

Department of Computer Science and Engineering (HTE)

A PROJECT BASED LABORATORY REPORT

ON

Simplex Method With Branch Bound Method

SUBMITTED BY:

|  |  |
| --- | --- |
| ID-NO | NAME |
| 2300031389 | DEEPIKAREDDY MANDAPATI |

UNDER THE ESTEEMED GUIDANCE OF

DR. B. BIKRAM KUMAR

Associate Professor



KONERU LAKSHMAIAH EDUCATION FOUNDATION

GREEN FIELDS, VADDESWARAM – 522502

Guntur Dst. Andra Pradesh. India.

Department of Computer Science and Engineering Honours



CERTIFICATE

This is to certify that the project-based laboratory report on

“Simplex Method With Branch Bound Method”

Submitted by 2300031389 - Deepika Reddy Mandapati to the Department of Computer Science and Engineering (Honours), Koneru Lakshmaiah Education Foundation in partial fulfilment of the requirements for the completion of a project in 23MT2004- Mathematical Programming course in II year B.Tech (CSE) is a bonafide record of the work carried out by her under my supervision during the Odd Semester of the Academic Year 2024-25.

Supervisor

DR. B. BIKRAM KUMAR

Associate Professor

ACKNOWLEDGEMENTS

It is a great pleasure for me to express my gratitude to our honourable President Sri Koneru Satyanarayana for having given the opportunity and platform with facilities in accomplishing the project-based laboratory report. I express the sincere gratitude to our Dean/Director/HOD, for his administration towards our academic growth. I express sincere gratitude to our Course coordinator and faculty. I record it as my privilege to deeply thank management for providing us efficient faculty and facilities to make our ideas into reality. I express my sincere thanks to our Project supervisor, Dr. B Bikram Kumar for  he novel association of ideas, encouragement, appreciation, and intellectual zeal, which motivated us to venture this project successfully. Finally, it is pleased to acknowledge the indebtedness to all those who have devoted themselves, directly or indirectly, to make this project report success.

- 2300031389

Deepika Reddy Mandapati

​​**Table of Contents**

​

​

|  |  |  |
| --- | --- | --- |
| S.No | Topics | Page Numbers |
| 1 | Introduction | 5 |
| 2 | Objectives | 6 |
| 3 | Advantages &Disadvantages | 7-8 |
| 4 | Algorithm | 9-11 |
| 5 | Code | 12-16 |
| 6 | Output | 17-18 |
| 7 | Conclusion | 19 |

**INTRODUCTION:**

**1. Simplex Method**

The Simplex Method is an algorithm widely used to solve Linear Programming (LP) problems, where the goal is to find the maximum or minimum value of a linear objective function subject to a set of linear equality and inequality constraints. LP problems are characterized by:

* A linear objective function (e.g., maximize profit or minimize cost).
* A set of linear constraints defining feasible solutions.
* Variables that can take any real (continuous) values within the constraints.

The Simplex Method operates on feasible solutions by iterating along the edges of the feasible region—a geometric shape called a polytope—until it finds the optimal vertex (corner point). It is efficient for many practical problems and is often used as the foundation for solving more complex problems, like integer programming.

**2. Branch and Bound Method**

While the Simplex Method is effective for LP problems with continuous variables, it cannot directly solve Integer Programming (IP) problems where some or all variables are restricted to integer values. The Branch and Bound Method is a systematic algorithm for solving IP problems by combining relaxation techniques with a strategic exploration of solution space.

* It starts by solving a relaxed version of the IP problem (without integer constraints) using a method like the Simplex.
* If the relaxed solution is not integer-valued, the solution space is split (branched) into sub

**OBJECTIVES:**

**1.Objectives of the Simplex Method:**

1. **Understand the Simplex Algorithm**: Learn how the Simplex Method iterates through the vertices of the feasible region to find the optimal solution for linear programming problems.
2. **Maximize or Minimize a Linear Function**: Use the Simplex Method to solve problems that involve maximizing or minimizing a linear objective function subject to linear constraints.
3. **Explore Feasible Regions**: Understand the concept of feasible regions and how the Simplex Method navigates through them efficiently to find the optimal vertex.
4. **Handle Constraints Effectively**: Apply the Simplex Method to problems with multiple constraints to determine feasible solutions within the given limits.

**2.Objectives of the Branch and Bound Method:**

1. **Understand the Branch and Bound Algorithm**: Learn how the Branch and Bound Method solves integer programming problems by systematically exploring feasible solutions.
2. **Handle Integer Constraints**: Apply Branch and Bound to problems that require integer values for decision variables, ensuring the solution respects integer constraints.
3. **Implement Branching**: Understand how the algorithm divides the problem into smaller subproblems (branching) and solves them recursively.
4. **Explore Feasible Regions with Bounding**: Learn how to use upper and lower bounds to prune branches and avoid unnecessary exploration of infeasible solutions.

**ADVANTAGES**

* **Simplex Method:**

1. **Efficiency in Practice**: Despite its worst-case exponential time complexity, the Simplex Method is highly efficient in practice and solves most problems in polynomial time.
2. **Applicability to Large Problems**: It can handle linear programming problems with a large number of variables and constraints, making it suitable for complex real-world problems.
3. **Optimal Solution**: The Simplex Method guarantees finding the optimal solution (if one exists) for linear optimization problems, as long as the problem is feasible.
4. **Versatility**: It is widely used in diverse fields, including economics, operations research, and supply chain management, due to its flexibility in solving linear programming problems.

* **Branch and Bound Method:**

1. **Solves Integer Problems:** The Branch and Bound Method is specifically designed to solve integer programming problems, where the decision variables must take integer values**.**
2. **Global Optimality:** It guarantees finding the global optimal solution by exploring all possible solutions systematically and pruning suboptimal ones.
3. **Flexibility**: The method can be applied to both linear and nonlinear integer programming problems, offering flexibility for a wide range of applications.
4. **Handles Complex Constraints:** It can handle complex constraints that may involve non-convex feasible regions, making it suitable for real-world problems with complicated constraints.

**DISADVANTAGES**

* **Simplex Method:**

1. **Worst-Case Exponential Time Complexity**: In the worst case, the Simplex Method can take an exponential amount of time, though this is rare in practice.
2. **Sensitive to Degeneracy**: In some cases, the Simplex Method can get stuck in cycles, especially in degenerate problems (when multiple optimal solutions exist), although this can be handled with specific rules.
3. **Non-Polynomial Worst-Case Performance**: Although efficient in practice, it does not guarantee polynomial time complexity, which may be an issue for very large or complex problems.
4. **No Guarantee for Non-Linear Problems**: The Simplex Method is designed for linear programming and does not directly apply to nonlinear optimization problems without modifications.

* **Branch and Bound Method:**

1. **Time Complexity:** The method can be very slow for large problems as it explores many potential solutions (branches), making it computationally expensive in practice.
2. **Memory Usage:** Since the algorithm stores multiple subproblems (branches) in memory, it can be memory-intensive, especially for large problems with many variables.
3. **Difficulty with Nonlinear Problems:** While it works well with integer linear problems, the method may face challenges when applied to non-linear problems, where it requires specialized modifications.
4. **No Guarantee of Speed:** Branch and Bound is an exhaustive search technique that, although guaranteed to find the optimal solution, does not guarantee fast convergence or efficiency.

**PSEUDOCODE/ALGORITHM:**

* **Simplex Method :**

1. Convert the problem to standard form:
   1. Ensure all constraints are equality constraints by introducing slack variables.
   2. Convert the objective function to the form: Maximize c1x1+c2x2+...+cnxnc\_1 x\_1 + c\_2 x\_2 + ... + c\_n x\_nc1​x1​+c2​x2​+...+cn​xn​.
2. Set up the initial Simplex tables:
   1. The tableau contains the objective function coefficients, constraint coefficients, and right-hand side values. The tableau should look like this:
3. Check for optimality:
   1. If all the coefficients in the bottom row (objective row) are non-negative, the current solution is optimal.
   2. If there are negative values in the bottom row, proceed to the next step.
4. Choose the pivot column:
   1. Identify the column with the most negative coefficient in the objective row. This column corresponds to the variable that should enter the solution (pivot column).
5. Choose the pivot row:
   1. For each positive entry in the pivot column, compute the ratio of the right-hand side (RHS) to the column value:
      1. Ratio=RHS/pivot column value​
      2. The row with the smallest positive ratio corresponds to the pivot row.
6. Perform the pivot operation:
   1. The intersection of the pivot column and pivot row is the pivot element. Normalize the pivot row by dividing all its elements by the pivot element.
   2. Adjust the other rows to zero out the pivot column by performing elementary row operations.
7. Update the tableau:
   1. After the pivot operation, update the tableau. The basic variable corresponding to the pivot row will now become non-basic, and the non-basic variable from the pivot column will become basic.
8. Repeat steps 3–7:
   1. Continue iterating by checking for optimality, selecting pivot columns and rows, and performing pivoting operations until the objective row contains only non-negative coefficients.
9. Obtain the solution:
   1. Once the optimal solution is reached (all objective row coefficients are non-negative), the values in the RHS column of the tableau represent the optimal solution for the decision variables. The value in the bottom-right cell represents the optimal objective value.

* **Branch and Bound Method:**

1. Solve the LP relaxation:
   1. Ignore the integer constraints (treat the problem as a continuous LP) and solve it using the Simplex method.
   2. If the solution satisfies the integer constraints, it is a potential optimal solution. If not, proceed to the next step.
2. Check the feasibility:
   1. If the LP solution is feasible and satisfies the integer conditions (i.e., the decision variables are integers), update the current best solution.
   2. If the solution is not integer, proceed with branching.
3. Branching:
   1. Identify the first non-integer variable in the LP solution.
   2. Create two new subproblems by adding new constraints that force the non-integer variable to take integer values.
      1. One subproblem forces the variable to be less than or equal to the floor of the value.
      2. The other subproblem forces the variable to be greater than or equal to the ceiling of the value.
4. Solve the relaxed LP subproblems:
   1. For each branch, solve the LP relaxation of the new subproblem by ignoring the integer constraints (like Step 1).
   2. Calculate the objective value for each subproblem. If the objective value is worse than the best found so far, prune (discard) that subproblem.
5. Bounding:
   1. For each subproblem, calculate the upper or lower bound on the objective function based on the LP solution. If the bound is worse than the current best solution, discard the subproblem (pruning).
6. Prune infeasible or suboptimal branches:
   1. If a subproblem has an infeasible solution (i.e., no feasible solution satisfies all the constraints) or if its bound is worse than the current best, prune it.
   2. If the solution of a subproblem is feasible and better than the current best, update the best solution.
7. Repeat branching and bounding:
   1. Continue branching and bounding until all subproblems have been either solved or pruned.
8. Obtain the optimal solution:
   1. The process stops when no further branches can improve the objective value. The best feasible solution found during the process is the optimal solution.

**PYTHON CODE**

* Simplex method:

import numpy as np

import math

def simplex(c, A, b):

tableau = to\_tableau(c, A, b)

iteration = 0

print("Initial Tableau:")

print\_tableau(tableau)

while can\_be\_improved(tableau):

print(f"\nIteration {iteration + 1}")

pivot\_position = get\_pivot\_position(tableau)

print(f"Pivot Element at Row {pivot\_position[0] + 1}, Column {pivot\_position[1] + 1}")

tableau = pivot\_step(tableau, pivot\_position)

print("Updated Tableau:")

print\_tableau(tableau)

iteration += 1

solution = get\_solution(tableau)

print("\nOptimal Solution Found:")

print(f"Objective function value: {tableau[-1][-1]}")

print(f"Variable values: {solution}")

return solution

def to\_tableau(c, A, b):

xb = [row + [rhs] for row, rhs in zip(A, b)]

z = c + [0]

return xb + [z]

def can\_be\_improved(tableau):

z = tableau[-1]

return any(x > 0 for x in z[:-1])

def get\_pivot\_position(tableau):

z = tableau[-1]

column = next(i for i, x in enumerate(z[:-1]) if x > 0)

restrictions = []

for eq in tableau[:-1]:

el = eq[column]

restrictions.append(math.inf if el <= 0 else eq[-1] / el)

row = restrictions.index(min(restrictions))

return row, column

def pivot\_step(tableau, pivot\_position):

new\_tableau = [[] for \_ in tableau]

i, j = pivot\_position

pivot\_value = tableau[i][j]

new\_tableau[i] = np.array(tableau[i]) / pivot\_value

for eq\_i, eq in enumerate(tableau):

if eq\_i != i:

multiplier = np.array(new\_tableau[i]) \* tableau[eq\_i][j]

new\_tableau[eq\_i] = np.array(tableau[eq\_i]) - multiplier

return new\_tableau

def get\_solution(tableau):

columns = np.array(tableau).T

solutions = []

for column in columns[:-1]:

solution = 0

if is\_basic(column):

one\_index = column.tolist().index(1)

solution = columns[-1][one\_index]

solutions.append(solution)

return solutions

def is\_basic(column):

return list(column).count(0) == len(column) - 1 and list(column).count(1) == 1

def print\_tableau(tableau):

for row in tableau:

print(" | ".join(f"{num:8.2f}" for num in row))

print("-" \* 50)

# Example

c = [3, 5]

A = [

[1, 2],

[3, 2]

]

b = [6, 12]

solution = simplex(c, A, b)

* **Branch and bound code:**

import numpy as np

from scipy.optimize import linprog

class BranchAndBound:

def \_\_init\_\_(self, c, A, b):

self.c = -np.array(c)

self.A = np.array(A)

self.b = np.array(b)

self.best\_solution = None

self.best\_value = float('-inf')

def solve(self):

print("Starting Branch and Bound...\n")

# Start branching

self.branch\_and\_bound([], [])

print("\nBranch and Bound completed.")

return self.best\_solution, -self.best\_value # Negate to get original max value

def branch\_and\_bound(self, integer\_indices, bounds):

solution = self.linear\_relaxation(bounds)

if solution is None:

print("Infeasible solution encountered, backtracking...\n")

return

current\_value = -solution.fun

print(f"Current relaxation solution: {solution.x}, Value: {current\_value}")

if current\_value <= self.best\_value:

print("Current value is not better than the best known solution, backtracking...\n")

return

x = solution.x

if all(int(x[i]) == x[i] for i in integer\_indices):

# Update the best solution

if current\_value > self.best\_value:

print(f"Found a new best integer solution: {x}, Value: {current\_value}\n")

self.best\_value = current\_value

self.best\_solution = x

return

for i in integer\_indices:

if int(x[i]) != x[i]:

break

print(f"Branching on variable index {i} with value {x[i]}...")

lower\_bound = bounds + [(i, 'le', int(x[i]))]

upper\_bound = bounds + [(i, 'ge', int(x[i]) + 1)]

print(f"Creating lower bound branch: x[{i}] <= {int(x[i])}")

self.branch\_and\_bound(integer\_indices, lower\_bound)

print(f"Creating upper bound branch: x[{i}] >= {int(x[i]) + 1}")

self.branch\_and\_bound(integer\_indices, upper\_bound)

def linear\_relaxation(self, bounds):

A\_mod = self.A.copy()

b\_mod = self.b.copy()

for var\_idx, relation, value in bounds:

if relation == 'le':

A\_mod = np.vstack([A\_mod, np.eye(1, len(self.c), var\_idx)[0]])

b\_mod = np.append(b\_mod, value)

elif relation == 'ge':

A\_mod = np.vstack([A\_mod, -np.eye(1, len(self.c), var\_idx)[0]])

b\_mod = np.append(b\_mod, -value)

try:

return linprog(c=self.c, A\_ub=A\_mod, b\_ub=b\_mod, method='highs', bounds=(0, None))

except:

return None

# Example usage:

# Objective: Maximize 3x1 + 5x2

# Constraints:

# x1 + 2x2 <= 6

# 3x1 + 2x2 <= 12

# x1, x2 >= 0 and integers

c = [3, 5]

A = [

[1, 2],

[3, 2]

]

b = [6, 12]

solver = BranchAndBound(c, A, b)

solution, max\_value = solver.solve()

print(f"Optimal solution: {solution}")

print(f"Optimal value: {max\_value}")

**Output:**

Simplex Method:

A screenshot of a computer program

Description automatically generated

Branch and Bound:

A screenshot of a computer

Description automatically generated

**CONCLUSION:**

The integration of the Simplex Method and Branch and Bound Method provides a powerful framework for solving both linear and integer programming problems.

1. **Simplex Method**: As demonstrated in this project, the Simplex Method is efficient for solving Linear Programming (LP) problems where decision variables are continuous. By iterating through the feasible region, it navigates towards the optimal solution. The algorithm guarantees optimality when feasible solutions exist, and despite its exponential worst-case time complexity, it performs efficiently for most practical problems. Its versatility and ability to handle large problems make it a crucial tool in optimization tasks, especially in industries such as operations research, logistics, and economics.
2. **Branch and Bound Method**: When dealing with Integer Programming (IP) problems, where variables must take integer values, the Branch and Bound method is indispensable. It leverages a systematic approach to explore solution spaces by branching and bounding, ensuring that the optimal integer solution is found. Though it can be computationally expensive for large problems, its ability to guarantee optimality and handle complex constraints gives it a significant advantage in solving integer programming problems, making it valuable in scheduling, resource allocation, and decision-making scenarios.

In conclusion, combining the Simplex and Branch and Bound Methods enhances the ability to solve both linear and integer programming problems, offering efficient solutions to real-world optimization challenges. However, it is important to consider the limitations of time and memory complexity, especially when dealing with large or complex problem instances. Future improvements could focus on optimizing these algorithms to enhance computational efficiency and memory usage, particularly for large-scale problems.