R.P.I. Math Report No. 200
September 17, 1991

An improved branch and bound algorithm for mixed integer nonlinear programs. [1] [2]

Brian Borchers and John E. Mitchell

Department of Mathematical Sciences
Rensselaer Polytechnic Institute
Troy, NY 12180

Abstract

This report describes a branch and bound code for zero–one mixed-integer nonlinear programs with convex objective functions and constraints. The code uses heuristics to detect subproblems which do not have integral solutions. When the code detects such a subproblem, it creates two new subproblems instead of solving the current problem to optimality. Computational results on sample problems show that these heuristics can significantly decrease solution times for some problems.

[1]Research partially supported by ONR Grant number N00014–90–J–1714.

[2]This paper is taken in part from a thesis to be submitted by Brian Borchers in partial fulfillment for the degree of Doctor of Philosophy in the Department of Mathematical Sciences at Rensselaer Polytechnic Institute.

Introduction

This report is concerned with zero–one mixed integer nonlinear programming problems of the form:

$$
\begin{array}{llrcl}
(MINLP) & \min & f(\mathbf{x},\mathbf{y}) & & \\
 & \text{subject to} & \mathbf{g}(\mathbf{x},\mathbf{y}) & \leq & 0 \\
 & & \mathbf{x} & \in & \{0,1\}^m \\
 & & \mathbf{y} & \leq & \mathbf{u} \\
 & & \mathbf{y} & \geq & \mathbf{l}
\end{array}
$$

Here $\mathbf{x}$ is a vector of m zero–one variables, $\mathbf{y}$ is a vector of n continuous variables, and $\mathbf{u}$ and $\mathbf{l}$ are vectors of upper and lower bounds for the continuous variables $\mathbf{y}$. The objective function f and the constraint functions $\mathbf{g}$ are assumed to be convex.

Mixed integer non-linear programs of this form arise in a number of applications, in areas as diverse as network design [13, 19], chemical process synthesis [5, 9, 16], product marketing [6], and capital budgeting [17, 20, 22]. Methods for solving mixed integer nonlinear programming problems are surveyed in [10, 12].

Branch and bound algorithms such as the ones described in [11, 17, 18] work by explicitly enumerating possible values of the zero–one variables until an optimal integer solution has been found. The algorithm begins by solving the continuous relaxation of the original problem. If a zero–one variable is fractional at optimality, the algorithm constructs two new subproblems, in which the variable is fixed at zero and one. The algorithm continues in this fashion by solving subproblems and creating new subproblems until an integer solution has been found and each remaining subproblem has a higher lower bound than the integer solution. This report describes a branch and bound algorithm which uses the sequential quadratic programming method to solve the subproblems, and which uses heuristics to determine when to split a subproblem into two new subproblems.

The remainder of this report is organized as follows: In section 1, we review the sequential quadratic programming method. In section 2, we describe the heuristics that our branch and bound algorithm uses. In section 3, we describe a method for calculating a lower bound on the value of a subproblem without solving the subproblem to optimality. In section 4, we describe our branch and bound algorithm in detail. Section 5 contains computational results for a number of sample problems. Our conclusions are presented in section 6. An appendix contains the formulation of a sample problem which was constructed for this paper.

1 Solving the Subproblems

At each stage of the branch and bound algorithm, we must solve a nonlinear programming problem of the form:

$$
(NLP) \qquad \begin{array}{rrcl}
\min & f(\mathbf{x},\mathbf{y}) & & \\
\text{subject to} & \mathbf{g}(\mathbf{x},\mathbf{y}) & \leq & 0 \\
 & \mathbf{x} & \leq & \mathbf{e} \\
 & \mathbf{x} & \geq & 0 \\
 & \mathbf{y} & \leq & \mathbf{u} \\
 & \mathbf{y} & \geq & \mathbf{l}
\end{array}
$$

where some of the zero–one variables in $\mathbf{x}$ may have been fixed at zero or one. Since this is a convex problem, we know that if it is feasible, it has a unique minimum. Furthermore, the Lagrangean for this problem is defined as:

$$L(\mathbf{x},\mathbf{y},\lambda) = f(\mathbf{x},\mathbf{y}) + \lambda^T \mathbf{g}(\mathbf{x},\mathbf{y})$$

where $\mathbf{x}$ and $\mathbf{y}$ are still subject to upper and lower bounds, and the Lagrangean multipliers λ are restricted to positive values. Since the problem is convex, the Lagrangean has a unique stationary point at $\mathbf{x}^*, \mathbf{y}^*, \lambda^*$, where $\mathbf{x}^*, \mathbf{y}^*$ is an optimal solution to (NLP), and λ^* is a set of optimal Lagrange multipliers, and λ_i is non-zero only if $\mathbf{g}(\mathbf{x}^*,\mathbf{y}^*)_i = 0$.

The method of sequential quadratic programming, described in [4, 8], attempts to find this stationary point and solve (NLP) by solving a sequence of quadratic programs with linear constraints. These quadratic subproblems are of the form:

$$
(QP) \qquad \begin{array}{rrcl}
\min & \nabla f^T \mathbf{p} + \frac{1}{2}\mathbf{p}^T (W)\mathbf{p} & & \\
\text{subject to} & A\mathbf{p} & = & -\mathbf{c}
\end{array}
$$

Here W is the Hessian of the Lagrangean with respect to the variables $\mathbf{x}$ and $\mathbf{y}$, the rows of A are the gradients of the active constraints, and $\mathbf{c}$ is the vector of values of the active constraints.

It can be shown that an optimal solution to (QP) gives a direction $\mathbf{p}$ which is identical to the direction (in the $\mathbf{x}$ and $\mathbf{y}$ variables) given by Newton's method applied to the problem of finding a stationary point of the Lagrangean. Furthermore, the Lagrange multipliers for (QP) are the Lagrange multipliers that would be obtained by an iteration of Newton's method applied to the problem of finding a stationary point of the Lagrangean. Thus one iteration of the SQP method is equivalent to an iteration of Newton's method searching for the stationary point of the Lagrangean.

Our code uses a routine called E04VCF from the NAG library [21] to solve (NLP). This routine uses a version of the SQP method in which the solution to (QP) is used as a search direction. The routine picks a step size that minimizes an augmented Lagrangean "merit function."

The E04VCF routine calls user supplied subroutines that compute the objective function, constraint functions, and their gradients. The E04VCF subroutine will execute until it reaches a limit on the number of iterations, it finds an optimal solution, or it encounters an error condition. If the routine has reached the limit on iterations, the user can restart the routine from where it left off.

2 Heuristics for Detecting Fractional Solutions

Since the sequential quadratic programming algorithm generates estimates of the variables $\mathbf{x}$, we can examine them after each iteration to see if a variable is converging to a fractional value. This leads to heuristics for determining early in the solution of the subproblem that the optimal solution will have fractional zero–one variables.

Our experimental code examines the values of the zero–one variables after every second iteration of the sequential quadratic programming algorithm. (The E04VCF routine won't stop after just one iteration. It must run for at least two iterations at a time.) If the current solution is not feasible with respect to both the linear and non-linear constraints, then no determination is made. If the current solution is feasible, and the value of x_i is between 0.001 and 0.999, and the difference between the previous value of x_i and the new value of x_i is less than a small tolerance ϵ, and the new value of x_i is closer to the old value of x_i than it is to either 0 or 1, than the code declares x_i fractional. The current version of the code uses a tolerance of 0.1, which was chosen after experimenting with several different values of ϵ.

The experimental code uses a very simple rule for selecting the branching variable. Of the variables which are considered fractional, the experimental code chooses the most fractional variable (i.e. the variable which has value closest to 0.5) to branch on. In choosing the next subproblem to work on, the experimental code picks the subproblem with the smallest estimated optimal value. This estimate is simply the objective value of the parent subproblem if it was solved to optimality, or the objective value at the last iteration if the parent subproblem was not solved to optimality.

3 Generating Lower Bounds

In the branch and bound algorithm, when an early decision to break the current subproblem into two new subproblems is made, a lower bound on the values of the two new subproblems is required. This bound can be used later to fathom the subproblems if a better integer solution is found.

Given a set of Lagrange multipliers λ, we can use Lagrangian duality to find a lower bound for the optimal value of the current subproblem. This lower bound is given by solving the nonlinear programming problem:

$$
(DUAL) \qquad \begin{array}{rrcl}
\min & f(\mathbf{x},\mathbf{y})+\lambda^T\mathbf{g}(\mathbf{x},\mathbf{y}) & & \\
\text{subject to} & \mathbf{x} & \leq & \mathbf{e} \\
 & \mathbf{x} & \geq & 0 \\
 & \mathbf{y} & \leq & \mathbf{u} \\
 & \mathbf{y} & \geq & \mathbf{l}
\end{array}
$$

where again, some of the zero–one variables may be fixed at zero or one.

This is a nonlinear programming problem with simple bound constraints that can easily be solved by a quasi-Newton method. Furthermore, if the multipliers λ are close to optimality, the minimum value of (DUAL) should be close to the optimal value of (NLP). However, this lower bounding procedure is likely to fail to give an improved lower bound if the Lagrange multipliers are not well chosen. For this reason, our algorithm doesn't attempt to use this lower bounding procedure until the Lagrange multipliers have had time to converge to their optimal values. If the lower bounding scheme fails, our algorithm will continue to work on the current subproblem until lower bounding succeeds or until an optimal solution has been found.

Our experimental code uses the NAG routine E04KBF to solve (DUAL). The initial guess is simply the current estimated solution $\mathbf{x}$ and $\mathbf{y}$ to the current subproblem. The Lagrangean function and its gradient are calculated automatically from the user supplied objective and constraint functions.

4 The Branch and Bound Algorithm

Our branch and bound algorithm for solving mixed integer nonlinear programs is as follows:

Algorithm 1

```
1. Call a user supplied routine to do any initialization needed by the routines
   which compute the constraint and objective functions.
2. Put the NLP relaxation of the problem into the branch and bound tree as
   the root.
3. While there are unexamined subproblems in the branch and bound tree
    (a) Pick a subproblem from the branch and bound tree. If all subproblems
        in the branch and bound tree have a higher lower bound than a known
        integer solution, then go to step 4.
    (b) Repeat the following steps until one of the conditions is satisfied.
        i. Take $S$ steps of the SQP algorithm.
```

   ii. *If the subproblem is infeasible, then drop it from consideration. Go back to step 3.*
   iii. *If the solution is optimal and the zero-one variables are all either zero or one, then we have found an integer solution. Record the value of this solution and go back to step 3.*
   iv. *If the solution is optimal and a zero-one variable is fractional, then split the current subproblem into two new subproblems with the fractional variable fixed alternately at zero and one. The objective value of the current subproblem provides a lower bound for the two new subproblems.*
   v. *If a zero-one variable appears to be converging to a fractional value, and the current solution is feasible with respect to the linear and non-linear constraints, then consider splitting the current subproblem.*
      - *If the current objective value is close to the lower bound for this problem, then branch without attempting to find a new lower bound.*
      - *If it appears that the Lagrange multipliers are close to their optimal values, then use the lower bounding procedure described in section 3 to find a lower bound. If this succeeds, then branch. If the lower bounding procedure fails, go back to step 3–(b).*
      - *Otherwise, go back to step 3–(b).*

4. *The optimal solution is simply the best integer solution found up to this point.*

In step 3-b-(i), the algorithm performs S steps of the SQP algorithm. S is a parameter that can be adjusted. If we set S to a very large value, then the algorithm will solve each subproblem to optimality within S steps or detect infeasibility, but it will never use the heuristics to detect a fractional variable. This is a conventional branch and bound algorithm for (MINLP). If S is set to a smaller value, the heuristics will come into play. Because of a restriction in the E04VCF subroutine, we were only able to use values of S that were greater than one. Our experimental code uses the smallest possible value $S = 2$.

In step 3–b–(ii) of our algorithm, we have to determine whether the current subproblem is infeasible. The E04VCF routine will automatically detect infeasibility with respect to the linear constraints. However, if a subproblem is infeasible with respect to one or more of its nonlinear constraints, the E04VCF routine does not automatically detect the condition. Instead, the routine will return with an indication that it stopped after it could not find an improved point. This could be because the problem was infeasible, or it could be that the SQP routine was simply having trouble finding a feasible point. In this situation, we examine the current solution. If the current solution is close to feasibility (within 10%), then we restart the SQP algorithm in hopes of eventually finding a feasible solution. If the SQP algorithm fails a second time, we declare the subproblem infeasible.

In step 3–b–(v) we need to determine whether the subproblem is feasible. If the current solution is within 1% of feasibility with respect to each of the linear and non-linear constraints, then we assume that the current subproblem is feasible. If the algorithm errs in considering a problem feasible, then it is possible that the algorithm could create two additional subproblems, which would in turn be infeasible. Although this would slow the algorithm down, it wouldn't prevent the algorithm from finding the correct solution. In practice, this hasn't been a problem.

We also need to determine in step 3–b–(v) whether or not to use the lower bounding procedure. A lower bound for the current subproblem is inherited from the parent subproblem. In some cases, fixing a variable at zero or one causes no change in the optimal objective value, and there is little point in attempting to find an improved lower bound. Thus if we have a feasible point with an objective value within 2% of the current lower bound, we skip the lower bounding procedure.

The experimental code makes use of three user supplied subroutines. The first routine, SETUPP, is used to initialize the continuous relaxation of the problem. SETUPP returns the total number of variables, number of integer variables, a matrix for any linear constraints, and feasibility tolerances for the linear and nonlinear constraints. A second routine, OBJFUN, calculates the objective function and its gradient. The third routine, CONFUN, calculates the non-linear constraint functions and their gradients. These routines are then linked with the experimental code to create a program for solving the actual problem.

5 Computational Results

We tested the experimental code on a number of problems taken from the literature. Sample problems one through four were taken from a paper by Duran and Grossman [3]. The first three of these problems come from the chemical engineering problem of designing a chemical processing system. The fourth problem comes from the area of product marketing [6]. The third and fourth problems also appear in [15]. Our fifth sample problem is example problem five from a paper by Floudas, Aggarwal, and Ciric [5]. (Other problems in this paper were non-convex.) Our sixth sample problem is example four from a paper by Kocis and Grossman [16]. This problem is a convex version of a chemical process design problem that was non-convex in its original formulation. The seventh problem is a topological network design problem which is described in detail in the appendix. A summary of the problems is presented in Table 1.

Two versions of the experimental code were developed. In the first version, the heuristics and lower bounding procedure are used after every other iteration of the SQP method. In the second version, the heuristics are not used. These codes were run under AIX on an IBM 3090-200S. The total number of SQP iterations, total number of subproblems created, total number of subproblems solved, and CPU times were recorded for each code on each problem. The CPU times were measured with the CPUTIME subroutine of the Fortran run-time library. Unfortunately, our experience has been that these CPU times are only accurate to about 5%, since repeated runs of the code show slight but consistent variations in the CPU time. This could be caused by an inaccuracy in the timing routines or by cache effects. These computational results are presented in Tables 2 and 3.

In several cases, there are slight differences between the two codes in how many subproblems were created and later solved. This can be explained by slightly different choices of the branching variables that were sometimes made by the two codes.

The code with heuristics worked about as well as the code without heuristics on the smaller problems. This is not surprising, since on these problems, the SQP method needed an average of 3 to 4 iterations to solve a subproblem, and there was little opportunity for the heuristics to come into play. On the more difficult problems 6 and 7, the SQP algorithm required more iterations to solve each subproblem, and the heuristics could be used successfully. For example, on problem 7, the code without heuristics used an average of 20 iterations on each subproblem that it solved, while the code with heuristics used an average of only 16 iterations per subproblem.

The paper by Paules and Floudas [15] includes computational results for our problems 3 and 4 which can be compared to our results. These problems were solved by both generalized Bender's decomposition (GBD) and an outer approximation algorithm on an IBM–3090 computer similar to the computer used in our study. On problem 3, GBD required 16.35 CPU seconds while the outer approximation algorithm required 7.45 CPU seconds. In contrast, our branch and bound codes needed only about 1 CPU second to solve the problem. On problem 4, GBD required over 800 CPU seconds to solve the problem. The outer approximation algorithm required between 16.09 and 25.63 CPU seconds depending on the starting point. Our codes both solved the problem in about 29 CPU seconds.

6 Summary and Conclusions

Our computational results indicate that the heuristics used in this experimental code were sometimes successful in reducing the number of iterations of the SQP method needed to solve the MINLP's. On some problems there was little or no improvement, while on other problems the code with heuristics reduced the number of SQP iterations by as much as 20%. In general, the heuristics worked better when a large number of SQP iterations were required for each subproblem. Thus there is some reason to hope that the heuristics might work well on larger mixed integer nonlinear programming problems.

Problem	Zero–One Vars	Continuous Vars	Linear Constraints	Nonlinear Constraints
1	3	3	4	2
2	5	6	11	3
3	8	9	19	4
4	25	5	5	25
5	4	3	4	5
6	24	22	72	1
7	10	100	41	0

Table 1: Characteristics of the sample problems

Problem	SQP Iters	Nodes Created	Problems Solved	CPU Time	Optimal Value
1	21	5	5	0.1	6.00976
2	41	15	9	0.3	73.0353
3	75	25	15	1.0	68.0097
4	461	101	81	28.6	8.06487
5	25	11	7	0.2	4.57958
6	123	23	17	8.5	285507
7	2862	211	145	84.4	15.6086

Table 2: Computational results for the code without heuristics

Problem	SQP Iters	Nodes Created	Problems Solved	CPU Time	Optimal Value
1	21	5	5	0.1	6.00976
2	37	13	9	0.3	73.0353
3	70	25	15	1.1	68.0097
4	461	99	81	28.7	8.06486
5	25	11	7	0.2	4.57958
6	90	21	15	6.6	285507
7	2238	175	137	67.8	15.6086

Table 3: Computational results for the code with heuristics

Appendix: Example Problem 7

Our seventh example problem is a small network design problem for a data communications network involving five cities (New York, Los Angeles, Chicago, Philadelphia, and Houston.) The distances between these cities and the populations of the cities were obtained from the 1990 World Almanac and Book of Facts [14]. The total flow between pairs of cities was made proportional to the product of the populations. These flows are given in Table 4. The cost of a link between two cities was made equal to the distance between the two cities in miles. These distances are given in Table 5.

City	New York	Los Angeles	Chicago	Houston	Philadelphia
New York	0	0.592	0.547	0.314	0.298
Los Angeles	0.592	0	0.245	0.141	0.134
Chicago	0.547	0.245	0	0.130	0.124
Houston	0.314	0.141	0.130	0	0.071
Philadelphia	0.298	0.134	0.124	0.071	0

Table 4: Flows Between Cities

City	New York	Los Angeles	Chicago	Houston	Philadelphia
New York	0	2786	802	1608	100
Los Angeles	2786	0	2054	1538	2706
Chicago	802	2054	0	1067	738
Houston	1608	1538	1067	0	1508
Philadelphia	100	2706	738	1508	0

Table 5: Distances Between Cities

The problem is to choose a set of links between cities and a routing of the data such that the total queueing delay is minimized subject to a budget constraint. Problems of this type are discussed in [1, 2, 7]. To formulate this problem, we number the nodes from 1 to 5, and Let x_{ij} be 0 if the arc (i,j) is in the network and 1 if the arc is not in the network. We'll assume that the communications links are bidirectional, so we will always have $x_{ij} = x_{ji}$. Let f_{ij} be the flow between nodes i and j. Let f_{ij}^{s} be the flow on arc (i,j) of data destined for node s.

Each link will have a nominal capacity of 1.0 in each direction. However, since the queueing delay would be infinite if the flow was at full capacity, we'll introduce a capacity constraint of 90%. This can be written as:

$$\sum_{s=1}^{n} f_{ij}^s \le 0.9 x_{ij}$$

There are 20 of these constraints, with one for each direction on each link.

Let F_i^s be the total flow of data from node i to node s for $i \neq s$. The network flow constraint at node i for data destined for node s can be written as:

$$\sum_{\text{arcs (i,j)}} f_{ij}^s - \sum_{\text{arcs (j,i)}} f_{ji}^s = F_i^s$$

For our five city problem, there are 20 such constraints.

Finally, we have a budget constraint:

$$\sum_{i<j} c_{ij} x_{ij} \le 8000$$

Here c_{ij} is the mileage between cities i and j. This constraint limits the network to a total of 8000 miles of links. (Without it, the optimal solution would involve every possible link.)

The objective is to minimize the total queueing delay in the network. This delay can be approximated by the sum over all arcs of the queueing delay on an individual arc, which is given by $D(f_{ij}) = f_{ij}/(1 - f_{ij})$, where $f_{ij} = \sum_{s=1}^{n} f_{ij}^s$ is the total flow on arc (i, j), and 1 is the normalized capacity of the link. Unfortunately, this function has a singularity which could cause numerical problems. To avoid this, we use the function:

$$D(f_{ij}) = \begin{cases} f_{ij}/(1 - f_{ij}) & \text{if } f_{ij} \le 0.9 \\ 9 + 100(f_{ij} - 0.9) + 1000(f_{ij} - 0.9)^2 & \text{Otherwise} \end{cases}$$

This alternative function matches the original function for all feasible flows (where $f_{ij} \le 0.9$), and is continuous and twice differentiable for all values of f_{ij}.

Thus the problem can be written as:

$$\begin{array}{llrcll}
(NET) & \min & \sum_{\text{arcs (i,j)}} D(f_{ij}) & & & \\
& \text{subject to} & \sum_{s=1}^{n} f_{ij}^s & \le & 0.9 x_{ij} & \text{for all arcs (i,j)} \\
& & \sum_{\text{arcs (i,j)}} f_{ij}^s - \sum_{\text{arcs (j,i)}} f_{ji}^s & = & F_i^s & \text{for all nodes } i \neq s \\
& & \sum_{i<j} c_{ij} x_{ij} & \le & 8000 & \\
& & f_{ij}^s & \ge & 0 & \text{for all } i, j, s \\
& & x_{ij} & = & x_{ji} & \text{for all } i \neq j \\
& & x_{ij} & \in & \{0, 1\} & \text{for all } i \neq j
\end{array}$$

In this form, the problem has 10 zero-one variables, 100 continuous variables, 41 linear constraints, and a nonlinear objective function. The optimal solution has a total queueing delay of 15.6086. It uses links between New York and all four of the other cities, a link between Houston and Chicago, and a link between Houston and Los Angeles. This solution uses 7901 miles of the 8000 mile budget.

References

[1] Dimitri Bertsekas and Robert Gallager. *Data Networks.* Prentice-Hall, Englewood Cliffs, N.J., 1987.

[2] Robert R. Boorstyn and Howard Frank. Large-scale network topological optimization. *IEEE Transactions on Communications*, 25:29–47, 1977.

[3] Marco A. Duran and Ignacio E. Grossmann. An outer-approximation algorithm for a class of mixed-integer nonlinear programs. *Mathematical Programming*, 36:307–339, 1986.

[4] R. Fletcher. *Practical Methods of Optimization.* John Wiley & Sons, New York, second edition, 1987.

[5] C. A. Floudas, A. Aggarwal, and A. R. Ciric. Global optimum search for nonconvex NLP and MINLP problems. *Computers and Chemical Engineering*, 13(10):1117–1132, 1989.

[6] B. Gavish, D. Horsky, and K. Srikanth. An approach to the optimal positioning of a new product. *Management Science*, 29(11):1277–1297, 1983.

[7] Mario Gerla and Leonard Kleinrock. On the topological design of distributed computer networks. *IEEE Transactions on Communications*, 25:48–60, 1977.

[8] Philip E. Gill, Walter Murray, and Margaret Wright. *Practical Optimization.* Academic Press, New York, 1981.

[9] I. E. Grossmann. Mixed-integer programming approach for the synthesis of integrated process flow-sheets. *Computers and Chemical Engineering*, 9:463–482, 1985.

[10] Omprakash K. Gupta and A. Ravindran. Nonlinear integer programming algorithms: A survey. *OPSEARCH*, 20(4):189–206, 1983.

[11] Omprakash K. Gupta and A. Ravindran. Branch and bound experiments in convex nonlinear integer programming. *Management Science*, 31(12):1533–1546, 1985.

[12] Pierre Hansen. Methods of nonlinear 0-1 programming. In P. L. Hammer, E. J. Johnson, and B. H. Korte, editors, *Discrete Optimization II*, Annals of Discrete Mathematics. North Holland, New York, 1979.

[13] H. H. Hoang. Topological optimization of networks: A nonlinear mixed integer model employing generalized benders decomposition. *IEEE Transactions on Automatic Control*, 27:164–169, 1982.

[14] Newspaper Enterprise Association Inc. *The World Almanac and Book of Facts*. Scripps Howard Company, 1990.

[15] Granville E. Paules IV and Christodoulos A. Floudas. APROS: Algorithmic development methodology for discrete-continuous optimization problems. *Operations Research*, 37(6):902–915, 1989.

[16] Gary R. Kocis and Ignacio E. Grossman. Global optimization of nonconvex mixed-integer nonlinear programming (MINLP) problems in process synthesis. *Industrial & Engineering Chemistry Research*, 27:1407–1421, 1988.

[17] D. J. Laughhunn. Quadratic binary programming with applications to capital-budgeting problems. *Operations Research*, 18(3):454–461, 1970.

[18] E. L. Lawler and D. E. Wood. Branch and bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[19] T.L. Magnanti and R. T. Wong. Network design and transportation planning: Models and algorithms. *Transportation Science*, 18(1):1–55, February 1984.

[20] J. C. T. Mao and B. A. Wallingford. An extension of Lawler and Bell's method of discrete optimization with examples from capital budgeting. *Management Science*, 15(2):51–60, 1968.

[21] NAG. *FORTRAN Library Manual Mark 13: Volume 3*. Numerical Analysis Group, Oxford, 1988.

[22] H. M. Weingartner. Capital budgeting of interrelated projects. *Management Science*, 12(7):485–516, 1966.