1

Introduction and Concepts

The core part of this book, Chapters 2 to 5, constitutes a comprehensive overview of different classes of MIP heuristics and describes individual heuristics in detail. In Chapter 6, we present a computational study that analyzes the impact of primal heuristics from different angles and presents two recent and innovative lines of research that extend and develop the field in promising new directions: primal heuristics for mixed-integer nonlinear programming (Chapter 7) and machine learning for primal heuristics (Chapter 8).

Before exploring these topics, this chapter introduces the main concepts and notation that we will use throughout the book. In Section 1.1, we formally introduce mixed-integer programs, while in Section 1.2 we give an overview of the different types of complete algorithms used for solving them, from the ubiquitous LP-based branch and bound to (less mainstream) primal methods. Then, in Section 1.3, we provide an overview of how primal heuristics are used inside MIP solvers: what their (typical) impact on the solving process is, and how they are scheduled within the branch-and-bound search. Then, we give a first categorization of the different types of primal heuristics inside a modern MIP solver. In Section 1.4, we introduce bound tightening and constraint propagation, essential building blocks in the design of (many) primal heuristics. In Section 1.5, we review the general concepts that are used when designing a heuristic algorithm for (general) combinatorial optimization problems, as those are relevant for the specific MIP case as well. Finally, we introduce additional concepts that are used when analyzing primal heuristics for MIP (Section 1.6) and for measuring their impact on the solving process (Section 1.7), most notably the primal integral.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

1.1 Notation

Every maths book needs a section that is an accumulation of definitions. So here is ours, introducing some core concepts of computational MIP.

Definition 1.1 (MIP problem) Let $m, n \in \mathbb{Z}_{\geqslant 0}$. Given a matrix $A \in \mathbb{R}^{m \times n}$, a right-hand-side vector $b \in \mathbb{R}^m$, an objective function vector $c \in \mathbb{R}^n$, a lower and an upper bound vector $l \in (\mathbb{R} \cup \{-\infty\})^n$, $u \in (\mathbb{R} \cup \{+\infty\})^n$, and a subset $\mathcal{I} \subseteq \mathcal{N} = \{1, \ldots, n\}$, the corresponding (linear) MIP problem is given by

$$\begin{aligned} \min \quad & c^\mathsf{T} x \\ \text{s.t.} \quad & Ax \leqslant b, \\ & l_j \leqslant x_j \leqslant u_j, \quad \text{for all } j \in \mathcal{N}, \\ & x_j \in \mathbb{R}, \quad \text{for all } j \in \mathcal{N} \setminus \mathcal{I}, \\ & x_j \in \mathbb{Z}, \quad \text{for all } j \in \mathcal{I}. \end{aligned} \tag{1.1}$$

Note that the format given in Definition 1.1 is very general. First, maximization problems can be transformed into minimization problems by multiplying all objective function coefficients by -1. Similarly, "$\geqslant$" constraints can be multiplied by -1 to obtain "$\leqslant$" constraints. Equations can be replaced by two opposite inequalities. We assume, without loss of generality, that $l_j \leqslant u_j$ for all $j \in \mathcal{N}$ and $l_j, u_j \in \mathbb{Z}$ for all $j \in \mathcal{I}$.

If $\mathcal{I} = \emptyset$, problem (1.1) is called a *linear programming* (LP) problem.

For a given MIP P, the set

$$\tilde{\mathcal{X}}(P) := \{x \in \mathbb{R}^n \mid Ax \leqslant b, \quad x \in [l, u], x_j \in \mathbb{Z} \text{ for all } j \in \mathcal{I}\},$$

is called the set of *feasible solutions* of the MIP. Let $c^\star \in \mathbb{R} \cup \pm\infty$ with

$$c^\star := \inf\{c^\mathsf{T} x \mid x \in \tilde{\mathcal{X}}\}.$$

If $c^\star = -\infty$, we call P *unbounded*; if $c^\star = +\infty$, we call it *infeasible*. If $c^\star$ is finite, we call it the *optimal solution value* of P. A solution $x \in \tilde{\mathcal{X}}$ is called an *optimal solution* if and only if $c^\mathsf{T} x = c^\star$. If $c = 0$, we call P a *feasibility problem*.

One of the most effective techniques in MIP is the use of relaxations to provide proven lower bounds on the optimal solution value of a given problem instance. The LP relaxation of a MIP arises by omitting the integrality constraints.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

Definition 1.2 (LP relaxation) Given a MIP P of the form (1.1), the LP

$$\begin{aligned} \min \quad & c^\mathsf{T} x \\ \text{s.t.} \quad & Ax \leqslant b, \\ & l_j \leqslant x_j \leqslant u_j, \quad \text{for all } j \in \mathcal{N}, \\ & x_j \in \mathbb{R}, \quad \text{for all } j \in \mathcal{N}, \end{aligned} \tag{1.2}$$

is called the *LP relaxation of P*.

The feasible region of the LP relaxation is a polyhedron. There is always an optimal solution of the LP relaxation that is attained in a vertex of this polyhedron. Throughout this book, we will typically refer to LP-feasible solutions as $\bar{x} \in \mathbb{R}^n$, while we refer to integer feasible solutions as $\tilde{x} \in \mathbb{R}^{n-|I|} \times \mathbb{Z}^{|I|}$. For a given linear constraint

$$\sum_{j \in \mathcal{N}} a_{ij} x_j \leq b_i,$$

and a given point $\bar{x}$, we call the value

$$\sum_{j \in \mathcal{N}} a_{ij} \bar{x}_j$$

the *activity* of the constraint i with respect to the point $\bar{x}$.

By knowing the optimal objective function values of a relaxation and of a feasible (integer) solution, we get a *dual bound* and a *primal bound*, respectively, for the optimal solution value of a MIP. To measure the quality of these bounds with respect to either the optimal solution value or each other, we use the notion of gap functions.

Definition 1.3 (primal gap) Let $\tilde{x}$ be a solution for a MIP, and let $x^\star$ be an optimal (or best known) solution for that MIP. We define the *primal gap* $\gamma^p \in [0, 1]$ of $\tilde{x}$ as

$$\gamma^p(\tilde{x}) := \begin{cases} 0, & \text{if } c^\mathsf{T} x^\star = c^\mathsf{T} \tilde{x} = 0, \\ 1, & \text{if } c^\mathsf{T} x^\star \cdot c^\mathsf{T} \tilde{x} < 0, \\ \frac{|c^\mathsf{T} x^\star - c^\mathsf{T} \tilde{x}|}{\max\{|c^\mathsf{T} x^\star|, |c^\mathsf{T} \tilde{x}|\}}, & \text{otherwise.} \end{cases}$$

The *primal-dual gap* is a typical measure given by MIP solvers during runtime. It is often referred to as an *optimality gap*, a name that might be considered slightly misleading since it does not explicitly describe the gap of any of the bounds to optimality, but is, rather, a worst-case estimation.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

Definition 1.4 (primal-dual gap) Let $\bar{x}$ be an optimal solution of a relaxation of a MIP and $\tilde{x}$ be a feasible solution for that MIP. We define the *primal-dual gap* $\gamma^{pd} \in \mathbb{R}_{\geq 0}$ of $\bar{x}$ and $\tilde{x}$ as

$$\gamma^{pd}(\bar{x}, \tilde{x}) := \begin{cases} 0, & \text{if } c^\mathsf{T}\bar{x} = c^\mathsf{T}\tilde{x} = 0, \\ \frac{c^\mathsf{T}\tilde{x} - c^\mathsf{T}\bar{x}}{|c^\mathsf{T}\bar{x}|}, & \text{if } c^\mathsf{T}\bar{x} \cdot c^\mathsf{T}\tilde{x} > 0, \\ \infty, & \text{otherwise.} \end{cases}$$

Inequalities only constrain variables against shifting their values into one direction. For the design of heuristics, it can be of interest to know into which direction a variable "is more constrained." This motivates Definition 1.5.

Definition 1.5 (variable locks) Let a MIP in the form (1.1) be given.

(i) The number of negative coefficients $\underline{\kappa}_j := |\{i \colon A_{ij} < 0\}|$ of a column $A_{\cdot j}$ is called the number of *down-locks* of the variable x_j.

(ii) The number of positive coefficients $\overline{\kappa}_j := |\{i \colon A_{ij} > 0\}|$ of a column $A_{\cdot j}$ is called the number of *up-locks* of the variable x_j.

Further, we call a variable x_j *trivially down-roundable*, if $\underline{\kappa}_j = 0$, and hence all coefficients of the corresponding column of matrix A are nonnegative. We call a variable x_j *trivially up-roundable*, if $\overline{\kappa}_j = 0$, and hence all coefficients of the corresponding column of matrix A are nonpositive.

1.2 Algorithms for Mixed-Integer Programming

Since MIP is an $\mathcal{NP}$-hard problem,[1] all known algorithms for general MIP have a worst-case exponential runtime. *Branch and bound* [108] is the most widely used algorithm to solve mixed-integer programs. State-of-the-art MIP solvers such as SCIP [2], FICO Xpress [60], Gurobi [85] and IBM ILOG CPLEX [101] all use *LP-based* branch and bound as a basic algorithm that is enhanced by various sophisticated subroutines to make the solvers efficient in practice.

In many cases, primal heuristics for MIP have been studied as subroutines of LP-based branch and bound; see in particular [18]. They might either be used globally, before the main part of the search, or locally for individual subproblems. The basic ideas of branch and bound are described in Section 1.2.1.

[1] 0–1 integer programming was among the 21 problems of Karp [103] for whom $\mathcal{NP}$-completeness was first proven. The $\mathcal{NP}$-completeness proof for the general integer case followed four years later [40].

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

Branch and bound, however, is not the only algorithm used to solve MIPs. A group of algorithms that is closely connected to primal heuristics are primal (exact) methods and are discussed in Section 1.2.2.

1.2.1 LP-Based Branch and Bound

input : A MIP P
output: An optimal solution $x^\star$ for P and the corresponding objective value $c^\star$

1 $\mathcal{L} \leftarrow \{P\}, \quad U \leftarrow \infty, \quad \tilde{x} \leftarrow$ **NULL**
2 **if** $\mathcal{L} = \emptyset$ **then**
3 | **return** $(\tilde{x}, U)$
4 **end**
5 Select $P_i \in \mathcal{L}, \quad \mathcal{L} \leftarrow \mathcal{L} \backslash \{P_i\}$
6 Solve LP relaxation of $P_i, \quad \bar{x} \leftarrow LP(P_i), \quad L_{\text{loc}} \leftarrow c(\bar{x})$
/* Bounding */
7 **if** $L_{loc} \geqslant U$ **then**
8 | **goto** line 2
9 **end**
10 **if** $\bar{x} \in P$ **then**
11 | $\tilde{x} \leftarrow \bar{x}, \quad U \leftarrow L_{\text{loc}}$
12 | **goto** line 2
13 **end**
14 Select $j \in I$: $\bar{x}_j \notin \mathbb{Z}$
/* Branching */
15 Split P_i into $P_{2i+1} := P_i \cup \{x_j \leqslant \lfloor \bar{x}_j \rfloor\}$
16 $P_{2i+2} := P_i \cup \{x_j \geqslant \lceil \bar{x}_j \rceil\}, \quad \mathcal{L} \leftarrow \mathcal{L} \cup \{P_{2i+1}, P_{2i+2}\}$
17 **goto** line 2

Algorithm 1 LP-based branch-and-bound algorithm to solve MIPs.

The idea of branch and bound is simple, yet effective: an optimization problem is recursively split into smaller subproblems, thereby creating a search tree and implicitly enumerating all potential assignments of the integer variables. The principle algorithm is given in Algorithm 1; a visualization is given in Figure 1.1.

The task of *branching* (line 15 in Algorithm 1.1) is to successively divide the given problem instance into smaller subproblems until the individual

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

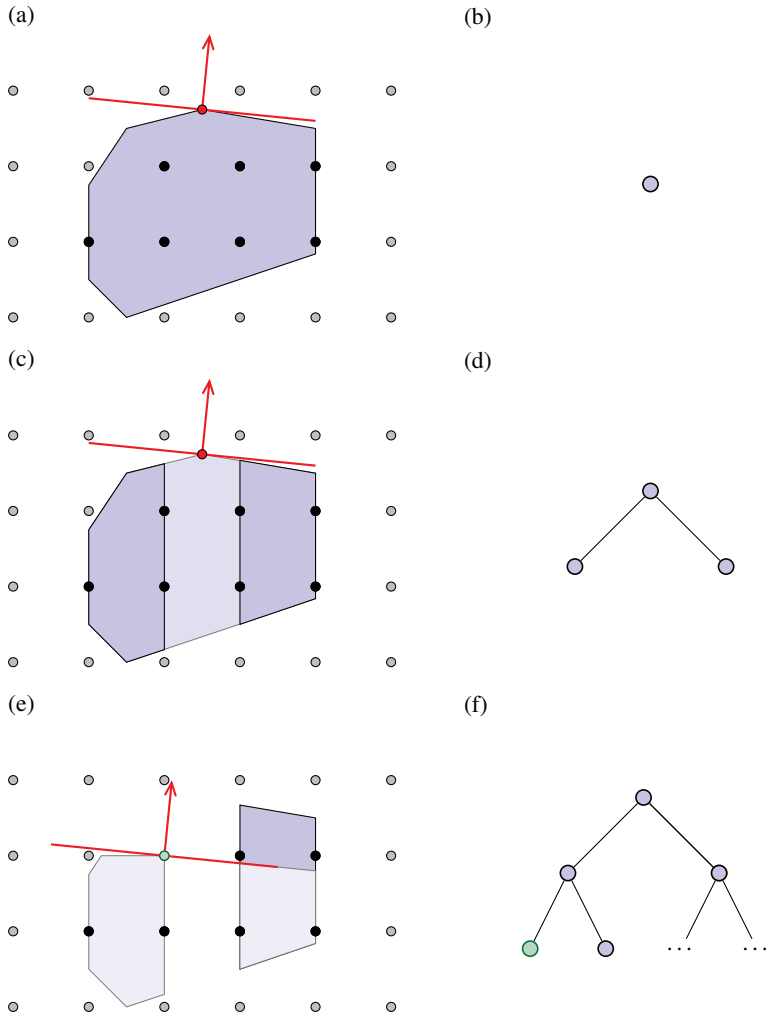

Figure 1.1 LP-based branch and bound: graphical representation. (a) The original MIP instance. (b) The root node represents the original problem. (c) Branching creates two subproblems. (d) One node for each subproblem. (e) Bounding cuts off suboptimal part. (f) The leaf node (green) is solved; the others will be recursively branched on.

subproblems are easy to solve. The best of all solutions found in the subproblems yields the global optimum. During the course of the algorithm a *branching tree* is created, with each node representing one of the subproblems. Compare Figures 1.1(a)–1.1(d). The original MIP is represented by the root node of the branching tree. Splitting the domain of one of the integer variables

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

by rounding the fractional LP solution up and down and installing the rounded values as new lower and upper bound, respectively, creates two disjoint subproblems. Those subproblems together contain all feasible integer solutions, but the minimum of its two LP relaxations will be strictly better than the LP relaxation of the original MIP. In the branching tree, the new problems are represented as children of the root node.

The intention of *bounding* (line 7 in Algorithm 1.1) is to avoid the complete enumeration of all potential integer assignments for the initial problem, which are usually exponentially many. If a subproblem's lower (dual) bound is greater than or equal to the global upper (primal) bound, that subproblem can be pruned. Lower bounds are calculated with the help of a relaxation, which is expected to be easy to solve. Upper bounds are found if the solution of the relaxation is also (integer) feasible for the corresponding subproblem. This is visualized in Figure 1.1(e).

Most commonly, the LP relaxation (1.2) is used for dual bounding. For MIPs, the LP relaxation is simply constructed by dropping the integrality conditions; see Definition 1.2.

Various techniques have been developed to improve this basic algorithm. Besides involved strategies for making good branching and subproblem selections, this includes supplementary procedures that help in tightening the lower and upper bounds. At each subproblem, domain propagation (see Section 1.4.2) can be performed to exclude values from the variables' domains.[2] The relaxation may be strengthened by adding further valid constraints (typically linear inequalities), which cut off the optimal solution of the relaxation, but retain all feasible solutions of the MIP. In the case where a subproblem is found to be infeasible, conflict analysis might be performed to learn additional valid constraints. Primal heuristics are used as supplementary methods to improve the upper bound. A good overview on the state of the art in computational mixed-integer linear programming can be found in [1, 116].

1.2.2 Primal Methods

A possible categorization of algorithms to solve optimization problems is to subdivide them into *primal* and *dual methods*. Loosely speaking, a primal method is an algorithm that produces a sequence of feasible, suboptimal solutions until it meets a criterion proving that the current incumbent solution is optimal. By contrast, a dual method is an algorithm that produces a sequence of infeasible, "super-optimal" solutions until it finds a first feasible point –

[2] The domain propagation applied at each subproblem is only one of the actions that are applied to strengthen the formulation. In particular, an aggressive strengthening is performed in the root node of the MIP so as to influence as much as possible the solving process. The actions in that phase are referred to as *preprocessing* (or *presolving*) [154].

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

which will be an optimum. As examples, consider the primal and dual simplex algorithms.

The added advantage of LP-based branch and bound is that it produces two sequences during the course of the algorithm, providing dual and primal bounds at the same time.

By the above classification, the *cutting plane method* [79, 80] by Gomory, is a dual method; it approaches the set of feasible solutions "from the outside," solving a sequence of relaxations. As soon as the relaxation finds a point that is feasible for the MIP, the proof of optimality comes "for free."

Since the mid 1990s, there has been a rising interest in using primal methods to solve MIPs. The principal ideas of primal methods, however, date back to the 60s and 70s. *Test set algorithms* and *integral basis methods* are two important groups of primal methods for (mixed-) integer programming. Both procedures require a known feasible solution as a starting point.

Test set algorithms are motivated by the Ford–Fulkerson [67] algorithm to compute maximum flows through a network. A *test set* for a given integer program P is a finite set of n-dimensional integral vectors $\mathcal{T} \subset \mathbb{Z}^n$, such that $c^\mathsf{T} t < 0$ for all $t \in \mathcal{T}$ and that for every nonoptimal feasible solution $\tilde{x}$ of P, there exists a $t \in \mathcal{T}$ such that $\tilde{x} + t$ is a feasible solution of P. If given a feasible start solution and a test set for a *pure integer (linear) programming* (IP) problem the algorithmic idea is straightforward: iteratively find an element of the test set that maintains feasibility when added to the solution. If no such element exists, the current solution is optimal. When Graver introduced the idea of test sets for integer programs, he showed that finite test sets exist for every feasible IP [83]. Weismantel gives a good overview on different methods to computationally obtain test sets for IPs [172].

The "simplified primal integer programming algorithm" in [175] can be seen as one of the first versions of an *integral basis method* or a *primal cutting plane method* for integer programming. The idea is to solve IPs only by means of the primal simplex algorithm, hence starting from a feasible solution (and an associated basis) and only performing simplex pivots that improve the objective, maintaining primal feasibility and integrality of the basic solution. Since this might not be possible in general, the constraint matrix is manipulated. In [175], this is done by adding a Gomory cut (and its slack variable) for the pivot row when conducting the ratio test of the simplex algorithm. This cut itself will then be chosen as the pivot row instead, and by construction the coefficient of the pivoting variable in the cut and the pivot ratio cancel out. As a consequence, both the cut's slack and the pivot variables take integral values in the new linear system that has been enhanced by one column and one row. A different understanding of this procedure is that it cuts off neighboring fractional

points of the incumbent feasible solution until it can make a simplex step that leads to a new incumbent.

Several extensions of this algorithm have been suggested. Notably, Haus et al. present an integral basis method that manipulates the columns of the matrix *without any* cuts being added [90]. By contrast, Letchford and Lodi [109] suggest several enhancements that make Young's algorithm converge more quickly by adding *more* cuts. The main improvements come from separating classes of cuts other than only Gomory cuts and by potentially adding several cuts per round, which is typical in dual cutting plane algorithms.

There is a smooth transition between primal methods and primal heuristics. On the one hand, some primal heuristics such as local branching or proximity search can be modified such that they become complete algorithms; see [62, 64]. In this case, each of their iterations will take an integer solution as an input and have either an improved integer solution or a proof of optimality as output, which is the general concept of primal methods. On the other hand, complete primal methods such as test set algorithms or the integral basis method could of course be run for a limited time as a primal heuristic within a branch-and-bound-based MIP solver.

1.3 Primal Heuristics inside MIP Solvers

In state-of-the-art MIP solvers, primal heuristics play a major role in finding and improving integer feasible solutions at an early stage of the solution process. Knowing good solutions early during optimization helps to prune the search tree and to simplify the problem via dual reductions. Further, knowing a solution proves the feasibility of a problem, and a practitioner might be satisfied with a solution that is proven to be within a certain gap to optimality.

A meaningful experiment in [91] categorized the impact of various components of branch-and-bound-based MIP solvers on the primal and dual sides of the solution process, measured by the primal and dual integral [17], respectively.[3] He considered branching rules, node selection, cutting plane generation, presolving and primal heuristics, and either deactivated all algorithms belonging to a certain component or, in the case of branching rule and node selection, switched to the vanilla default rules of random branching and a depth-first search.

One result of this experiment was that, not surprisingly, primal heuristics have a much larger impact on the primal side of the solution process than on the

[3] The primal integral is defined formally in Section 1.7.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

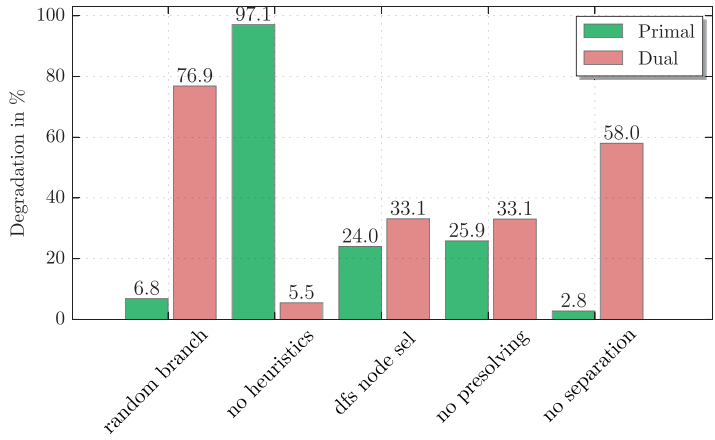

Figure 1.2 Degradation (in percent) of the average primal and dual integral of the MIP solver SCIP when deactivating individual parts of the solver (or changing them to vanilla rules for branching and node selection). Picture by G. Hendel [91].

dual side. Figure 1.2 shows that primal heuristics are by far the most important solver component for achieving good primal performance. Without them, the average primal integral, roughly speaking a measure of the convergence of the primal solution quality, nearly doubles.

These results are in line with those obtained in [6] with a similar experiment, this time based on the commercial MIP solver CPLEX. The results therein show that disabling primal heuristics gives an overall slowdown of approximately 28% in shifted geometric mean, and the slowdown gets higher when restricting to the harder models, that is, models that take more time to solve to proven optimality. Most importantly, primal heuristics play an important role in being able to solve models at all: a non-negligible share of models becomes unsolvable if primal heuristics are disabled.

Heuristics can be called at many different points of the branch-and-bound search. The MIP solver SCIP knows 13 different timings at which external heuristic plugins can be invoked. These include:

- before a node of the branch-and-bound tree is solved;
- after a node of the branch-and-bound tree has been solved;
- after domain propagation, but before LP solving at a node;
- after LP solving, but before branching;
- during LP solving, in each pricing round;
- after the final node of a dive in the branch-and-bound tree;

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

- various timing points for nodes with LP solves;
- before presolving starts;
- after presolving, before the initial root LP; and
- in between presolving rounds.

Clearly, some overall strategy is needed to coordinate the execution of all the different heuristics within a MIP solver. As usual, a trade-off needs to be sought: on the one hand, we do not want to miss finding good solutions early on in the process; on the other hand, we do not want the primal heuristics to become too time-consuming, in particular on models where they are not very effective.

Common approaches include setting frequencies at which each heuristic should be called, or setting a resource limit on how much should be spent in the solving process executing primal heuristics (e.g., 10% of the running time). In addition, some expensive heuristics are given a chance to run only once during the solution process, usually at the root.

Unfortunately, little is published about these strategies, with two notable exceptions. In [93], multi-armed bandit theory is applied for deciding how to split the heuristic resource budget on the different sub-MIP heuristics implemented in SCIP. The idea is that as we run each of them, we can collect statistics about their success rate on the current model, and bias future calls (within the same solution process) toward the most successful ones, but still giving the other heuristics a chance to run from time to time. Finally, in [27], the overall solution process is formally split into the following three phases:

- feasibility phase, where the solver tries to find a first feasible solution;
- improving phase, where the solver has found a feasible solution, but not yet the optimal one;
- proof phase, where the solver has already found the optimal solution and it *just* needs to prove optimality.

The intuition is that a different tuning of the solver components in general, and of primal heuristics in particular, should be used in the three phases. For example, primal heuristics could be disabled in the last phase, or they could be more aggressive if we are still in the first. Of course, precisely knowing when the transition between the improving and the proof phases happens is a challenge in itself, thus making the strategy adaptation a complex task. The recent line of research on the use of *machine learning* techniques within MIP (see, e.g., [12] for a recent survey on the topic) is a potentially promising direction for leveraging data to tackle the above challenge. Some of the work devoted

to the use of modern statistical learning for primal heuristics is discussed in Chapter 8.

Also, it is of utmost importance to balance the time invested versus expected success when designing individual primal heuristics. One common principle of achieving this is the so-called *fast-fail* (or *first-fail*) strategy. This idea has its origins in the constraint programming and artificial intelligence community [89] as a rationale for branching strategies, and was transferred to MIP heuristics in [18]. The rule of [89]: "To succeed, try first where you are most likely to fail" can be interpreted as prioritizing the most critical, or wavering, decisions early on. For variable fixing strategies, for example, this means attempting to fix variables that appear hardest to make feasible first. In a constraint-based view, this means aggressively resolving constraints first that are highly violated, rather than "locking in" constraints that are already feasible or almost feasible. This strategy offers two main benefits. First, it is easier to rectify the implications of a decision early on, when many variables are still unfixed. Second, if those early critical decisions indeed lead to infeasibility, the heuristic fails early, and at least minimizes the wasted computational time, hence the term "fail fast." The opposite strategy – that naively appears more natural to many humans – would be to fix variables or resolve constraints in a way that at each step minimizes the risk of getting an infeasible situation. In many situations, this pushes those hard decisions to the very end, where there is less freedom to resolve their implications, hence making success less likely, and ultimately often leads to running into an infeasible situation only after investing a lot of computational effort.

We end the section by presenting two possible categorizations of primal heuristics. The first categorization is based on the computational expensiveness of the building blocks that the heuristic is allowed to use. In this regard, heuristics inside a MIP solver are usually split into:

- *LP-free heuristics*, that is, those algorithms that are based solely on constraint propagation and other logical reasoning, but that not are allowed to solve linear programs (with the possible exception of solving a final LP in the mixed-integer case after all integer values have been assigned a value).
- *LP-based heuristics*, that is, those algorithms that are allowed to solve LPs, possibly multiple times.
- *sub-MIP heuristics*, that is, those algorithms that even allow the solution of mixed-integer programs, which are typically a (quite) restricted version of the main problem, with strict limits.

Alternatively, we can categorize heuristics depending on their purpose, that is, if their main objective is to construct a feasible solution from scratch (*start*

heuristics), or rather to improve upon an already known feasible solution (*improving* heuristics). Typically, but not necessarily, LP-free and and LP-based heuristics are start heuristics, while most sub-MIP heuristics are improving heuristics, defining and exploring neighborhoods associated with the known feasible solutions.

The vast majority of the primal heuristics discussed in the next chapters belong to only one of the classes above, with some notable exceptions. Nevertheless, the heuristics will be presented by grouping them into further (smaller) categories so as to better highlight their characteristics. We will also try to discuss them by referring to the concepts introduced in this section, namely the suitability of each group of heuristics to be invoked in specific points of the branch-and-bound search and its different phases defined above.

1.4 Rounding and Constraint Propagation

The two major components of LP-based heuristics are *rounding* and *constraint propagation*.

1.4.1 Rounding

In the MIP context, we refer to rounding as the operation that is applied to turn an LP-feasible solution $\bar{x}$ into a mixed-integer vector $\hat{x}$. Namely, for each $j \in \mathcal{I}$, if $\bar{x}_j$ is integer, then $\hat{x}_j = \bar{x}_j$. Otherwise, $\hat{x}_j = \lfloor \bar{x}_j \rceil$, which denotes assigning to variable x_j the integer value nearest to the fractional one $\bar{x}_j$. The rest of the solution does not change, that is, for all $j \in \mathcal{N} \setminus \mathcal{I}$, $\hat{x}_j = \bar{x}_j$. It is easy to see that the resulting vector $\hat{x} \in \mathbb{R}^{n-|\mathcal{I}|} \times \mathbb{Z}^{|\mathcal{I}|}$, but that it does not necessarily satisfy the linear constraints $Ax \leqslant b$ in the MIP (1.1).

The rounding operation is performed on all variables at the same time, and it is the most naive way of trying to recover MIP feasibility from an LP-feasible solution. In practice, rounding is performed sequentially on one or few variables at a time by taking into account its effect, for example, through constraint propagation, the topic of Section 1.4.2.

1.4.2 Constraint Propagation

Constraint propagation is a very general concept that appears under different names in different fields of computer science and mathematical programming. It is essentially a form of inference that consists of explicitly forbidding values – or combinations of values – for some problem variables. Constraint

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

propagation is used as a subroutine inside many practical heuristics, and inside branch and bound as well.

To get a practical constraint propagation system, two questions need to be answered:

- What does it mean to propagate a single constraint? In our particular case, this means understanding how to propagate a general linear constraint with both integer and continuous variables. The logic behind this goes under the name of bound strengthening (a form of preprocessing) in the integer programming community [124, 154].
- How do we coordinate the propagation of the whole set of constraints defining our problem?

In the remaining part of this section we will first describe bound strengthening and then describe the basic concepts of constraint propagation systems, following the propagator-based approach given by Schulte and Stuckey in [158].

Bound Strengthening

Bound strengthening [1, 81, 97, 124, 154] is a preprocessing technique that, given the original domain of a set of variables and a linear constraint on them, tries to infer tighter bounds on the variables. We will now describe the logic behind this technique in the case of a linear inequality of the form

$$\sum_{j \in C^+} a_j x_j + \sum_{j \in C^-} a_j x_j \leq b,$$

where C^+ and C^- denote the index set of the variables with positive and negative coefficients, respectively. We will assume that *all* variables are bounded: simple extensions can be made to deal with unbounded (continuous) variables and equality constraints.

For a given linear constraint and a given point $\bar{x}$, we call the value

$$\sum_{j \in C^+} a_j \bar{x}_j + \sum_{j \in C^-} a_j \bar{x}_j,$$

the *activity* of the constraint i with respect to the point $\bar{x}$.

In order to propagate the constraint above, the first step is to compute the minimum ($\underline{\alpha}$) and maximum ($\overline{\alpha}$) *activity level* [41] of the constraint, namely

$$\underline{\alpha} = \sum_{j \in C^+} a_j l_j + \sum_{j \in C^-} a_j u_j,$$

$$\overline{\alpha} = \sum_{j \in C^+} a_j u_j + \sum_{j \in C^-} a_j l_j.$$

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

Now we can compute updated upper bounds for variables in C^+ as

$$\overline{u}_j = l_j + \frac{b - \underline{\alpha}}{a_j}, \tag{1.3}$$

and updated lower bounds for variables in C^- as

$$\overline{l}_j = u_j + \frac{b - \underline{\alpha}}{a_j}. \tag{1.4}$$

Moreover, for variables constrained to be integer, we can also apply the floor $\lfloor \cdot \rfloor$ and ceiling $\lceil \cdot \rceil$ operators to the new upper and lower bounds, respectively. The maximum activity level is used analogously for constraints in $\geq$ form.

Let's consider, for example, the following system of linear constraints over five variables:

$$\begin{aligned} 3x_1 + 2x_3 + x_4 &\leq 5, \\ x_4 - 6x_5 &\geq 0, \\ x_1 + x_2 - x_5 &\leq 0, \\ x_1 &\in \{0, 1\}, \\ x_2 &\in \{0, 1\}, \\ x_3 &\in \{0, \ldots, 5\}, \\ x_4 &\in \{0, \ldots, 10\}, \\ x_5 &\in \{0, 1\}. \end{aligned} \tag{1.5}$$

The minimum activity $\underline{\alpha}$ of the first constraint is zero, and that gives updated upper bounds $x_4 \leq 5$ and $x_5 \leq 2$ (for the last upper bound, integrality is exploited). Then, the updated maximum activity $\overline{\alpha}$ for the second constraint implies the upper bound $x_5 \leq 0$ (again, exploiting integrality). Finally, the last constraint derives $x_1 = x_2 = 0$. Note that bound strengthening, being allowed to exploit integrality information, can derive bounds that are invalid for the linear programming relaxation, and thus strengthen the model: in the example above, the fractional solution $(0, 5/6, 0, 5, 5/6)$ is violated by the derived bounds.

It is worth noting that no propagation is possible in the case that the maximum potential activity change due to a single variable, computed as

$$\max_j \{|a_j(u_j - l_j)|\},$$

is not greater than the quantity $b - \underline{\alpha}$. This observation is very important for the efficiency of the propagation algorithm, since it can save several useless propagator calls.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

Finally, for linear constraints of special form, there often exist stronger or faster propagation algorithms. For example, a knapsack constraint, where all variables are binary and all coefficients are integer, can be propagated by making use of integer arithmetic instead of floating point arithmetic. Set covering constraints, that is, constraints of the form

$$x_{j_1} + x_{j_2} + \cdots + x_{j_k} \geq 1,$$

where all variables are binary, admit only one kind of propagation, namely fixing the remaining variable to 1 once all the others appearing in the constraint are fixed to 0, and propagating a set of set-covering constraints can be implemented very efficiently via the so-called *two-watched literals scheme* [133]. For an overview of specialized propagators for linear constraints, we refer to [1].

Propagation Algorithm

Now that we have described what it means to propagate a single linear constraint, we move on to describe how the propagation of the different constraints is coordinated within a MIP solver. The basic idea is that one does not just need to propagate all linear constraints once, as reductions obtained when propagating one constraint can lead to further reductions from constraints that we already propagated. Intuitively, we need to keep propagating the relevant constraints until a fix point is reached, that is, no more propagations can be found. It is also important to note that, in general, we do not just use the linear constraints of the model during propagation, but also other related global structures, for example, the clique table or the implication graph (see, e.g., [1] and Section 2.5).

A general framework for describing the behavior of (efficient) constraint propagation systems can be found in the constraint programming [149] literature, most notably in [157, 158]. In the remaining part of this section, we will introduce the general concepts and describe how they map to the MIP case.

Constraint propagation systems are built upon the basic concepts of *domain*, *constraint* and *propagator*.

A *domain D* is the set of values a solution x can possibly take. In our case, the domain is defined by

$$\begin{aligned} l_j \leqslant x_j \leqslant u_j, &\quad \text{for all } j \in \mathcal{N}, \\ x_j \in \mathbb{R}, &\quad \text{for all } j \in \mathcal{N} \setminus \mathcal{I}, \\ x_j \in \mathbb{Z}, &\quad \text{for all } j \in \mathcal{I}. \end{aligned} \tag{1.6}$$

In full generality, a *constraint c* is a relation among a subset of variables listing the tuples allowed by the constraint itself. However, this (very general)

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

definition is of little use from the computational point of view. In the MIP case, constraints are obviously defined as the linear constraints of the model, but in general we also propagate other structures: those also play the role of constraints during propagation. In order to have a more practical, yet still general, abstraction, constraint propagation systems implement constraints through so-called *propagators*.

A propagator p implementing a constraint c is a function that maps domains to domains and that satisfies the following conditions:[4]

- p is a *decreasing* function, that is, $p(D) \subseteq D$ for all domains. This guarantees that propagators only remove values.
- p is a *monotonic* function, that is, if $D_1 \subseteq D_2$, then $p(D_1) \subseteq p(D_2)$.
- p is *correct* for c, that is, it does not remove any tuple allowed by c.
- p is *checking* for c, that is, all domains D corresponding to solutions of c are *fixpoints* for p, that is, $p(D) = D$. In other words, for every domain D in which all variables involved in the constraint are fixed and the corresponding tuple is valid for c, we must have $p(D) = D$.

Again, propagators can be associated not just with the linear constraints of the model, but with other global structures such as the clique table or the implication graph. In addition, propagators can be used to implement *dual* reductions, that is, bound changes that are not necessarily valid for all feasible solutions of the model at hand, but that still guarantee that at least one optimal solution is left. The most common example is the so-called reduced cost fixing [140], but others such as *orbital fixing* [144] are also common in modern MIP solvers.

A *propagation solver* for a set of propagators R and some initial domain D finds a fixpoint for propagators $p \in R$.

A basic propagation algorithm is outlined in Algorithm 2. On input, the propagator set is partitioned into the sets P_f and P_n, depending on the known fixpoint status of the propagators for domain D – this feature is essential for implementing efficient incremental propagation. The algorithm maintains a queue Q of pending propagators (initially P_n). At each iteration, a propagator p is popped from the queue and executed. At the same time, the set K of variables whose domains have been modified is computed and all propagators that share variables with K are added to Q (hence they are *scheduled* for execution).

The complexity of this algorithm is highly dependent on the domain of the variables. For integer (finite domain) variables, the algorithm terminates in a finite number of steps, although the complexity is exponential in the size of

[4] In general, a constraint c is implemented by a collection of propagators; we will consider only the case where a single propagator suffices.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

input : a domain D
input : a set P_f of (fixpoint) propagators p with $p(D) = D$
input : a set P_n of (non-fixpoint) propagators p with $p(D) \subseteq D$
output: an updated domain D

$Q = P_n$
$R = P_f \cup P_n$
while Q *not empty* **do**
  $p = \texttt{Pop}\ (Q)$
  $D = p(D)$
  K = set of variables whose domain was changed by p
  $Q = Q \cup \{q \in R : var(q) \cap K \neq \emptyset\}$
end
return D

Algorithm 2 Basic propagation engine.

the domain (it is however polynomial in the pure binary case, provided that the propagators are also polynomial, which is usually the case). For continuous variables, this algorithm may not converge in a finite number of steps, as shown in the following example (Hooker [97]):

$$\begin{cases} \alpha x_1 - x_2 \geq 0, \\ -x_1 + x_2 \geq 0, \end{cases} \tag{1.7}$$

where $0 < \alpha < 1$ and the initial domain is $[0, 1]$ for both variables; it can be easily seen that the upper bound on x_1 converges only asymptotically to zero. In a first round of propagation, the upper bound of x_2 would be tightened to α due to the first inequality, and in turn the upper bound of x_1 would be tightened to α due to the second inequality. In the next round, the upper bound of x_2 would be tightened to α^2 due to the first inequality, leading to an upper bound of α^2 for x_1, and so on, as shown in Figure 1.3. So, in practice, Algorithm 2 is stopped after some predefined number of iterations or if the reduction in the domain of variable falls below some given threshold.

1.5 General Heuristic Concepts

While primal heuristics for MIP clearly exploit the specific properties and tools of the MIP paradigm, for example, the global view of the problem given by the MIP formulation and the ability to solve LP relaxations or even related

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

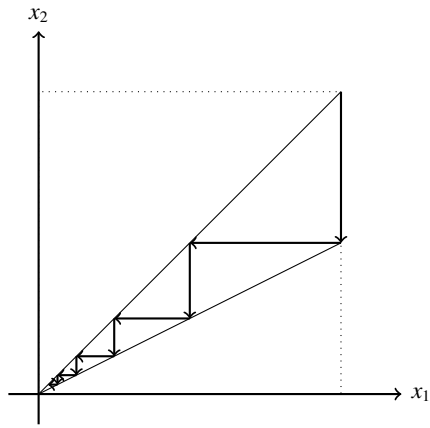

Figure 1.3 A nasty example for constraint propagation.

(sub-)MIPs, they can often be described and interpreted with the language of general mathematical heuristics for combinatorial optimization problems. In particular, many constructive heuristics follow (more or less strictly) a greedy approach, while most improvement heuristics are based on local search. In this section, we give a general overview and definition of those basic concepts.

1.5.1 The Greedy Paradigm

A very natural approach for solving combinatorial problems is to use a strategy that takes a sequence of decisions, with a set of different possible choices at each step. If at each step we always pick the choice that is locally optimal, that is, it is the best choice given the decisions we have made so far, and we never backtrack on those decisions, then we have a *greedy* algorithm [48, chapter 16].

Greedy algorithms are not guaranteed to return the optimal solution (or even a feasible solution) to an optimization problem, but in some cases they do. The effectiveness of a greedy approach by far depends on the structure of the problem: for some specific classes of problems, such as minimum spanning tree, shortest path and continuous knapsack (to name a few), there are greedy algorithms that solve the problem to proven optimality. In less ideal, but still good cases, they always return a feasible solution and are able to guarantee a constant worst-case approximation ratio; for example, the greedy algorithm returns a solution that is at most $1 - 1/e$ away from the optimum in submodular function maximization [141], where e is the Euler constant. Similarly, with some care, the greedy strategy gives a 2-approximation for binary knap-

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

sacks [125]. For other combinatorial problems, the best guarantee is no longer constant: the approximation is $O(\log n)$ for both the set covering problem and the *traveling salesman problem* (TSP), where the heuristic is known as nearest neighbor. Finally, no guarantee whatsoever can be given for a general MIP. Another substantial benefit of greedy strategies, besides their simplicity, is that they are typically quite fast. With appropriate data structures, greedy algorithms can be often implemented in linear time, potentially with an additional logarithmic factor in case sorting is needed. So, even if they are not guaranteed to find a solution, they are usually worth a try.

Inside MIP solvers, diving heuristics, that is, methods that simulate a dive in the branch-and-bound tree by iteratively fixing a variable and resolving the LP relaxation, are prime examples of greedy strategies.

1.5.2 Local Search

Another approach for designing heuristics for a given optimization problem is that of local search. Again, the concept is very natural: given a known feasible solution, often referred to as "reference" or "candidate" solution, it is usually worth looking for a better one in a properly defined neighborhood in the solution space. Then, the approach can be repeated until a locally optimal solution is found.

For structured optimization problems, local search heuristics rely on ad hoc definitions of neighborhood, based on so-called *moves*. For example, permutation problems (e.g., the TSP) are amenable to 2-opt neighborhoods, where a solution is in the neighborhood of the reference solution if it can be obtained from the latter by swapping two elements in the sequence. Another (even simpler) neighborhood is 1-opt: given a reference solution, this is defined as the set of solutions that can be obtained by changing the value of a single variable. Note that a 1-opt neighborhood might not contain any additional solution (e.g., in the TSP case). On the one hand, the size (and structure) of the neighborhood has a direct effect on the complexity of the method. Small neighborhoods are clearly faster to explore, but the chances of finding an improving solution are smaller, so there is a higher risk of getting stuck. On the other hand, very large neighborhoods can be prohibitively expensive to explore, at least exhaustively.

In the MIP case, we often try to sidestep the issue by applying a so-called large neighborhood search (LNS) approach, in which we define a neighborhood implicitly by adding constraints (often, but not necessarily, variable fixings), and then apply MIP technology recursively on the resulting sub-MIP. While this is quite elegant, it is still quite challenging to predict a priori how expensive each such sub-MIP is to solve, so proper limits and an outer logic

to adjust the size of the subproblems is needed to get a practical method (see Chapter 2).

Independently from the choice of neighborhood, we also need a sound strategy to escape local optima, and we have a great variety of options, namely:

- we can restart the local search from a different solution, obtained by sampling or perturbation;
- we can change the acceptance criterion when exploring a neighborhood, allowing for non-improving solutions to be accepted;
- we can change the neighborhood (either in size or structure);
- all of the above.

There is a vast literature in the *meta*heuristic community [78] on strategies based on local search: iterated local search (ILS), tabu search, variable neighborhood search (VNS), simulated annealing and random walks, just to name a few, have all proven quite successful on specific optimization problems, and are tools of wide applicability, although they often require extensive tuning to give the best results on a given class of problems. We note though that those meta approaches, while crucial when designing standalone heuristics from scratch, are less important within the MIP framework, where we rely on branch and bound itself to let the heuristics escape local optima.

Finally, note that local search is in many ways complementary to greedy methods: greedy strategies can be used to find feasible solutions, while a subsequent local search phase (i) improves their quality, and (ii) makes sure that those are at least locally optimal.

1.5.3 Population-Based Methods

The local search methods described in Section 1.5.2 are so-called single solution methods: they move from feasible solution to feasible solution, and the currently explored neighborhood is defined starting from the current solution only. These methods are of course allowed to (and often do) store past solutions in a so-called *solution pool*, but those do not affect the behavior of the algorithm. By contrast, population-based methods maintain a set of feasible solutions at each iteration, and they explore the solution space based on some properties of the whole set.

The most well-known population-based algorithms by far are *genetic algorithms* [132] that are inspired by natural evolution: at each iteration, the current generation of individuals (i.e., set of solutions) is mutated/combined to obtain the next generation, and selection is applied according to fitness (i.e., the objective function). While the intuition of evolutionary algorithms is clear,

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

their design is less trivial: many different steps need to be specified, often in a problem-specific fashion, and there are many parameters to be tuned in order to get the best results on a given class of problems. However, once designed, they are also, in general, quite easy to implement and they parallelize trivially.

Evolutionary algorithms are not predominant in MIP, for the same reason as for local-search meta schemes, but some limited form of them, based on the MIP solution pool, is actually implemented in all state-of-the-art MIP solvers.

1.5.4 Diversification vs. Intensification

To be successful, a primal heuristic needs to be able to find good quality solutions in an exponentially sized solution space, and to do so in a reasonable amount of time. In order to achieve such goal, when designing a primal heuristics two opposite goals must be continuously kept in balance. On the one hand, we want to sample different parts of the solution space in order to reduce the chances of getting stuck in an unpromising region: this goal is called diversification, and it is often achieved through a random component. On the other hand, we also need to invest more resources in the promising parts, as sampling alone is not going to find the proverbial needle in a haystack quickly enough on average; this is called intensification, and it is often (although not necessarily) based on some form of local search.

Most of the heuristics that we will describe in the book can be analyzed under the lens of diversification vs. intensification. A greedy algorithm, for example, does neither; it neither diversifies (to the contrary, being deterministic it would always yield the same outcome), nor intensifies, and this is why a single greedy approach is rarely a good option on general problems. At least on the first front, the greedy scheme can be improved by incorporating randomization. The most common approach is the so-called GRASP scheme [59], where at each step, instead of just picking the locally optimal solution, we build a shortlist of best candidates and choose randomly within them. To the contrary, a pure local search method intensifies very well but lacks diversification; indeed, all the meta schemes mentioned in the previous sections can be interpreted as a way to add diversification to the basic method (and thus escape local optima).

1.6 Reference Points, Information and Statistics

In this section, we introduce some extra definitions and concepts that help discuss and analyze primal heuristics for MIP.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

1.6.1 Reference Points

A variety of reference points are of interest for primal heuristics.

The *incumbent solution*, that is, the best feasible solution that has been found so far during a running MIP solve, is used as a reference point for the vast majority of improvement heuristics. In particular, large neighborhood search heuristics such as RINS [51], proximity search [65], or local branching [62] make use of the incumbent. Guided diving [51] is an example of a diving heuristic that requires an incumbent solution.

Other feasible solutions are of comparable interest. Crossover is a large neighborhood search heuristic that requires more than one solution. Furthermore heuristics might use infeasible integer points as reference, for example, feasibility pump or some variants of local branching or proximity search. Finally, a partial assignment of the integer variables might be used to initialize local search heuristics. For example, this can be provided as an input to a MIP solver.

Just as every MIP might have many different optimal solutions, there might also be multiple optima of the LP relaxation. Many MIP heuristics consider an optimal solution of the LP relaxation as a reference point. Besides that, alternative optimal or feasible solutions for the LP relaxation might be used to guide heuristics. Another important reference point is the so-called analytic center.

The analytic center x^{ac} of a bounded polyhedron given in equality form ($Ax = b, x \geqslant 0$) has been introduced by Sonnevend [163] and is defined as

$$x^{\text{ac}} = \operatorname{argmin}\left\{ -\sum_{j \in \mathcal{N}} \ln x_j : Ax = b \right\}. \tag{1.8}$$

The analytic center can be efficiently computed by using a barrier algorithm. Note that the strong convexity of the logarithm implies that the analytic center of a bounded polyhedron is uniquely defined. It maximizes the distance to the boundary due to the logarithm tending to minus infinity when its argument is going to zero. If the polyhedron is a simplex, the analytic center is also the barycenter of the polyhedron [164].

1.6.2 Solving Statistics

The *pseudocosts* [13, 72] of a variable are a statistic that is collected during the course of a branch-and-bound algorithm. Pseudocosts track the history of how much the dual bound improved when branching on a given variable in previous nodes. These statistics can then be used to estimate how much the objective will change when the bounds of the variables are tightened.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

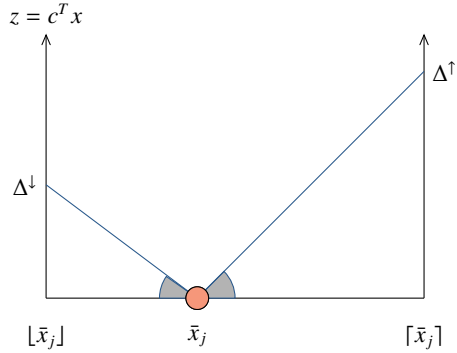

Figure 1.4 Graphical representation of pseudocosts update.

Pseudocosts are updated after a bound change of a variable has been performed (e.g., due to branching or a fix in a diving heuristic) and the LP relaxation of the tightened problem has been solved. When updating the pseudocosts, the objective gain per unit change in variable x_j is computed. Assume we change the lower bound of x_j, that is, we rounded it up. Then, the upward gain ς_j^+ is computed as

$$\varsigma_j^+ := \frac{\Delta^{\uparrow}}{\lceil \bar{x}_j \rceil - \bar{x}_j}, \tag{1.9}$$

where $\Delta^{\uparrow}$ is the difference between the optimal LP objectives of the subproblem with x_j rounded up and the optimal LP objective before conducting the rounding. The downward gain ς_j^- when rounding down a variable is computed accordingly. The thin, light blue line in Figure 1.4 illustrates the operation. These estimation formulas are based on the assumption that the objective increases linearly in both directions.

Let σ_j^+ denote the sum of ς_j^+ over all subproblems that were created by branching upward on x_j and whose LP relaxation has already been solved and was feasible. Further, let ν_j^+ be the number of such problems. Then the upward pseudocosts of variable x_j are calculated as the arithmetic mean of all objective gains (per unit step length) observed by branching upward on x_j, namely

$$\Psi_j^+ := \frac{\sigma_j^+}{\nu_j^+}. \tag{1.10}$$

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

The downward pseudocosts Ψ_j^- are computed analogously from subproblems that were created by branching downward on x_j and whose LP relaxation has already been feasibly solved.

The *upward inference value* of a variable x_j, $j \in \mathcal{I}$, is defined analogously to the pseudocosts as

$$\tilde{\varsigma}_j^+ := \frac{\tilde{\sigma}_j^+}{\tilde{\nu}_j^+}, \tag{1.11}$$

where $\tilde{\sigma}_j^+$ is the sum of the number of domain reductions found by propagation taken over all subproblems that were created by branching upward on x_j for which domain propagation has already been applied, and $\tilde{\nu}_j^+$ is the number of such subproblems. Again, the downward inference $\tilde{\varsigma}_j^-$ value is defined analogously.

Branching statistics are useful information for primal heuristics to make decisions, but heuristics can also contribute to these statistics. Rapid learning [21] takes this to the extreme. It is applies a fast depth-first branch-and-bound search without solving LPs for a limited number of nodes, initializing, among other statistics, inference values.

1.7 Measuring the Impact of Primal Heuristics

When implementing optimization software, two questions naturally arise: how does the new code perform with respect to existing code, and which are the best settings for a particular algorithm? This goes back to the early days of operations research. Hoffman et al. reported a first computational experiment to compare different implementations of linear programming algorithms in 1953 [95]. Just as researchers and software vendors want to distinguish their code on general test sets, a user wants to tune their optimization software for a particular set of problems. However, all parties require suitable criteria for measuring the performance of a software implementation.

In mathematical programming, the running time to optimality, the number of branch-and-bound nodes and the number of simplex or interior point iterations are commonly used performance measures. All of these mostly depend on the convergence of the dual bound since, in practice, it typically takes much longer until the dual bound converges to the optimal value than it takes to find a primal solution of the optimal value. This is not only an empirical observation, but also indicated by complexity theory. Finding a solution for a certain objective value is $\mathcal{NP}$-complete, but proving a certain bound is co-$\mathcal{NP}$-complete and thereby

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

most likely not even in $\mathcal{NP}$, is hence arguably harder! Feasibility certificates (hence solutions) are of linear size, but a MIP optimality certificate is not – it can be an exponential tree or set of cuts.

Furthermore, measures such as time to optimality or number of branch-and-bound nodes make most sense for instances that can actually be solved within a given time limit – whereas employing heuristics is particularly worthwhile for hard instances that cannot be solved to proven optimality within a reasonable time.

For primal heuristics, it seems logical to consider measures that consider primarily the primal bound and that are independent of an eventual timeout. Examples of such measures are the time needed to find a first feasible solution, an optimal solution, or a solution within a certain gap to optimality (see, e.g., [96]). Each of these has its individual strengths and weaknesses. The time to first solution entirely disregards the solution quality: for about one quarter ($^{23}/_{87}$) of the MIPLIB 2010 [106] benchmark instances, a trivial solution of all variables set to their lower bound (or all to their upper bound) is feasible, but most of the time such a solution does not provide valuable information to the user – it corresponds to obvious extreme cases such as utilizing all available resources when trying to minimize the number of used resources or producing no goods when trying to maximize the amount of produced goods. Particularly when analyzing heuristics embedded in a complete solver, the time to the first solution mainly measures the time needed for preprocessing and solving the root node relaxation; the MIP solvers CPLEX, GUROBI and XPRESS find solutions for the vast majority of the MIPLIB 2010 benchmark instances during root node processing. Instead, the time to optimal solution ignores that slightly suboptimal but practically sufficient solutions might have been found long before. Finally, taking the time to a certain gap is an attempt to balance this, but the choice of the threshold is arbitrary by design.

Altogether, the most important consideration for primal heuristics is the trade-off between speed and solution quality. None of the above performance measures gives a complete assessment of this trade-off. In [17], Berthold introduces a new performance measure that takes into account the whole solution process, and is especially targeted at benchmarking primal heuristics. It measures the progress of the primal bound's convergence toward the optimal solution over the entire solving time.

Assume the objective function values of intermediate incumbent solutions and the points in time when they have been found to be given – for a particular MIP solver, a certain problem instance and a fixed computational environment. This information can be gathered from the log files that standard MIP solvers produce.

Definition 1.6 (primal gap function) Let $t_{\max} \in \mathbb{R}_{\geqslant 0}$ be a limit on the solution time of a MIP solver. Its *primal gap function* $p\colon [0, t_{\max}] \mapsto [0, 1]$ is defined as follows:

$$p(t) := \begin{cases} 1, & \text{if no incumbent until time } t, \\ \gamma^p(\tilde{x}(t)), & \text{with } \tilde{x}(t) \text{ being the incumbent at time } t, \text{ otherwise.} \end{cases}$$

Here, γ^p is the primal gap as defined in Definition 1.3. The primal gap function $p(t)$ is a step function that changes whenever a new incumbent is found. It is monotonically decreasing, one at $t = 0$, and zero from the point at which the optimal solution is found.

Definition 1.7 (primal integral) Let $T \in [0, t_{\max}]$ and let $t_i \in [0, T]$ for $i \in \{1, \ldots, I-1\}$ be the points in time when a new incumbent solution is found, $t_0 = 0$, $t_I = T$. The *primal integral* $P(T)$ of a run is defined as

$$P(T) := \int_{t=0}^{T} p(t)\, dt = \sum_{i=1}^{I} p(t_{i-1}) \cdot (t_i - t_{i-1}).$$

The fraction $P(t_{\max})/t_{\max}$ can be seen as the average solution quality during the search process. In other words, the smaller $P(t_{\max})$ is, the better is the expected quality of the incumbent solution if the solver is stopped at an arbitrary point in time.

Berthold [17] suggests using $P(t_{\max})/t_{\max}$ for measuring the quality of primal heuristics. This measure features two simple, but important attributes. First, whenever a better solution is found at the same point in time, $P(t_{\max})$ decreases. Second, whenever the same solution is found at an earlier point in time, $P(t_{\max})$ decreases. Briefly, the primal integral favors finding good solutions early. For the performance measures discussed so far, at most one of these two attributes holds in general.

Consider Figure 1.5 for a visualization of the primal integral. The thick red line shows the development of the primal bound when using heuristics; the red-shaded area corresponds to the primal integral of that solution process. The thick green line shows the development of the primal bound when not using heuristics; the green-shaded (plus the red-shaded) area corresponds to the primal integral of that solution process. The primal integral of the run with heuristics is smaller (hence better) since solutions with a small gap are found earlier.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

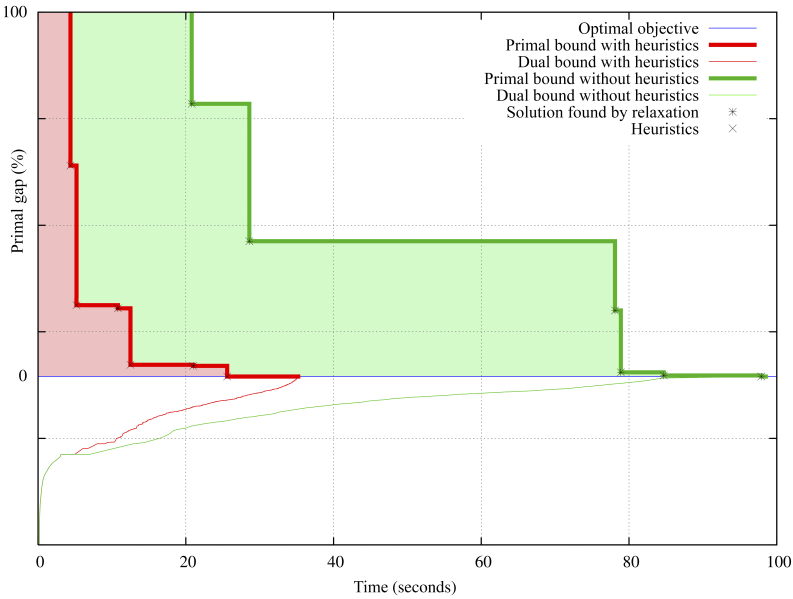

Figure 1.5 Visualization of the primal integral for two solution processes of the same instance.

1.8 Quiz

It's quiz time! This section is designed to test your understanding of the concepts covered in this chapter. Each question is followed by four possible answers, only one of which is correct. If you are unsure about some questions, you might find it helpful to work through the corresponding section again. Remember, sometimes the process of elimination can be as valuable a tool as direct problem-solving. By engaging with this quiz, you'll reinforce your knowledge and identify areas requiring further study. Take your time, think carefully and check the answers in the Appendix when you are done.

Question 1 The feasible region of the LP relaxation of a MIP is a polyhedron

- ☐ only if all variables are bounded;
- ☐ always;
- ☐ only if there are no equalities;
- ☐ never.

Question 2 Consider a simple binary knapsack problem in standard form (all positive data):

- ☐ all variables are up-locked;

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press

- ☐ all variables are down-locked;
- ☐ all variables have zero locks;
- ☐ all variables are both up- and down-locked.

Question 3 In an LP-based branch-and-bound algorithm, the primal-dual gap between the current best primal bound and current best dual bound

- ☐ changes erratically;
- ☐ is monotonically decreasing and reaches zero when the algorithm terminates;
- ☐ is monotonically increasing and reaches one when the algorithm terminates;
- ☐ stays constant, as it is a property of the instance;

Question 4 Within the context of branch and bound, primal heuristics are

- ☐ useless, branch and bound is an exact method and does not need heuristics;
- ☐ required for branch and bound to converge;
- ☐ practically relevant and effective at reducing the primal gap;
- ☐ irrelevant for the optimality proof.

Question 5 Consider an integer program with all binary variables. Constraint propagation is

- ☐ a complete and polynomial inference scheme;
- ☐ a complete but non-polynomial inference scheme;
- ☐ an incomplete but polynomial inference scheme;
- ☐ an incomplete and non-polynomial inference scheme.

Question 6 In a MIP context, a greedy heuristic followed by a local search procedure

- ☐ is guaranteed to find a feasible solution;
- ☐ always finds the optimal solution if the local search neighborhood is a 2-opt;
- ☐ is guaranteed to find a feasible solution with a $O(\log n)$ worst-case approximation ratio;
- ☐ has no guarantees.

Question 7 The primal integral

- ☐ is independent of the chosen time limit;
- ☐ can only be approximated with a finite summation;
- ☐ is a measure balancing solution quality and runtime;
- ☐ is always a number in $[0, 1]$.

https://doi.org/10.1017/9781009574792.002 Published online by Cambridge University Press