=1189645589`, 2023. [Online; accessed 24-March-2024].

[2] Wikipedia contributors. Bland's rule — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Bland%27s_rule&oldid=1185843384`, 2023. [Online; accessed 24-March-2024].