Discrete Optimization 19 (2016) 79–102

Contents lists available at ScienceDirect

Discrete Optimization

www.elsevier.com/locate/disopt

Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning

David R. Morrison[a], Sheldon H. Jacobson[b], Jason J. Sauppe[c,*], Edward C. Sewell[d]

[a] *Inverse Limit, USA*

[b] *Department of Computer Science, University of Illinois at Urbana–Champaign, 201 North Goodwin Avenue, Urbana, IL 61801, USA*

[c] *Department of Computer Science, University of Wisconsin–La Crosse, 1725 State Street, La Crosse, WI 54601, USA*

[d] *Department of Mathematics and Statistics, Southern Illinois University Edwardsville, 1 Hairpin Drive, Edwardsville, IL 62025, USA*

ARTICLE INFO

Article history:
Received 2 September 2014
Received in revised form 7 October 2015
Accepted 20 January 2016
Available online 16 February 2016

MSC:
90-02
90C57
90C10
90C27

Keywords:
Branch-and-bound
Discrete optimization
Integer programming
Search strategies
Cyclic best first search
Survey

ABSTRACT

The branch-and-bound (B&B) algorithmic framework has been used successfully to find exact solutions for a wide array of optimization problems. B&B uses a tree search strategy to implicitly enumerate all possible solutions to a given problem, applying pruning rules to eliminate regions of the search space that cannot lead to a better solution. There are three algorithmic components in B&B that can be specified by the user to fine-tune the behavior of the algorithm. These components are the search strategy, the branching strategy, and the pruning rules. This survey presents a description of recent research advances in the design of B&B algorithms, particularly with regards to these three components. Moreover, three future research directions are provided in order to motivate further exploration in these areas.

1. Introduction

The branch-and-bound (B&B) framework is a fundamental and widely-used methodology for producing exact solutions to NP-hard optimization problems. The technique, which was first proposed by Land and Doig [1], is often referred to as an *algorithm*; however, it is perhaps more appropriate to say that B&B

* Corresponding author. Tel.: +1 608 785 6807.
E-mail addresses: drmorr0@gmail.com (D.R. Morrison), shj@illinois.edu (S.H. Jacobson), sauppejj@gmail.com (J.J. Sauppe), esewell@siue.edu (E.C. Sewell).

http://dx.doi.org/10.1016/j.disopt.2016.01.005
1572-5286/

encapsulates a family of algorithms that all share a common core solution procedure. This procedure implicitly enumerates all possible solutions to the problem under consideration, by storing partial solutions called **subproblems** in a tree structure. Unexplored nodes in the tree generate children by partitioning the solution space into smaller regions that can be solved recursively (i.e., **branching**), and rules are used to prune off regions of the search space that are provably suboptimal (i.e., **bounding**). Once the entire tree has been explored, the best solution found in the search is returned. An early overview of the core B&B algorithm was provided by Lawler and Wood [2]; the solution procedure is also covered in the excellent texts by Nemhauser and Wolsey [3], Bertsimas and Tsitsiklis [4], and Papadimitriou and Steiglitz [5].

However, in the above framework there are three components which are left unspecified, but which can have significant impacts on the performance of the algorithm. These components are the **search strategy** (i.e., the order in which subproblems in the tree are explored), the **branching strategy** (i.e., how the solution space is partitioned to produce new subproblems in the tree), and the **pruning rules** (i.e., rules that prevent exploration of suboptimal regions of the tree). Clausen [6] gives an overview of these different components and how they affect algorithm performance for the Traveling Salesman Problem, the Graph Partitioning Problem, and the Quadratic Assignment Problem.

Substantial research has been done to develop both general-purpose and problem-specific algorithmic extensions to these three components that can improve the performance of B&B. The goal of this paper is therefore to provide a survey of modern advances in the theory of B&B algorithms, particularly with regards to the above three components. In addition, three important research directions are highlighted that are currently unresolved or unstudied in the literature. These research directions are:

(i) The formulation of new search strategies to drive a B&B algorithm towards an optimal solution quickly.
(ii) An analysis of how the branching strategy affects both the search and verification phase, along with a study of new branching strategies.
(iii) The development of a unified theory for pruning in B&B algorithms, which will enable better theoretical bounds to be proved on the performance of such algorithms.

The paper is organized as follows: Section 2 provides a formal description of the generic B&B algorithm, together with a discussion of how the three core components relate to each other. The subsequent three sections each focus in detail on one of these components; Section 3 describes commonly-used search strategies and their comparative advantages and disadvantages. Section 4 describes different options for branching strategies, and Section 5 discusses pruning rules and dominance relations. This section also discusses extensions to B&B such as branch-and-cut or branch-and-price, which both serve to improve the pruning rules used in the search procedure. Finally, Section 6 provides some concluding remarks.

2. Branch-and-bound

2.1. Algorithm overview

Define an optimization problem as $\mathcal{P} = (X, f)$, where X (called the **search space**) is a set of valid solutions to the problem, and $f : X \to \mathbb{R}$ is the **objective function**. The goal is to find an **optimal** solution $x^* \in \arg\min_{x \in X} f(x)$. To solve $\mathcal{P}$, B&B iteratively builds a **search tree** T of **subproblems**, or subsets of the search space. Additionally, a feasible solution $\hat{x} \in X$ (called the **incumbent solution**) is stored globally. At each iteration, the algorithm selects a new subproblem $S \subseteq X$ to explore from a list L of **unexplored** subproblems; if a solution $\hat{x}' \in S$ (called a **candidate incumbent**) can be found with a better objective value than $\hat{x}$ (i.e., $f(\hat{x}') < f(\hat{x})$), the incumbent solution is updated. On the other hand, if it can be proven that no solution in S has a better objective value than $\hat{x}$ (i.e., $\forall x \in S, f(x) \geq f(\hat{x})$), the

subproblem is **pruned** (or **fathomed**), and the subproblem is **terminal**. Otherwise, child subproblems are generated by partitioning S into an exhaustive (but not necessarily mutually exclusive) set of subproblems $S_1, S_2, \ldots, S_r$, which are then inserted into T. Once no unexplored subproblems remain, the best incumbent solution is returned; since subproblems are only fathomed if they contain no solution better than $\hat{x}$, it must be the case that $\hat{x} \in \arg\min_{x \in X} f(x)$. Pseudocode for the generic B&B procedure is given in Algorithm 1.

Algorithm 1: Branch-and-Bound(X, f)

```
1 Set L = {X} and initialize x̂
2 while L ≠ ∅:
3     Select a subproblem S from L to explore
4     if a solution x̂' ∈ {x ∈ S | f(x) < f(x̂)} can be found: Set x̂ = x̂'
5     if S cannot be pruned:
6         Partition S into S_1, S_2, ..., S_r
7         Insert S_1, S_2, ..., S_r into L
8     Remove S from L
9 Return x̂
```

With respect to this pseudocode, the search strategy affects the order in which nodes are selected for exploration in Line 3 of Algorithm 1; the branching strategy affects the number of children and the way the subproblem is partitioned (Line 6, Algorithm 1), and the pruning rules in Line 5 of Algorithm 1 determine whether or not S can be fathomed. Often, an initial incumbent solution can be found (Line 1, Algorithm 1) via a heuristic procedure (see, for example, [7]). Algorithm 1 is guaranteed to terminate when X is finite and the partitioning procedure creates child subproblems S_i that are proper subsets of S at each subproblem S.

Note that the set X (and all subproblems) is generally given implicitly; that is, given an element x, membership in a particular subproblem can be checked efficiently, and a partition of any subproblem can be computed efficiently without knowing all the members of X. Then, the complexity of B&B algorithms is related to two factors: the **branching factor** b of the tree, which is the maximum number of children generated at any node in the tree, and the **search depth** d of the tree, which is the length of the longest path from the root of T to a leaf. Thus, any B&B algorithm operates in $O(Mb^d)$ worst-case running time, where M is a bound on the length of time needed to explore a subproblem; however, the presence of pruning rules can substantially improve the algorithm performance.

2.2. Relationships between algorithm components

There are two important phases of any B&B algorithm: the first is the **search** phase, in which the algorithm has not yet found an optimal solution x^*. The second is the **verification** phase, in which the incumbent solution is optimal, but there are still unexplored subproblems in the tree that cannot be pruned. Note that an incumbent solution cannot be proven optimal until no unexplored subproblems remain; also note that the delineation between the search phase and the verification phase is not known until the algorithm terminates. In a slight abuse of terminology, a problem $\mathcal{P}$ is said to be **solved** if the B&B algorithm has completed the verification phase. In this case, the algorithm is said to have produced a **certificate of optimality**.

The three algorithmic components (search strategy, branching strategy, and pruning rules) each play a distinct role in B&B algorithms with respect to these two phases of operation (see Fig. 1). In particular, the choice of search strategy primarily impacts the search phase. To see this, suppose the pruning rules only depend on the value of the incumbent solution (e.g., they compare a subproblem's lower bound to the

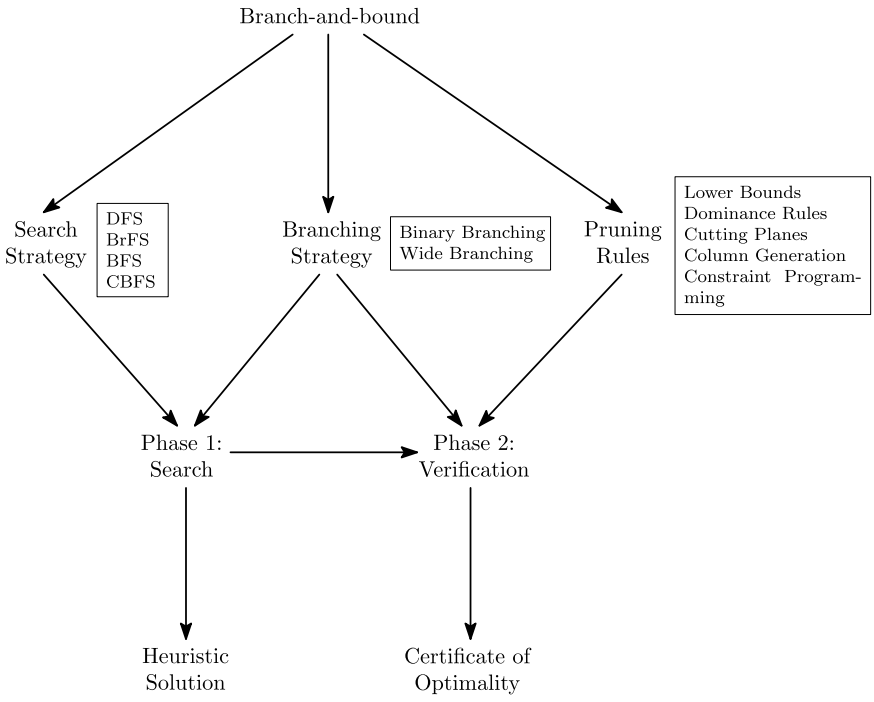


Fig. 1. A diagram of the three main B&B components. The search strategy and the pruning rules primarily impact the search phase and verification phase, respectively, whereas the branching strategy impacts both.

incumbent value). In this setting, any search strategy must explore the same set of subproblems once an optimal solution is found.

On the other hand, pruning rules are often targeted at the verification phase, particularly in the case of objective-based bounding which can be relatively weak before an optimal (or near-optimal) solution is known. In this case, if the incumbent solution has a poor objective value early in the search process the lower bounds will not be able to prune effectively, even if they are very tight. However, there are also situations in which pruning rules contribute to the search phase, such as when cutting planes in a mixed integer program (MIP) are used to identify feasible solutions.

The third B&B component, the branching strategy, has significant impacts on both the search phase and the verification phase. By branching appropriately at subproblems, the strategy can guide the algorithm towards optimal solutions. Once the search phase has concluded, an appropriate branching strategy can help limit the branching decisions that are made in order to prevent unnecessary work from being performed to produce a certificate of optimality.

There are two important reasons to improve performance of the B&B algorithm during the search phase. First, if the algorithm terminates before producing a certificate of optimality, the incumbent solution can still be returned as a heuristic solution, which may be sufficient in some problems. An example of this behavior can be seen in [8] where B&B is used to improve upper bounds for large simple assembly line balancing instances even if a certificate of optimality cannot be obtained. Another approach by Guzelsoy et al. [9] called **restrict-and-relax** allows B&B to relax previously-made branching decisions in order to more quickly find a good feasible solution.

Secondly, finding an optimal solution earlier in the search phase has a direct impact on the size of the search tree (and thus the time necessary to verify optimality), since no further nodes with bounds greater than the optimum value need to be explored. Intuitively, this is the rationale behind the result of Dechter and Pearl [10] showing that best-first search explores the fewest number of subproblems of any search strategy.

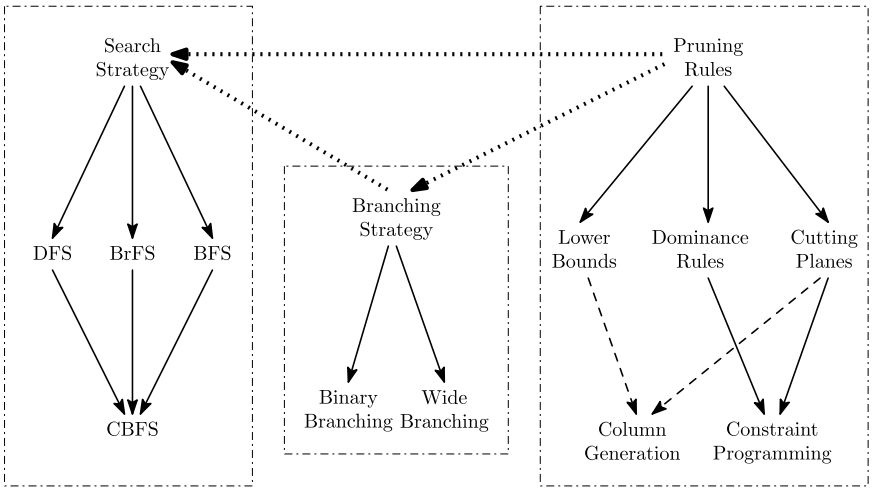


Fig. 2. A diagram of relationships between various algorithm components.

However, there are relatively few recent results in the literature studying the impacts of search strategy and branching strategy on the performance of B&B algorithms. Instead, most work focuses on pruning rules, which are the most useful during verification. The development of more advanced search strategies and branching strategies are therefore two important research directions highlighted by this survey.

Fig. 2 shows the internal relationships between the different types of pruning rules, branching strategies, and search strategies. In this figure, solid lines indicate a generalization relationship. For instance, as discussed in Section 5.5, many constraint programming techniques generalize cutting planes and dominance relations. Furthermore, the Cyclic Best-First Search (CBFS) strategy is a generalization of Depth-First Search (DFS), Breadth-First Search (BrFS), and Best-First Search (BFS) (see Section 3.4). Column generation techniques, while not strictly a generalization of other techniques, are closely connected to lower bounding and cutting plane techniques (in essence, column generation adds cutting planes to the dual optimization problem to improve the computed lower bound). This diagram also highlights the third important research direction presented in this survey, namely the lack of a strong generalizing theory of the many types of pruning rules used in B&B algorithms.

Fig. 2 also shows the relationships between the pruning rules, the branching strategy, and the search strategy used by an algorithm. In particular, the choice of pruning rules often impacts or limits the choices that can be made in the other two areas. For example, as discussed in Section 5.4, if column generation is used to improve lower bounds, the choice of branching strategies that can be used is limited. Moreover, if dominance relations are used, this may cause BrFS to become a desirable search strategy, since it has the property of never exploring a dominated subproblem. Finally, the choice of branching strategy can itself impact the choice of search strategy. For instance, if the branching strategy chosen produces a particularly unbalanced tree, the CBFS strategy can balance the search process, or variants of DFS can limit the depth explored at any stage in the algorithm.

2.3. Integer programming

It is worth mentioning one common specialization of the generic B&B procedure to integer programming problems. In this setting, the set X is described by a vector of integer variables y and a set of (linear) algebraic constraints given by $Ay \leq b$, where A is the **constraint matrix** and b is a set of constraint

bounds. The objective function $f(y)$ is defined as $c'y$, where c is a real-valued vector and $'$ denotes the transpose operator.

In this setting, bounds are commonly produced by solving the **LP relaxation** of the problem, where the integrality constraints on y are relaxed. Branching decisions are imposed by adding additional constraints to the problem to shrink the feasible region without removing any optimal integral solutions. For example, the standard integer branching rule selects a variable y_i with fractional value α in the LP relaxation and creates two new branches, one with $y_i \leq \lfloor \alpha \rfloor$, and one with $y_i \geq \lceil \alpha \rceil$. If no fractional variables y_i exist, a new candidate incumbent has been found.

Many different optimization problems can be formulated as integer programs, and the LP relaxation often provides tight bounds in practice. Thus, a number of B&B techniques have been developed specifically for this setting. Moreover, many very efficient software packages (both commercial and freeware) exist for solving integer programs using B&B techniques, including CPLEX [11], SYMPHONY [12], Gurobi [13], LINDO [14], SCIP [15], Xpress-MP [16], and CBC [17].

2.4. Assessing performance

As mentioned previously, B&B algorithms have a worst-case running time of $O(Mb^d)$, where b is the branching factor and d is the search depth, or height, of the tree. In practice, however, the performance of a B&B algorithm is often measured using a metric such as computational running time or the number of nodes explored. Such metrics allow for the comparison of various search strategies, branching strategies, and pruning rules, often by evaluating them on benchmark libraries (e.g., the MIPLIB 2010 library for mixed integer programming [18]). One difficulty with such assessments is that actual algorithm performance can be highly sensitive to initial conditions. For example, in mixed integer programming, permuting the rows of the constraint matrix should not affect performance in theory, but it often does in practice [19]. Similarly, starting the B&B procedure with an optimal incumbent solution should ensure that any search strategy explores the same set of nodes when the branching strategy is fixed and pruning is done using only the lower bounds; this is not always observed in practice, however. As such, any comparison of B&B components should take these variations into account [19].

While these sources of variability often serve as confounding factors when measuring performance, they can also be turned into an advantage. For example, Fischetti and Monaci [20] propose to exploit performance variability by running a B&B solver for a limited duration under many different initial conditions. The configuration that performs best is then selected to run to completion.

2.5. Extensions

The generic B&B procedure given in Algorithm 1 can be extended in many different ways. One such extension is the combination of various heuristic methods with the B&B framework. Heuristics can be used to find initial feasible solutions prior to starting the B&B search [21,22] and to improve existing incumbent solutions [23–25]. In other cases hybrid algorithms are developed which use a B&B search algorithm to solve subproblems within a general heuristic search framework [26,27].

Another notable extension to the generic B&B procedure is the application of parallel programming techniques. A standard approach is to parallelize the exploration of subproblems. Many new issues arise here, particularly with the choice of branching strategy, as well as the communication of bounds, dominance, and other pruning rules across different processors [28]. In contrast to this approach, Carvajal et al. [29] attempt to make use of parallelism by maintaining separate search trees across machines and sharing information such as bounds and cuts between them. Koch et al. [30] discuss some of the challenges and open questions regarding the parallelization of B&B algorithms.

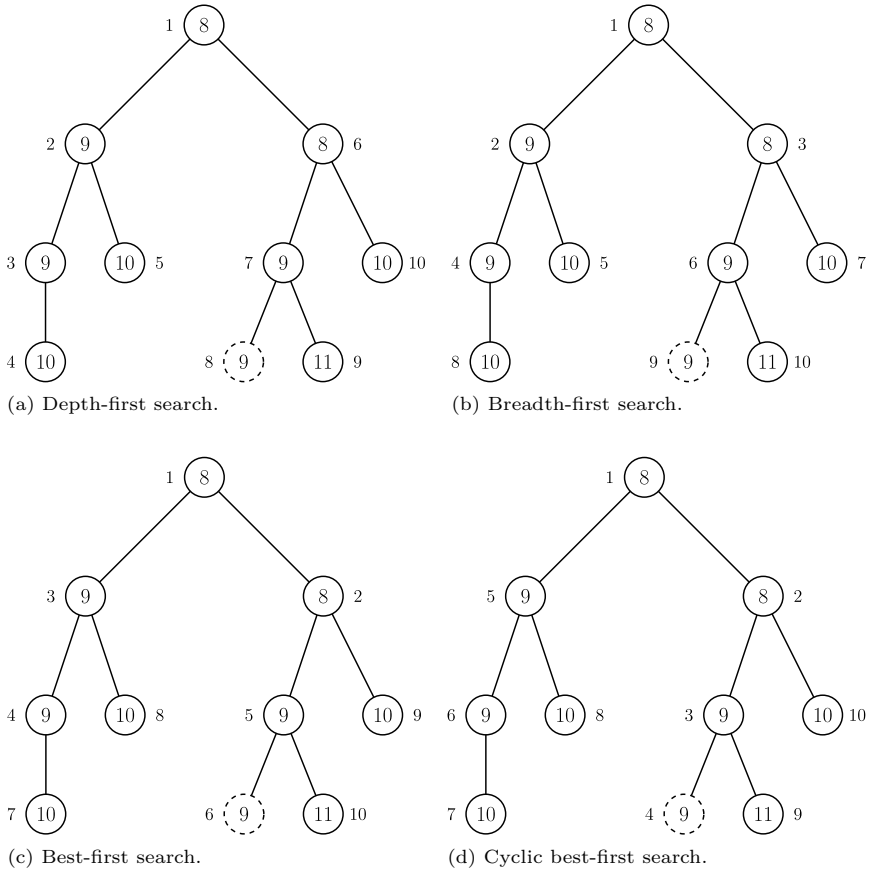


Fig. 3. Subproblem exploration order for different search strategies. The dashed subproblem is optimal, numbers inside nodes are subproblem lower bounds, and numbers outside the nodes indicate exploration order. The algorithm starts with an incumbent solution of value 10. BFS and CBFS use the lower bound as the measure-of-best, with ties broken arbitrarily. Note that this requires that a subproblem's lower bound is computed prior to inserting it into the list of unexplored subproblems.

3. Search strategies

The search strategy in a B&B algorithm determines the order in which unexplored subproblems in T are selected for exploration. The choice of search strategy has potentially significant consequences for the amount of computation time required for the B&B procedure, as well as the amount of memory used. In some cases, for very large or challenging problems, it may be necessary to choose a search strategy that requires low memory usage; however, for problems in which memory is not a concern, other search strategies exist which may find an optimal solution very quickly, and thus explore potentially fewer subproblems. A comparison of some search strategies is given in [31]. In this section, a discussion of common search strategies, along with their strengths and weaknesses is given. Fig. 3 shows a small search tree, and the order in which nodes are explored for several different search strategies.

3.1. Depth-first search

The **depth-first search** (DFS) strategy (sometimes called depth-first search with backtracking, or last-in, first-out search; see Section 11.4 in [3]) is a search strategy used in many different graph algorithms in addition to B&B [32,33]. It can be implemented by maintaining the list of unexplored subproblems L as a stack. The algorithm removes the top item from the stack to choose the next subproblem to explore, and

when children are generated as a result of branching, they are inserted on the top of L. Thus, the next subproblem that is explored is the most recently generated subproblem.

However, a slight modification to this algorithm can be made to produce a substantial savings in memory usage. In particular, if the children of a subproblem can be ordered in some way, then DFS does not need to store the entire list of unexplored subproblems (which can grow quite large) over the course of the algorithm. Instead, the search strategy only stores the path from the root of T to the current subproblem; at each subproblem along this path, it also stores the index of the last-explored child subproblem. At the current subproblem, the next unexplored child is selected for exploration. If no unexplored children remain, the algorithm **backtracks** to the closest ancestor node with unexplored children.

In addition to its low memory requirements, another advantage of DFS arises when solving integer programming problems that use the LP relaxations as lower bounds. Since many branching decisions do not significantly change the structure of the LP relaxation between parent and child nodes, the LP solver can often reuse information from the parent LP solution as a starting point for the child LP solution. Reusing the optimal basis and its LU factorization when solving the child LP is known as **hot starting**, while reusing only the optimal basis is known as **warm starting** [34,35]. When DFS proceeds from a parent to a child, it can employ hot starting because both the optimal basis and LU factorization are readily available; other search strategies would need to store the appropriate information in memory in order to make use of these techniques. Because storing factorizations is memory-intensive, hot starting is typically not used with other search strategies.

Two problems arise with the use of the DFS strategy. The first problem is that naïve implementations of DFS do not use any information about problem structure or bounds, which means the search process can spend large amounts of exploration time in poor regions of the search space. A related phenomenon, called **thrashing**, occurs when different regions of the search space all fail for the same or similar reasons [36]. For instance, perhaps the presence of a single branching constraint always leads to infeasibility, but the algorithm must explore many more subproblems before the infeasibility is detected.

A different problem arises when the search tree is extremely unbalanced. In other words, if some optimal solutions are close to the root, but there exist long paths in T that do not lead to an optimal solution, DFS can (unluckily) choose many long, bad paths before it explores a path leading to an optimal solution. However, this computation time often could be avoided via pruning rules if the search strategy instead chose to explore a short optimal path first. In fact, this behavior of DFS was first noticed on problems where the search tree had unbounded depth [37], but the same problem exists in trees with a few extremely long paths.

A host of variants to the depth-first search strategy exist that attempt to overcome these limitations. One common variant is the **iterative deepening** DFS algorithm [38], which imposes a limit on the depth of subproblems explored by DFS; if the search process is not able to prove optimality using this depth limit, the depth is increased and the search is restarted from the root. This ensures that the search does not spend unneeded time exploring extremely long paths in the search tree while retaining the low memory overhead of regular DFS; the disadvantage is that subproblems near the root of the tree are explored multiple times.

Another algorithm called **interleaved depth-first search** by Meseguer [39] tries to overcome thrashing-type behavior by performing depth-first search from multiple locations in the search tree at once. This strategy can be performed by sequentially selecting exactly one subproblem to explore from each different DFS path in the search tree before returning to the first search path. This algorithm can improve performance over standard DFS with a relatively limited increase in memory usage (a single stack needs to be maintained for each search path).

A third variant of DFS is **depth-first search with complete branching**, which tries to exploit problem structure by selecting the next child subproblem to explore as the one with the best computed lower bound [40]. This method explores the search tree more intelligently, at the expense of increased memory

usage, since all child subproblems must be generated when a subproblem is explored. However, if the tree has a relatively small branching factor, this increased memory usage is not likely to be significant.

3.2. Breadth-first search

Breadth-first search (BrFS) is the opposite of DFS in that it is implemented with a first-in, first-out, or queue, data structure. BrFS explores all subproblems that are at a fixed distance from the root before exploring any deeper subproblems. The BrFS strategy has the advantage of always finding an optimal solution that is closest to the root of the tree, thus operating well on unbalanced search trees. However, since complete solutions are usually at larger depths, BrFS is generally unable to exploit pruning rules that compare against the current incumbent solution. For this reason, the memory requirements for BrFS are quite high, and it is generally not used in a B&B context. Two exceptions are in the presence of dominance relations, which can prune effectively even in the absence of a good incumbent solution, and if a good incumbent solution can be found effectively by some other means, for example with a good heuristic search [41]. It is also worth noting that in the absence of pruning rules, the iterative deepening DFS strategy explores the same sequence of nodes as BrFS with substantially lower memory requirements.

3.3. Best-first search

In settings where sufficient memory is available to store the entire unexplored search tree, the **best-first search** (BFS) strategy is often used. This strategy makes use of a heuristic **measure-of-best** function $\mu : 2^X \to \mathbb{R}$, which computes a value $\mu(S)$ for every unexplored subproblem, and selects as the next subproblem to explore the one minimizing μ. If $\mu(S) \leq \min_{x \in S} f(x)$ (that is, the measure-of-best function never overestimates the best solution in a subproblem), the measure-of-best function is **admissible**. In the presence of an admissible μ, BFS is also called the A* algorithm [10], or sometimes **best-bound** search. BFS can easily be implemented by storing the list of subproblems in a heap data structure, using the value of μ as the key [42].

There are many choices for the measure-of-best function; one common choice is a lower bound on the value of the best solution in the subproblem. If the lower bound is strongly correlated with the subproblem objective values, this measure-of-best will encourage exploration of subproblems with better solutions. However, in practice, lower bounds may not be a good proxy for the objective function value. For example, in integer programming problems which use the LP relaxation as a lower bound, a small lower bound may just indicate that the structure of the problem allows the LP to "cheat" in ways that the IP cannot. To overcome this, other candidate measure-of-best functions are heuristics which estimate the quality of a solution, such as in [41]. Additionally, Shi and Ólafsson [43] use a probabilistic function called the **promising index** to estimate the quality of a solution, and commercial solvers such as CPLEX use a heuristic function to estimate the objective value of a particular subproblem.

Best-first search offers a number of significant advantages over DFS; because it is not tied to exploring one specific branch of the tree before any other, it is often able to find good solutions earlier in the search process. In fact, this notion has been formalized in a theorem by Dechter and Pearl [10] that states that, assuming an admissible μ with no ties and no dominance relations, BFS explores the fewest number of subproblems of any search strategy that has access to the same heuristic function and other information.

However, as observed in [41], BFS does still have one potential drawback—if there exist many subproblems in T for which $\mu(S) = f(x^*)$, depending on the tie-breaking rule used, BFS may spend much time in middle regions of the search tree and never explore an optimal solution (see Fig. 3(c), in which nodes 3, 4, and 5 are explored before the optimal solution is found). In this situation, BFS may be slower than some other

strategy. To overcome this, many BFS implementations employ **diving** heuristics or other heuristic methods to drive a subproblem towards a new incumbent solution that can aid in pruning [44,45].

3.4. Cyclic best-first search

A more recent search strategy that has been used successfully in a number of scheduling applications is called **cyclic best-first search**. Originally called distributed best-first search [46], the strategy can be thought of as a hybrid algorithm between DFS and BFS. While BFS is implemented by using a single heap data structure to store all unexplored subproblems, CBFS divides the unexplored subproblems over a collection of heaps, referred to as **contours**. When a new subproblem is identified, it is inserted into one of the contours according to a set of rules (e.g., all subproblems at depth d get stored in a contour associated with depth d). To explore the search space, the CBFS strategy repeatedly iterates through all of the non-empty contours, selecting the best subproblem (according to a measure-of-best function μ) from each contour before moving on to the next one. For example, if contours are organized by depth, CBFS will explore the best subproblem at depth 0, then depth 1, and so on; upon reaching the bottom of the search tree, it will repeat the process starting from depth 0.

By separating subproblems into contours, the measure-of-best function effectively assigns to each subproblem a local ranking within a single contour instead of a global ranking. Thus, contours can be used to group subproblems that are more directly **comparable**. Cycling ensures that each contour is visited frequently, which serves to diversify the search and can aid in generating incumbent solutions (e.g., depth-based contours ensure that a leaf of the search tree is explored every cycle).

Other papers in the literature have used cyclic behavior to guide branching algorithms. For instance, in [43], a randomized branching heuristic algorithm called **nested partitioning** is presented, which shares many similarities with an early-terminating B&B algorithm. One variant of this algorithm uses a cycling mechanism that is similar to that of CBFS. Moreover, Choi et al. [47] propose an algorithm called cyclic best-first search which uses a cycling mechanism to perform some intelligent backtracking in a heuristic algorithm. While this algorithm shares a name with the CBFS strategy described herein, it has different behavior.

Two important results have been shown about the CBFS strategy by Morrison et al. [48]: first, the number of subproblems for which CBFS generates children is at most $K|\text{BFS}|$, where $|\text{BFS}|$ is the number of subproblems explored by BFS using the same μ, branching strategy, and pruning rules. Intuitively, this is because on every pass through the non-empty contours, CBFS explores at least one subproblem that BFS would also explore. The second result establishes the generality of CBFS—it says that for any other search strategy $\mathcal{A}$, there exists a set of contours that allows CBFS to emulate $\mathcal{A}$, in that CBFS explores the same sequence of subproblems as $\mathcal{A}$. The CBFS strategy can be implemented using a red–black tree and a collection of heaps.

First research direction

The first research direction highlighted by this survey asks how the search strategy can be used to improve performance of B&B algorithms. In particular, this research direction asks the question, “How can the search strategy be tailored for different problems?” There are a number of proposed approaches to answering this question:

One approach might be to use different contour definitions for CBFS that accelerate the search towards optimal solutions. While CBFS has produced some remarkable results in a few different settings, the strategy is not well-understood in general. Another possible approach involves the measure-of-best function; many

B&B algorithms using BFS or CBFS use an admissible μ. However, are there are any potential gains that can be made using a non-admissible measure-of-best, or a probabilistic μ (as in [43] or [49])?

A third possible approach to this problem would use machine learning techniques in conjunction with B&B to learn new search strategies. In particular, is it possible for an algorithm to learn a search strategy that produces optimal solutions quickly? There are a number of interesting challenges to this approach, including how such algorithms can be trained and how to avoid over-fitting the search strategy to a particular problem or problem instance.

4. Branching strategies

The choice of branching strategy determines how children are generated from a subproblem. Branching strategies can be categorized into two groups: **binary** branching strategies and non-binary, or **wide**, branching strategies. Additionally, due to the prevalence of integer programming problems, there is a plethora of literature devoted to branching strategies in integer programming. These will be discussed separately at the end of the section.

4.1. Binary branching

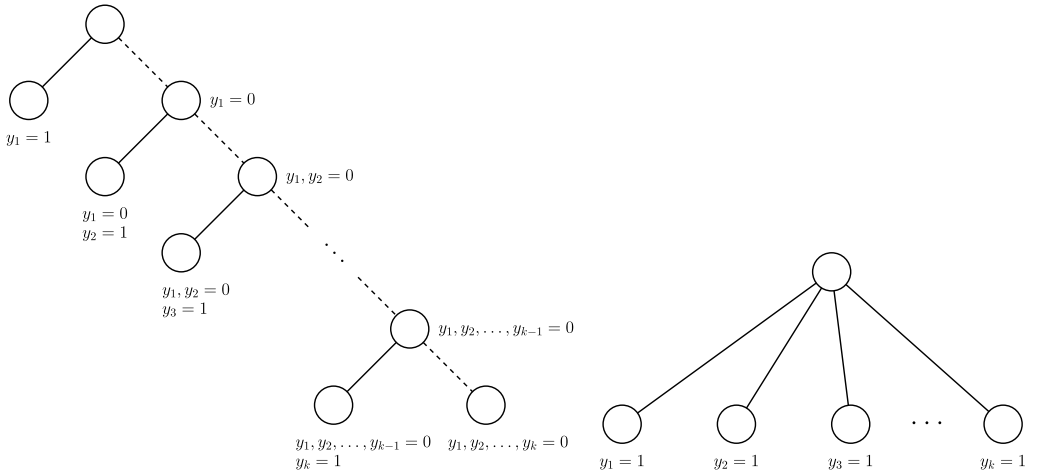
Binary branching strategies focus on subdividing a subproblem S into two mutually-exclusive, smaller subproblems. For example, in the knapsack problem, which seeks a maximum-cost selection of items to fit inside a storage bin with fixed capacity, a binary branching strategy for a subproblem S selects some unassigned item and creates two branches, one in which the item is included in the knapsack, and one in which the item is excluded from the knapsack [50]. Most binary branching strategies are variants of this idea. The integer branching scheme for integer programming, described in Section 2.3, is another binary branching scheme.

In some cases, the mechanism for performing the partitioning is more complicated. In the graph coloring branch-and-price solver of Mehrotra and Trick [51], branching is performed by either adding edges or contracting vertices of the graph to force a pair of non-adjacent vertices to either share the same color or use different colors. Similarly, for the branch-and-price solver for the generalized assignment problem [52], in which a series of tasks is assigned to a group of workers to maximize profit, branching is performed by either including or excluding all schedules that assign a particular task to a worker.

4.2. Wide branching

In contrast to binary branching are wide branching strategies (the term "wide branching" was first introduced in [53], though previous algorithms have used non-binary branching strategies). Wide branching focuses on selecting one element from a set of different options. For example, in B&B algorithms to compute maximum cliques or independent sets in a graph, a set of unused vertices is maintained for each subproblem, and each unused vertex generates a child with that vertex added to the child's set [54,55]. Wide branching methods allow for potentially large reductions in the size of the search tree. In the above example, a binary branching strategy would have to consider each unused vertex individually, creating a long sequence of subproblems. This long sequence can be bypassed with the wide branching technique (see Fig. 4).

One important setting in which a wide branching strategy has been used successfully is with **special ordered sets** (SOS), introduced by Beale and Tomlin [56] and Beale and Forrest [57]. There are two types of special ordered sets, denoted by SOS1 and SOS2. An SOS1 is a set of elements for which at most one element can be used in a solution, and an SOS2 is a set of elements for which at most two adjacent elements can be used in a solution. SOS2 are useful for modeling piecewise linear approximations of nonlinear



(a) Binary branching. (b) Wide branching.

Fig. 4. Binary branching versus wide branching. Given a set of k elements from which one must be selected, binary branching must explicitly reject elements $1, 2, \ldots, i-1$ before creating a branch that considers element i. Conversely, wide branching can consider each of them immediately.

optimization problems [58,59]. Wide branching strategies can be used in B&B to handle problems with SOS variables; for example, when an SOS1 is selected for branching, the strategy creates one branch for each element in the set. The subproblem for this set uses the chosen element and excludes all others. Finally, the branching strategy creates one subproblem which uses no elements from the set. This strategy can be generalized to handle SOS2, as well.

Two potential problems arise with wide branching strategies. The first is that such strategies usually do not create mutually-exclusive branches, so it is possible to arrive at the same subproblem from several different paths. It is generally possible to work around this problem using a lexicographic ordering rule [60], dominance rules (discussed in Section 5.2), or nogood recording (discussed in Section 5.5). A second problem arises if the number of branches that can be created at a particular subproblem is very large. In this case, the algorithm could get stuck generating children at a particular subproblem and never move on to explore new regions of the search space. Moreover, if the number of generated children is very large at every subproblem, the size of the search tree will grow much more quickly. There are two potential ways around this problem; the first sets an arbitrary cap on the number of children that can be generated at a subproblem. If the branching factor ever exceeds this limit, any additional children are just discarded. This technique, used in the simple assembly line balancing solver of Sewell and Jacobson [41], can prove optimality if the branching factor is never exceeded; otherwise, it performs as a heuristic. The other method uses a **delayed branching** technique, where after a certain number of children have been generated, the strategy delays generation of the remaining children in the hopes that when it returns to the node, better bounds may have been computed that allow it to prune children more effectively. This method was used in a branch-and-price solver for graph coloring by Morrison et al. [53].

4.3. Branching in integer programs

Given that many optimization problems can be modeled using integer programming, substantial effort has been devoted to branching strategies for integer programming problems. The branching strategy is generally divided into two phases: selecting a variable or set of variables to branch on, and creating child subproblems by imposing bounds on these variables to force them away from fractional values. The choice of branching

variable can significantly impact the performance of the algorithm, and many different techniques exist to choose good branching variables. An overview of variable selection strategies for MIP is presented here; more details can be found in [61].

One commonly-used, easy-to-implement rule is called the **most fractional** or **most infeasible** rule, which selects the variable y_i whose fractional part is closest to 0.5 as the branching variable. However, as shown by Achterberg et al. [61], this branching rule is in general no better than selecting a branching variable at random in terms of computational time required and number of subproblems explored. Therefore, a number of more advanced techniques have been proposed in the literature to improve the performance of B&B integer programming solvers. The opposite branching rule, the **least fractional** rule selects y_i such that its fractional part is furthest from 0.5; this rule is less commonly used and is often outperformed by other methods (see, for example, [62]).

Another strategy for variable selection in MIP is **pseudocost branching** [63], which attempts to predict the per-unit change in the objective function for each candidate branching variable, based on past experience in the tree. By branching on a variable that is expected to produce a significant change in the objective function, it is more likely that the generated subproblems can be pruned. One difficulty with pseudocost branching is that no information about past branching behavior is available at the beginning of the algorithm, so the pseudocosts for each variable must be initialized in some way. One strategy is to set the pseudocost for each variable to the value of its coefficient in the objective function, as these values would be the actual pseudocosts if all variables were independent.

Strong branching is another method of variable selection that branches on the variable that induces the most change in the objective function [61,64]. This strategy was first introduced by Applegate et al. [65] in the context of solving the traveling salesman problem, where it was one of several improvements that allowed the authors to solve 20 previously unsolved instances from TSPLIB [66]. Strong branching computes the LP relaxation objective value of the children of a subproblem S for a subset of the candidate branching variables, and then selects for branching the variable that induces the most change in the objective. **Full strong branching** considers all candidate branching variables before selecting one. This approach often leads to small search trees, but the time required to perform variable selection is often prohibitive [61].

Linderoth and Savelsbergh [67] proposed to combine strong branching and pseudocost branching by applying strong branching only to variables with uninitialized pseudocosts. In this way, they are able to overcome the difficulty of pseudocost initialization as well as avoid the prohibitive computational time that would be required to apply strong branching at every subproblem. A related method called **hybrid strong/pseudocost branching** employs strong branching in the upper levels of the search tree and pseudocosts in the lower levels where more branching history is available.

Subsequent work by Achterberg et al. [61] introduced the method of **reliability branching**, which is a further refinement of strong branching with pseudocosts. Specifically, reliability branching applies strong branching to variables with uninitialized pseudocosts as well as variables whose pseudocosts have been deemed unreliable—that is, for which there is not enough historical information in the branching process to construct pseudocost estimates that are believed to be reliable. Achterberg et al. [61] also provide a classification of branching strategies involving strong branching and pseudocost branching that demonstrates how the various approaches are related through their parameters.

Recent research by Pryor and Chinneck [68] has explored the use of branching rules that try to find feasible integer solutions to the problem quickly. They achieve this by branching on variables that induce change in the largest number of *variables* in the problem (as opposed to the largest change in objective value, as with pseudocost branching). The somewhat surprising result in their paper shows that branching on variables with the smallest probability of satisfying constraints in the LP often leads to integer feasibility more quickly, because this will require a large number of other variables in the LP to change to satisfy the constraints.

Another recent branching method developed by Fischetti and Monaci [69] is called **backdoor branching**. This technique solves an auxiliary integer program to determine a small set of variables that should be branched on before any others; this auxiliary program is a set covering problem which computes a **backdoor**—that is, a set of variables which, if branched upon early in the search process, yield a small search tree.

Finally, Gilpin and Sandholm [70] use information-theoretic results to guide the search process by branching so as to remove **uncertainty** from subproblems in the search tree. Subproblems close to the root in the search tree have a large amount of uncertainty, since few variables have been fixed; terminal subproblems have no uncertainty, since all variables have assumed integer values. To do this, they treat the values of fractional variables as probabilities, and compute the **entropy** (i.e., the amount of uncertainty) for each candidate branching variable, selecting the one with the least entropy to branch upon.

Second research direction

The second research direction highlighted by this survey focuses on the development of better branching strategies for B&B algorithms. An important question here is: "How should the B&B algorithm branch in order to generate the smallest number of unhelpful subproblems?" In this context, an unhelpful subproblem is a subproblem that does not lead to any optimal or near-optimal solutions. Such subproblems are problematic because they do not provide any gain for the algorithm but may still require substantial work to explore and prune.

However, by considering a different partitioning scheme for the branching strategy, it may be possible to avoid exploring some of these unhelpful subproblems. Therefore, it is recommended to study how different branching strategies affect performance times and number of nodes explored for a particular B&B algorithm. Finally, it may be beneficial to explore "hybrid" binary-wide branching strategies, which employ wide branching high in the search tree and switch to binary branching in lower regions, or vice versa.

As in the first research direction, machine learning techniques may also be beneficial here. For instance, Karzan et al. [71] demonstrate how better branching rules can be obtained for binary integer programming problems by incorporating a learning phase to identify sets of branching decisions that lead to fathomed nodes, and use the learned clauses to then choose branching variables in a restart phase of the algorithm.

5. Pruning and dominance rules

A critical aspect of B&B search is the choice of pruning rules used to exclude regions of the search space from exploration. Note that for a fixed branching strategy, any node that cannot be pruned by the pruning rules must be explored by *any* search strategy, even if an optimal solution is known before the search begins. The only way to reduce the size of the search tree in this case is to use better pruning rules. There are many different classes of pruning rules, but they are usually problem-specific and must be derived anew for each problem type under consideration. Again because of its prevalence, many pruning rules are focused on integer programming problems.

5.1. Lower bounds

The most common way to prune is to produce a lower bound on the objective function value at each subproblem, and use this to prune subproblems whose lower bound is no better than the incumbent's solution value. Lower bounds are computed by relaxing various aspects of the problem. For example, in the simple assembly line balancing problem and its variants, in which tasks must be scheduled to a group of capacity-limited machines so as to satisfy certain precedence requirements, one common relaxation is to compute the

optimal solution value ignoring the precedence constraints [72,8]. In general, as many different lower bounds can be computed as necessary; some lower bound computations may be easy to perform, whereas others may be more computationally intensive. Thus a common practice is to attempt to prune using the easy lower bounds first, and then move on to the more complex, but tighter, lower bounds if the easy methods are unsuccessful.

If the problem can be formulated as an integer program, the optimal value of the LP relaxation is an extremely common lower bound choice. The quality of the LP relaxation value is measured by the **integrality gap** of the formulation, that is, the ratio between the best integer solution and the best LP relaxation value across all problem instances. However, there may be many different ways to formulate the problem using integer programming, and some of these problems may have tighter integrality gaps than others. Thus, one technique for improving lower bounds is to derive a new formulation with a tighter integrality gap [73]. The branch-and-cut and branch-and-price algorithms described in Sections 5.3 and 5.4 are common methods for exploiting integer programming formulations with tighter bounds. A related approach for polynomial programming problems called the **reformulation–linearization technique** (RLT) transforms a mathematical program with polynomial objective function and constraints into a linear program, and uses the resulting LP bound to prune in B&B algorithm to find global optimal solutions to the polynomial program [74].

Another method for deriving lower bounds on integer programming problems is through duality. Though there is no strong duality theorem for integer programming, one can still arrive at a notion of weak duality. Given an integer program $\min\{f(y) \mid Ay \leq b, y \in \mathbb{Z}\}$, the **Lagrangian relaxation** problem is $\mathcal{P}(\lambda) = \min\{f(y) + \lambda(b - Ay) \mid y \in \mathbb{Z}\}$, where λ is a non-positive vector of real-valued weights called **Lagrange multipliers**. The optimal solution value for the Lagrangian relaxation is always bounded above by the value of the optimal solution to the original problem. Thus, the best bound possible may be computed as the solution to the **Lagrangian dual** problem, $\max_{\lambda \leq 0} \mathcal{P}(\lambda)$. The Lagrangian dual problem can be solved using **subgradient optimization**, a modification of Newton's method for piecewise linear concave functions [4]. Integer programming duality methods have been used in [72,75–77], among others.

5.2. Dominance relations

In contrast to lower bounding rules, dominance relations allow subproblems to be pruned if they can be shown to be **dominated** by some other subproblem—in other words, if subproblem S_1 dominates subproblem S_2, this means that for any solution that is a descendant of S_2, there exists a complete solution descending from S_1 that is at least as good. Thus, it suffices to just explore S_1. Dominance relations, first studied by Kohler and Steiglitz [78], are closely related to the Bellman equations from dynamic programming [79]. Note that, as shown by Ibaraki [80], it is not always true that using dominance relations will improve the quality of the search process; however, there are many cases in which dominance relations will improve the search.

There are two primary types of dominance relations, memory-based and non-memory-based. Memory-based dominance rules compare unexplored subproblems to other problems previously generated and stored in the tree [81]. As the name implies, memory-based dominance rules require the entire search tree to be stored for the duration of the algorithm, instead of just the unexplored subproblems. However, this may allow for additional pruning to be performed that would be otherwise impossible.

Non-memory-based dominance relations do not require the dominating state to have been previously generated in the search process—instead, non-memory-based dominance rules are able to imply the *existence* of a dominating subproblem, regardless of whether it has been explored or generated. Such rules have the advantage that they do not require additional memory to store the generated search tree, but they may not be able to prune the same subset of problems that memory-based dominance rules can. Fischetti and Salvagnin [82] make use of non-memory-based dominance rules in a B&B solver for generic mixed integer

programming problems. By solving an auxiliary problem, their solver is able to identify whether a node in the tree is dominated by some other node (which need not have been explored yet).

In B&B algorithms that employ dominance, the BrFS strategy has the useful property that it never explores a dominated subproblem, as long as the dominance relations are formulated in such a way so that subproblems are only compared if they are within the same level of T (see, for example, [83,41]).

Note that care must be taken when implementing dominance rules to avoid mutual dominance relations. In particular, depending on the structure of the dominance rules employed, cycles of dominating subproblems $S_1, S_2, \ldots, S_k$ could exist where S_{i+1} dominates S_i, and S_1 dominates S_k [84]. In such cases, at least one subproblem in the dominance cycle must not be pruned; often, this can be accomplished using some lexicographic ordering rule.

Dominance relations play an important role in B&B algorithms for solving integer programs with a high degree of symmetry. In such problems, a particular solution may have many equivalent representations that appear as separate subproblems within the search tree (e.g., in graph coloring, the labels of color classes can be permuted without changing the solution). If the B&B algorithm fails to recognize this symmetry, it may end up performing redundant work that could otherwise be avoided. To address this difficulty, Margot [85,86] introduces **isomorphic pruning**, which uses lexicographic inequality tests at each node in the search tree to determine if the node can be pruned. Ostrowski et al. [87] propose **orbital branching**, which identifies equivalent variables at a particular node in the search tree using the group-theoretic concept of an **orbit** and constructs two branches, one with an arbitrary variable in the orbit fixed at one, and another with all variables fixed at zero. Thus, this strategy prunes off child nodes that correspond to fixing each of the other variables in the orbit to one.

5.3. Cutting planes

The discussion in this section is restricted to problems that can be formulated as (mixed) integer programs. A significant advance in the theory of linear and integer programming was developed by Gomory [88], who introduced the idea of **cutting planes**. A cutting plane is a constraint that can be added to an integer program to tighten the feasible region without removing any integer solutions. This fundamental idea was applied to B&B by Padberg and Rinaldi [89] to develop an algorithm called **branch-and-cut**. In this algorithm, new cutting planes (sometimes called **valid inequalities**) are added to the LP relaxation at every subproblem in the search tree (note that a valid inequality is a global constraint—it must apply at the LP relaxation of the root subproblem). The algorithm of Padberg and Rinaldi [89] is specific to the well-known traveling salesman problem, but in [90] a generalization of branch-and-cut for binary integer programs is presented.

There are a number of different types of valid inequalities for general mixed integer programs; surveys can be found in [91,92]. The initial cutting planes described by Gomory are called **Gomory (mixed integer) cuts**, and are based on the structure of the simplex tableau. These cuts were shown to be of both practical and theoretical interest by Balas et al. [93]. Chvátal [94] considered these cutting planes from a geometric perspective and showed that a finite number of such cuts can be added to a pure integer program to yield the convex hull of the integer feasible solutions. Subsequent work focused on extending Chvátal's cuts to mixed integer programs. Cook et al. [95] introduced **split cuts** to do this, while Nemhauser and Wolsey [96] introduced **mixed integer rounding (MIR) cuts**; the two approaches are equivalent [96].

Lovász and Schrijver [97] and Balas et al. [98] introduced **lift-and-project** cuts for 0–1 mixed integer programming. Lift-and-project operates by lifting the LP relaxation into a higher-dimensional space by adding additional variables, finding valid inequalities in this higher-dimensional space, and then projecting the valid inequalities back into the original space by deleting the extra variables. Balas and Perregaard [99]

demonstrated the relationship between lift-and-project cuts, intersection cuts [100], and Gomory mixed integer cuts.

Cutting planes can also be generated by exploiting local structure in a mixed integer program. One well-known set of inequalities are **cover inequalities**, which are derived from viewing individual constraints in the MIP as separate knapsack problems [101,102]. Other examples of structural cuts include **flow cover inequalities** [103] and **weight inequalities** [104]. Marchand et al. [92] provide more examples of cuts derived from problem structure.

Marchand [101] and Marchand and Wolsey [105] show that many cuts based on the structural properties of a MIP can be derived as MIR cuts using only the initial set of constraints in the MIP formulation. These results are utilized by Marchand and Wolsey [105] in a heuristic procedure for generating cuts for arbitrary mixed integer programs. For a fractional solution to the LP relaxation, the heuristic separation procedure repeatedly aggregates a subset of constraints from the original formulation to form a single constraint, applies bound information to generate a mixed knapsack set, and then searches for (complemented) MIR inequalities that are violated.

Cutting planes for MIP have led to significant improvements in solver performance over the years. Early work by Crowder et al. [106] demonstrated how cover inequalities could be used to solve large, sparse 0–1 integer programs in a reasonable amount of computing time. Balas et al. [93] showed the effectiveness of Gomory mixed integer cuts in a branch-and-cut framework for 0–1 mixed integer programs; as a result, Gomory cuts are included in all modern MIP solvers today. Computational results by Balas and Saxena [107] have demonstrated the effectiveness of split cuts in reducing the integrality gap through experiments with the split closure on MIPLIB instances; separating such cuts in practice remains computationally difficult but progress is being made [108]. A computational study of the various cutting plane methods available in CPLEX 12.5 can be found in [109]. The results from those experiments indicate that MIR cuts are the most effective, followed by Gomory cuts.

Another method of generating valid inequalities is through decomposition methods such as **Benders' decomposition** [110,111]. A decomposition method splits apart a problem into a **master problem** and one or more **slave problems** (sometimes referred to as "subproblems", but this terminology is avoided herein to avoid confusion with the search tree subproblems). For example, in Benders' decomposition, a set of **complicating variables** in the integer program are identified which drive the intractability of the problem. The non-complicating variables are then projected out of the integer program. The master problem therefore seeks a solution to the new integer program, and the slave problem (which is assumed to be easy to solve) either determines that the master problem is feasible for the original integer program or produces a constraint that it violates (an algorithm to solve such a slave problem is also known as a **separation oracle**, because it separates feasible solutions from infeasible ones). These **Benders' cuts** can then be added in to the master problem. Hernández-Pérez and Salazar-González [112] give an example of using Benders' cuts in a branch-and-cut context to solve the traveling salesman problem with both pickups and deliveries. The idea of decomposition will be even more important in the next section, which discusses branch-and-price.

One interesting question with regards to this method of pruning involves the interplay between cutting plane generation and branching. In many cases, the set of generated cuts is too large to allow all of them to be added, and it is often computationally expensive to generate new cuts, so at some point cutting planes are no longer generated and branching occurs. However, the question of when to stop generating cutting planes and start branching is an important problem when implementing a branch-and-cut algorithm [113,114].

5.4. Column generation

Another very common method for solving difficult integer programs is known as branch-and-price; in a sense, branch-and-price can be thought of as the dual algorithm to branch-and-cut. Here, instead of adding

new constraints to the (primal) master problem, a separation oracle for the dual of the master problem is used to add new constraints to the dual. This procedure is known as **column generation**, since new constraints in the dual master problem correspond to new variables (or constraint matrix columns) in the primal master problem. The column generation approach was first described in [115], together with a decomposition method called **Dantzig–Wolfe decomposition**. Detailed descriptions of branch-and-price algorithms are given in [116,117].

The Dantzig–Wolfe decomposition method identifies **coupling constraints** in the original integer program that are "complicating" (in the same sense as for Benders' decomposition) and retains them to form the master problem. One or more slave problems, or **pricing problems**, then identify variables with the potential to improve the value of the master problem objective function and adds them to the set of variables in the master problem; this is accomplished using a separation oracle for the dual of the master problem to identify variables with negative reduced cost. The coupling constraints link together the variables identified by the pricing problems, and a new constraint is introduced to enforce that solutions to the master problem are a convex combination of variables introduced by the slave problem. However, unlike Benders' decomposition, the master problem in these settings is typically easier to solve, whereas the pricing problem is usually NP-hard.

Moreover, additional complexity is added when attempting to incorporate column generation with a branching strategy, because typical branching rules usually interfere with the structure of the pricing problem. In other words, once some branching decisions have been fixed, new negative-cost variables must be found that respect the branching decisions. This is often related to the kth-**shortest-path** problem, which is NP-hard [118]. However, despite the apparent difficulty of incorporating column generation with a branching algorithm, the Dantzig–Wolfe decomposition procedure can often substantially tighten the lower bounds produced in the tree, while at the same time reducing problem symmetry leading to thrashing behavior, and thus branch-and-price has been empirically shown to produce substantial computational gains in practice.

To avoid interfering with the pricing problem structure, most branch-and-price algorithms use alternative branching strategies that do not disrupt the structure of the pricing problem. The branching strategy for graph coloring [51] or for the generalized assignment problem [52] (see Section 4) are two such examples. Another general-purpose branching strategy for branch-and-price algorithms involves branching on the original (non-decomposed) problem variables [119]. Morrison et al. [53] branch on the master problem variables directly, but use a wide branching strategy to limit the number of branching decisions that interfere with the pricing problem structure.

Due to the intractability of the pricing problem in many IP models, significant research has also gone into ways to solve the pricing problem. Gualandi and Malucelli [120] use constraint programming techniques (Section 5.5) to more efficiently solve the pricing problem in a graph coloring branch-and-price solver, and Easton et al. [121] use a similar approach for a sports timetabling problem. Morrison et al. [122] use a data structure called a zero-suppressed binary decision diagram, or ZDD, to characterize the pricing problem and generate solutions quickly. Their scheme has the added advantage that traditional branching strategies can be used without slowing down the pricing problem procedure.

Finally, a new field of research is emerging in **branch-and-cut-and-price** algorithms, which use separation oracles to produce new constraints for both the primal and dual master problems. These methods suffer from many of the same problems as branch-and-price algorithms, since now the pricing problem must respect both the branching decisions and the additional cutting planes added to the problem. However, de Aragão and Uchoa [123] developed a new method called **robust branch-and-cut-and-price** which further reformulates the master problem to eliminate the interference of cuts and branching decisions with the pricing problem. This method has been used with success in a number of vehicle routing and other graph problems [124,125].

5.5. Constraint programming techniques

Recently, interest has increased in using constraint programming techniques for solving optimization problems. Constraint programming is a subfield of artificial intelligence that has been very successful in solving logic problems such as SAT. Techniques from constraint programming have many potential applications to B&B algorithms, as well as other applications in the broader field of optimization (for a complete discussion of the field of constraint programming and applications in AI and OR, see [126]). A comparison of operations research and constraint programming techniques along with a discussion of their application to the fixed-charge network flow problem can be found in [127].

Two primary ideas behind many constraint programming techniques for B&B algorithms are **constraint propagation** and **nogood learning** [128]. Constraint propagation, also known as **node presolving** in the mixed integer programming community [129], exploits the repeated application of logical inference rules in an attempt to derive contradictions that allow a subproblem to be pruned. On the other hand, a nogood is a structural property of the problem that has been proven to not lead to a feasible or optimal solution by complete exploration of some subtrees. Nogoods are generally learned over the course of the algorithm, and enable the search to check the validity of subproblems under consideration. The process of learning nogoods is referred to as **conflict analysis**. Marques-Silva and Sakallah [130] present a SAT solver that makes use of this technique. Constraint programming techniques in B&B share many commonalities with other pruning techniques such as cutting planes and dominance relations. However, as pointed out by Caseau and Laburthe [131], one significant difference is that constraint programming techniques are usually local techniques that tighten bounds at a particular subproblem, unlike the global pruning rules introduced by dominance or valid inequalities.

An example of constraint propagation rules is given in [132]; in this paper, a technique called **domain filtering** is used to solve the maximum clique problem in a B&B context. Here, two results are proven showing when it is impossible for vertices in a graph to participate in a maximum clique, and these results are iteratively applied (or propagated) at each subproblem in T. If the propagation reduces this list of candidate vertices to the empty set, the subproblem is fathomed.

Constraint propagation techniques often have close relations to lower bounding techniques. Fahle [132] show that their domain filtering rule subsumes seven out of eight common lower bounds for the maximum clique problem. Moreover, Li et al. [133] use a similar constraint propagation rule to aid in the computation of lower bounds that can be used for pruning in a Max-SAT solver (the Max-SAT problem seeks an assignment that satisfies the maximum number of clauses in a Boolean formula).

Conversely, nogoods are more closely related to dominance relations and cutting planes. For instance, Sandholm and Shields [134] learn a sequence of nogoods (in this case, invalid assignments to sets of variables) that can be added as cuts to the integer program. These nogoods are derived by constraint propagation based on the branching decisions. Achterberg [135] makes use of conflict analysis to derive valid constraints which are used for node presolving in a branch-and-cut solver for mixed integer programming. Fischetti and Salvagnin [82] introduce a variant of nogoods called **pruning moves** in their generic mixed integer programming solver.

Third research direction

There is often a significant discrepancy between the performance of B&B algorithms in practice and the theoretical guarantees that can be proven about performance. As observed in Section 2.1, the worst-case performance of B&B is $O(Mr^d)$, where r is the branching factor, d is the depth of the tree, and M is a bound on the time needed to explore a subproblem. Pruning rules are the primary way that this complexity is reduced in practice. However, most pruning rules are somewhat ad hoc and problem specific. The desire

for better bounds on the behavior of B&B is well-known, and is expressed poignantly in the following quote by George Nemhauser:

> "I have always wanted to prove a lower bound about the behavior of branch and bound, but I never could". (in a personal communication with [136])

Therefore, the third research direction proposed by this survey is the development of a unified theory for pruning in branch-and-bound, with the goal of improving theoretical bounds on algorithm performance.

This research direction is motivated by the observation that in many cases, the distinguishing lines between the types of pruning rules mentioned are not very strict. For example, lower bound rules can be viewed as a weak form of dominance relations by observing that subproblems leading to incumbent solutions dominate any subproblems with a worse lower bound. Furthermore, many constraint programming techniques are closely related to dominance relations or cutting planes. The creation of a unifying pruning theory would provide a generic way to capture these connections.

Such a unification theory would potentially provide three significant advantages in the theory of B&B algorithms. First, insights gained from this theory may drive more advanced pruning rules that can prune search trees more effectively in the future. Secondly, such a theory would provide a problem-agnostic way to talk about pruning, and may generate a generic class of pruning rules that could be applied across a wide range of settings. Thirdly, a unification theory may enable tighter theoretical bounds to be proved about the performance of B&B, and thus close the gap between the $O(Mr^d)$ asymptotic analysis and the observed behavior in practice.

6. Conclusion

In this survey of the branch-and-bound framework, a comprehensive study of the current state-of-the-art for each of three different algorithm components is presented, with the goal of acting as a starting point for future research that is conducted in these areas. These components are the search strategy, the branching strategy, and the pruning rules. Since it is unlikely to have any one combination of these parameters that works uniformly for all possible problems, it is valuable to understand the choices that are available for these components, along with their strengths and limitations.

Furthermore, three major research questions have been posed as a starting point for researchers: the first is concerned with the development of new search strategies to drive the search towards an optimal solution more quickly; the second focuses on the study of branching strategies that limit the generation of unimportant subproblems; and the third proposes the creation of a unified theory for pruning rules to provide a generic framework for pruning in branch-and-bound.

Acknowledgments

This research has been supported in part by the Air Force Office of Scientific Research (FA9550-10-1-0387), the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (32 CFR 168a), and the National Science Foundation Graduate Research Fellowship Program (DGE-1144245). Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, the United States Air Force, the Department of Defense, or the United States Government. The majority of this work was completed while the first and third authors were Ph.D. students at the University of Illinois at Urbana–Champaign.

References

[1] A.H. Land, A.G. Doig, An automatic method for solving discrete programming problems, Econometrica (1960) 497–520.
[2] E.L. Lawler, D.E. Wood, Branch-and-bound methods: A survey, Oper. Res. 14 (1966) 699–719.
[3] G. Nemhauser, L. Wolsey, Integer and Combinatorial Optimization. Vol. 18, Wiley, New York, 1988.
[4] D. Bertsimas, J.N. Tsitsiklis, Introduction to Linear Optimization, Athena Scientific, 1997.
[5] C.H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Courier Dover Publications, 1998.
[6] J. Clausen, Branch and bound algorithms-principles and examples. Technical Report, Department of Computer Science, University of Copenhagen, 1999.
[7] E. Malaguti, M. Monaci, P. Toth, An exact approach for the vertex coloring problem, Discrete Optim. 8 (2011) 174–190.
[8] D.R. Morrison, E.C. Sewell, S.H. Jacobson, An application of the branch, bound, and remember algorithm to a new simple assembly line balancing dataset, European J. Oper. Res. (2013) in press.
[9] M. Guzelsoy, G. Nemhauser, M. Savelsbergh, Restrict-and-relax search for 0–1 mixed-integer programs, EURO J. Comput. Optim. 1 (2013) 201–218.
[10] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of A*, J. ACM 32 (1985) 505–536.
[11] IBM Corp., IBM ILOG CPLEX Optimization Studio V12.5, 2014.
[12] L. Ladanyi, T. Ralphs, M. Guzelsoy, A. Mahajan, SYMPHONY 5.5.0, 2014. URL: https://projects.coin-or.org/SYMPHONY.
[13] Gurobi Optimization, Inc., 2014. Gurobi Optimizer 5.6.
[14] LINDO Systems, Inc., 2014. LINDO API 8.0.
[15] Konrad-Zuse-Zentrum für Informationstechnik Berlin. SCIP Optimization Suite 3.0.1, 2014. URL: http://scip.zib.de/scip.shtml.
[16] Fair Isaac Corporation (FICO), 2014. Xpress Optimization Suite.
[17] COIN-OR, 2014. COIN-OR Branch and Cut. https://projects.coin-or.org/Cbc.
[18] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. Bixby, E. Danna, G. Gamrath, A. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. Steffy, K. Wolter, MIPLIB 2010: Mixed integer programming library version 5, Math. Program. Comput. 3 (2011) 103–163.
[19] E. Danna, Performance variability in mixed integer programming, in: Workshop on Mixed Integer Programming, 2008.
[20] M. Fischetti, M. Monaci, Exploiting erraticism in search, Oper. Res. 62 (2014) 114–122.
[21] M. Fischetti, F. Glover, A. Lodi, The feasibility pump, Math. Program. Ser. A 104 (2005) 91–104.
[22] L. Bertacco, M. Fischetti, A. Lodi, A feasibility pump heuristic for general mixed-integer problems, Discrete Optim. 4 (2007) 63–76.
[23] M. Fischetti, A. Lodi, Local branching, Math. Program. 98 (2003) 23–47.
[24] E. Danna, E. Rothberg, C. Le Pape, Exploring relaxation induced neighborhoods to improve MIP solutions, Math. Program. 102 (2005) 71–90.
[25] E. Danna, C. Le Pape, Branch-and-price heuristics: A case study on the vehicle routing problem with time windows, in: G. Desaulniers, J. Desrosiers, M.M. Solomon (Eds.), Column Generation, 2005, pp. 99–129.
[26] A.P. French, A.C. Robinson, J.M. Wilson, Using a hybrid genetic-Algorithm/Branch and bound approach to solve feasibility and optimization integer programming problems, J. Heuristics 7 (2001) 551–564.
[27] K. Büdenbender, T. Grünert, H. Sebastian, A hybrid tabu Search/Branch-and-Bound algorithm for the direct flight network design problem, Transportation Sci. 34 (2000) 364–380.
[28] B. Gendron, T.G. Crainic, Parallel Branch-and-Branch Algorithms: Survey and Synthesis, 1994.
[29] R. Carvajal, S. Ahmed, G. Nemhauser, K. Furman, V. Goel, Y. Shao, Using diversification, communication and parallelism to solve mixed-integer linear programs, Oper. Res. Lett. 42 (2014) 186–189.
[30] T. Koch, T. Ralphs, Y. Shinano, Could we use a million cores to solve an integer program? Math. Methods Oper. Res. 76 (2012) 67–93.
[31] T. Ibaraki, Theoretical comparisons of search strategies in branch-and-bound algorithms, Int. J. Comput. Inf. Sci. 5 (1976) 315–344.
[32] S.W. Golomb, L.D. Baumert, Backtrack programming, J. ACM 12 (1965) 516–524.
[33] R. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (1972) 146–160.
[34] A. Atamtürk, M.W.P. Savelsbergh, Integer-programming software systems, Ann. Oper. Res. 140 (2005) 67–124.
[35] J. Chinneck, Practical optimization: A gentle introduction. 2015. http://www.sce.carleton.ca/faculty/chinneck/po.html.
[36] V. Kumar, Algorithms for constraint-satisfaction problems: A survey, AI Mag. 13 (1992) 32.
[37] D.J. Slate, L.R. Atkin, CHESS 4.5—The Northwestern University chess program, in: P.W. Frey (Ed.), Chess Skill in Man and Machine, Springer, New York, 1983, pp. 82–118.
[38] R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, Artificial Intelligence 27 (1985) 97–109.
[39] P. Meseguer, Interleaved depth-first search, in: IJCAI, 1997, pp. 1382–1387.
[40] A. Scholl, R. Klein, Balancing assembly lines effectively—a computational comparison, European J. Oper. Res. 114 (1999) 50–58.
[41] E.C. Sewell, S.H. Jacobson, A branch, bound, and remember algorithm for the simple assembly line balancing problem, INFORMS J. Comput. 24 (2012) 433–442.
[42] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, third ed., MIT Press, 2009.
[43] L. Shi, S. Ólafsson, Nested partitions method for global optimization, Oper. Res. 48 (2000) 390–407.

[44] R.E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, R. Wunderling, MIP: theory and practice—closing the gap, in: M.J.D. Powell, S. Scholtes (Eds.), System Modelling and Optimization, in: IFIP—The International Federation for Information Processing, vol. 46, Springer US, 2000, pp. 19–49.
[45] T. Achterberg, T. Berthold, T. Koch, K. Wolter, Constraint integer programming: A new approach to integrate CP and MIP, in: L. Perron, M.A. Trick (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, in: Lect. Notes Comput. Sci., vol. 5015, Springer, Berlin, Heidelberg, 2008, pp. 6–20.
[46] G.K. Kao, E.C. Sewell, S.H. Jacobson, A branch, bound, and remember algorithm for the $1 \mid r_i \mid \sum t_i$ scheduling problem, J. Sched. 12 (2009) 163–175.
[47] S. Choi, M. Shin, J. Cha, Loss reduction in distribution networks using cyclic best first search, in: M.L. Gavrilova, O. Gervasi, V. Kumar, C.J.K. Tan, D. Taniar, A. Laganá, Y. Mun, H. Choo (Eds.), Computational Science and Its Applications—ICCSA 2006, in: Lecture Notes in Computer Science, vol. 3984, Springer, Berlin, Heidelberg, 2006, pp. 312–321.
[48] D.R. Morrison, J.J. Sauppe, E.C. Sewell, S.H. Jacobson, Cyclic best-first search: Using contours to guide branch-and-bound algorithms. Technical Report, University of Illinois, Urbana-Champaign, 2014.
[49] M. Dür, V. Stix, Probabilistic subproblem selection in branch-and-bound algorithms, J. Comput. Appl. Math. 182 (2005) 67–80.
[50] P.J. Kolesar, A branch and bound algorithm for the knapsack problem, Manage. Sci. 13 (1967) 723–735.
[51] A. Mehrotra, M.A. Trick, A column generation approach for graph coloring, INFORMS J. Comput. 8 (1996) 344–354.
[52] M. Savelsbergh, A branch-and-price algorithm for the generalized assignment problem, Oper. Res. 45 (1997) 831–841.
[53] D.R. Morrison, J.J. Sauppe, E.C. Sewell, S.H. Jacobson, A wide branching strategy for the graph coloring problem, INFORMS J. Comput. 26 (2014) 704–717.
[54] L. Babel, A fast algorithm for the maximum weight clique problem, Computing 52 (1994) 31–38.
[55] S. Held, W. Cook, E.C. Sewell, Maximum-weight stable sets and safe lower bounds for graph coloring, Math. Program. Comput. 4 (2012) 363–381.
[56] E.M.L. Beale, J.A. Tomlin, Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables, in: Fifth Annual International Conference on Operational Research, Tavistock Publications, 1970, pp. 447–454.
[57] E.M.L. Beale, J.J.H. Forrest, Global optimization using special ordered sets, Math. Program. 10 (1976) 52–69.
[58] I. de Farias Jr., E. Johnson, G. Nemhauser, A generalized assignment problem with special ordered sets: a polyhedral approach, Math. Program. 89 (2000) 187–203.
[59] C. D'Ambrosio, A. Lodi, Mixed integer nonlinear programming tools: a practical overview, 4OR 9 (2011) 329–349.
[60] A.M. Geoffrion, An improved implicit enumeration approach for integer programming, Oper. Res. 17 (1969) 437–454.
[61] T. Achterberg, T. Koch, A. Martin, Branching rules revisited, Oper. Res. Lett. 33 (2005) 42–54.
[62] F. Ortega, L.A. Wolsey, A branch-and-cut algorithm for the single-commodity, uncapacitated, fixed-charge network flow problem, Networks 41 (2003) 143–158.
[63] M. Benichou, J.M. Gauthier, P. Girodet, G. Hentges, G. Ribiere, O. Vincent, Experiments in mixed-integer linear programming, Math. Program. 1 (1971) 76–94.
[64] T. Achterberg, Constraint integer programming (Ph.D. thesis), 2007.
[65] D. Applegate, R.E. Bixby, V. Chvátal, W. Cook, Finding cuts in the TSP. Technical Report 95-05, DIMACS, 1995.
[66] G. Reinelt, TSPLIB—a traveling salesman problem library, ORSA J. Comput. 3 (1991) 376–384.
[67] J.T. Linderoth, M.W.P. Savelsbergh, A computational study of search strategies for mixed integer programming, INFORMS J. Comput. 11 (1999) 173.
[68] J. Pryor, J.W. Chinneck, Faster integer-feasibility in mixed-integer linear programs by branching to force change, Comput. Oper. Res. 38 (2011) 1143–1152.
[69] M. Fischetti, M. Monaci, Backdoor branching, in: O. Günlük, G.J. Woeginger (Eds.), Integer Programming and Combinatoral Optimization, in: Lecture Notes in Computer Science, vol. 6655, Springer, Berlin, Heidelberg, 2011, pp. 183–191.
[70] A. Gilpin, T. Sandholm, Information-theoretic approaches to branching in search, Discrete Optim. 8 (2011) 147–159.
[71] F.K. Karzan, G.L. Nemhauser, M.W.P. Savelsbergh, Information-based branching schemes for binary linear mixed integer problems, Math. Program. Comput. 1 (2009) 249–293.
[72] M. Vilà, J. Pereira, A branch-and-bound algorithm for assembly line worker assignment and balancing problems, Comput. Oper. Res. 44 (2014) 105–114.
[73] S. Arora, B. Bollobas, L. Lovász, Proving integrality gaps without knowing the linear program, in: The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings, 2002, pp. 313–322.
[74] H.D. Sherali, C.H. Tuncbilek, A global optimization algorithm for polynomial programming problems using a reformulation-linearization technique, J. Global Optim. 2 (1992) 101–112.
[75] J. Desrosiers, R. Jans, Y. Adulyasak, Improved column generation algorithms for the job grouping problem. Technical Report G-2013-26, Les Cahiers du GERAD, 2013.
[76] B. Gendron, P. Khuong, F. Semet, A Lagrangian-Based Branch-and-Bound Algorithm for the Two-Level Uncapacitated Facility Location Problem with Single-Assignment Constraints. Technical Report CIRRELT-2013-21, CIRRELT, 2013.
[77] D.T. Phan, Lagrangian duality and branch-and-bound algorithms for optimal power flow, Oper. Res. 60 (2012) 275–285.
[78] W.H. Kohler, K. Steiglitz, Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems, J. ACM 21 (1974) 140–156.
[79] R. Bellman, The theory of dynamic programming, Bull. Amer. Math. Soc. 60 (1954) 503–515.
[80] T. Ibaraki, The power of dominance relations in branch-and-bound algorithms, J. ACM 24 (1977) 264–279.

[81] E.C. Sewell, J.J. Sauppe, D.R. Morrison, S.H. Jacobson, G. Kao, A BB&R algorithm for minimizing total tardiness on a single machine with sequence dependent setup times, J. Global Optim. 54 (2012) 791–812.
[82] M. Fischetti, D. Salvagnin, Pruning moves, INFORMS J. Comput. 22 (2010) 108–119.
[83] T. Nazareth, S. Verma, S. Bhattacharya, A. Bagchi, The multiple resource constrained project scheduling problem: A breadth-first approach, European J. Oper. Res. 112 (1999) 347–366.
[84] E. Demeulemeester, B.D. Reyck, W. Herroelen, The discrete time/resource trade-off problem in project networks: a branch-and-bound approach, IIE Trans. 32 (2000) 1059–1069.
[85] F. Margot, Pruning by isomorphism in branch-and-cut, Math. Program. Ser. B 94 (2002) 71–90.
[86] F. Margot, Exploiting orbits in symmetric ilp, Math. Program. 98 (2003) 3–21.
[87] J. Ostrowski, J. Linderoth, F. Rossi, S. Smriglio, Orbital branching, Math. Program. 126 (2011) 147–178.
[88] R.E. Gomory, Outline of an algorithm for integer solutions to linear programs, Bull. Amer. Math. Soc. 64 (1958) 275–278.
[89] M. Padberg, G. Rinaldi, A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems, SIAM Rev. 33 (1991) 60–100.
[90] E. Balas, S. Ceria, G. Cornuéjols, Mixed 0–1 programming by lift-and-project in a branch-and-cut framework, Manage. Sci. 42 (1996) 1229–1246.
[91] G. Cornuéjols, Valid inequalities for mixed integer linear programs, Math. Program. 112 (2008) 3–44.
[92] H. Marchand, A. Martin, R. Weismantel, L. Wolsey, Cutting planes in integer and mixed integer programming, Discrete Appl. Math. 123 (2002) 397–446.
[93] E. Balas, S. Ceria, G. Cornuéjols, N. Natraj, Gomory cuts revisited, Oper. Res. Lett. 19 (1996) 1–9.
[94] V. Chvátal, Edmonds polytopes and a hierarchy of combinatorial problems, Discrete Math. 4 (1973) 305–337.
[95] W. Cook, R. Kannan, A. Schrijver, Chvátal closures for mixed integer programming problems, Math. Program. 47 (1990) 155–174.
[96] G. Nemhauser, L. Wolsey, A recursive procedure to generate all cuts for 0–1 mixed integer programs, Math. Program. 46 (1990) 379–390.
[97] L. Lovász, A. Schrijver, Cones of matrices and set-functions and 0–1 optimization, SIAM J. Optim. 1 (1991) 166–190.
[98] E. Balas, S. Ceria, G. Cornuéjols, A lift-and-project cutting plane algorithm for mixed 0–1 programs, Math. Program. 58 (1993) 295–324.
[99] E. Balas, M. Perregaard, A precise correspondence between lift-and-project cuts, simple disjunctive cuts, and mixed integer Gomory cuts for 0–1 programming, Math. Program. 94 (2003) 221–245.
[100] E. Balas, Intersection cuts-a new type of cutting planes for integer programming, Oper. Res. 19 (1971) 19–39.
[101] H. Marchand, A polyhedral study of the mixed knapsack set and its use to solve mixed integer programs (Ph.D. thesis), Faculté des Sciences Appliquées, Université catholique de Louvain, 1998.
[102] A. Atamtürk, Cover and pack inequalities for (mixed) integer programming, Ann. Oper. Res. 139 (2005) 21–38.
[103] M. Padberg, T.J. Van Roy, L.A. Wolsey, Valid linear inequalities for fixed charge problems, Oper. Res. 33 (1985) 842–861.
[104] A. Martin, R. Weismantel, Contributions to general mixed integer knapsack problems. Technical Report, Konrad-Zuse-Zentrum für Informationstechnik, 1997.
[105] H. Marchand, L. Wolsey, Aggregation and mixed integer rounding to solve MIPs, Oper. Res. 49 (2001) 363–371.
[106] H. Crowder, E. Johnson, M. Padberg, Solving large-scale zero–one linear programming problems, Oper. Res. 31 (1983) 803–834.
[107] E. Balas, A. Saxena, Optimizing over the split closure, Math. Program. 113 (2008) 219–240.
[108] M. Fischetti, D. Salvagnin, Approximating the split closure, INFORMS J. Comput. 25 (2013) 808–819.
[109] T. Achterberg, R. Wunderling, Mixed integer programming: Analyzing 12 years of progress, in: M. Jünger, G. Reinelt (Eds.), Facets of Combinatorial Optimization—Festschrift for Martin Grötschel, Springer, 2013, pp. 449–481.
[110] J.F. Benders, Partitioning procedures for solving mixed-variables programming problems, Numer. Math. 4 (1962) 238–252.
[111] A.M. Geoffrion, Generalized benders decomposition, J. Optim. Theory Appl. 10 (1972) 237–260.
[112] H. Hernández-Pérez, J. Salazar-González, A branch-and-cut algorithm for a traveling salesman problem with pickup and delivery, Discrete Appl. Math. 145 (2004) 126–139.
[113] M. Jünger, G. Reinelt, S. Thienel, Practical problem solving with cutting plane algorithms in combinatorial optimization, in: W. Cook, L. Lovász, P. Seymour (Eds.), Combinatorial Optimization, in: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 20, 1995, pp. 111–152.
[114] J.E. Mitchell, Branch-and-cut algorithms for combinatorial optimization problems, in: Handbook Appl. Optim., 2002, pp. 65–77.
[115] G.B. Dantzig, P. Wolfe, Decomposition principle for linear programs, Oper. Res. 8 (1960) 101–111.
[116] C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, P.H. Vance, Branch-and-price: Column generation for solving huge integer programs, Oper. Res. 46 (1998) 316–329.
[117] M.E. Lübbecke, J. Desrosiers, Selected topics in column generation, Oper. Res. 53 (2005) 1007–1023.
[118] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, first ed., W. H. Freeman, 1979.
[119] F. Vanderbeck, Branching in branch-and-price: a generic scheme, Math. Program. 130 (2011) 249–294.
[120] S. Gualandi, F. Malucelli, Exact solution of graph coloring problems via constraint programming and column generation, INFORMS J. Comput. 24 (2012) 81–100.
[121] K. Easton, G. Nemhauser, M. Trick, Solving the travelling tournament problem: A combined integer programming and constraint programming approach, in: E. Burke, P.D. Causmaecker (Eds.), Practice and Theory of Automated Timetabling IV, in: Lect. Notes Comput. Sci., vol. 2740, Springer, Berlin, Heidelberg, 2003, pp. 100–109.
[122] D.R. Morrison, E.C. Sewell, S.H. Jacobson, Solving the pricing problem in a generic branch-and-price algorithm using zero-suppressed binary decision diagrams, 2014. arXiv:1401.5820 [cs.DS].

[123] M.P. de Aragão, E. Uchoa, Integer program reformulation for robust branch-and-cut-and-price algorithms, in: Mathematical Program in Rio: a Conference in Honour of Nelson Maculan, 2003, pp. 56–61.
[124] R. Fukasawa, H. Longo, J. Lysgaard, M.P.D. Aragão, M. Reis, E. Uchoa, R.F. Werneck, Robust branch-and-cut-and-price for the capacitated vehicle routing problem, Math. Program. 106 (2006) 491–511.
[125] E. Uchoa, R. Fukasawa, J. Lysgaard, A. Pessoa, M.P. de Aragão, D. Andrade, Robust branch-cut-and-price for the capacitated minimum spanning tree problem over a large extended formulation, Math. Program. 112 (2008) 443–472.
[126] F. Rossi, P.V. Beek, T. Walsh, Handbook of Constraint Programming, first ed., Elsevier Science, 2006.
[127] H. Kim, J.N. Hooker, Solving fixed-charge network flow problems with a hybrid optimization and constraint programming approach, Ann. Oper. Res. 115 (2002) 95–124.
[128] P. van Beek, Backtracking search algorithms, in: F. Rossi, P. van Beek, T. Walsh (Eds.), Foundations of Artificial Intelligence, in: Handbook of Constraint Programming, vol. 2, Elsevier, 2006, pp. 85–134 (Chapter 4).
[129] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, D. Weninger, Progress in presolving for mixed integer programming. Technical Report 13-48, Zuse Institute, Berlin, 2013.
[130] J.P. Marques-Silva, K.A. Sakallah, GRASP: A search algorithm for propositional satisfiability, IEEE Trans. Comput. 48 (1999) 506–521.
[131] Y. Caseau, F. Laburthe, Improving branch and bound for jobshop scheduling with constraint propagation, in: M. Deza, R. Euler, I. Manoussakis (Eds.), Combinatorics and Computer Science, in: Lecture Notes in Computer Science, vol. 1120, Springer, Berlin, Heidelberg, 1996, pp. 129–149.
[132] T. Fahle, Simple and fast: Improving a branch-and-bound algorithm for maximum clique, in: R. Möhring, R. Raman (Eds.), Algorithms—ESA 2002, in: Lect. Notes Comput. Sci., vol. 2461, Springer, Berlin, Heidelberg, 2002, pp. 485–498.
[133] C.M. Li, F. Manyà, J. Planes, Exploiting unit propagation to compute lower bounds in branch and bound max-SAT solvers, in: P.V. Beek (Ed.), Principles and Practice of Constraint Programming—CP 2005, in: Lecture Notes in Computer Science, vol. 3709, Springer, Berlin, Heidelberg, 2005, pp. 403–414.
[134] T. Sandholm, R. Shields, Nogood learning for mixed integer programming, in: Workshop on Hybrid Methods and Branching Rules in Combinatorial Optimization, Montréal, 2006.
[135] T. Achterberg, Conflict analysis in mixed integer programming, Discrete Optim. 4 (2007) 4–20.
[136] R.J. Lipton, K. Regan, Branch and bound—why does it work? 2012. URL: http://rjlipton.wordpress.com/2012/12/19/branch-and-bound-why-does-it-work/.