Branch-and-Bound Algorithms

Chapter · January 2011

DOI: 10.1002/9780470400531.eorms0116

Kiavash Kianfar
Texas A&M University
27 PUBLICATIONS **209** CITATIONS

BRANCH-AND-BOUND ALGORITHMS

KIAVASH KIANFAR
Department of Industrial and Systems Engineering, Texas A&M University, College Station, Texas

For most discrete optimization problems, complete (or explicit) enumeration of solution space to find the optimal solution is out of question because even for small problem sizes the number of solution points in the feasible region is extremely large (e.g., even if a single point can be enumerated in 10^{-10} s, enumerating all points in a relatively small problem with only 75 binary variables will take about 120,000 years!). *Branch-and-bound* is a general purpose approach to *implicitly* enumerate the feasible region and was first introduced by Land and Doig [1] for solving integer programming problems. It works based on a few simple principles to avoid enumerating every solution point explicitly.

Although branch-and-bound is often discussed in the context of integer programming, it is actually a general approach that can also be applied to solve many combinatorial optimization problems even when they are not formulated as integer programs. For example, there are efficient branch-and-bound algorithms specifically designed for job shop scheduling or quadratic assignment problems, which are based on the combinatorial properties of these problems and do not use their integer programming formulations. Nevertheless, it is true that almost all integer programming solvers use branch-and-bound to solve IP problems and, therefore, application of branch-and-bound in the context of IP is of special importance. In this article we present the main concepts of branch-and-bound in a general setting and then discuss some problem-specific details in the context of integer programming. Many of these details can be extended to more general settings other than IP depending on the problem that is being solved.

BRANCH-AND-BOUND BASIC IDEAS

In this section we explain the main concepts of branch-and-bound as a general discrete optimization approach. Many of branch-and-bound concepts discussed here are based on Nemhauser and Wolsey [2] and Wolsey [3] the reader can refer to these resources for further reading. Consider a general combinatorial optimization problem like

$$z = \max\left\{f(x) : x \in S\right\}.$$

Branch-and-bound is a divide and conquer strategy, which decomposes the problem to subproblems over a *tree* structure, which is referred to as *branch-and-bound tree*. The decomposition works based on a simple idea: If S is decomposed into S^1 and S^2 such that $S = S^1 \cup S^2$, and we define subproblems $z^k = \max\{f(x) : x \in S^k\}$ for $k = 1, 2$, then $z = \max_k z^k$. Each subproblem represents a *node* on the tree. Fig. 1 shows a schematic of a branch-and-bound tree. The main problem with feasible region S (for simplicity we call it problem S) is at the *root node,* and is then divided into two subproblems (with feasible regions) S^1 and S^2, where we have $S^1 \cup S^2 = S$. The process of dividing a node subproblem into smaller subproblems is called *branching* and subproblems S^1 and S^2 are called *branches* created at node S. In Fig. 1 subproblem S^1 is further branched into smaller subproblems and so on. The branching does not necessarily have to be two-way, and multiway branching is also possible.

Observe that branching indefinitely will only result in explicit enumeration of the feasible region of S. Therefore to avoid explicit enumeration, in branch-and-bound whenever possible a branch is pruned (or fathomed), meaning that its subproblem is not divided anymore. In other words, the feasible region of the node subproblem is implicitly enumerated without branching any deeper. But when can we prune a branch? The main idea is to use bounds on the objective value of subproblems intelligently to prune branches. That is why the method is called *branch-and-bound*.

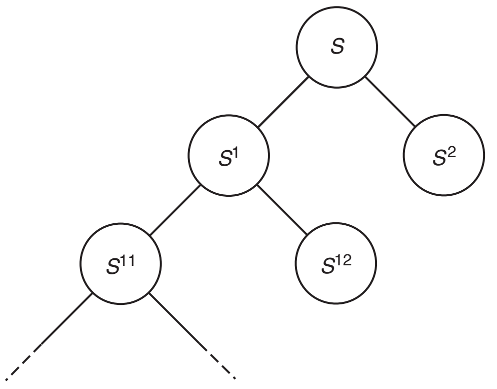


Figure 1. Subproblems in a branch-and-bound tree.

Fig. 2 shows the flow chart of the branch-and-bound algorithm in a general context. For a maximization problem, a lower bound $\underline{z}$ is updated throughout the algorithm and is used in pruning the nodes. The updating of the lower bound is normally triggered by finding an optimal solution with a better objective value at a node subproblem. Clearly, an optimal solution at a node problem gives a lower bound on z. Additionally, the branch-and-bound algorithm must have a mechanism to calculate an upper bound for the objective value of a node subproblem. For example, in IP solving the linear programming (LP) relaxation of the subproblem gives an upper bound on the IP objective value. As observed in Fig. 2, in general, a node is pruned if one of the following cases happens:

Pruning by infeasibility. If the feasible region of a node subproblem is empty, the node is naturally pruned.

Pruning by bound. If the upper bound calculated for a node subproblem is not greater than the lower bound on z, that is, $\bar{z}^i \leq \underline{z}$, then the node is pruned because there is no point in searching the feasible region of that node when we know that the best objective value we can obtain is not better than a solution we already know.

Pruning by optimality. When it is possible to find the optimal solution to a node subproblem S^i, then the node is pruned and its solution is stored as the incumbent if its objective value is better than the best we know so far. The lower bound $\underline{z}$ is also updated in this case.

If a node cannot be pruned based on any of the above conditions then new branches are created to decompose the node subproblem into smaller problems. The algorithm stops when all nodes are pruned, that is, there is no active node remaining. The optimal solution will be the incumbent solution with objective value $\underline{z}$. The finiteness of the number of steps in a branch-and-bound algorithm has been theoretically studied and proved within a general formulation in works such as Refs 4 and 5.

This provides a complete high level view to branch-and-bound. In a more technical level there are many detailed questions that must be addressed in implementing the algorithm. The answers to many of these questions depend upon the particular problem being solved. Some of these questions are as follows [3]:

- How should the upper bounds be calculated?
- What is the appropriate balance between the time spent to find an upper bound and the strength of the upper bound?
- How should we do the branching, that is, how should a node subproblem be decomposed into smaller problems?
- How many branches should be created at each branching? Two or more?
- Should the branching rule be an a priori rule or should it adapt as the algorithm proceeds?
- How should we choose the next active node to consider?

As mentioned, almost all solvers for IP problems use branch-and-bound. In the next section, we discuss the LP-based branch-and-bound algorithm for solving an IP problem specifically and address the questions above in the context of solving the IP problem.

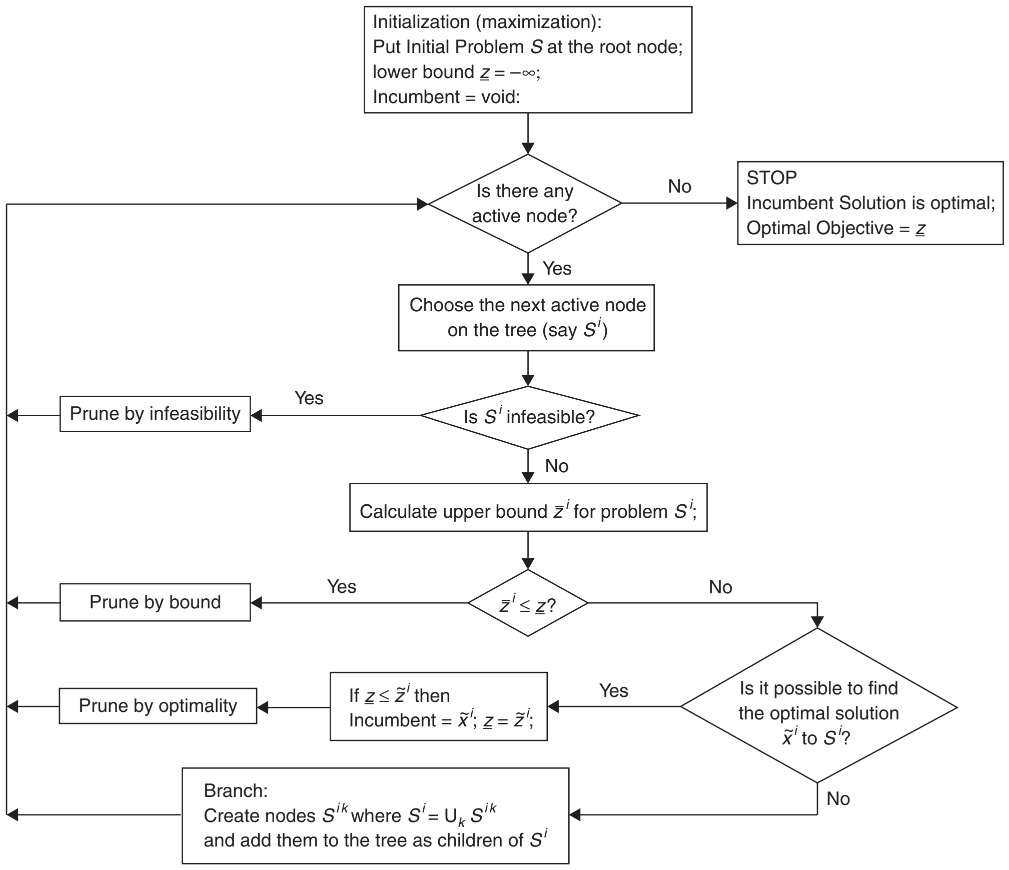

Initialization (maximization):
Put Initial Problem S at the root node;
lower bound z = −∞;
Incumbent = void:
Is there any active node?
No
STOP
Incumbent Solution is optimal;
Optimal Objective = z
Yes
Choose the next active node on the tree (say S^i)
Is S^i infeasible?
Yes
Prune by infeasibility
No
Calculate upper bound z̄^i for problem S^i;
z̄^i ≤ z?
Yes
Prune by bound
No
Is it possible to find the optimal solution x̃^i to S^i?
Yes
If z ≤ z̃^i then Incumbent = x̃^i; z = z̃^i;
Prune by optimality
No
Branch:
Create nodes S^ik where S^i = ∪_k S^ik and add them to the tree as children of S^i

Figure 2. General branch-and-bound algorithm.

LP-BASED BRANCH-AND-BOUND TO SOLVE INTEGER PROGRAMMING PROBLEMS

Consider an integer programming problem like

$$z = \max\{cx : x \in S\},$$

where $S = \{x : x \in Z^n \cap P\}$ and P is a polyhedron. As before we refer to this problem as problem S. For simplicity we talk about pure integer programming problems but what follows can be easily extended to mixed integer programming problems too. Figure 3 shows the flow chart of the branch-and-bound algorithm for solving the IP problem S. This flow chart is a special case of the general flow chart of Fig. 2 customized in order to solve the IP problem.

Illustrative Example. Consider the IP problem S:

$$\begin{aligned} \max z = 3x_1 + 2x_2 \\ 3x_1 + 4x_2 \le 12 \\ 2x_1 + x_2 \le 5 \\ x_1, x_2 \ge 0 \text{ and integer.} \end{aligned}$$

The branch-and-bound tree for solving this problem is shown in Fig. 4.

Observe in Fig. 4 how the calculation of bound in the right branch helps us to prune the left branch and therefore saves us the effort of further enumerating the solutions in that branch.

Now let us address in more detail some of the questions we posed at the end of the section titled “Branch and Bound Basic

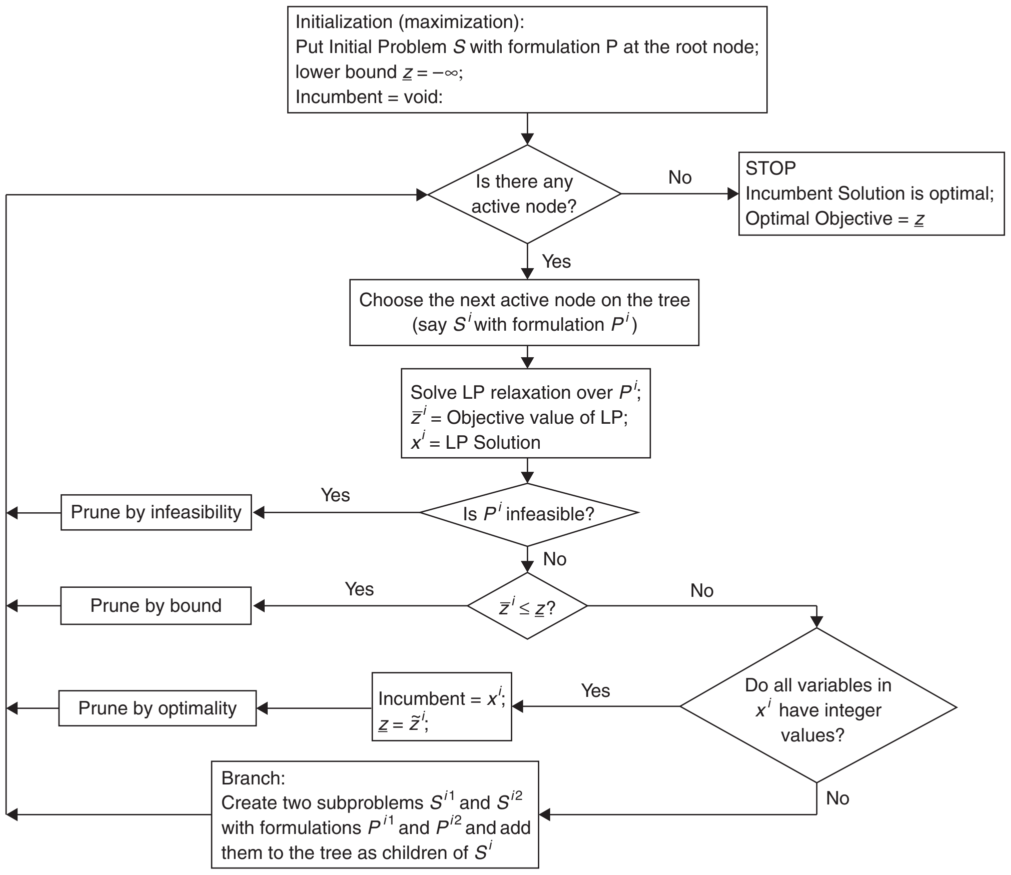
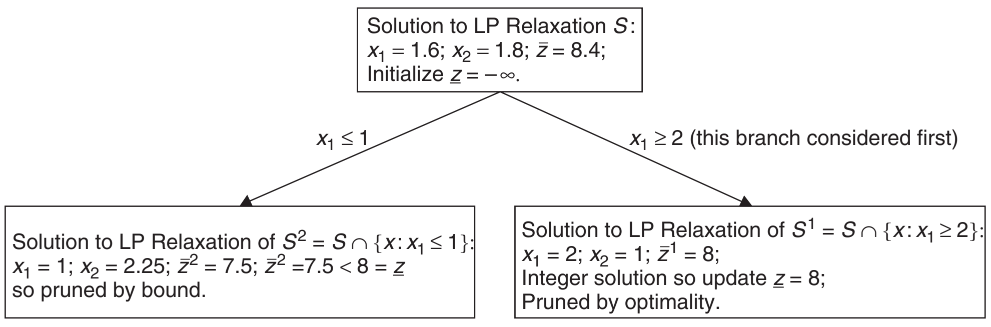

Initialization (maximization):
Put Initial Problem S with formulation P at the root node;
lower bound $\underline{z} = -\infty$;
Incumbent = void:
Is there any active node?
No
STOP
Incumbent Solution is optimal;
Optimal Objective = $\underline{z}$
Yes
Choose the next active node on the tree (say S^i with formulation P^i)
Solve LP relaxation over P^i;
$\bar{z}^i$ = Objective value of LP;
x^i = LP Solution
Yes
Prune by infeasibility
Is P^i infeasible?
No
Yes
Prune by bound
$\bar{z}^i \le \underline{z}$?
No
Prune by optimality
Incumbent = x^i;
$\underline{z} = \bar{z}^i$;
Yes
Do all variables in x^i have integer values?
No
Branch:
Create two subproblems S^{i1} and S^{i2} with formulations P^{i1} and P^{i2} and add them to the tree as children of S^i

Figure 3. Branch-and-bound flow chart for solving an IP problem.


Solution to LP Relaxation S:
$x_1 = 1.6$; $x_2 = 1.8$; $\bar{z} = 8.4$;
Initialize $\underline{z} = -\infty$.
$x_1 \le 1$
$x_1 \ge 2$ (this branch considered first)
Solution to LP Relaxation of $S^2 = S \cap \{x : x_1 \le 1\}$:
$x_1 = 1$; $x_2 = 2.25$; $\bar{z}^2 = 7.5$; $\bar{z}^2 = 7.5 < 8 = \underline{z}$
so pruned by bound.
Solution to LP Relaxation of $S^1 = S \cap \{x : x_1 \ge 2\}$:
$x_1 = 2$; $x_2 = 1$; $\bar{z}^1 = 8$;
Integer solution so update $\underline{z} = 8$;
Pruned by optimality.

So the optimal IP solution is $x_1 = 2$; $x_2 = 1$; $z = 8$.

Figure 4. Branch-and-bound tree for the example problem.

Ideas" with respect to branch-and-bound for solving the IP problem.

Bounding

Solving the LP relaxation of any IP subproblem S^i gives an upper bound on its objective value. This is the bound that is most commonly used in practice. The LP is usually solved using simplex-based algorithms. On very large models interior point methods may be best for solution of the first LP [3]. A desirable feature of LP relaxation with simplex is that an optimal or near-optimal basis of the problem can be stored so that the LP relaxation in subsequent nodes can be reoptimized rapidly.

Branching

An important question is how to branch, meaning how to split a subproblem into smaller subproblems. Here we review the most common methods.

Single Variable Branching. The simplest idea to split the feasible region of a subproblem S^i with formulation P^i is to pick an integer variable with fractional value in the LP relaxation optimal solution, say x_j with value $\bar{x}_j$, and create branches by adding simple linear constraints to formulation P^i as follows:

$$S^{i1} = S^i \cap \{x : x_j \leq \lfloor \bar{x}_j \rfloor\}$$
$$S^{i2} = S^i \cap \{x : x_j \geq \lceil \bar{x}_j \rceil\}.$$

S^{i1} is usually referred to as the *left (down) branch* and S^{i2} is referred to as the *right (up) branch*. This is a desirable choice because clearly we have $S^i = S^{i1} \cup S^{i2}$ and $S^{i1} \cap S^{i2} = \emptyset$, and furthermore, the current LP solution is not feasible for any of S^{i1} and S^{i2} so in the absence of multiple optima for LP the upper bound would strictly decrease in each of these branches. This branching scheme was first introduced in Ref. 6.

A generalization of the above branching scheme is using branches such as $S^{i1} = S^i \cap \{x : dx \leq d_0\}$ and $S^{i2} = S^i \cap \{x : dx \geq d_0 + 1\}$ in which d and d_0 are integer. However, in practice, only the simplest form which is the above single variable branching is used in all solvers.

With the single variable branching strategy, an important question is which variable should be chosen out of all variables with fractional value? A recent comprehensive study on this issue is done in Ref. 7; also see Refs 2, 3, 8, 9. An old and common rule is the *most fractional variable*. If C is the set of all integer variables with fractional LP relaxation value then the chosen variable is

$$\text{argmax}_{j \in C} \min\{f_j, 1 - f_j\},$$

where $f_j = \bar{x}_j - \lfloor \bar{x}_j \rfloor$. When branching is based on this rule, it is recommended that the $x_j \leq \lfloor \bar{x}_j \rfloor$ branch is considered first only if $f_j \leq 1 - f_j$. Otherwise the $x_j \geq \lceil \bar{x}_j \rceil$ should be selected first. Example in Fig. 4 obeys this rule. Accordingly the most fractional variable, that is, x_1, is chosen as the branching variable and then the right branch is selected after the root node because $f_1 = x_1 - \lfloor x_1 \rfloor = 0.6 > 0.4 = 1 - (x_1 - \lfloor x_1 \rfloor) = 1 - f_1$.

The most fractional rule is simple to implement but in Ref. 7, it is shown computationally that this rule is not better than just any random variable selection rule. More effective strategies have been proposed and studied over the years, which are more sophisticated. The method of *pseudocost branching* goes back to [10] and works based on calculating a pseudocost by keeping a history of success (change in LP relaxation value) of the left and right branching performed on each variable. The branching variable with the highest pesudocost is chosen each time. Different functions have been proposed for the pseudocost such as maximum of sum of left and right *degradations* (decrease in upper bound) [11], maximum of the smaller of the left and right degradations [10]; also see Ref. 8.

Strong branching [8,12] is in a sense an extreme effort to find the best branching variable. In its full version, at any node one tentatively branches on each candidate variable with fractional value and solves the LP relaxation for right and left branches for each variable either to optimality or for a specified number of dual simplex iterations. The degradation in bound for left and right branches are calculated and the variable is picked on the basis of the function of these degradations [3,8]. In addition to a number of dual simplex method iterations, the computational effort of strong branching can also be limited by defining a much smaller set of candidate branching variables.

An effective variable selection strategy was proposed in Ref. 7 called *reliability branching*. This method integrates strong and pseudocost branching. A threshold is defined on the number of previous branchings involving a variable. Below this threshold the pseudocost is considered unreliable and strong branching is used but above the threshold the pseudocost method is used.

We also note that many solvers provide the user with the option of assigning *user-specified priorities* to the integer variables. When branching, the solver picks the variable with a fractional value that has the highest assigned priority. This is especially useful when the user has some insight regarding the importance of variables based on what they represent in the application underlying the model [3].

GUB Branching. Many IP models contain the so-called *Generalized Upper Bound* (GUB) or *Special Ordered Set* (SOS) constraint of the form

$$\sum_{j=1}^{n} x_j = 1$$

with $x_j \in \{0, 1\}$ for $j = 1, \dots, n$. If the single variable branching on one of the variables $x_j, j = 1, \dots, n$, is performed then the two branches will be $S^{i1} = S^i \cap \{x : x_j = 0\}$ and $S^{i2} = S^i \cap \{x : x_j = 1\}$. Because of the GUB constraint, $\{x : x_j = 0\}$ will leave $n - 1$ possibilities $\{x : x_i = 1\}_{i \neq j}$ while $\{x : x_j = 1\}$ leaves only one possibility. So S^{i1} is usually much larger than S^{i2} and the tree is unbalanced. A more balanced split is desired and GUB branching proposed in Ref. 13 is a way to get a balanced tree which works as follows: The user provides a special order of variables in the GUB set, say $j_1, \dots, j_n$. Then the two branches will be as follows:

$$S^{i1} = S^i \cap \{x : x_{ji} = 0, i = 1, \dots, r\}$$
$$S^{i2} = S^i \cap \{x : x_{ji} = 0, i = r + 1, \dots, n\},$$

where $r = \min\left\{t : \sum_{i=1}^{t} \overline{x}_{ji} \geq \frac{1}{2}\right\}$. The number of nodes is usually significantly reduced with this branching scheme compared to the single variable branching.

Node Selection

Having a list of active (unpruned) nodes, the question is which node should be examined next. There are two categories of rules for this purpose: *static rules* that determine in advance the order of node selection and *adaptive rules* which use the information about the status of active nodes to choose a node. For a complete review of different node selection strategies see Ref. 8. Among static rules the *depth-first* and the *best-bound* (or *best-first*) are the two well-known extremes.

Depth-First Node Selection. Depth-first rule also known as *last in, first out* (LIFO) is as follows: if the current node is not pruned the next node is one of its children; and if it is pruned the next node is found by *backtracking* which means the next node is the child of the first node on the path from the current node to the root node with an unconsidered child node. Obviously, this is a completely a priori rule if a rule is specified to select between left and right children of a node. This rule has known advantages:

1. For pruning the tree a good lower bound is needed. The depth-first method descends quickly in the tree to find a first feasible solution which gives a lower bound that hopefully causes the pruning of many future nodes.
2. The depth-first method tends to minimize the memory requirements for storing the tree at any given time during the algorithm.
3. Passing from a node to its immediate child has the advantage that the LP relaxation can be easily resolved by adding just an additional constraint.

However, the depth-first search can result in an extremely large search tree. This is the result of the fact that we may need to consider many nodes that would have been fathomed if we had a better lower bound.

Best-Bound Node Selection. In this rule the next node is the active node with the best (largest) upper bound. With this rule one would never branch a node whose upper bound is less than the optimal value. As a result this rule minimizes the number of the nodes considered. However, the memory requirements for this method may become prohibitive if good lower bounds are not found early leading to relatively little pruning. In terms of reoptimizing the LP relaxations this method is also at a disadvantage because one LP problem has little relation to the next one.

Adaptive Node Selection. Adaptive methods make intelligent use of information about the nodes to select the next node. The *estimate-based methods* attempt to select nodes that may lead to improved integer feasible solutions. The *best projection* criterion [14] and the *best estimate* criterion found in Refs 10 and 15 are among these methods.

Many adaptive methods are *two-phase* methods that essentially use a hybrid of depth-first and best-bound searches. At the beginning the depth-first strategy is used to find a feasible solution and then the best-bound method is used [16]. Variations of two-phase methods using estimate-based approaches are also proposed [15,17]. Some other suggested rules use an estimation of the optimal IP solution to avoid considering *superfluous* nodes, that is, the nodes in which $\bar{z}^i < z$. The tree is searched in a depth-first fashion as long as $\bar{z}^i$ is greater than the estimate. After that a different criterion such as best-bound is used [10,11].

EXTENSIONS OF BRANCH-AND-BOUND

In solving integer programming *problems*, pure branch-and-bound is seldom used. In most cases cutting planes are added to the root problem or the node subproblems to tighten the feasible region of the LP relaxation and hence obtain better bounds faster. A branch-and-bound algorithm in which cutting planes are used is known under the general name of *branch-and-cut* [2,3,18].

In branch-and-cut at any node, after optimizing the LP relaxation, a *separation* problem is solved to find valid inequalities for feasible integer solutions, which are violated by the LP relaxation solution. These valid inequalities are then added to the problem and the problem is reoptimized to improve the LP relaxation bound. Branching happens when no further valid inequalities can be found. Of course the amount of effort spent to find valid inequalities is one of the parameters of the algorithm that should be decided. We refer the reader to the article titled ***Branch and Cut*** in this encyclopedia for further information.

Another extension of branch-and-bound is *branch-and-price* [2,3,19]. When the number of variables in an integer program is huge, a solution method is using *column generation* within the branch-and-bound framework. More specifically, implicit pricing of nonbasic variables is used to generate new columns or to prove LP optimality at a node of the branch-and-bound tree. Branching happens when no columns price out to enter the basis and the LP solution does not satisfy integrality constraints. We note that if cutting planes are also used in branch-and-price the algorithm is called *branch-cut-price.* We refer the reader to the article titled ***Branch-Price-and-Cut Algorithms*** in this encyclopedia for further information.

PARALLEL BRANCH-AND-BOUND

The divide and conquer nature of branch-and-bound makes it a suitable framework for attacking huge problems using today's parallel computing capabilities. For a survey of parallel branch-and-bound algorithms refer to Ref. 20. There are three types of parallelism that can be implemented for a branch-and-bound algorithm: Type 1 is parallel execution of operations on generated subproblems, for example, parallel bounding operations for each subproblem to accelerate execution. Type 2 consists of building the tree in parallel by performing operations on several subproblems simultaneously. Type 3 is building several trees in parallel. In each tree some operation such as branching, bounding or selection is performed differently but the trees share their information and use the best bounds among themselves. For further reading refer to Ref. 20.

REFERENCES

1. Land AH, Doig AG. An automated method of solving discrete programming problems. Econometrica 1960;28(2):497–520.
2. Nemhauser GL, Wolsey LA. Integer and combinatorial optimization. New York: Wiley-Interscience; 1988.
3. Wolsey LA. Integer programming. New York: Wiley; 1998.
4. Bertier P, Roy B. Procédure de résolution pour une classe de problèmes pouvant avoir un caractère combinatoire. Cah Cent Étud Rech Oper 1964;6:202–208.
5. Balas E. A note on the Branch-and-bound Principle. Oper Res 1968;16(2):442–445.
6. Dakin RJ. A tree search algorithm fo mixed integer programming. Comput J 1965;8: 250–255.
7. Achterberg T, Koch T, Martin A. Branching rules revisited. Oper Res Lett 2005;33:42–54.
8. Linderoth JT, Savelsbergh MWP. A computational study of search strategies for mixed integer programming. INFORMS J Comput 1999;11:173–187.
9. Lodi A. Mixed integer programming computation. In: Junger M, Liebling T, Naddef D, *et al*., editors. 50 years of integer programming 1958–2008. Berlin: Springer; 2010. pp. 619–645.
10. Benichou M, Gauthier JM, Girodet P, *et al*. Experiments in mixed-integer programming. Math Program 1971;1:76–94.
11. Gauthier JM, Ribière G. Experiments in mixed-integer linear programming using pseudocosts. Math Program 1977;12:26–47.
12. Applegate D, Bixby RE, Chvátal V, *et al*. The traveling salesman problem: a computational study. Princeton (NJ): Princeton University Press; 2007.
13. Beale EML, Tomlin JA. Special facilities in a generalized mathematical programming system for nonconvex problems using ordered sets of variables. In: Lawrence J, editor. Proceedings of the 5th Annual Conference on Operational Research. London: Tavistock Publications; 1970. pp. 447–457.
14. Mitra G. Investigation of some branch-and-bound strategies for the solution of mixed integer linear programs. Math Program 1973;4:155–170.
15. Forrest JJH, Hirst JPH, Tomlin JA. Practical solution of large scale mixed integer programming problems with UMPIRE. Manage Sci 1974;20:736–773.
16. Eckstein J. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. SIAM J Optim 1994;4:794–814.
17. Beale EML. Branch-and-bound methods for mathematical programming systems. In: Hammer PL, Johnson EL, Korte BH, editors. Discrete optimization II. Amsterdam: North Holland Publishing Co.; 1979.
18. Caprara A, Fischetti M. Branch-and-cut algorithms. In: Dell'Amico M, Maffioli F, Martello S, editors. Annotated bibliographies in combinatorial optimization. New York: Wiley; 1997. pp. 45–63.
19. Barnhart C, Johnson EL, Nemhauser GL, *et al*. Branch-and-price: column generation for solving huge integer programs. Oper Res 1998;46(3):316–329.
20. Gendron B, Crainic TG. Parallel branch-and-bound algorithms: survey and synthesis. Oper Res 1994;42(6):1042–1066.