EURO Advanced Tutorials on Operational Research
Series Editors: M. Grazia Speranza · José Fernando Oliveira

Vittorio Maniezzo
Marco Antonio Boschetti
Thomas Stützle

Matheuristics

Algorithms and Implementations

EURO
THE ASSOCIATION OF EUROPEAN OPERATIONAL RESEARCH SOCIETIES

Springer

EURO Advanced Tutorials on Operational Research

Vittorio Maniezzo • Marco Antonio Boschetti •
Thomas Stützle

Matheuristics

Algorithms and Implementations

Springer

Vittorio Maniezzo
University of Bologna
Cesena, Italy

Marco Antonio Boschetti
University of Bologna
Cesena, Italy

Thomas Stützle
Université Libre de Bruxelles
Brussels, Belgium

ISSN 2364-687X ISSN 2364-6888 (electronic)
EURO Advanced Tutorials on Operational Research
ISBN 978-3-030-70276-2 ISBN 978-3-030-70277-9 (eBook)
https://doi.org/10.1007/978-3-030-70277-9

Preface

Nomina sunt consequentia rerum, stated Justinian I, some 15 centuries ago.[1]

Language serves not only to express thoughts but also to make possible thoughts that could not exist without it, as annotated a *maître à penser* closer to our times.[2] It was thus an unavoidable consequence of the state of things that a word emerged to describe an ever more active area of interest, where mathematical programming (MP) techniques, originally developed for solving to optimality combinatorial problems, were primarily used to quickly find possibly suboptimal solutions or, conversely, problems for which heuristic methods had difficulties providing satisfactory solutions began to be attacked with MP techniques under constrained computational resource availability, thus abandoning the guarantee to obtain optimal solutions.

The literature on search for optimal or fast but possibly suboptimal solutions of combinatorial optimization problems has very often intersected in the past, but it has usually kept these two concerns well separated, thereby defining separate methods to achieve either objective. Indeed, while we are all aware of the massive amount of mathematical contributions that made up the theoretical corpus of exact methods, specific literature for nonproblem-specific heuristic methods has been slow to emerge.

One of the first contributions along this last line is due to Polya (1945), whose book "How to Solve It" has a focus on the general class of "mathematical" problems, i.e., problems that can be modeled and solved by mathematical techniques. The book proposes strategies that are, therefore, applicable to design optimization algorithms. However, the book was better received by psychology scholars than by mathematicians, as it also contributed to studies on how the human brain is able to effectively cope with difficult problems. This possibly unexpected initial impact

[1]Names are the consequence of things, Institutiones, II, 7, 3. This is the usual formulation, though the actual citation reads *nos ... consequentia nomina rebus esse studentes*.

[2]Russell (1948). Citations are discussed at the end of the chapter, for each content chapter, in compliance with editorial criteria.

can be remarked also noticing that some of the first heuristics for the assignment problem (expressively named "the method of divine intuition," "the method of daily quotas," or "the method of predicted yield") were presented in an address delivered in 1949 at a meeting of the American Psychological Association while discussing a job assignment problem (Thorndike 1950).

Actually, for the first decades of their history, heuristics were presented as inherently linked to a problem.[3] An early example is a study made on the transportation problem by Tolstoi (1930), who presented a heuristic to solve a concrete transportation problem coming from cargo transportation along the Soviet railway network. However, it is in the 1950s that the advent of sufficient computing resources made heuristics operational techniques, and several problems were attacked, such as, for example, max flow with a "flooding technique," shortest path, and traveling salesman problem,[4] among others.

The two main basic heuristic approaches, namely constructive and local search, began to be repeatedly applied, but without any high-level, problem-independent algorithmic formalization. To get this, one had to wait until the 1960s and the 1970s, with the rise of metaheuristics, which finally brought about a wide scientific literature entirely dedicated to heuristic algorithms. However, metaheuristic contributions drew very little from established MP results.

Things began to change, thanks to increased MP effectiveness, which together with increased hardware performance permitted to efficiently solve ever larger NP-hard problem instances, and thanks to a major game changer, MIP solvers. These last are themselves results of MP efforts and have proven capable of efficiently solving larger and larger instances of mixed-integer problem instances, both generic and deriving from specific combinatorial optimization problems. However, following Lodi (2013), it is worth remarking, that:

- MIP solvers are used for a large portion as heuristics,
- MIP solvers are heuristic in nature,
- the computation for NP-hard problems is intrinsically heuristic,
- heuristic decisions and techniques are hidden everywhere in MIP solvers.

Even a MIP solver, a paradigmatic example of an exact system, leans, therefore, much toward a model-based heuristic technique.

Anyway, a good understanding of effective methods for solving NP-hard problems was reached, and effective code was available to be included in a solution where an NP-hard problem appeared as a subproblem for the overall objective. This kindled the interest in researching how these new opportunities could best be exploited in the design of heuristic methods. Several directions were investigated, and this leads to the Byzantine proposition that opens the section. A number of researchers were studying how to make use of MP results to improve heuristic search

[3]Most of the following citations are drawn from Schrijver (2005).

[4]See Boldyreff (1955), Rosenfeld (1956), Croes (1958), and Morton and Land (1955). The last citation proposing a linear programming approach with a 3-exchange heuristic, thus a local search.

effectiveness, enough of them to define a focused scientific community. The focus of the research of this community was given the name "*matheuristics*," after the name of the international workshop series that had the first edition in 2006[5] and continued biennially ever since.

Literally, hundreds of publications appeared on proceedings and journals under this heading, accompanied after a while by several review papers.[6] This book represents a further effort to compendiate these works, providing a unified presentation of what we deem to be the most innovative and effective contributions to the field.

Actually, the scope of the book does not encompass all kinds of matheuristic contributions. Matheuristics leverage on the mutual benefits of mathematical models and heuristic techniques, and as such they can derive both from the use of mathematical models in heuristic design and from the use of heuristics as modules of exact MP approaches. This last possibility has been effectively studied, too, but it is not covered in this book. We refer the interested reader to the review by Puchinger et al. (2009).

The book is composed of three parts.

The first part is not strictly about matheuristics but presents preliminary material. All algorithms in the text are in fact proposed as general, problem-independent methods and are then detailed in a specific application, to show what is implied when implementing the general method in a specific case. All implementations are complete with traces of runs on one same instance of generalized assignment problem (GAP). Therefore, we preliminary review the state of the art of research on the GAP, introduce the test case instance, and, more importantly, show the application to the GAP of several MP methods (relaxations, decompositions, bounds, etc.), which will then be used as modules of later described matheuristics. A remark about notation: throughout the book, we tried to keep it uniform and consistent, despite the heterogeneity found in the original sources. The result is not utterly rigorous but hopefully light and accurate enough to be unequivocal and accessible.

All search algorithms, when put to use, are run under the control of user-defined parameters, whose configuration greatly affects the search performance. This is also true for matheuristics and for their runs reported in the book. We include, therefore, in the preliminary part a section reviewing the state of the art of automated parameter setting to allow the reader to get the best performance from her or his possibly self-designed algorithms.

The second part of the book describes how MP methods have been used as modules complementing known heuristics, which in our selection are all metaheuristics. We have already remarked that early metaheuristics proposals made very little use of MP results. This began to change at the turn of the millennium, when a number of the most significant metaheuristics were hybridized with MP. We

[5]See Maniezzo (2006).

[6]See, for example, Boschetti et al. (2009), Fischetti and Fischetti (2018), Maniezzo et al. (2009a,b), Maniezzo et al. (2020).

show how this has been done both for metaheuristics that update a single solution at each iteration (simulated annealing, tabu search, iterated local search, variable neighborhood search, GRASP, ejection chains) and for those that maintain sets of solutions (evolution strategies, genetic algorithms, ant colony optimization, scatter search with path relinking).

The third part of the book presents innovative approaches that are both of general applicability and not derived from adaptations of previous metaheuristics. These are algorithms that were explicitly designed based on mathematical models but with heuristic objectives and represent the current state of the art of a still lively research area, as testified by the variety of the proposed approaches.

A few of them pre-date the introduction of the term matheuristic but share its common objective. Very large-scale neighborhood search (a denotation that, when taken in its least normative acception, could be used almost as a synonym of the term matheuristics), dynasearch, RINS, local branching, and beam search are among them. Other methods were designed and presented in the milieu of matheuristics and include kernel search, the corridor method, decomposition-based heuristics, and the fore-and-back heuristic.

All these methods are proposed as pseudocodes suitable for any combinatorial optimization problem and are then detailed specifically for the GAP. The pseudocode for the GAP has actually been implemented in C++ and run on the common simple GAP instance, so that significant events appearing in the run traces can be commented in the text, and the runs can be replicated. A possibly arguable choice we made when reporting the run traces is to keep the relevant variables indexed as in the C++ code, thus as in the results it provides. This means that arrays and matrices are indexed from 1 throughout the text when in the context of formulations, in coherence with most optimization literature, but are indexed from 0 in the trace logs and in their comments.

The code is available at the URL https://github.com/maniezzo/MatheuristicsGAP as plain C++ and is provided with an optional Visual Studio project, in the future hopefully along with encompassing projects for other IDEs. The code can be directly applied to any GAP instance encoded according to the JSON format specified in the first section of the book.

Every software solution can inherently be expanded and updated, and this is all the more true for the accompanying code of this text, which is based on the code written only for teaching purposes by one of the authors, and as such it has been mainly validated only on simple instances. The code is purposefully kept at minimum complexity. We are quite aware that much more efficient implementations are possible by the inclusion of further elements. We leave the effort of optimizing performance to the interested students and provide our codebase only as a skeleton, a basement upon which possibly building even marketable commodities, besides further research achievements. We are, however, committed to maintain the repository, and we hope that whoever finds bugs, inaccuracies, or improvements that do not imply significant additions will contribute to it.

References

Boldyreff AW (1955) Determination of the maximal steady state flow of traffic through a railroad network. J Oper Res Soc Am 3:443–465

Boschetti MA, Maniezzo V, Roffilli M, Bolufé Röhler A (2009) Matheuristics: Optimization, simulation and control. In: Blesa M, Blum C, Di Gaspero L, Roli A, Sampels M, Schaerf A (eds) Hybrid metaheuristics, HM 2009. Lecture notes in computer science, vol 5818. Springer

Croes GA (1958) A method for solving traveling-salesman problems. Operations Research 6:791–812

Fischetti M, Fischetti M (2018) Matheuristics. In: Handbook of heuristics, vol 1–2, pp 121–153

Lodi A (2013) The heuristic (dark) side of MIP solvers. In: Talbi E (ed) Hybrid metaheuristics. Studies in computational intelligence, vol 434. Springer, pp 272–294

Maniezzo V (2006) Matheuristics 2006 conference web portal. https://astarte.csr.unibo.it/Matheuristics2006/

Maniezzo V, Stützle T, Voß S (eds) (2009a) Matheuristics: Hybridizing metaheuristics and mathematical programming. Annals of information systems, vol 10. Springer

Maniezzo V, Voß S, Hansen P, (eds) (2009b) Special issue on mathematical contributions to metaheuristics. J Heuristics 15:197–199

Maniezzo V, Stützle T (eds) (2020) Special issue: Matheuristics and metaheuristics. Int Trans Oper Res 27:1

Morton G, Land A (1955) A contribution to the "travelling-salesman" problem. J Roy Stat Soc B 17:185–194

Polya G (1945) How to solve it. Princeton University Press

Puchinger J, Raidl GR, Pirkwieser S (2009) Metaboosting: Enhancing integer programming techniques by metaheuristics. In: Maniezzo V, Stützle T, Voß S (eds) Matheuristics. Hybridizing metaheuristics and mathematical programming. Annals of information systems. Springer

Rosenfeld L (1956) Unusual problems and their solutions by digital computer techniques. In: Proceedings of the western joint computer conference, San Francisco, CA, pp 79–82

Russell B (1948) Human knowledge: Its scope and limits, Section: Part II: Language, Chapter I: The uses of language, vol 60. Simon and Schuster, New York

Schrijver A (2005) On the history of combinatorial optimization (till 1960). In: Aardal K, Nemhauser G, Weismantel L (eds) Handbook of discrete optimization. Elsevier, pp 1–68

Thorndike RL (1950) The problem of the classification of personnel. Psychometrika 15:215–235

Tolstoi AN (1930) Methods of finding the minimal total kilometrage in cargo transportation planning in space (in Russian). In: Transportation Planning, TransPress of the National Commissariat of Transportation, Moscow, vol I, pp 23–55

Cesena, Italy Vittorio Maniezzo
Cesena, Italy Marco Antonio Boschetti
Brussels, Belgium Thomas Stützle

Contents

Acronyms

ACO	Ant colony optimization
BS	Beam search
CO	Combinatorial optimization
DP	Dynamic programming
GAP	Generalized assignment problem
GAs	Genetic algorithms
GRASP	Greedy randomized adaptive search procedure
ILS	Iterated local search
IP	Integer programming
ILP	Integer linear programming
LP	Linear programming
LR	Lagrangian relaxation
JSON	JavaScript Object Notation
KS	Kernel search
MP	Mathematical programming
MIP	Mixed-integer programming
MILP	Mixed-integer linear programming
NLP	Nonlinear programming
RINS	Relaxation induced neighborhood search
SR	Surrogate relaxation
TSP	Traveling salesman problem
VLSNS	Very large-scale neighborhood search
VND	Variable neighborhood descent
VNS	Variable neighborhood search

List of Algorithms

Part I
Contextual Issues

Chapter 1
The Generalized Assignment Problem

1.1 Introduction

The generalized assignment problem (GAP) asks to assign n *clients* to m *servers* in such a way that the assignment cost is minimized, provided that all clients are assigned to a server and that the *capacity* of each server is not exceeded.

A mathematical model can be obtained by defining:[1]

- an index set $I = \{1, \ldots, m\}$ of servers;
- an index set $J = \{1, \ldots, n\}$ of clients;
- an assignment cost matrix $\mathbf{c} = [c_{ij}], i \in I, j \in J$, where each element c_{ij} specifies the cost for assigning client j to server i;
- a request amount matrix $\mathbf{q} = [q_{ij}], i \in I, j \in J$, where each element q_{ij} specifies the amount requested by client j to server i, in case it is to be serviced by it;
- a capacity vector $\mathbf{Q} = [Q_i], i \in I$, where each element Q_i specifies the capacity of server i.

The decision variables can be represented by a matrix $\mathbf{x} = [x_{ij}], i \in I, j \in J$, of binary variables, each one specifying whether client j is assigned to server i or not. Let $\mathbf{x}_i$ denote the ith row of matrix $\mathbf{x}$, $i \in I$, $\mathbf{x}^j$ the jth column, $j \in J$.

Formally, the problem can now make use of a knapsack of capacity Q_i for each server i and asks to assign each client to exactly one knapsack minimizing the total

[1]Vectors and matrices are denoted by small bold letters throughout the text, and these definitions depart from the standard for compliance with much of the literature on the GAP. Furthermore, as stated in Sect. 1.1, remind that all variables in this chapter are indexed from 1, but in the computational traces they will appear as indexed from 0.

V. Maniezzo et al., *Matheuristics*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-70277-9_1

assignment cost, without exceeding the server capacity:

$$(GAP) \qquad z_{GAP} = \min \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \tag{1.1}$$

$$s.t. \quad \sum_{i \in I} x_{ij} = 1, \qquad j \in J \tag{1.2}$$

$$\sum_{j \in J} q_{ij} x_{ij} \leq Q_i, \qquad i \in I \tag{1.3}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I, j \in J \tag{1.4}$$

The objective function (1.1) minimizes the assignment cost, constraints (1.2) are often called *assignment constraints* (more properly, they are *semi-assignment constraints*), constraints (1.3) are the *capacity constraints*, and constraints (1.4) are the *integrality constraints*.

In a feasible solution there is only one element valued 1 in any column $\mathbf{x}^j$, $j \in J$: let σ_j be the index of the corresponding row, i.e., $\sigma_j = i$ if $x_{ij} = 1\}$. A feasible solution can thus be represented also by a vector $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_n)$.

The problem can appear also in its maximization version, substituting the cost matrix **C** with a profit matrix **P**, and in its continuous version, substituting the integrality constraints (1.4) with the bounding constraints $0 \leq x_{ij} \leq 1, i \in I, j \in J$. The minimization and maximization versions are equivalent and can be trivially transformed into one another, while the continuous version provides a bound to its integer counterpart (e.g., a lower bound for a minimization problem).

An example of a GAP instance, named *example8x3*, which will be used throughout this book to present the different solution algorithms that will be covered, is shown in Fig. 1.1. In this instance we have $J = \{1, 2, 3, 4, 5, 6, 7, 8\}$, thus $n = 8$ clients, and $I = \{1, 2, 3\}$, thus $m = 3$ servers. The server capacities are 160, 90, and 70, respectively, thus $\mathbf{Q} = (160, 90, 70)$. The assignment costs and client requests are presented columnwise next to the corresponding client, thus, for example, client 1 requests 48 from server 1, or 38 from server 2, or 28 from server 3. Assigning it to server 1 costs 10, to server 2 costs 22, and to server 3 costs 60.

As an aside, all instances referenced in this book, along with all major GAP test sets, are available from the server available in (Maniezzo 2019) in JSON (JavaScript Object Notation) format. Specifically, this example can be downloaded from the URL http://astarte.csr.unibo.it/gapdata/homemade/example8x3.json obtaining this file content

```
{ "name":    "example8x3",
  "numcli":  8,
  "numserv": 3,
  "cost":    [[10,11,12,13,14,15,16,17],
             [22,24,26,28,30,32,34,36],
             [60,64,68,72,76,80,84,88]],
  "req":      [[48,47,46,45,44,43,42,41],
             [38,37,36,35,34,33,32,31],
             [28,27,26,25,24,23,22,21]],
  "cap":      [160,90,70]
}
```

which is a JSON object consisting of 6 fields, the first (*name*) being the name of the instance, the second (*numcli*) is n, the number of clients, the third (*numserv*) is m, the number of servers, the fourth (*cost*) is the row-wise representation of the cost matrix **c**, the fifth (*req*) is the row-wise representation of the requests matrix **q**, and the sixth (*cap*) is the capacity vector **Q**. As reported in the download site, the optimal cost for this instance is 325, while its linear relaxation provides a lower bound of 231.45, thus certifying an integrality gap of over 40%.

It is interesting to analyze the search space of this instance. Some relevant data is presented in Table 1.1.

Formulation GAP makes use of $n \times m$ binary decision variables, and therefore it induces a search space of size $2^{n \times m}$. An alternative formulation is based on n integer variables constrained to take values in the set $\{1, \dots, m\}$ and implicitly enforces

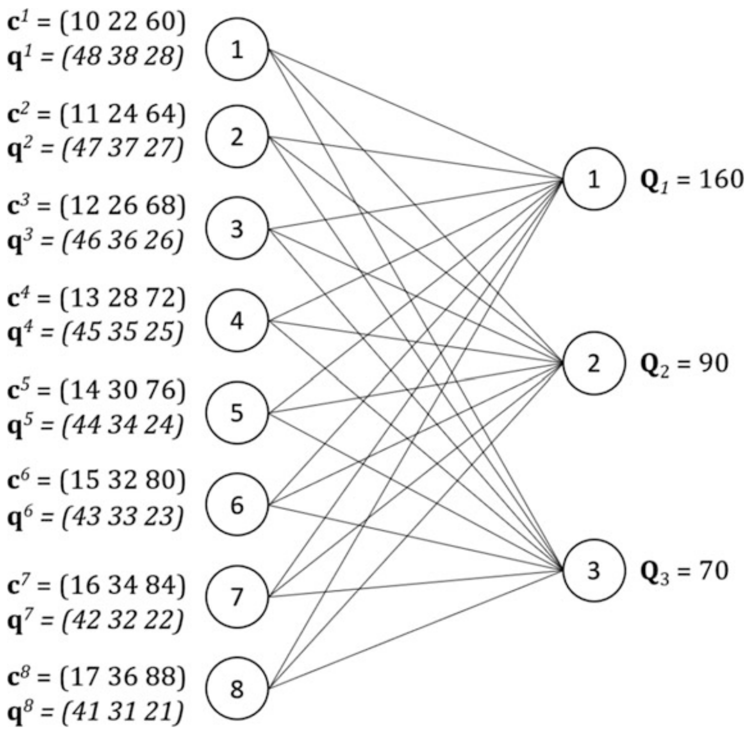


Fig. 1.1 Example: instance *example8x3*

Table 1.1 The search space of instance example3x8

- The number of solutions is $2^{24} = 16777216$, by making use of formulation (GAP).
- The number of solutions feasible for the assignments only is $3^8 = 6561$.
- The number of feasible solutions is 110, as determined by an exhaustive scan of all assignment-feasible solutions.
- The range of costs of the feasible solutions goes from 325 to 343 and their distribution is best fit by a binomial distribution (see Fig. 1.2).
- The descriptive statistics of the cost distribution for the feasible solutions are: mean = 332.91, median = 332.5, mode = 331, and std. deviation = 4.17 (thus, it is roughly normal).

the assignment constraints, permitting a much more compact representation of solutions. We will use either.

Below we show the two alternative representations of one same solution of instance *example3x8*.

$$\boldsymbol{\sigma} = (1, 1, 1, 2, 2, 3, 3, 3) \qquad \mathbf{x} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

The generalized assignment problem appears often in the literature, both in itself and as a subproblem in some specialized situations. It was first introduced as a model for applications that "include assigning software development tasks to programmers, assigning jobs to computers in computer networks, scheduling variable length television commercials into time slots, and scheduling payments on accounts where contractual agreements specify 'lump sum' payments. Other applications include fixed charge plant location models in which customer requirements must be satisfied by a single plant, and communication network design models with certain node capacity constraints" (Ross and Soland 1975). Actual reports have been presented for routing problems, early examples using GAP as a subproblem that asks to assign customers to vehicles in the framework of a vehicle routing problem, or using GAP to model the assignment of batch jobs to flexible manufacturing systems (FMSs), where the GAP is a subproblem, in a production planning context. Reported applications became then very varied, a non-exhaustive list includes telecommunication protocols, allocation of workers to departments, batch loading and scheduling in production planning, job assignment, covering assignment, load balancing, material supply, database partitioning and allocation to processors, and healthcare.

1.1.1 NP-hardness

Problem GAP is NP-hard in the strong sense therefore, most importantly from the viewpoint of heuristic solving, even the feasibility question is NP-complete. A generic instance in fact is not guaranteed to be feasible because of the capacity

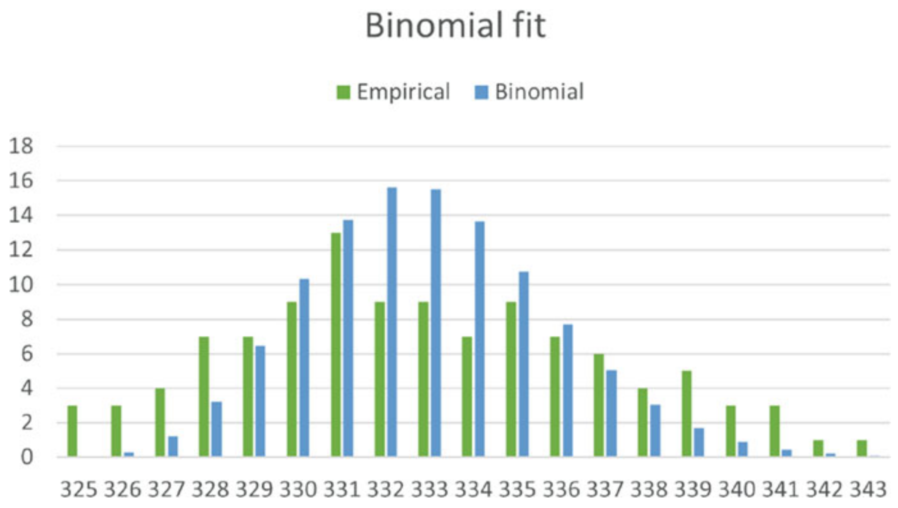


Fig. 1.2 Solution cost distribution for instance *example8x3*

constraints (1.3), while assignment constraints alone would always be satisfiable. In general, when capacity constraints are not extremely loose, the feasibility region is small when compared to the solution set, as was shown for the instance *example8x3*, see Table 1.1.

To prove the NP-completeness of the feasibility question, it is easy to design a reduction of an instance of problem *Partition* to an instance of GAP. Problem Partition, reputedly NP-complete, asks to determine if it is possible to partition a set of numbers into two subsets, such that the sum of the elements of each subset is the same. This can be reduced to GAP by associating each number with a client and introducing two servers, each with a capacity equal to half of the sum of the numbers. If all assignment costs are the same and each client requests to each server an amount equal to its associated number, determining the feasibility of the GAP corresponds to solving problem Partition. Note that costs are irrelevant for the feasibility problem defined on the original instance.

It is moreover possible to convert the feasibility problem into an optimization one, which is guaranteed to admit a feasible solution by adding to the original problem GAP a dummy high-cost server capable of accommodating any unassigned client. If the dummy server is not used, the original instance is feasible, if it is necessary to resort to it, the original instance is infeasible. However, to ascertain this, it is necessary to solve to optimality the extended instance, thus incurring in the NP complexity.

1.1.2 Special Cases and Extensions

The GAP is a fundamental problem in combinatorial optimization and, given its structural simplicity and significance, it appears in restricted or generalized form in several different situations.

Two obvious special cases appear when there is only one server, in which case we have a knapsack problem (adding a dummy variable for unassigned items), or when the server capacities and client requests are all equal to 1, in which case it can be reconducted to a matching in a bipartite graph.

Looking at formulation GAP, another immediate special case appears when each client incurs in equal costs and has the same requests for any server (i.e., $c_{ij} = c_j$ and $q_{ij} = q_j, i \in I, j \in J$), in which case the problem becomes a multiple knapsack problem with equality constraints (again adding a dummy variable for unassigned items), whereas if we have server-specific costs but equal requests we deal with a capacitated facility location without fixed costs.

GAP generalizations have been proposed along several different directions. Among the best known ones from the literature, we point out the following ones.

One of the first extensions proposed considered mutual interactions among clients assigned to one same server. Formally, it substituted inequalities (1.3) by $\sum_{j \in J} q_{ij}x_{ij} + \sum_{h \in J} k_{ih}(\prod_{l \in J} x_{il}) \leq Q_i, i \in I$, thus introducing a nonlinearity in the capacity constraints; $k_{ih}, i \in I, h \in J$, are coefficients that model the capacity interaction between x_{il} variables and can be positive, negative, or zero.

In the *bottleneck* version of the GAP, the objective function (1.1) is substituted by $\min \left\{\max_{ij} c_{ij}x_{ij}\right\}$. This permits to minimize the worst assignment in the solution, thereby achieving a good equity in the assignment costs among the different clients.

Another early extension accounted for multiple resource types, so that the assignment of a client to a server implied the consumption of a number of different resources. This gives rise to the so-called multi-resource or multi-constraint GAP. Formally, inequalities (1.3) were substituted by $\sum_{j \in J} q_{ijk}x_{ij} \leq Q_{ik}, i \in I, k \in K$, where K is the set of resource types.

A somewhat similar extension was studied by the name of *Multilevel GAP*. In this case, the servers can operate at different efficiency levels, where typically a higher efficiency implies a higher cost. The decision variables are now defined with 3 indices, $x_{ijk}, i \in I, j \in J, k \in K$, where K is the set of the efficiency levels, and cost and request coefficients get the same dimensionality, so that, for example, constraints (1.3) become $\sum_{j \in J} \sum_{k \in K} q_{ijk}x_{ijk} \leq Q_i, i \in I$, and a further constraint set forces each server to operate at a single efficiency level for servicing a client request.

The so-called elastic GAP extension permits limited violations of the server capacities, but violations are penalized. As usual in these cases, two additional decision variables are added for each constraint that can be violated. In the case of this particular application to the GAP, the capacity constraints were modeled as equalities, and two auxiliary continuous upper bounded variables were introduced for each capacity constraint, a deficiency u_i and an excess v_i. The objective function

(1.1) then becomes $\sum_{i\in I}\sum_{j\in J} c_{ij}x_{ij} + \sum_{i\in I}(d_i u_i + e_i v_i)$, with appropriate d_i and e_i cost coefficients, and constraints 1.3 become $\sum_{j\in J} q_{ij}x_{ij} + u_i - v_i = Q_i, i \in I = \{1, \dots, m\}$.

Stochastic versions of the GAP have also been studied, where uncertainty could be associated either with the server capacities or with the presence of the clients. This is, in the first case, equivalent to say that the capacity of the server is not precisely known and, in the second case, that the client can be present or not; in case they are present, they have known requests and costs.

In the *quadratic* version of the GAP, it is assumed to have a given "traffic" among clients, $t_{jk}, j, k \in J$, along with a given "distance" among servers, $d_{ih}, i, h \in I$. Besides assignment costs, we also want to have intense traffic on "short" connections. This reflects on an augmented objective function, which becomes $\sum_{i\in I}\sum_{j\in J} c_{ij}x_{ij} + \sum_{i\in I}\sum_{h\in I}\sum_{j\in J}\sum_{k\in J} t_{jk}d_{ih}x_{ij}x_{hk}$.

Finally, a *multi-objective*, specifically, bi-objective version of the GAP was studied, where the two objectives could be time and cost of travels, or processing time and cost of the assignment of a job to a machine, where, for example, the decisionmaker wants to minimize at the same time the total execution time and the total cost for the assignment of jobs to machines. The problem is solved by an LP-based heuristic for the case when both objective functions are linear.

1.2 Lower Bounds

The relevance and the structural simplicity of the GAP have made it a primary candidate for studying and testing different bounding approaches. As several lower bound contributions are relevant also for the design of some of the matheuristic codes presented in this book, we introduce here the most impactful ones.

1.2.1 Linear Relaxation

The linear relaxation of formulation GAP (*LP bound*) is obtained by simply substituting constraints (1.4) with $x_{ij} \geq 0, i \in I, j \in J$. Its application leads to a split assignment of some client. It is known that "the number of non-unique assignments is less than or equal to the number of machines (*servers*) of which the capacity is used completely" (Cattrysse and Van Wassenhove 1992).

It is possible to design instances with unbounded integrality gap, an example being the $n = 2$, $m = 2$ instance with costs $[[0, 0], [(k + 1), (k + 1)]]$, requests $[[k + 1, k + 1], [1, 1]]$, and capacities $[2k + 1, 2]$ (see Sect. 1.1 for the data format), which has a percent integrality gap of at least k for any positive k.

Intuitively, when the instances have many more clients than servers and loose capacity constraints, the LP bound can be tight, whereas when capacity constraints become critical, the LP bound is not informative, and the LP solution can be completely unrelated to any optimal integer solution.

The LP bound for the *example8x*3 is the solution of the following formulation:

$$
\begin{aligned}
z_{LP} = min\ & 10x_{11} + 11x_{12} + 12x_{13} + 13x_{14} + 14x_{15} + 15x_{16} + 16x_{17} + 17x_{18} + \\
& 22x_{21} + 24x_{22} + 26x_{23} + 28x_{24} + 30x_{25} + 32x_{26} + 34x_{27} + 36x_{28} + \\
& 60x_{31} + 64x_{32} + 68x_{33} + 72x_{34} + 76x_{35} + 80x_{36} + 84x_{37} + 88x_{38} \\
s.t.\ & x_{11} + x_{21} + x_{31} = 1 \\
& x_{12} + x_{22} + x_{32} = 1 \\
& x_{13} + x_{23} + x_{33} = 1 \\
& x_{14} + x_{24} + x_{34} = 1 \\
& x_{15} + x_{25} + x_{35} = 1 \\
& x_{16} + x_{26} + x_{36} = 1 \\
& x_{17} + x_{27} + x_{37} = 1 \\
& x_{18} + x_{28} + x_{38} = 1 \\
& 48x_{11} + 47x_{11} + 46x_{11} + 45x_{11} + 44x_{11} + 43x_{11} + 42x_{11} + 41x_{11} \leq 160 \\
& 38x_{21} + 37x_{21} + 36x_{21} + 35x_{21} + 34x_{21} + 33x_{21} + 32x_{21} + 31x_{21} \leq 90 \\
& 28x_{31} + 27x_{31} + 26x_{31} + 25x_{31} + 24x_{31} + 23x_{31} + 22x_{31} + 21x_{31} \leq 70 \\
& x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18} \geq 0 \\
& x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{27}, x_{28} \geq 0 \\
& x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}, x_{37}, x_{38} \geq 0
\end{aligned}
$$

An optimal LP solution, of cost 231.45 and with the values rounded to the third decimal, is as follows:

$$
\mathbf{x} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0.773 & 1 & 1 & 1 \\ 0 & 0.305 & 1 & 1 & 0.227 & 0 & 0 & 0 \\ 1 & 0.695 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}
$$

Notice that the number of clients with fractional assignments is equal to the number of servers minus one.

1.2.2 Lagrangian Relaxation

Lower bounds for the GAP can be obtained by means of Lagrangian relaxation by either dualizing constraints (1.2) or constraints (1.3) or both. However, dualizing both is an inefficient choice from a computational point of view, providing no better solution, with higher computational effort than keeping the assignment constraints, having more parameters to fit.

If we relax the assignment constraints (1.2), we are left with $|I|$ knapsack problems, one for each server. Since the subproblem is not polynomial, solving the corresponding Lagrangian dual can produce a bound better than the LP one.

Formally, this relaxation associates an unrestricted Lagrangian penalty λ_j, $j \in J$, to each constraint (1.2). The problem becomes the following problem LGAPA:

$$(LGAPA) \quad z_{LGAPA} = \min \sum_{i \in I} \sum_{j \in J} (c_{ij} + \lambda_j) x_{ij} - \sum_{j \in J} \lambda_j \tag{1.5}$$

$$s.t. \sum_{j \in J} q_{ij} x_{ij} \leq Q_i, \qquad i \in I \tag{1.6}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I, j \in J \tag{1.7}$$

$$\lambda_j \ unrestricted, \qquad j \in J \tag{1.8}$$

Given the penalty vector $\boldsymbol{\lambda} = (\lambda_j)$, problem LGAPA decomposes into m 0-1 knapsack problems, one for each server $i \in I$.

In the case of *example8x3*, this formulation would be instantiated as follows:

$$\begin{aligned}
z_{LRA} = \min\ & (10 + \lambda_1)x_{11} + (11 + \lambda_2)x_{12} + (12 + \lambda_3)x_{13} + (13 + \lambda_4)x_{14} + \\
& (14 + \lambda_5)x_{15} + (15 + \lambda_6)x_{16} + (16 + \lambda_7)x_{17} + (17 + \lambda_8)x_{18} + \\
& (22 + \lambda_1)x_{21} + (24 + \lambda_2)x_{22} + (26 + \lambda_3)x_{23} + (28 + \lambda_4)x_{24} + \\
& (30 + \lambda_5)x_{25} + (32 + \lambda_6)x_{26} + (34 + \lambda_7)x_{27} + (36 + \lambda_8)x_{28} + \\
& (60 + \lambda_1)x_{31} + (64 + \lambda_2)x_{32} + (68 + \lambda_3)x_{33} + (72 + \lambda_4)x_{34} + \\
& (76 + \lambda_5)x_{35} + (80 + \lambda_6)x_{36} + (84 + \lambda_7)x_{37} + (88 + \lambda_8)x_{38} + \\
& - \lambda_1 - \lambda_2 - \lambda_3 - \lambda_4 - \lambda_5 - \lambda_6 - \lambda_7 - \lambda_8 \\
s.t.\ & 48x_{11} + 47x_{11} + 46x_{11} + 45x_{11} + 44x_{11} + 43x_{11} + 42x_{11} + 41x_{11} \leq 160 \\
& 38x_{21} + 37x_{21} + 36x_{21} + 35x_{21} + 34x_{21} + 33x_{21} + 32x_{21} + 31x_{21} \leq 90 \\
& 28x_{31} + 27x_{31} + 26x_{31} + 25x_{31} + 24x_{31} + 23x_{31} + 22x_{31} + 21x_{31} \leq 70
\end{aligned}$$

$$x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18} \in \{0, 1\}$$

$$x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{27}, x_{28} \in \{0, 1\}$$

$$x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}, x_{37}, x_{38} \in \{0, 1\}$$

Solving this instance, with a penalty vector (values rounded to the third decimal):

$$\boldsymbol{\lambda} = [111.139, 111.792, 113.420, 114.66, 116.009, 116.630, 117.328, 118.261]$$

as determined by solving the corresponding *Lagrangian dual* problem, yields a lower bound value of 324.077, corresponding to the solution:

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

which is infeasible due to the assignments, as the fourth and fifth clients are assigned to more than one server. However, the bound is capable of proving the optimality of a possible heuristic solution of cost 325, if this last was obtained by some other means. This is because the integrality gap would be less than 1; therefore, no integer solution could exist of cost less than 325.

If we relax the capacity constraints, we are left with only the assignment constraints, thus with a totally unimodular constraint matrix.

This relaxation associates a non-negative Lagrangian penalty λ_i, $i \in I$, to each constraint (1.3). The problem becomes the following problem LGAPC:

$$(LGAPC) \quad z_{LGAPC} = \min \sum_{i \in I} \sum_{j \in J} (c_{ij} + \lambda_i q_{ij}) x_{ij} - \sum_{i \in I} \lambda_i Q_i \tag{1.9}$$

$$\text{s.t.} \sum_{i \in I} x_{ij} = 1, \qquad j \in J \tag{1.10}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I, j \in J \tag{1.11}$$

$$\lambda_i \geq 0, \qquad i \in I \tag{1.12}$$

Given the penalty vector $\boldsymbol{\lambda} = (\lambda_i)$, problem LGAPC can be trivially solved by setting to 1, for each $j \in J$, the variable x_{ij} for which the cost $(c_{ij} + \lambda_i q_{ij})$ is minimum, $i \in I$. All remaining x_{ij} are set to 0. The subproblem can thus be trivially solved by inspection, by assigning each client to its least cost server, thus the Lagrangian bound cannot be better than the LP bound.

In the example, this formulation would be instantiated as follows:

$$
\begin{aligned}
z_{LRC} = \min\ & (10+48\lambda_1)x_{11} + (11+47\lambda_1)x_{12} + (12+46\lambda_1)x_{13} + (13+45\lambda_1)x_{14} + \\
& (14+44\lambda_1)x_{15} + (15+43\lambda_1)x_{16} + (16+42\lambda_1)x_{17} + (17+41\lambda_1)x_{18} + \\
& (22+38\lambda_2)x_{21} + (24+37\lambda_2)x_{22} + (26+36\lambda_2)x_{23} + (28+35\lambda_2)x_{24} + \\
& (30+34\lambda_2)x_{25} + (32+33\lambda_2)x_{26} + (34+32\lambda_2)x_{27} + (36+31\lambda_2)x_{28} + \\
& (60+28\lambda_3)x_{31} + (64+27\lambda_3)x_{32} + (68+26\lambda_3)x_{33} + (72+25\lambda_3)x_{34} + \\
& (76+24\lambda_3)x_{35} + (80+23\lambda_3)x_{36} + (84+22\lambda_3)x_{37} + (88+21\lambda_3)x_{38} + \\
& -160\lambda_1 - 90\lambda_2 - 70\lambda_3 \\
s.t.\ & x_{11} + x_{21} + x_{31} = 1 \\
& x_{12} + x_{22} + x_{32} = 1 \\
& x_{13} + x_{23} + x_{33} = 1 \\
& x_{14} + x_{24} + x_{34} = 1 \\
& x_{15} + x_{25} + x_{35} = 1 \\
& x_{16} + x_{26} + x_{36} = 1 \\
& x_{17} + x_{27} + x_{37} = 1 \\
& x_{18} + x_{28} + x_{38} = 1 \\
& x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18} \in \{0, 1\} \\
& x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{27}, x_{28} \in \{0, 1\} \\
& x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}, x_{37}, x_{38} \in \{0, 1\} \\
& \lambda_1, \lambda_2, \lambda_3 \geq 0
\end{aligned}
$$

Solving this instance, with a penalty vector (values rounded to the third decimal)

$$\boldsymbol{\lambda} = [1.199, 1.081, 0]$$

as determined by solving the corresponding *Lagrangian dual* problem, yields a lower bound value of 231.445, corresponding to the solution:

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

which is infeasible for the capacities, as it exceeds the capacity of server 1, but which yields a lower bound equivalent to the linear relaxation of the instance.

1.2.3 Lagrangian Decomposition

Another way to compute a Lagrangian bound is by applying Lagrangian decomposition . This was used, in the case of GAP, with the following reformulation:

$$(LDGAP) \qquad z_{LDGAP} = \min \alpha \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} + \beta \sum_{i \in I} \sum_{j \in J} c_{ij} y_{ij} \tag{1.13}$$

$$s.t. \quad \sum_{i \in I} x_{ij} = 1, \qquad j \in J \tag{1.14}$$

$$\sum_{j \in J} q_{ij} y_{ij} \leq Q_i, \qquad i \in I \tag{1.15}$$

$$x_{ij} = y_{ij}, \qquad i \in I, j \in J \tag{1.16}$$

$$x_{ij}, y_{ij} \in \{0, 1\}, \qquad i \in I, j \in J \tag{1.17}$$

Here, the capacity constraints are formulated over **x** variables and the assignment constraints over **y**, with the added condition that the two sets of variables must take the same values (1.16). It is furthermore possible to gauge the impact on the solution cost of the two sets of constraints by means of parameters α and β.

It is now possible to relax in a Lagrangian fashion constraints (1.16), obtaining a subproblem in **y** variables asking to find an otherwise unrestricted least cost client assignment, and a subproblem in **x** variables, which asks to solve $|I|$ knapsack problems.

For the convex combination $\alpha + \beta = 1$ and for the optimal Lagrangian multipliers, the bound obtained by Lagrangian decomposition can be stronger than that obtained by a straight Lagrangian relaxation of one of the constraint sets.

In the example, this decomposition would lead to the following subproblems:

$$\begin{aligned} z_{LDA} = \min \ & (\alpha 10 + \lambda_{11})x_{11} + (\alpha 11 + \lambda_{12})x_{12} + (\alpha 12 + \lambda_{13})x_{13} + (\alpha 13 + \lambda_{14})x_{14} + \\ & (\alpha 14 + \lambda_{15})x_{15} + (\alpha 15 + \lambda_{16})x_{16} + (\alpha 16 + \lambda_{17})x_{17} + (\alpha 17 + \lambda_{18})x_{18} + \\ & (\alpha 22 + \lambda_{21})x_{21} + (\alpha 24 + \lambda_{22})x_{22} + (\alpha 26 + \lambda_{23})x_{23} + (\alpha 28 + \lambda_{24})x_{24} + \\ & (\alpha 30 + \lambda_{25})x_{25} + (\alpha 32 + \lambda_{26})x_{26} + (\alpha 34 + \lambda_{27})x_{27} + (\alpha 36 + \lambda_{28})x_{28} + \\ & (\alpha 60 + \lambda_{31})x_{31} + (\alpha 64 + \lambda_{32})x_{32} + (\alpha 68 + \lambda_{33})x_{33} + (\alpha 72 + \lambda_{34})x_{34} + \\ & (\alpha 76 + \lambda_{35})x_{35} + (\alpha 80 + \lambda_{36})x_{36} + (\alpha 84 + \lambda_{37})x_{37} + (\alpha 88 + \lambda_{38})x_{38} \\ s.t. \ & x_{11} + x_{21} + x_{31} = 1 \\ & x_{12} + x_{22} + x_{32} = 1 \\ & x_{13} + x_{23} + x_{33} = 1 \\ & x_{14} + x_{24} + x_{34} = 1 \end{aligned}$$

$$
\begin{aligned}
& x_{15} + x_{25} + x_{35} = 1 \\
& x_{16} + x_{26} + x_{36} = 1 \\
& x_{17} + x_{27} + x_{37} = 1 \\
& x_{18} + x_{28} + x_{38} = 1 \\
& x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18} \in \{0, 1\} \\
& x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{27}, x_{28} \in \{0, 1\} \\
& x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}, x_{37}, x_{38} \in \{0, 1\}
\end{aligned}
$$

and

$$
\begin{aligned}
z_{LDC} = \min\ & (\beta 10 - \lambda_{11})y_{11} + (\beta 11 - \lambda_{12})y_{12} + (\beta 12 - \lambda_{13})y_{13} + (\beta 13 - \lambda_{14})y_{14} + \\
& (\beta 14 - \lambda_{15})y_{15} + (\beta 15 - \lambda_{16})y_{16} + (\beta 16 - \lambda_{17})y_{17} + (\beta 17 - \lambda_{18})y_{18} + \\
& (\beta 22 - \lambda_{21})y_{21} + (\beta 24 - \lambda_{22})y_{22} + (\beta 26 - \lambda_{23})y_{23} + (\beta 28 - \lambda_{24})y_{24} + \\
& (\beta 30 - \lambda_{25})y_{25} + (\beta 32 - \lambda_{26})y_{26} + (\beta 34 - \lambda_{27})y_{27} + (\beta 36 - \lambda_{28})y_{28} + \\
& (\beta 60 - \lambda_{31})y_{31} + (\beta 64 - \lambda_{32})y_{32} + (\beta 68 - \lambda_{33})y_{33} + (\beta 72 - \lambda_{34})y_{34} + \\
& (\beta 76 - \lambda_{35})y_{35} + (\beta 80 - \lambda_{36})y_{36} + (\beta 84 - \lambda_{37})y_{37} + (\beta 88 - \lambda_{38})y_{38} \\
s.t.\ & 48y_{11} + 47y_{11} + 46y_{11} + 45y_{11} + 44y_{11} + 43y_{11} + 42y_{11} + 41y_{11} \le 160 \\
& 38y_{21} + 37y_{21} + 36y_{21} + 35y_{21} + 34y_{21} + 33y_{21} + 32y_{21} + 31y_{21} \le 90 \\
& 28y_{31} + 27y_{31} + 26y_{31} + 25y_{31} + 24y_{31} + 23y_{31} + 22y_{31} + 21y_{31} \le 70 \\
& y_{11}, y_{12}, y_{13}, y_{14}, y_{15}, y_{16}, y_{17}, y_{18} \in \{0, 1\} \\
& y_{21}, y_{22}, y_{23}, y_{24}, y_{25}, y_{26}, y_{27}, y_{28} \in \{0, 1\} \\
& y_{31}, y_{32}, y_{33}, y_{34}, y_{35}, y_{36}, y_{37}, y_{38} \in \{0, 1\}
\end{aligned}
$$

Solving this instance, with the penalty vector Λ reported in (1.18), which was determined by solving the corresponding *Lagrangian dual* problem (values rounded to the second decimal)

$$
\Lambda = \begin{bmatrix} 106.71 & 107.42 & 107.85 & 108.64 & 109.10 & 109.61 & 109.55 & 110.17 \\ 100.09 & 99.99 & 100.84 & 101.74 & 101.90 & 102.20 & 102.25 & 103.84 \\ 86.68 & 80.52 & 79.92 & 78.40 & 78.03 & 76.93 & 75.82 & 74.60 \end{bmatrix} \tag{1.18}
$$

and with parameters $\alpha = \beta = 0.5$, yields a lower bound value of 324.205, corresponding to the solutions of (1.19) and (1.20).

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{1.19}$$

$$\mathbf{y} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{1.20}$$

The combination of these two solutions is infeasible, but it is nevertheless capable of proving the optimality of a possible heuristic solution of cost 325, if this last was obtained by some other means.

1.2.4 Surrogate Relaxation

Surrogate relaxation is less common than the techniques discussed above, but its principle is simple, as it suggests to replace a set of constraints with their linear combination, i.e., their weighted sum. Determining the best set of surrogate multipliers, that is the one that maximizes the obtained lower bound, gives rise to the *surrogate dual* problem. An important theoretical result is that surrogate duality gaps are at least as small as Lagrangian duality gaps, possibly smaller.

A possible surrogate relaxation for the GAP yields the following reformulation:

$$(SR) \qquad z_{SR} = \min \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} \tag{1.21}$$

$$s.t. \quad \sum_{i \in I} x_{ij} = 1, \qquad j \in J \tag{1.22}$$

$$\sum_{i \in I} \sum_{j \in J} \mu_i q_{ij} x_{ij} \leq \sum_{i \in I} \mu_i Q_i, \tag{1.23}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I, j \in J \tag{1.24}$$

where the $\mu_i, i = 1, \ldots, m$ are the nonrestricted surrogate coefficients.

In the example, this formulation would be instantiated as follows:

$$\begin{aligned} z_{LRC} = \min\ & 10x_{11} + 11x_{12} + 12x_{13} + 13x_{14} + 14x_{15} + 15x_{16} + 16x_{17} + 17x_{18} + \\ & 22x_{21} + 24x_{22} + 26x_{23} + 28x_{24} + 30x_{25} + 32x_{26} + 34x_{27} + 36x_{28} + \\ & 60x_{31} + 64x_{32} + 68x_{33} + 72x_{34} + 76x_{35} + 80x_{36} + 84x_{37} + 88x_{38} \end{aligned}$$

$$
\begin{aligned}
s.t.\ & x_{11} + x_{21} + x_{31} = 1 \\
& x_{12} + x_{22} + x_{32} = 1 \\
& x_{13} + x_{23} + x_{33} = 1 \\
& x_{14} + x_{24} + x_{34} = 1 \\
& x_{15} + x_{25} + x_{35} = 1 \\
& x_{16} + x_{26} + x_{36} = 1 \\
& x_{17} + x_{27} + x_{37} = 1 \\
& x_{18} + x_{28} + x_{38} = 1 \\
& 48\mu_1 x_{11} + 47\mu_1 x_{11} + 46\mu_1 x_{11} + 45\mu_1 x_{11} + \\
& 44\mu_1 x_{11} + 43\mu_1 x_{11} + 42\mu_1 x_{11} + 41\mu_1 x_{11} + \\
& 38\mu_2 x_{21} + 37\mu_2 x_{21} + 36\mu_2 x_{21} + 35\mu_2 x_{21} + \\
& 34\mu_2 x_{21} + 33\mu_2 x_{21} + 32\mu_2 x_{21} + 31\mu_2 x_{21} + \\
& 28\mu_3 x_{31} + 27\mu_3 x_{31} + 26\mu_3 x_{31} + 25\mu_3 x_{31} + \\
& 24\mu_3 x_{31} + 23\mu_3 x_{31} + 22\mu_3 x_{31} + 21\mu_3 x_{31} \leq 160\mu_1 + 90\mu_2 + 70\mu_3 \\
& x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18} \in \{0, 1\} \\
& x_{21}, x_{22}, x_{23}, x_{24}, x_{25}, x_{26}, x_{27}, x_{28} \in \{0, 1\} \\
& x_{31}, x_{32}, x_{33}, x_{34}, x_{35}, x_{36}, x_{37}, x_{38} \in \{0, 1\}
\end{aligned}
$$

Solving this instance, with surrogate multipliers $\boldsymbol{\mu} = [0.5, 0.5, 0]$, which is a heuristic solution of the corresponding *surrogate dual* problem, yields a lower bound value of 240, corresponding to the solution:

$$
\mathbf{x} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}
$$

which is infeasible for the capacities, as it exceeds the capacity of server 1, but which produces a lower bound better than the LP bound.

1.2.5 Benders Decomposition

Benders decomposition provides yet another way to decompose the original GAP problem and obtain a lower bound. This is achieved by decomposing the problem into a Benders' *master problem* (BMP), which could fix some of its variables, and a

Benders' *subproblem* (BSP), whose dual permits to add the new so-called Benders' cuts to the master. The bound is obtained by solving MP to optimality.

This decomposition comes natural when there are two sets of variables, which is not the case of the GAP. In order to decompose the GAP, one possibility is to partition the decision variables, keeping some in the master and the remaining ones in the subproblem. A second decision to take is about which constraints to keep in BMP. As Benders' cuts will be added to BMP, most likely making it NP-hard, it is sensible to keep in BMP hard constraints, in our case capacity constraints.

A possible formulation of a decomposition that partitions the decision variables between those making reference to a subset I_1 of servers and those making reference to the servers in I_2, with $I_1 \cup I_2 = I$ and $I_1 \cap I_2 = \emptyset$, is as follows.

The master problem is

$$z_{BMP} = \min z \tag{1.25}$$

$$s.t.\ z \geq \sum_{i \in I_1} \sum_{j \in J} c_{ij} x_{ij} \tag{1.26}$$

$$\sum_{j \in J} q_{ij} x_{ij} \leq Q_i, \qquad i \in I_1 \tag{1.27}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I_1, j \in J \tag{1.28}$$

and the subproblem keeps all assignment constraints, plus the capacity constraints of the servers in I_2. Its formulation, where the variables set in BMP are denoted as $\bar{x}_{ij}$, $i \in I_1$, $j \in J$, is as follows:

$$(BSP) \qquad z_{BSP} = \min \sum_{i \in I_2} \sum_{j \in J} c_{ij} x_{ij} \tag{1.29}$$

$$s.t.\ \sum_{i \in I_2} x_{ij} = 1 - \sum_{i \in I_1} \bar{x}_{ij}, \qquad j \in J \tag{1.30}$$

$$\sum_{j \in J} q_{ij} x_{ij} \leq Q_i, \qquad i \in I_2 \tag{1.31}$$

$$x_{ij} \geq 0, \qquad i \in I_2, j \in J \tag{1.32}$$

The dual of BSP, denoted DSP, can be obtained by associating a free variable w_j to each constraint (1.30) and a nonpositive variable v_i to each constraint (1.31):

$$z_{DSP} = \max \sum_{j \in J} (1 - \sum_{i \in I_1} \bar{x}_{ij}) w_j + \sum_{i \in I_2} Q_i v_i \tag{1.33}$$

$$s.t.\ w_j + q_{ij} v_i \leq c_{ij}, \qquad i \in I_2, j \in J \tag{1.34}$$

$$v_i \leq 0, \qquad i \in I_2 \tag{1.35}$$

where w_j variables have a coefficient of 1 whenever the corresponding client j was not assigned to a server in I_1 (therefore, $\sum_{i \in I_1} \bar{x}_{ij} = 0$), and a coefficient of 0, therefore they do not appear in the sum, whenever the corresponding client j was assigned to a server in I_1 (therefore, $\sum_{i \in I_1} \bar{x}_{ij} = 1$).

Solving DSP to optimality yields the variable values $\bar{w}_j$, $j \in J$, and $\bar{v}_i$, $i \in I_2$. These values are used to generate the Benders' cut

$$z \geq \sum_{j \in J} \bar{w}_j + \sum_{i \in I_2} \bar{v}_i - \sum_{j \in J} \bar{w}_j x_{ij}$$

which will be added to MP to ascertain whether the bound can be improved.

In the case of the example of Fig. 1.1 with $I_1 = \{1\}$ and $I_2 = \{2, 3\}$, the BMP is

$$\begin{aligned} z_{BMP} = \min\ & z \\ s.t.\ & z - 10x_{11} - 11x_{12} - 12x_{13} - 13x_{14} - 14x_{15} - 15x_{16} - 16x_{17} - 17x_{18} \geq 0 \\ & 48x_{11} + 47x_{11} + 46x_{11} + 45x_{11} + 44x_{11} + 43x_{11} + 42x_{11} + 41x_{11} \leq 160 \\ & x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{16}, x_{17}, x_{18} \in \{0, 1\} \end{aligned}$$

and the dual of BSP at the first iteration (where all $\bar{x}_{ij}$ are zero) is

$$\begin{aligned} z_{BDP} = max\ & w_1 + w_2 + w_3 + w_4 + w_5 + w_6 + w_7 + w_8 + 90v_2 + 70v_3 \\ s.t.\ & w_1 + 38v_2 \leq 22; \quad w_2 + 37v_2 \leq 24 \\ & w_3 + 36v_2 \leq 26; \quad w_4 + 35v_2 \leq 28 \\ & w_5 + 34v_2 \leq 30; \quad w_6 + 33v_2 \leq 32 \\ & w_7 + 32v_2 \leq 34; \quad w_8 + 31v_2 \leq 36 \\ & w_1 + 28v_3 \leq 60; \quad w_2 + 27v_3 \leq 64 \\ & w_3 + 26v_3 \leq 68; \quad w_4 + 25v_3 \leq 72 \\ & w_5 + 24v_3 \leq 76; \quad w_6 + 23v_3 \leq 80 \\ & w_7 + 22v_3 \leq 84; \quad w_8 + 21v_3 \leq 88 \\ & v_2, v_3 \leq 0 \end{aligned}$$

Solving BDP permits to identify the cut

$$z - 22x_{11} - 24x_{12} - 26x_{13} - 28x_{14} - 30x_{15} - 32x_{16} - 34x_{17} - 36x_{18} \geq 232$$

which can be added to MP. Solving to optimality, this expanded MP formulation produces a lower bound of value $z_{BMP} = 232$.

1.2.6 Column Generation

A further way to decompose the GAP can be obtained following the *Dantzig–Wolfe decomposition* guidelines. Dantzig–Wolfe decomposition results in an iterative procedure that successively approximates the linear relaxation of problem P by decomposing it into a sequence of smaller and easier *subproblems*, which collectively contribute to the definition of the *master problem*. The subproblems in fact dynamically generate the columns of the master problem whose LP solution, at the end of the generation process, provides a bound to the problem of interest.

In the case of the GAP, we are again faced with the decision of whether to delegate to the subproblem the assignment or the capacity constraints. In this case, it is much more immediate to keep the assignments in the master and to push the capacities to the subproblems, thus originating the following LP relaxation of the set partitioning formulation for the master problem:

$$(MDW) \qquad z_{MDW} = \min \sum_{t \in T} c_t x_t \tag{1.36}$$

$$s.t. \sum_{t \in T} a_{jt} x_t = 1, \qquad j \in J \tag{1.37}$$

$$\sum_{t \in T} s_{it} x_t = 1, \qquad i \in I \tag{1.38}$$

$$x_t \geq 0, \qquad t \in T \tag{1.39}$$

In this case, the decision variables x_t, $t \in T$, indicate whether the set of clients specified in the column should be assigned to the server, again specified in the column, or not. The index set T, therefore, indicates all client subsets, which are feasible for the capacities for some server, and is therefore a set with a size growing exponentially with the number of clients. If we denote by J^t the subset of clients specified by column t and by σ^t the server, they are assigned to in that column, we have that $c_t = \sum_{j \in J^t} c_{\sigma^t j}$, and the constraint coefficients are all 0-1 coefficients, such that $a_{jt} = 1$ iff $j \in J^t$ and $s_{it} = 1$ iff $i = \sigma^t$.

Since the set T quickly becomes too large to be generated in its entirety, it must be dynamically generated in successive iterations. The generation of the columns of problem MDW, i.e., the *column generation* process, is instantiated by including in T a subset of columns that is collectively feasible for the assignments, for example, the columns corresponding to a heuristic solution, and it goes on ascertaining whether further columns can improve the current MDW solution.

A column should be added to T if it is feasible for the capacity and if it permits to decrease the cost z_{MDW}, i.e., if it has a negative reduced cost with respect to the problem defined on the current set T. To identify such columns, we have to solve the subproblem that, similarly to Sect. 1.2.2, decomposes into m knapsack problems, one for each server. The optimal solution of each of these knapsacks could result in a new column, provided that it produced a negative reduced cost with respect to the current MDW.

To compute the reduced cost of a knapsack, we associate a dual variable v_j to each constraint (1.37) and a dual variable u_i to each constraint (1.38). Given a master dual solution ($\bar{\mathbf{u}}$,$\bar{\mathbf{v}}$), the pricing subproblem for a server $i \in I$ therefore becomes

$$(SDW_i) \qquad z_{SDW_i} = \min \sum_{j \in J} (c_{ij} - \bar{v_j})\xi_{ij} \tag{1.40}$$

$$s.t. \sum_{j \in J} q_{ij}\xi_{ij} \leq Q_i, \tag{1.41}$$

$$x_{ij} \in \{0, 1\}, \qquad j \in J \tag{1.42}$$

and its optimal solution $\boldsymbol{\xi}^*$ will enter MDW as a new column if its reduced cost is negative, i.e., if $\sum_{j \in J}(c_{ij} - \bar{v_j})\xi^*_{ij} - \bar{u}_i < 0$.

This possibility to add a new column to the master formulation, which is—as mentioned—the LP relaxation of a set partitioning formulation of the problem, grants the basic building block of a column generation procedure, which can therefore be instantiated on the formulation provided by the Dantzig–Wolfe decomposition.

In the case of the example of Fig. 1.1, we can initialize the master with the solution provided by the simple constructive heuristic presented in Sect. 1.3.1: $\boldsymbol{\sigma} = (1, 1, 1, 2, 2, 3, 3, 3)$ of cost 343. The initial MDW is

$$\begin{array}{rlllll}
z_{MDW} = \min & 33x_1 & +58x_2 & +252x_3 & & \\
s.t. & x_1 & & & = 1 \\
 & x_1 & & & = 1 \\
 & x_1 & & & = 1 \\
 & & x_2 & & = 1 \\
 & & x_2 & & = 1 \\
 & & & x_3 & = 1 \\
 & & & x_3 & = 1 \\
 & & & x_3 & = 1 \\
 & x_1 & & & = 1 \\
 & & x_2 & & = 1 \\
 & & & x_3 & = 1 \\
 & x_1, & x_2, & x_3 & \geq 0
\end{array}$$

Upon calling the column generation procedure, 28 columns get included into MDW, leading to the final tableau presented in Fig. 1.3, which has a fractional solution (0, 0, 0, 0, 0, 0, 0, 0, 0, 0.333, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0.333, 0, 0.333, 0, 0, 0.333, 0, 0, 0.333, 0.333, 0) of cost 325, expectedly better than the other bounds presented in this section. This bound permits to certify the optimality of any feasible solution with such cost, for example one provided by the simple heuristic algorithms that will be introduced in the following Sect. 1.3. Figure 1.3 shows in the top line the variable/column identifiers, in row *obj* the coefficients of the objective function, in rows *a0-a7* the coefficients of constraints (1.37), and in rows *s0-s2* the coefficients of constraints (1.38).

1.3 Upper Bounds

Research on heuristics for the GAP has been very rich, we refer to specific surveys to cover it all. In this chapter, we introduce only elements and contributions that will be expanded in the matheuristic approaches detailed in the following chapters.

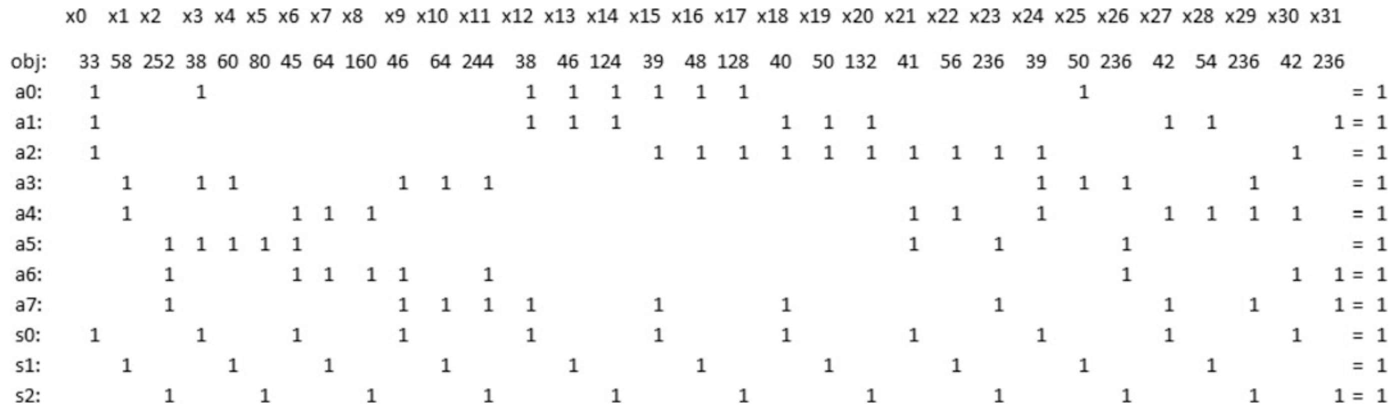
Since the problem of determining the feasibility of a GAP instance is NP-complete, we have that, unless P = NP, the GAP admits no polynomial-time approximation algorithm with fixed worst-case performance ratio. Heuristic algorithms can only be assessed a posteriori, on test instances of interest.

Historically, as mentioned in the preface, heuristic algorithms were first designed specifically for the different problems, then metaheuristics emerged. Algorithms of the first wave can usually be classified into two main approaches: constructive heuristics and local search heuristics.

1.3.1 Constructive Heuristics

Constructive heuristics for combinatorial optimization problems share a main approach starting from an empty solution, ordering the possible components to include according to some specific measure, and adding the best component in the order into a partial solution, provided that they do not make the partial solution infeasible until a complete solution is obtained.

The way the ordering is defined and utilized differentiates the algorithms. It is, in fact, possible to order and later select at each iteration among only a subset of the possible components, which compose the *constructive neighborhood* of the solution to expand, and this subset can be defined in different ways. The way the constructive neighborhood is defined characterizes the resulting algorithms: for example, in the case of minimum spanning trees, Kruskal's algorithm orders just once all edges before starting to insert (*static* ordering), while Prim's has the neighborhood, thus an ordering, redefined at each insertion step (*adaptive* ordering).



	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16	x17	x18	x19	x20	x21	x22	x23	x24	x25	x26	x27	x28	x29	x30	x31		
obj:	33	58	252	38	60	80	45	64	160	46	64	244	38	46	124	39	48	128	40	50	132	41	56	236	39	50	236	42	54	236	42	236		
a0:	1			1									1	1	1	1	1	1								1							=	1
a1:	1												1	1	1				1	1	1							1	1			1	=	1
a2:	1															1	1	1	1	1	1	1	1	1	1						1		=	1
a3:		1		1	1					1	1	1													1	1	1			1			=	1
a4:		1					1	1	1													1	1		1			1	1	1	1		=	1
a5:			1	1	1	1	1															1		1			1						=	1
a6:			1				1	1	1	1		1															1				1	1	=	1
a7:			1							1	1	1	1			1			1					1				1		1		1	=	1
s0:	1			1			1			1			1			1			1			1			1			1			1		=	1
s1:		1			1			1			1			1			1			1			1			1			1				=	1
s2:			1			1			1			1			1			1			1			1			1			1		1	=	1

Fig. 1.3 Final Dantzig–Wolfe final tableau for instance *example8x3*

To make this introduction more specific, we note that the components of a GAP solution are the arcs connecting clients to servers, as determined by x_{ij} decision variables. The general structure of a GAP constructive heuristic is therefore that is presented in Algorithm 1.[2]

The constructive neighborhood $\mathcal{N}_C(\tilde{\mathbf{x}})$ of a partial solution $\tilde{\mathbf{x}}$ is here implicitly given by the set of all unassigned client allocations that do not result in server overloads.

Algorithm 1: Generic constructive heuristic for the GAP

```
1 function Constructive;
  Input :
  Output: A feasible solution $\bar{\mathbf{x}}$ or an indication of failure
2 Let $\tilde{x}_{ij} = 0, i \in I, j \in J$;                // Empty partial solution
3 Make a list lstComp of all components, $(i, j)$ pairs in the case of the GAP, ordered by a
    suitable criterion;                                   // $\mathcal{N}_C(\tilde{\mathbf{x}}) = f(i, j),\ i \in I, j \in J$
4 repeat
5     if $lstComp = \emptyset$ then return fail;
6     $(i, j)$ = pop(lstComp);
7     if $(i, j)$ can be feasibly added to the solution then Set $\tilde{x}_{ij} = 1$;
8 until complete solution;
9 return $\tilde{\mathbf{x}}$;
```

Two main differences can exist among alternative constructive approaches. The first and foremost lies in the ordering criterion of step 3, the second in the decision whether to order the components only once (static ordering), as in Algorithm 1, or at the beginning of each loop, moving instruction 3 inside the repeat loop (adaptive ordering). Notice that the algorithm, at step 5, can terminate with just an indication of failure. However, in this case as in the case of Algorithm 2, for the purpose of fixing heuristics as introduced in Sect. 1.4, it is possible to dictate it to return the partial solution as it was able to construct it.

One possible complete instantiation of Algorithm 1 is presented in Algorithm 2. This algorithm makes use of two orderings: the external one orders clients by increasing regrets, where the regret of a client is the difference in cost between its best allocation and its second best, while the internal ordering given a client considers all servers in the order of increasing client requests. The (i, j) pairs therefore are not identified by a single *pop* from a list of components, but by two successive *pop*s from two different lists.

When we apply Algorithm 2 to instance *example8x3*, we get the regrets (12, 13, 14, 15, 16, 17, 18, 19), and therefore the client ordering for the outer loop is $lstCli = (8, 7, 6, 5, 4, 3, 2, 1)$, that is, client 8 is the first one to choose its cheapest server, client 7 is the second one, and so on. In this instance, server 3 is always the least requested, and server 1 the most requested. Thus, the outer loop proceeds to

[2]In this, and in all algorithms in the text, the input of all GAP instance data is implicit.

Algorithm 2: Simple constructive heuristic for the GAP

```
1  function SimpleConstructive;
   Input  :
   Output: A feasible solution $\tilde{\mathbf{x}}$ or an indication of failure
2  Let $\tilde{x}_{ij} = 0, i \in I, j \in J$;
3  Make a list lstCli of all clients, ordered by increasing regrets;
4  repeat
5      $j$ = pop(lstCli);
6      Make a list $lstServ$ of all servers, ordered by increasing requests of client $j$;
7      while $lstServ \neq \emptyset$ do
8          $i$ = pop(lstServ);
9          if pair $(i, j)$ can be feasibly added to the solution then
10             Set $\tilde{x}_{ij} = 1$;
11             go to 5;
12         end
13     end
14     if $lstServ = \emptyset$ then return fail;
15 until complete solution;
16 return $\tilde{\mathbf{x}}$;
```

assign client 8 to server 3, client 7 to server 3, and client 6 to server 3. Client 5 must be assigned to server 2 because the residual capacity of server 3 (amounting to 4) is not enough to satisfy the request by client 5 ($q_{25} = 24$). At the end of the procedure, we get the solution of equation (1.43), of cost 343.

$$\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{1.43}$$

Algorithm 2 is very simple, but it is not very effective and it has difficulties in finding feasible solutions for tightly constrained instances.

A worthwhile extension of this approach introduces a dummy server $m + 1$ with capacity big enough to accommodate all clients and very high assignment costs. If the original instance is feasible, an optimal solution to the extended instance is also an optimal solution to the original one.

If a feasible solution to an original instance is produced, the dummy machine can be dropped from further consideration. If this is not the case, it is possible to utilize the partial solution to the original instance and try to complete it by means of a heuristic or of a general IP solver, or else to apply any of the fixing heuristics described in Sect. 1.4.

1.3.2 Local Search Heuristics

Local search heuristics implement algorithms that try to improve a given initial solution by iteratively analyzing a subset of the search space “close” to it. They share a common general structure.

Preliminarily, we must define what “close” means. To this end, we define a specific *neighborhood function* that, given a solution $\mathbf{x}$, identifies the subset of neighboring solutions $N(\mathbf{x})$, i.e., of solutions that are assumed not to differ much from the seed one. This subset is called the *local search neighborhood* (or simply the $neighborhood$) of $\mathbf{x}$.

Algorithm 3: Generic local search heuristic for the GAP

```
1  function GenericLocalSearch;
   Input  : A feasible solution $\mathbf{x}$
   Output: A feasible solution $\mathbf{x}$
2  Find $\mathbf{x}' \in N(\mathbf{x}) : z_{GAP}(\mathbf{x}') \le z_{GAP}(\mathbf{x}''), \forall \mathbf{x}'' \in N(\mathbf{x})$;
3  if $z_{GAP}(\mathbf{x}') < z_{GAP}(\mathbf{x})$ then
4      $\mathbf{x} = \mathbf{x}'$;
5      go to 2;
6  end
7  return $\mathbf{x}$;
```

The solution update at step 4 is called a *move*. This could be made toward the best solution in the neighborhood, as proposed in Algorithm 3 (best-improving case), or toward the first-improving solution that is found while exploring the neighborhood in step 2 (first-improving case).

Much research has been devoted to local search heuristics; the differences mainly lying in the specific neighborhood function to use, and in the way the neighborhood is explored in step 2. There is a trade-off to face, because larger neighborhoods could result in larger improvements but require more exploration time. In this section, we assume to make exhaustive neighborhood explorations, but we already mention that state-of-the-art proposals often make use of exponential sized neighborhoods. This is the case of matheuristics algorithms, which will be presented in Sects. 5, 6, and 8 of this book.

Sticking to simple exhaustive exploration, a simple local search for the GAP is presented in Algorithm 4. The neighborhood of a solution is defined to be the set of all feasible solutions differing for the allocation of only one client. Since a move, in this case, is implemented by removing one client allocation, leaving its server empty of it, and adding it to another one, following a common practice the neighborhood is named *opt10*, while *opt11* implies swapping the allocation of two clients, *opt21* removing two clients from a server and one from another server and swapping their allocations, and so on. Alternative names can be found, such as *1-1 exchange*, *2-1 exchange*, etc.

Algorithm 4: opt10 local search heuristic for the GAP

```
1  function opt01;
   Input  : A feasible solution x
   Output: A feasible solution x̄
2  foreach $j \in J$ do
3      Let $i \in I : x_{ij} = 1$;
4      foreach $k \in I \setminus \{i\}$ do
5          if the reassignment is feasible and $c_{kj} < c_{ij}$ then
6              Let $x_{ij} = 0$ and $x_{kj} = 1$;
7          end
8      end
9  end
10 return x;
```

If we apply Algorithm 4 to the solution produced by Algorithm 2, obviously we cannot get any improvement, given the constructive criterion used. More effective local searches will be presented in the second part of this book.

An effective polynomial-time combination of constructive and local search heuristic is named MTHG. It makes use of the same regrets introduced for Algorithm 2 and integrates the constructive procedure with a local search phase.

Algorithm MTHG initially sorts in decreasing order, for each client $j \in J$, the benefit of the assignments to the different servers $i \in I$, as measured by a specific function f_{ij}. Then it makes available, at each iteration in the inner loop, the best and the second best assignments, thus the regret of client j. Client j is then assigned to the best feasible server, whose residual capacity is accordingly updated. In the second phase of the algorithm, the current solution is improved through local exchanges.

Alternative benefit functions have been computationally tested, and good results were obtained by the functions: $f_{ij} = -c_{ij}$, $f_{ij} = -c_{ij}/q_{ij}$, $f_{ij} = -q_{ij}$, $f_{ij} = -q_{ij}/Q_i$.

The pseudocode of algorithm MTHG is proposed as Algorithm 5.

Upon applying Algorithm 5 to the instance of Fig. 1.1, we get the solution reported in Eq. (1.44), of cost 339.

$$\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{1.44}$$

1.4 Fixing Heuristics

Section 1.1 mentioned that the decision version of the GAP is NP-hard in the strong sense, and therefore the decision problem associated with the feasibility of the GAP

Algorithm 5: MTHG heuristic for the GAP

```
1  function MTHG;
   Input  : $f_{ij}$
   Output: A feasible solution $\bar{\mathbf{x}}$ or an indication of failure
   // Construction
2  Let $J' = J$;
3  while $J' \neq \emptyset$ do
4      Find the client $j^*$ with the greatest regret that can be feasibly assigned;
5      if there exists a client that can be assigned to no server then
6          return fail;
7      else
8          Let $i^*$ be the server $j^*$ has to be assigned to;
9          Set $\bar{x}_{i^*j^*} = 1$;
10         Let $J' = J' \setminus \{j^*\}$;
11     end
12 end
   // Local search
13 for j=1 to n do
14     Let $i' = i : \bar{x}_{ij} = 1$;
15     Let $F = \{c_{ij} : i$ a feasible server for $j\}$;
16     if $F \neq \emptyset$ then
17         Let $c_{ij} = \min F$;
18         if $c_{ij} < c_{i'j}$ then
19             Let $\bar{x}_{i'j} = 0$;
20             Let $\bar{x}_{ij} = 1$;
21         end
22     end
23 end
24 return $\bar{\mathbf{x}}$;
```

is an NP-complete problem itself. There is no guarantee, unless $P = NP$, to be able to determine in polynomial time if a given instance is satisfiable; therefore, there is also no guarantee to be able to convert in polynomial time an infeasible solution into one that satisfies all constraints.

As many heuristic algorithms generate infeasible solutions in their working (for example, Lagrangian heuristics, genetic algorithms, diving heuristics), it is important to have the means to quickly recover feasibility, or at least to try to. This gave rise to much research on fixing approaches; the most simple ones will be introduced in this section, while those more mathematically based will be dealt with in Sect. 3.7.1.

Given the problem constraints, infeasibilities can be due to three reasons:

- failed assignment of some clients,
- overassignment of some clients,
- exceeded capacity of some servers.

The underlying complexity of the relevant subproblem reflects on the effort needed to try to recover from these infeasibilities, where the first two causes are

$$\begin{matrix} 1 & 1 & 1 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{0} & 0 & 0 & 1 & 1 \end{matrix} \quad \rightarrow \quad \begin{matrix} 1 & 1 & 1 & \mathbf{0} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 1 & 1 \end{matrix}$$

Fig. 1.4 Feasibility recovering through 1-1 interchange

usually quickly dealt with, trying to determine whether it is easy or impossible to get a feasible solution, and the third one can be challenging on difficult instances.

Often, all that is tried is a simple interchange heuristic, for example, a 1-0 interchange (see Algorithm 4) or a 1-1 interchange, where assignments that lead to excessive use of some server are undone and alternatives are verified. This may not be very effective, but it is fast. Figure 1.4 shows an example of a 1-1 interchange move (the two bold assignment variables exchange their values) that permits to recover feasibility moving from a solution of cost 280, infeasible because of server 1, to a feasible solution of cost 339.

Most contributions from the literature are variations of this basic approach, specifying in different ways which clients to try to relocate first or which servers to try to allocate them to. More complex approaches have been tried, too. Leaving the more mathematically grounded to Sect. 3.7.1, we recall here two metaheuristic-based ones.

In the first one, interchanges have been integrated with a *data perturbation* approach, where problem data are temporarily perturbed in order to find feasible solutions of not too different instances, in the hope to be able to adapt them to the instance of interest.

In the second case, *genetic algorithms* have been used to achieve near optimal solutions. This was done either framing the GAP as a multi-objective problem, where two populations are evolved, one to optimize the objective function and the other one to achieve feasible solutions, or in a more standard genetic approach, where the fitness directs search also toward feasibility achievement.

1.5 Exact Algorithms

Exact algorithms are of indirect interest to this text, which is primarily concerned with heuristic approaches. Traditionally, exact and heuristic solution methods had little to share besides solutions or solution components, but the strong point of matheuristics is exactly the possibility of exploiting the synergies of the two worlds; therefore, ideas on how to efficiently run a complete exploration of the search space can be and have been included into matheuristic algorithms. This notwithstanding the fact that in current years, for the GAP, hardly any specialized exact code is competitive with a well-configured application of a general purpose MIP solver.

Unsurprisingly, given the relevance of the GAP, several different exact methods have been tested to solve it to optimality, including branch-and-bound, branch-and-cut, column generation, and dynamic programming. We refer the interested reader to relevant surveys for comprehensive accounts.

The first and the majority of attempts were based on depth-first branch-and-bound approaches, using different bounds among those presented in Sect. 1.2, with Lagrangian bounds being particularly investigated.

The analysis on the polyhedral structure of the feasible region led to the design of branch-and-cut approaches, whereas branch-and-price, i.e., column generation, is based on the Dantzig–Wolfe decomposition. The two approaches can be combined into a branch-and-cut-and-price code.

A no-branching approach can be based on dynamic programming, which can moreover be useful for computing subproblems arising in the search process, for example, in computing costs in a Lagrangian bound.

1.6 Related Literature

The generalized assignment problem was introduced by Ross and Soland (1975). A good early review can be found in Martello and Toth (1990), and its complexity is discussed in Garey and Johnson (1979) and Sahni and Gonzalez (1976). Chalmet and Gelders (1976) and later Mitrović-Minić and Punnen (2008) have introduced the possibility to transform the feasibility version into an optimization one. Comprehensive introductions can be found in the surveys of Cattrysse and Van Wassenhove (1992) and of Öncan (2007). The applications listed in Sect. 1.1 can be found in Fisher and Jaikumar (1981), Mazzola et al. (1989), Bressoud et al. (2003), Campbell (1999), Dobson and Nambimadom (2001), Drexl (1991), Foulds and Wilson (1997), Harvey et al. (2006), Higgins (1999), Pirkul and Gavish (1986), and Ruland (1999).

The *bottleneck* version of the GAP was studied in Mazzola and Neebe (1993), the *multiple resource types* by Gavish and Pirkul (1985), the *multilevel* GAP by Glover et al. (1979), and the *elastic* GAP by Nauss (2004). As for non-ILP models, the *quadratic* GAP was studied by Lee and Ma (2004) and by Cordeau et al. (2006) and the *multi-objective* GAP was presented by Zhang and Ong (2007).

The result on the maximum number of split assignments in the LP relaxation is due to Benders and van Nunen (1983), and the possible dominance of LP relaxation by Lagrangian relaxation is due to Geoffrion (1974). Lagrangian decomposition is described in Guignard and Kim (1987) and Beasley (1993), and its application to the GAP, along with a surrogate relaxation, in Jörnsten and Nasberg (1986). Surrogate relaxation was first proposed for integer programming by Glover (1965, 1968), and then theoretical analyses of surrogate duality have been proposed by Greenberg and Pierskalla (1970) and Glover (1975). Its possible dominance of LR is in Greenberg and Pierskalla (1970).

Benders decomposition was introduced in Benders (1962) and was first used in matheuristics in Boschetti and Maniezzo (2009). Dantzig–Wolfe decomposition was introduced in Dantzig and Wolfe (1960).

Surveys on heuristics for the GAP can be found in Cattrysse and Van Wassenhove (1992), Lee and Ma (2004), Martello and Toth (1990), Morales and Romeijn (2004), and Öncan (2007). Heuristic MTHG was proposed by Martello and Toth (1990).

Fixing heuristics have been proposed in Jörnsten and Nasberg (1986), Wilson (1997), Jeet and Kutanoglu (2007), Mazzola et al. (1989), Chu and Beasley (1997), and Narciso and Lorena (2007), among others. Data perturbation for fixing was proposed by Jeet and Kutanoglu (2007) and genetic algorithms by Chu and Beasley (1997) and by Wilson (1997).

Surveys on exact approaches to the GAP can be found in Cattrysse and Van Wassenhove (1992), Martello and Toth (1990), Morales and Romeijn (2004), and Öncan (2007). Branch and bounds can be found in Ross and Soland (1975), Martello and Toth (1981), Fisher et al. (1986), Cattrysse et al. (1998), and Nauss (2003). An analysis of the polyhedral structure of the GAP feasible region is in Gottlieb and Rao (1990) and De Farias et al. (2000), branch-and-cut in Avella et al. (2010), branch-and-price in Savelsbergh (1997), and branch-and-cut-and-price in Pigatti et al. (2005). Finally, dynamic programming for the GAP was used in Posta et al. (2012).

References

Avella P, Boccia M, Vasilyev I (2010) A computational study of exact knapsack separation for the generalized assignment problem. Comput Optim Appl 45(3):543–555

Beasley JE (1993) Lagrangian relaxation. In: Reeves CR (ed) Modern heuristic techniques for combinatorial problems. Wiley, New York, pp 243–303

Benders JF (1962) Partitioning procedures for solving mixed-variables programming problems. Numer Math 4:280–322

Benders JF, van Nunen JA (1983) A property of assignment type mixed linear programming problems. Oper Res Lett 32:47–52

Boschetti M, Maniezzo V (2009) Benders decomposition, Lagrangean relaxation and metaheuristic design. J Heurist 15:283–312

Bressoud TC, Rastogi R, Smith MA (2003) Optimal configuration for BGP route selection. In: IEEE INFOCOM 2003, twenty-second annual joint conference of the IEEE computer and communications societies, San Francisco, CA, vol 2, pp 916–926

Campbell GM (1999) Cross-utilization of workers whose capabilities differ. Manage Sci 45(5):722–732

Cattrysse DG, Van Wassenhove LN (1992) A survey of algorithms for the generalized assignment problem. Eur J Oper Res 60(3):260–272

Cattrysse DG, Degraeve Z, Tistaert J (1998) Solving the generalised assignment problem using polyhedral results. Eur J Oper Res 108(3):618–628

Chalmet LG, Gelders LF (1976) Lagrangian relaxations for a generalized assignment-type problem. In: Proceedings of the second european congress operations research. North-Holland, Amsterdam, pp 103–109

Chu PC, Beasley JE (1997) A genetic algorithm for the generalised assignment problem. Comput Oper Res 24(1):17–23

Cordeau JF, Gaudioso M, Laporte G, Moccia L (2006) A memetic heuristic for the generalized quadratic assignment problem. INFORMS J Comput 18(4):433–443

Dantzig GB, Wolfe P (1960) Decomposition principle for linear programs. Oper Res 8:101–111

De Farias IR, Johnson EL Jr, Nemhauser GL (2000) A generalized assignment problem with special ordered sets: a polyhedral approach. Math Program Ser A 89:187–203

Dobson G, Nambimadom RS (2001) The batch loading and scheduling problem. Oper Res 49(1):52–65

Drexl A (1991) Scheduling of project networks by job assignment. Manage Sci 37(12):1590–1602

Fisher ML, Jaikumar R (1981) A generalized assignment heuristic for vehicle routing. Networks 11:109–124

Fisher ML, Jaikumar R, Van Wassenhove LN (1986) A multiplier adjustment method for the generalized assignment problem. Manage Sci 32(9):1095–1103

Foulds L, Wilson J (1997) A variation of the generalized assignment problem arising in the New Zealand dairy industry. Ann Oper Research 69:105–114

Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. W. H. Freeman & Co, New York, NY

Gavish B, Pirkul H (1985) Efficient algorithms for solving multiconstraint zero-one knapsack problems to optimality. Math Program 33(1):31–78

Geoffrion AM (1974) Lagrangean relaxation for integer programming. Math Program Stud 2:82–114

Glover F (1965) A multiphase-dual algorithm for the zero-one integer programming problem. Oper Res 13:879–919

Glover F (1968) Surrogate constraints. Oper Res 16:741–749

Glover F (1975) Surrogate constraint duality in mathematical programming. Oper Res 23:434–451

Glover F, Hultz H, Klingman D (1979) Improved computer based planning techniques, part 2. Interfaces 9(4):12–20

Gottlieb ES, Rao MR (1990) Generalized assignment problem. Valid inequalities and facets. Math Program Ser A 46(1):31–52

Greenberg HJ, Pierskalla WP (1970) Surrogate mathematical programming. Oper Res 18:924–939

Guignard M, Kim S (1987) Lagrangean decomposition: a model yielding stronger Lagrangean bounds. Math Program 39:215–228

Harvey NJA, Ladner RE, Lovász L, Tamir T (2006) Semi-matchings for bipartite graphs and load balancing. J Algorith 59(1):53–78

Higgins AJ (1999) Optimizing cane supply decisions within a sugar mill region. J Sched 2:229–244

Jeet V, Kutanoglu E (2007) Lagrangean relaxation guided problem space search heuristic for generalized assignment problems. Eur J Oper Res 182(3):1039–1056

Jörnsten K, Nasberg M (1986) A new Lagrangian relaxation approach to the generalized assignment problem. Eur J Oper Res 27:313–323

Lee CG, Ma Z (2004) The generalized quadratic assignment problem. Tech. Rep. 5, Department of Mechanical and Industrial Engineering, University of Toronto, Toronto

Maniezzo V (2019) Bridging the GAP. Some generalized assignment problem instances. http://astarte.csr.unibo.it/gapdata/gapinstances.html

Martello S, Toth P (1981) An algorithm for the generalized assignment problem. In: Brans J (ed) Operations research ’81. North-Holland, Amsterdam, pp 589–603

Martello S, Toth P (1990) Knapsack problems: algorithms and computer implementations. Wiley, New York, NY

Mazzola JB, Neebe AW (1993) An algorithm for the bottleneck generalized assignment problem. Comput Oper Res 20(4):355–362

Mazzola JB, Neebe AW, Dunn CVR (1989) Production planning of a flexible manufacturing system in a requirements planning environment. Int J Flex Manuf Syst 1:115–142

Mitrović-Minić S, Punnen AP (2008) Very large-scale variable neighborhood search for the generalized assignment problem. J Interdiscipl Math 11(5):653–670

Morales DR, Romeijn HE (2004) The generalized assignment problem and extensions. In: Du D, Pardalos PM (eds) Handbook of combinatorial optimization. Springer, Boston, pp 259–311

Narciso MG, Lorena L (2007) Lagrangean/surrogate relaxation for generalized assignment problems. Eur J Oper Res 182:1039–1056

Nauss RM (2003) Solving the generalized assignment problem: an optimizing and heuristic approach. INFORMS J Comput 15(3):249–266

Nauss RM (2004) The elastic generalized assignment problem. J Oper Res Soc 55:1333–1341

Öncan T (2007) A survey of the generalized assignment problem and its applications. Inf Syst Oper Res 45(3):123–141

Pigatti A, Poggi de Aragao M, Uchoa E (2005) Stabilized branch-and-cut-and-price for the generalized assignment problem. Electron Notes Discr Math 19:389–395

Pirkul H, Gavish B (1986) Computer and database location in distributed computer systems. IEEE Trans Comput 35:583–590

Posta M, Ferland JA, Michelon P (2012) An exact method with variable fixing for solving the generalized assignment problem. Comput Optim Appl 52(3):629–644

Ross GT, Soland RM (1975) A branch and bound algorithm for the generalized assignment problem. Math Program 8(1):91–103

Ruland KS (1999) A model for aeromedical routing and scheduling. Int Trans Oper Res 6:57–73

Sahni S, Gonzalez T (1976) P-complete approximation problems. J ACM 23(3):555–565

Savelsbergh MWP (1997) A branch-and-price algorithm for the generalized assignment problem. Oper Res 45:831–841

Wilson JM (1997) A genetic algorithm for the generalised assignment problem. J Oper Res Soc 48:804–809

Zhang CW, Ong HL (2007) An efficient solution to biobjective generalized assignment problem. Adv Eng Softw 38:50–58

Chapter 2
Automatic Design for Matheuristics

2.1 Introduction

To organize a matheuristic, we can use a large variety of techniques. These organize the search intensification and diversification, and they guide in their way the problem-specific components. Traditionally, design and development of matheuristics is involved in a manual, experimental approach that is mostly guided by the experience of the algorithm designers. This way the process is guided by over-generalizations from the designer's previous experience and by implicit independence assumptions between parameters and algorithm components. In addition to these, the manual process has a number of other disadvantages. Some are that it limits the number of designs that would be involved, it hides the efforts that have been dedicated to their development, it loses the number of design alternatives that are involved, or it simply makes the design process irreproducible.

The implementation effort that is associated with the design of matheuristic algorithms is very variable. Basic versions that consist of few numerical parameters can be, in principle, designed conventionally with little effort and can reach good performance. However, when very high performance is required, the design of matheuristics profits strongly from the exploitation of problem-specific knowledge, the time invested in designing and tuning the algorithms, and the use of insights into algorithm behavior and the interplay with problem characteristics. With this, they go along, at least in tendency, with the importance of the parameters. The matheuristics can be seen as a rather general technique, varying their parameters as categorical or ordinal parameters (more on it in Sect. 2.2.1) together with specialized algorithm components usually in the form of numerical parameters.

There have been a number of methods that have adopted numerical and also categorical and ordinal techniques. A first one is the determination of numerical parameters. For a few parameters, it is the fractional design or variants of it, but it has the disadvantages that it is related to the chosen parameter settings. A different approach is to configure numerical parameters with numerical optimization

V. Maniezzo et al., *Matheuristics*, EURO Advanced Tutorials
on Operational Research, https://doi.org/10.1007/978-3-030-70277-9_2

algorithms. One can use fractional design techniques also if one has categorical or ordinal parameters, yet they turn infeasible if the parameters are many. Then, instead, there are several methods one can choose from, be it heuristics search techniques, statistical methods, or model-based techniques. Here we can cite ParamILS and gender-based genetic algorithm as heuristics search methods, F-race and iterated F-race as statistical racing techniques, and sequential model-based configuration. These techniques deal with the stochasticity of the algorithms and have successfully dealt with tens or sometimes hundreds of parameters.

A first approach to these automatic algorithm configuration techniques consists in the use of numerical techniques for the configuration of the parameter setting of matheuristics algorithms. This includes algorithms that are fully specified according to their numerical distributions and alternative algorithm components. Yet, although the numerical results may already improve significantly the performance of the algorithm, the effective importance of the automatic algorithm configuration techniques is in their essential role of making the categorical and ordinal choices of a matheuristic. It is an alternative design method that relies on a design paradigm: defining an appropriate design space that encodes alternative algorithm design choices and numerical parameter. Then a computation-intensive process explores this design space to find high-performing algorithm instantiations. In a variety of research efforts, this method has been shown to be feasible as we may see in a few examples in this chapter.

The chapter is organized as follows. First, in Sect. 2.2, we highlight the settings of parameters that lead to the parameter configuration problem and—to an increasing automatization of design. In Sect. 2.3, we give a number of successful examples that highlight the potential of an automated design of methods. We then go through an example with the irace software package in Sect. 2.4, showing with a short example of iterated local search for the generalized assignment problem (GAP), how it works.

2.2 Parameter Configuration

In this section, we give an overview of the type of parameter settings that we have in algorithm deployments and define what the configuration of parameters is. Then we go through the developments of automatic parameter configurators and show three examples: ParamILS, SMAC, and irace.

2.2.1 Parameters

The available choices for the parameter configuration problem depend on the type of the parameters. On one side we have discrete options available. These are *categorical* parameters if no details are given to allow distinguishing the value of the choices. These can be the choice of a simulated annealing or a tabu search as local search procedure or the choice of different perturbation types in an iterated local search. Sometimes it may be possible to give to the values an ordering without having a clear distance measure available. This gives place to *ordinal* parameters, with an example being {small, medium, large}. In the GAP, it could be the size of a neighborhood or the value of a lower bound. Finally, there are parameters that are numerical in nature. These can be real-valued parameters, such as the temperature in simulated annealing or the evaporation rate in ant colony optimization, or integer ones, such as the population size in memetic algorithms or the tabu list length in tabu search. These numerical parameters one can have are algorithm-wide parameters if they depend on the search method. For example, this is the case with the population size for a memetic algorithm. However, a parameter is *conditional*, if it depends on others that are to be set usually as a categorical or an ordinal one. For example, whether having a temperature parameter depends on whether one has a simulated annealing as local search or a tabu search.

Another issue is the size of the parameter settings. For many cases like the categorical and the ordinal parameters, these will be not addressed, as it is commonly the same for each of them. However, the numerical parameters need a range whose size, that is, the minimum and the maximum number, and precision, that is, the significant digits, impacts the search. In case of doubt, one may opt for ranges that are maybe too wide. They require more experiments than smaller ranges, but they can give better results. Only if the experiments are too few, it could be problematic and one would be too demanding with the size or the precision.

2.2.2 Parameter Configuration Problem

We can now describe the parameters of an algorithm in more detail. The performance-optimizing algorithm configuration can be described by the set of variables of any type, that is, they can be numerical, ordinal, and categorical. Every parameter $\theta_i, 1, \ldots, N_p$, is given an associated type t_i and a domain D_i, so that we have a parameter vector $\theta = (\theta_1, \ldots, \theta_{N_p}) \in \Theta$, where Θ is the space of possible parameters settings.

The goal in the design of algorithms then is to optimize for some problem Π the performance of an algorithm for a specific instance distribution $\mathscr{I}$ of problem Π. In formal ways, when applied to a problem instance π_i, the performance of the algorithm is the cost measure $\mathscr{C}(\theta, \pi_i) \colon \Theta \times \mathscr{I} \rightarrow R$. Usually, this will be a random variable with a typically unknown distribution, if during the search the algorithm involves stochastic decisions. Yet by executing an algorithm on a specific instance, one can measure realizations of the random variable. Another element of stochasticity is the fact that each instance π_i can be seen as being obtained from some random instance distribution $\mathscr{I}$. The performance $F_{\mathscr{I}}(\theta) \colon \Theta \rightarrow R$ of a configuration is then defined as function for an instance distribution $\mathscr{I}$.

The usual approach is to define $F_{\mathscr{I}}(\theta)$ with respect to the expected cost $E[\mathscr{C}(\theta, \pi_i)]$ of θ if applied to an instance distribution. The definition of $F_{\mathscr{I}}(\theta)$ determines how to compare configurations over a set of instances. If cost values across different instances are not comparable on a same scale, rank-based measures such as the median or the sum of ranks may be more meaningful. The precise value of $F_{\mathscr{I}}(\theta)$ can generally not be computed in an analytic way, but it can be estimated by sampling. This means that one obtains realizations $c(\theta, \pi_i)$ of the random variable $\mathscr{C}(\theta, \pi_i)$ by running an algorithm configuration θ on instances that have been sampled according to $\mathscr{I}$.

We can then say that the algorithm configuration problem is to identify an algorithm configuration θ^* where it holds

$$\theta^* = \arg\min_{\theta \in \Theta} F_{\mathscr{I}}(\theta). \tag{2.1}$$

In summary, the algorithm configuration problem is a stochastic optimization problem where the types of variables are numerical, categorical, ordinal, or of any other type. Each of these variables has a specific domain of distribution and a specific type of constraints. The stochasticity of the configuration task is determined by the types of the stochasticity of the algorithm and the stochasticity that stems from the sampling of the problem instances.

Due to the stochastic nature of the configuration process, it is of importance to let this process to generalize to unseen instances. As a result, this process is done in two phases. In a first *training* phase, a high-performing configuration is searched ideally by a large training size. Clearly, it is important that this training sets it also in shape and in size similar to the test phase. This is also ensured by a specific application context that has, for example, the available computation times in which the algorithms are employed. In a second *test* phase, this configuration is then evaluated as the true configuration on an independent test set. This generalization to the test instances is the real setting. This also reflects the nature of the setting. In a first phase, the training phase, an algorithm is designed and trained for a specific target application, and in a second phase, the test phase, the algorithm is deployed to solve new, previously unseen instances.

2.2.3 Automatic Algorithm Configuration

While in the old matheuristics literature the design of parameters is pre-set and mainly presented as a manual trial-and-error process, in the current literature the automatic algorithm processing is given a distinctive place and gives rise to a computationally intensive process. A configurator gets as input the name of the variables, the type of the variables, and the values of the parameters. For the categorical and ordinal parameters, it gets the discrete sets of values, such as $\{x, y, z\}$ and $\{small, medium, large\}$, and for the numerical parameters the real-valued variables, for example, [0, 1], or the integers, for example, {0, 1, 2, . . . , 10}. Additionally, the configuration tools may receive information about which parameters depend on each other, that is, the conditional parameters, which parameters have forbidden values, and any other detailed information that may be helpful.

The configurator sets these parameter values to one or several configurations, depending on whether it is a population-based algorithm or a single search-based configurator. These configurations are evaluated on training instances, and the evaluations of the candidate configurations are returned to the configurator. This process of returning the configurations and solving them by the software is repeated until some measure of interest is achieved. The most common measures are the amount of computational experiments or a measure of the computation time. The first is commonly a measure that is used for metaheuristics and consists in the number of experiments executed. The second, the computation time, be it CPU or wall-clock time, is usually the time that is allowed for a specific computation. Independent of the specific termination, the best configuration is returned at the end of the process. Usually, also some other information on the configurations process is returned such as other high-quality configurations or statistics of the configuration.

A general overview of these interactions between the configurator, the parameter definition, the calls of the software, and the return of the solution costs is given in Fig. 2.1.

We can now come to highlight some of the main approaches. We can classify them as experimental design techniques, continuous optimization techniques, heuristic search techniques, surrogate model-based configurators, and non-iterated and iterated racing approaches.

Experimental Design Techniques To avoid the immediate pitfalls of trial-and-error processes, various researchers have adopted statistical techniques, such as hypothesis testing for evaluating the statistical significance of performance

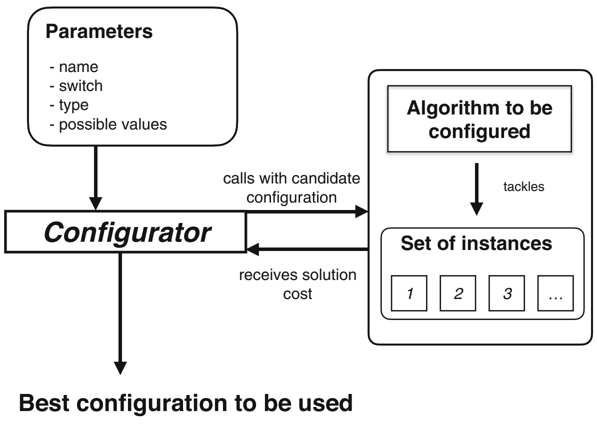


Fig. 2.1 Generic view on the main interaction of an automatic configuration technique with the configuration scenario

differences, or experimental design techniques such as factorial or fractional factorial designs and response surface methodologies. While often experimental design techniques such as ANOVA have been applied using manual intervention, several efforts have been made to exploit such techniques to make them more automated. An example in this direction is the CALIBRA approach, which applies Taguchi designs and a refinement local search to tune five parameters. A different approach is based on the exploitation of response surface methodologies. Yet, a common disadvantage of these methods is that many of these are limited to few variables and that many of these have the variables fixed.

Continuous Optimization Techniques A different approach is numerical optimization in the form of continuous optimization. It first has to be mentioned that in this case it is often feasible to integrate integer and continuous function optimization by a continuous relaxation approach in the form of truncation and rounding. Independent of this, such things are often feasible if the algorithm either has only numerical parameters or has already been fine-tuned by setting all the right value of categorical or ordinal parameters. One of the first is the mesh-adaptive direct search (MADS) mechanism, which features a mathematical program for the search and a large set of functions for the heuristic evaluations. About the same time, Nannen and Eiben have done some work on the REVAC algorithm. In a computational study of various instances among which were bound optimization by quadratic approximation (BOBYQA), Covariance matrix adaptation evolution strategy (CMAES), Iterated F-race, and MADS, they found that for small instances with two to four parameters the best performance was obtained by BOBYQA, while beyond that parameters the best performances were obtained by a modified version of CMAES. This performance was further improved by post-selection in automated configuration, where first a subset of configurations is defined and afterward it undergoes a more careful evaluation.

Heuristic Search Techniques If instead one has to make also categorical or ordinal parameter choices, one has a number of heuristic choices available. One of the first choices was the meta-genetic algorithm that was proposed by Grefenstette in the year 1986; but, there the performance was too weak to be impressive. It was in the first and in the second decades of this century that these methods became promising. One of the first was the ParamILS algorithm; we will go in more detail in Sect. 2.2.4. More recent is the gender-based genetic algorithm that is rather good for exact algorithms and that has also been adapted in recent years to make use of surrogate models. Other approaches are linear genetic programming algorithms that evolve evolutionary algorithms, the OPAL system that extends MADS to include categorical and binary parameters, the EVOCA evolutionary algorithm, and the heuristic oriented racing algorithm (HORA).

Surrogate Model-Based Configurators The evaluation of configurations is typically the most computationally demanding part of an automatic configuration method, since it requires actually executing the target algorithm being tuned. Several methods aim to reduce this computational effort by using surrogate models to predict the cost value of applying a specific configuration to one or several instances.

Based on the predictions, one or a subset of the most promising configurations are then actually executed and the prediction model is updated according to these evaluations. Among the first surrogate-based configuration method is sequential parameter optimization (SPOT). A more general method also using surrogate models is the sequential model-based algorithm configuration (SMAC). In fact, this is currently one of best configurators and we will revise it in some more detail in Sect. 2.2.5.

(Iterated) Racing Approaches Finally, other systems investigate racing methods for selecting one or various configurations among a (huge) sets of candidate configurations. They use F-race or other sequential statistical testing methods to do the racing. A race can be organized based on the design of experimental techniques, randomly generated configurations, or problem-specific information. If the race is iterated, this iteration can be based on problem instance-based methods. For example, iteration i is based on the best configuration of the previous race $i - 1$. In Sect. 2.2.6 we will see an example of a configurator based on the iterated racing approach.

Independent of the automated algorithm configuration, the usage of such a system has some advantages. A first is that one has the parameters for the system. Here it is important that not only one needs the "usual" parameters but one can also extend those parameters beyond those that are in use. We further may distinguish between categorical and ordinal parameters on the one side, and between numerical parameters on the other. The categorical and ordinal ones are the normal parameters, which discriminate major parts of the algorithm that before where in many designs pre-given. The numerical parameters are those that are available such as the temperature parameter in simulated annealing. However, when one has an automatic configuration system available, one can also deal with numerical parameter that otherwise would be pre-set. One can also distinguish between conditional parameters, which only may play an essential role for some parameters of a system.

Apart from the parameters, one has the further advantage that there is a clear separation between the training instances and test instances. Ideally, there is only the distinction between training and test instances in the sense that there are differences only in, say, random seeds. A further advantage is the clear role of the configurator and the algorithms that are configured. This comprises the budget, special features of the configurator, and so on. Last but not least, a final advantage is that with improvements of the configurators we may also improve performance of the configured algorithm.

2.2.4 ParamILS

We are now able to present in more detail some popular configurators. We start with ParamILS, which is an ILS algorithm in the search space of the parameters. A first difference from the other parameter configurators is that it is a fully categorical parameter system. In other words, real and integer parameters have to be converted into discrete values. This can be done either by introducing a grid or, if there is some knowledge available, by introducing a specific parameter space, or simply at random.

As initialization of ParamILS, a default configuration is required. In addition to this, r random initial configurations are determined, and the best of these $r + 1$ configurations builds the initial solutions. This is done as random configurations may be better than the default configuration.

The local search in ParamILS is a first-improvement local search with a random-improvement basis. Each time this random-improvement scheme finds a new improvement, it takes it and continues with the random-improvements scheme that is randomly initialized. This is continued until all the neighborhoods are examined or the budget is exhausted.

When a local search is terminated, the better of the two local minima is kept and a perturbation is done. With a probability of $1 - p_R$, this is done by actually taking the better of the two and introducing a k randomly chosen parameter to obtain a new $\theta^{'}$ as the solution. By default, $k = 3$ random parameters are set. Yet with a probability of p_R, the perturbation consists in a random contribution that is generated from which the next ILS iteration is started. By default, p_R is set to 0.01.

Of course, deciding which configuration is better is not trivial. ParamILS offers two variants: BasicILS and FocusedILS. BasicILS chooses the best of two configurations after having them evaluated them on n_e instances, where n_e is a given set on instances. Such setting can be approximately right or rather be very wrong. In other words, it is simply not a trivial task to set n_e: too low, it will risk not to provide enough estimates; too high, it will risk to waste enough candidates to be seen. The FocusedILS version compares two configurations increasingly, until one configuration dominates the other. Dominance between two configurations is established as follows. Let the two configurations θ_1 and θ_2 be evaluated on $n_1 > n_2$ instances. Then configuration θ_1 dominates θ_2 if we have that $\hat{F}(\theta_1, n_2) \leq \hat{F}(\theta_2, n_2)$, where $\hat{F}(\theta, n)$ gives the cost estimate of a configuration θ on n instances. Hence, a configuration dominates another one if it has been used on as many instances as the other and it has a lower cost estimate. If θ_1 is not dominating the other one, the number of instances on which the second one will be evaluated is redone. When a new instance improves over the old one, the number of instances will also be increased.

Often, ParamILS is used to configure for run-time. For example, this is very often the case when configuring exact algorithms such as CPLEX, Gurobi, or FICO Xpress for specific tasks. Therefore, ParamILS implements a procedure that is called adaptive capping, which prunes search early for the evaluation of potentially poor performing configurations. When it is combined with the above dominance criterion, adaptive capping can strongly reduce the computing time.

ParamILS is publically available at http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/, the version at the time of writing was 2.3.8. It has been shown to excel in a number of configuration tasks and, in particular, it led to speedups of several orders of magnitude for mixed-integer programming or propositional satisfiability problems.

2.2.5 SMAC

Sequential model-based algorithm configuration (SMAC) is another configurator proposed by some of the same authors of ParamILS. It is a configurator, which uses a surrogate model search of the parameter space and, as opposed to ParamILS, handles both real and integer parameters and categorical parameters. This means that it does not need only discrete numerical parameters as ParamILS.

SMAC starts from an initial configuration, which is typically the algorithm default. When no algorithm default is available, a random one can be given. With this input, the procedure loops through the following steps. First, a random forest model is learned. Following this, a set of candidate configurations is determined by following n_{LS} elite configurations. Each of them is the candidate of the n_{LS} elite configurations, which are determined as the best-improvement local searches on the configuration space; each time, an algorithm is selected according to the expected improvement criterion. In addition, n_r randomly generated configurations are generated, where each one is evaluated according to the expected improvement criterion. Then, SMAC sorts the $n_{LS} + n_R$ configurations according their expected improvement and executes them in this order. The process of evaluating the instances is the same as for FocusedILS. It uses the dominance criterion, the adaptive capping, and increases the number of instances on which the incumbent is evaluated. This evaluation process is terminated after a specific termination time. Then the next overall iteration of SMAC is invoked, which starts first with the learning of the new random forest.

While the high-level search process that is implemented by SMAC follows these stages, there are a number of rather intricate steps that make the process much more involved. Some of these are special treatments of censored data, which can improve the predictions of SMAC. Another one is the inclusion of instance features for improving the predictions. All in all they make SMAC currently one of the top-performing configurators.

SMAC is available online. The SMAC v2.08 is written in Java and the version SMAC v3 is written in Python3 and is continuously tested in Python3.5 and Python3.6. The versions of SMAC v2.08 and SMAC v3 are virtually the same, but the version SMAC v2.08 gives slightly better performance. The SMAC v3 is a free software, which one can redistribute or modify it under the terms of the 3-clause BSD license.

2.2.6 irace

Last but not least is the irace system that implements an iterated racing for the determination of the best configuration. It is based on three subsequent iterations. First, it generates candidate configurations from a statistical model. Second, it selects the best algorithm candidate configurations by a statistical racing method that is either F-race or paired student's t-test. Third, it updates the probability model of the system to give the most promising candidates a higher probability of being used.

In the first step, irace has to sample new configurations. In the first iteration, when no results are available, it has various possibilities. If nothing is known the first generation may be generated by random sampling, that is, all the first generation is random. Of course, one or also various candidates may be given as default configurations. From the second iteration, each of the surviving configurations, called elites, forms a (partial) model. Each of these elites will have numerical, ordinal, and categorical parameters. If it is numerical and ordinal, then the probabilistic parameters are drawn from a truncated normal distribution (μ_i^j, σ_i^j), where μ_i^j is the mean value of the ith parameter of the jth elite configuration and σ_i^j is the standard deviation. The truncation is happening because the parameter is in a range $[x_l, x_u]$, where x_l and x_u are the lower and the upper boundaries of the parameter. If this is an integer or an ordinal parameter, then it is rounded as $\mathrm{round}(x - 0.5)$. If instead with X_d we have a categorical parameter with levels $X_d \in \{x_1, x_2, \ldots, x_{n_d}\}$, then a new value is sampled from $\mathscr{P}^{j,z}$. This distribution is set to the uniform distribution at the beginning and then at each iteration is set to the sampling as follows:

$$\mathscr{P}^{j,z}(X_d = x_j) = \mathscr{P}^{j-1,z}(X_d = x_j) \cdot \left(1 - \frac{j-1}{N^{iter}}\right) + \Delta\mathscr{P} \tag{2.2}$$

where

$$\Delta\mathscr{P} = \begin{cases} \dfrac{j-1}{N^{iter}} & \text{if } x_j = x_z \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where N^{iter} is the estimated number of iteration, j is the current iteration, and x_z is the current elite. This way, the entries of the elite probability for a categorical parameter are always the ones with the highest probability.

After the configurations are generated, they undergo a race. In a race, all the configurations are evaluated on a first instance, then on a second one, and so on. As soon as the configurations are deemed to be poor, they are eliminated. There are different criteria how this is managed, which in part depends on the objective function, which can be quality-wise or time-wise. If it is quality-wise, then the usual thing is to take a test. In fact, one has either the F-race, which is based on the non-parametric Friedman's two-way analysis of variance by ranks or the paired student's t-test with or without p-value corrections. The first one, the F-race, can be used especially if the non-parametric test is worthwhile, such as when the quality-wise setting is very different. For the second choice, we would recommend to do the test without the p-value corrections, since it eliminates much more quickly the poor candidates. If the test is time-wise, then, in the version irace 3, one has an extension that better handles run-time minimization. In particular, it handles a dominance criterion among configurations. A configuration θ_1 dominates θ_2 if it has more instances N_1 than N_2 and it holds that cost $c(\theta_1, 1, \ldots, N_2) \leq c(\theta_2, 1, \ldots, N_2)$. This maintains the trajectory, and it works very well together with the student's t-test.

After a race is terminated, either because the minimum elite number of configurations is reached or because the computational budget is reached, one makes the update step of the elite configurations. This is done by numerical and ordinal parameters centering these at the point of their mean and diminishing their standard deviations as it is foreseen in the irace software. For categorical parameters, the elite probability and the other probabilities are set as indicated above.

The irace software is publicly available at http://iridia.ulb.ac.be/irace/ or at https://cran.r-project.org/web/packages/irace. Currently, it is available as version irace 3.4.

2.3 Design of Algorithms

In what follows, we claim that advantages of automatic configuration go from an automatic configuration of existing algorithm to integration of an algorithm engineering process and advanced uses of the automated configuration software. It is especially this last one where the systematic use of automated configuration software has real advantages.

2.3.1 Automated Configuration of Existing Algorithms

One of the first uses of automated configuration software is to determine parameter settings that would improve performance. This is feasible with relatively minor efforts. The reason is that the default setting can always be used if no advanced parameter setting is found. Yet, this is only rarely the case and often one can find parameter settings that substantially improve over it. There are various reasons for this. One is that the default values are for a very different instance distribution and often the new parameters are very different from the old ones. Another reason is that running times of the new application scenarios are very different from the ones that had been assumed. Yet another reason may be that the default parameter settings are not well attached with the available instances.

There has been a number of papers about this, especially in the initial steps of the automated configuring methods. One of the first papers in this direction is the paper of Hutter et al. on the boosting of a verifications procedure in which they observed speedups of up to 500 times the default. Similar examples were obtained when employing them in the parametric case study on mixed-integer programming methods such as CPLEX. Other examples include the REVAC method, which reached improvements on the solution quality, or the irace methods, which found improvements on the solution quality for various methods.

When we want to compare algorithms, we have to do the same thing: all of them should ideally be run on the same parameter setting for each algorithm. For this task, the automatic configuration methods make good performance available for all algorithms. In particular, they make the parameter setting, if they are equally good of course, less an issue of the particular system designer than intrinsic to the system. That is, the features of the system designer are not or, at least, less critical. All the algorithm parameters then have the same advantages as the single algorithms: the tuned versus the default parameters, the very different time or solution quality parameters, or the poor integration with the instances. Another advantage may be that an old algorithm may be surprisingly good when offering new parameters.

2.3.2 Integration into an Algorithm (Re-)engineering Process

One way to use automatic configuration methods is to apply them only at the end of the development. But it may be much better to use automatic configuration methods already in each step of the engineering or re-engineering of the algorithms. It is in these steps—when it is most useful to see whether it actually improves the algorithm or not. A first approach in this direction was the design of an existing particle swarm optimization algorithm for a large-scale function optimization competition. This approach went through a six-step engineering set that included a local search, call and control strategy of the local search, the particle swarm optimization rules that were set to the original ones, the bound constraint handling when calling the particle swarm optimization, the stagnation handling, and finally the restarts. Here iterated F-race was used at each step to configure up to ten parameters of the algorithm. Although the configuration was done on only 19 functions of dimension ten, it was found that the scaling was valid up to function dimension 1000. In a similar undertaking, Hartung and Hoos did an engineering of a parameterized algorithm used for clustered editing, and Balaparash et al. developed some high-performing algorithm for the probabilistic traveling salesman problem and a single vehicle routing problem with stochastic demands and customers.

2.3.3 Advanced Uses of Configurators

A major research area is how to design new algorithms. This can be done at least in two ways: top-down approaches and bottom-up approaches. Both approaches use the same set of algorithm components and follow rules on how to combine these components (Fig. 2.2). Yet they differ in how they deal with the problem.

In the top-down approaches there is always one single basic algorithm template, but different algorithm components are used. One of the first top-down approaches is the SATenstein approach. This approach could initialize five algorithmic blocks that go from performing diversification, over WalkSAT-based algorithms, dynamic local search algorithms, over G^2WSAT variants to updates of the data structures. While with SATenstein a new state-of-the-art solver for SAT local search algorithms could be generated, it is also clear that the local search is heavily optimized. Other groups have explored several metaheuristics. These include several metaheuristics that are single objective ones such as a framework for continuous optimization for ACO. In this case, they have shown that the numerical optimizer is outperforming the best previously known ACO. The same was obtained for the generalized artificial bee colony framework. Another research area is multi-objective optimization, which originated in 2010 with the help of multi-objective ant colony optimization and then went on with the TP + PLS framework and with an automatic design of multi-objective evolutionary algorithms. The most recent works explore an automated multi-objective evolutionary algorithm (AutoMOEA) that could instantiate a Pareto-based, an indicator-based, or a weight-based MOEA. This was possible through a preference relation that was set partitioning, quality, and diversity based. In its latest setting, AutoMOEA could generate algorithms to solve problems with multi-objective and with many-objective evolutionary algorithm with up to ten objectives that outperformed the best of the self-tuned algorithms these AutoMOEA where compared to.

A second method is the bottom-up approach, which tries to make it potentially more complex algorithms based on low-level rules frequently in the form of

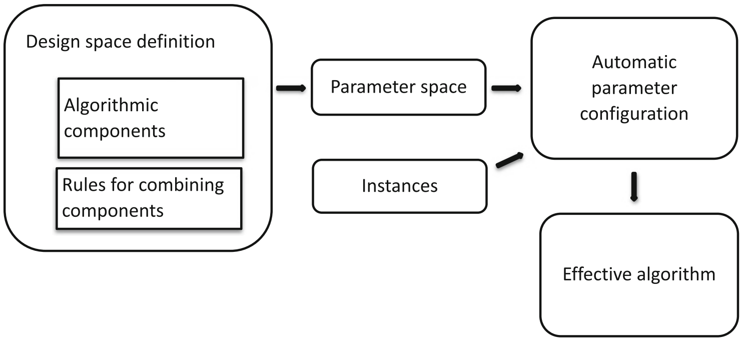


Fig. 2.2 General approach for deriving configurable algorithm frameworks

grammars. A first bottom-up approach is again for the SAT problem, based on a grammar-based approach. In fact, this is the approach that is commonly followed in the hyper-heuristic setting. In principle, it is a methodology that wants to generate constructive and perturbation heuristics and often these methods are based on grammatical evolution and gene expression programming. A different approach was followed by few others. They noticed that with standard software like irace they could replace grammatical evolution, a well established variant of grammar-based genetic programming, in a much more directed way. They first made a version that was based on Paradiseo as a white-box framework for the flexible design of metaheuristics. Later an framework called EMILI was designed from scratch and so far the results are very promising, producing state-of-the-art algorithms for a number of permutation flowshop problems.

2.4 An Example: irace

Let us continue this chapter with an example on irace configuring a simple iterated local search (ILS), which you can see also in Sect. 3.4 of this book, for the generalized assignment problem (GAP). For this, we assume that we have installed irace in R and we have a first ILS algorithm for the GAP. Now it is the time to improve this simple algorithm by configuring some of the parameters.

First, we have to notice that we do have infeasible and feasible solutions. Here we deal with it by generating all feasible solutions by counting infeasible ones with a penalty of 100 per integer infeasibility step. In other words, if we have, say, 27 infeasibilities, then we count them as $27 \cdot 100 = 2700$ in addition to the feasible part that is as usual in the GAP.

We have a few parameters of the ILS that we can change. The first part is the initial solution. Here we assume that the initial solution is a random one.

Next, we can address the acceptance criterion of our ILS. In the first place we have a better criterion, where we only accept improving solutions. As a slight change we may also accept the same solution quality as the best one. This may be better if we have many solutions in the neighborhood that are of the same quality. A rather different acceptance criterion would be one that accepts every new solution as the current one. An intermediate to this may be a temperature-based one. One common choice is to use the simulated annealing one; see Sect. 3.2 for details of simulated annealing. It always accepts a solution if it is better or equal to the current one. It instead accepts with a probability $\exp(-\delta/T_{Tcurrent})$ a worsening solution, depending on the worsening δ and the temperature $T_{Tcurrent}$. Here the temperature is kept constant throughout the run of the algorithm. Hence, we have two parameters of the acceptance options. The first, is a categorical acceptance criterion with 4 criteria. The second is a conditional parameter for the temperature $T_{Tcurrent}$, set only if the simulated annealing acceptance criterion is chosen as the first acceptance parameter.

As the next item we have the perturbation. Here we implement two types of perturbation. In the first case, we do a fixed setting that is kept constant independent of problem size, that is, perturbation is equal to p, with $p \in (2, 98)$. In the second case, we do a variable size perturbation in the style of variable neighborhood search (VNS); see Sect. 3.5 on VNS. We use a value of 2, which is the smallest possible value larger than the local search step, and configure the upper limit p, bounded to be at most 98. Hence, we also have two parameters for the perturbation: the first is the type of perturbation, fixed or variable, and the second is the value p.

As the last one, we can choose the way we use the local search. As first, we can choose no local search at all. This is fast, but we cannot assume it to be good enough. Hence, we use three more instances with local search. The first is a kind of first-improvement local search. It first examines the local search improvements that can be done for a specific server and if it is an improvement it applies it. Otherwise it goes to the next server. Once it has searched the server for each client, it stops the local search. In addition, it does a local search only if the initial solution is found to be feasible. However, notice that if a solution is found infeasible but better than the previous best one, it anyway replaces it. The two other local searches also perform a local search iteration when they are in an infeasible solution. The first is the very basic version of the second one, and it does one time the local search for each server when the infeasibility counts as one if present. In the second one an infeasible solution gets an *alpha* penalty, where alpha is an integer. In the case of the second implementation a true local minima is identified, that is, a server is visited more than once except if the solution is already a local optima. We know that we could identify more local searches as, for example, we could apply a two-opt one neighborhood. With this, also the final solution would probably improve. Yet, we leave this for further work, and we have two parameters as the local search: one is the type of local search, which is a categorical one with four parameters; the second is the value of *alpha* if the second local search is chosen.

With this, we can define the parameter file. This file has several inputs for each parameter, which are listed as

```
<name> <switch> <type> <range>  [ | <condition>].
```

Each entry then has the following meaning.

name	This is the name of the parameter as an alphanumeric string.
switch	A label for the parameter, which is a string that is used in the target runner of the system. In the default provided in the package it would be "$-l$" or similar. Note that here the user has to take care of giving the right size. Especially, the separator is important here and has to be how the code expects it.
type	This is the type of the parameter. It can be either real, integer, ordinal, or categorical represented by r, i, o, and c, respectively. Numerical parameter can be also sampled as r,log or as i,log for real and integer, respectively.
values	This is the range or the set of values. It is given as (0,1) or as (a, b, c, d).

```
# name            switch  type  values       [conditions (using R syntax)]
p_paramater        "-p "   i     (2, 98)
p_fixed_variable  "-z "   c     (0, 1)
acceptance         "-y "   c     (0, 1, 2, 3)
temperature_value "-f "   r     (0.0, 100.)  | Acceptance %in% c(3)
local_search       "-l "   c     (0, 1, 2, 3)
alpha              "-m "   i     (1, 100)     | Local_search %in% c(3)
```

Fig. 2.3 Parameter file (parameters.txt) for configuring ILS-GAP

condition	If the condition is false, then no value is assigned to the parameter. If the condition is true, then the condition must be in the same syntax as the valid R logical expression. The condition may use the same name or also use a different one as long as it does not have any cycles.

An example of the parameter file for the ILS can be seen in Fig. 2.3. We have six parameters, of which four are independent (two parameters for the perturbation, one for the choice of the acceptance criterion, and one for the choice of the local search). In addition there are two conditional parameters, one for the temperature parameter and the other for the *alpha* parameter. Recall that the temperature is a real parameter, which is as default 4 digits after the comma, and that the *alpha* is an integer within 1 and 100.

The `targetRunner` is the auxiliary program that actually runs the algorithm. The `targetRunner` is by default a minimization program. In the case of the GAP, where the default can be either a minimization or a maximization, it would have, if it is a maximization, returned to irace the result multiplied by -1. Yet, here we assume that it is minimization. Then the `targetRunner` is made out this

```
<id.configuration> <id.instance> <seed>
          <instance> [bound] <configuration>
```

The information is given as follows:

id.configuration	unique identifier of the configuration.
id.instance	unique identifier of the instance.
seed	seed for the random number generation; ignore it for a deterministic algorithm.
instance	instance to be used.
bound	optional execution time bound.
configuration	the pairs of the label and values of the parameters that describe the configuration.

So, for example, for us the program to run is ./ilsgap and then comes the fixed parameters of the candidate configuration, the instance ID, the seed of the parameter setting, and the instance that is run. Next, there would be a bound if it was an exact algorithm; but in this case it is not provided by irace, as the target algorithm is a metaheuristic. Then comes the part of the parameters. An important part is that the program (./ilsgap in our case) has to put a real number as the output, which

```
#!/bin/sh
CONFIG_ID=$1
INSTANCE_ID=$2
SEED=$3
INSTANCE=$4
CONFIG_PARAMS=$*

FIXED_PARAMS=" -t 3 "
STDOUT="c$CONFIG_ID-$INSTANCE_ID.stdout"
/home/stuetzle/Algorithms/GAP/GAP_new/ilsgap $FIXED_PARAMS \
                        -i $INSTANCE $CONFIG_PARAMS > $STDOUT
COST=$(tail -n 1 $STDOUT)
echo "$COST"
exit 0
```

Fig. 2.4 Target runner that configures the GAP for the ILS-GAP

corresponds to the real cost measure in our case. The working directory of the targetRunner then is the running directory set by the option `execDir`, which allows to run different runs of irace if necessary. The target runner is in Fig. 2.4 here in terms of a shell script.

Once we have the algorithm running, we need some training instances and some test instances. First, we have the training instances. The two options for irace are `trainInstancesDir` and `trainInstancesFile`, which is, however, empty as a default. In our case, we take the `trainInstancesDir`, which is `./GAP_instances/GAP_training`. We have generated instances of type C and D from Chu and Beasley, each of the instance classes with 100 and 200 clients for I with servers for 5, 10, and 20 for J. These are considered not too trivial for our purposes. For the C instances, we have generated the r_{ij} as integers from $U(5, 25)$ and the c_{ij} as integers from $U(10, 50)$ and as the bound has been generated as $b_i = 0.8 \cdot \sum_{j \in J} r_{ij}/m$. For the D instances, we have generated instances of r_{ij} as integers from $U(1, 100)$ and c_{ij} are integers from $111 - r_{ij} + e$, where the e are integers from $U(-10, 10)$. We again use $b_i = 0.8 \cdot \sum_{j \in J} r_{ij}/m$. For each size, we have generated 5 instances so that we have 60 instances. We verified that all instances were feasible for the best of our training elements. As test instances, we have used the instances of type A, B, C, and D that are available. We have not taken the instances A and B as training instances since they are known to be rather easy. As such, it is the 100 and 200 for I and the 5, 10, 20 for J. Note that the size of the instances is the same; this could have also been made different for the training set, but we leave this for later.

The irace has a number of further options. One is to have a text file that contains one or several initial configurations. In the case of the ILS on the GAP we can do so by starting from a random solution, with the better acceptance criterion, and do a perturbation with 5 random move and a first-improvement local search with an alpha. Hence, we would have the following ILS-GAP scenario as in Fig. 2.5.

```
## Initial candidate configuration for irace
initial  local search  perturbation  acceptance alpha
random   3             5             0          10
```

Fig. 2.5 Parameter file for specifying an initial configuration for ILS-GAP

Other options would be to forbid configurations or the repair of configurations, which are however not needed in our case. Something that should be useful is to run irace in parallel, which we do here by imposing 30 runs in parallel. Another thing that is useful in irace is the testing of the configuration. Here it may be the final best configuration `testNbElites = 1`, which is also the default that is returned. An alternative maybe is the first best configuration of every iteration of irace, which is done by `testIterationElites = 1`. With this, we can now run irace.

In the end, Fig. 2.6 gives the outcome of the execution of irace. The figure presents the first part of the output file up to the end of the first iteration and the last part comprising the last iteration and the final result. We note that the parameters and the details are both understandable. In the lower part of the figure, it is given the best configuration found, which is configuration 692 with the perturbation parameter 3 with variable perturbation, which varies between 2 and 3 perturbations, an acceptance criterion that is better or equal, and a local search that is doing the first-improvement local search. The two parameters temperature and alpha are not used. If one wants to do more analysis with irace, this can be done by the analysis of the results.

2.5 Related Literature

In this chapter we have seen how to perform parameter configuration for a matheuristics. We first argue that many people have been aware that categorical and numerical parameters are important (Birattari et al. 2002; Hutter et al. 2009, 2011; López-Ibáñez et al. 2016) in addition to only the numerical ones (Audet and Orban 2006; Nannen and Eiben 2006; Yuan et al. 2012). We then have seen the parameter configuration problem in a first version, as it was done by Birattari in his PhD thesis (Birattari 2004). We have classified the field into five classes. First of all, we used experimental design techniques that were used by Coy et al. (2001), Montgomery (2012), Ridge and Kudenko (2010), Ruiz and Maroto (2005), among others. Out of these, the CALIBRA approach (Adenso-Díaz and Laguna 2006) and the Coy et al. (2001) are well known, and next we highlighted the MADS (Audet and Orban 2006) and the REVAC ones as exemplary continuous techniques (Nannen and Eiben 2006, 2007; Smit and Eiben 2010). In our own work we have found CMAES to be among the best (Yuan et al. 2012, 2013). The majority of the systems fall in what we call heuristic search techniques. The first we know is that of Grefenstette from 1986. However, the majority belongs to the

```
#------------------------------------------------------------------------------
# irace: An implementation in R of (Elitist) Iterated Racing
# Version: 3.4.1.9fcaeaf
# Copyright (C) 2010-2020
# Manuel Lopez-Ibanez     <manuel.lopez-ibanez@manchester.ac.uk>
# Jeremie Dubois-Lacoste
# Leslie Perez Caceres    <leslie.perez.caceres@ulb.ac.be>
#
# This is free software, and you are welcome to redistribute it under certain
# conditions.  See the GNU General Public License for details. There is NO
# WARRANTY; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#
# irace builds upon previous code from the race package:
#     race: Racing methods for the selection of the best
#     Copyright (C) 2003 Mauro Birattari
#------------------------------------------------------------------------------
# installed at: /home/stuetzle/R/x86_64-redhat-linux-gnu-library/3.5/irace
# called with: --exec-dir=execdir6 --parallel 32 --mpi 1 --seed 1
Warning: A default scenario file './scenario.txt' has been found and will be read
# 2020-09-03 11:33:14 CEST: Initialization
# Elitist race
# Elitist new instances: 1
# Elitist limit: 2
# nbIterations: 4
# minNbSurvival: 4
# nbParameters: 6
# seed: 1
# confidence level: 0.95
# budget: 5000
# mu: 5
# deterministic: FALSE

# 2020-09-03 11:33:14 CEST: Iteration 1 of 4
# experimentsUsedSoFar: 0
# remainingBudget: 5000
# currentBudget: 1250
# nbConfigurations: 208
# Markers:
     x No test is performed.
     c Configurations are discarded only due to capping.
     - The test is performed and some configurations are discarded.
     = The test is performed but no configuration is discarded.
     ! The test is performed and configurations could be discarded but elite configurations are preserved.
     . All alive configurations are elite and nothing is discarded

+-+-----------+-----------+-----------+---------------+-----------+--------+-----+----+------+
| |   Instance|      Alive|       Best|      Mean best| Exp so far|  W time|  rho|KenW|  Qvar|
+-+-----------+-----------+-----------+---------------+-----------+--------+-----+----+------+
|x|          1|        208|        180|    3667.000000|        208|00:06:40|   NA|  NA|    NA|
|x|          2|        208|        169|    3224.000000|        416|00:13:00|+0.93|0.97|0.0399|
|x|          3|        208|          2|    4428.333333|        624|00:03:16|+0.78|0.85|0.5011|
|x|          4|        208|        180|    6688.000000|        832|00:12:45|+0.78|0.84|0.4350|
|-|          5|        123|        180|    5596.800000|       1040|00:13:09|+0.73|0.78|0.4833|
|-|          6|        103|        180|    6929.500000|       1163|00:03:41|+0.69|0.74|0.4941|
+-+-----------+-----------+-----------+---------------+-----------+--------+-----+----+------+
Best-so-far configuration:         180    mean value:     6929.500000
Description of the best-so-far configuration:
    .ID. perturbation_paramater perturbation_fixed_variable acceptance local_search temperatur_value alpha .PARENT.
180  180                      8                           1          0            1               NA    NA       NA


...


|.|         23|          4|        537|    6033.607143|         12|00:00:00|-0.02|0.01|0.7727|
|.|         16|          4|        537|    5923.172414|         12|00:00:00|-0.02|0.02|0.7678|
|.|          6|          4|        537|    6172.466667|         12|00:00:00|-0.02|0.01|0.7712|
+-+-----------+-----------+-----------+---------------+-----------+--------+-----+----+------+
Best-so-far configuration:         692    mean value:     6171.900000
Description of the best-so-far configuration:
    .ID. perturbation_paramater perturbation_fixed_variable acceptance local_search temperatur_value alpha .PARENT.
692  692                      3                           1          1            1               NA    NA      605

# 2020-09-03 17:46:39 CEST: Elite configurations (first number is the configuration ID; listed from best to worst
according to the mean value):
    perturbation_paramater perturbation_fixed_variable acceptance local_search temperatur_value alpha
692                      3                           1          1            1               NA    NA
709                      3                           1          1            1               NA    NA
537                      2                           1          0            1               NA    NA
605                      3                           1          1            1               NA    NA
# 2020-09-03 17:46:39 CEST: Stopped because there is not enough budget left to race more than the minimum (4)
# You may either increase the budget or set 'minNbSurvival' to a lower value
# Iteration: 13
# nbIterations: 13
# experimentsUsedSoFar: 4972
# timeUsed: 0
# remainingBudget: 28
# currentBudget: 28
# number of elites: 4
# nbConfigurations: 4
# Best configurations (first number is the configuration ID; listed
from best to worst according to the mean value):
    perturbation_paramater perturbation_fixed_variable acceptance local_search temperatur_value alpha
692                      3                           1          1            1               NA    NA
709                      3                           1          1            1               NA    NA
537                      2                           1          0            1               NA    NA
605                      3                           1          1            1               NA    NA
# Best configurations as commandlines (first number is the configuration ID; same order as above):
692  -p 3 -z 1 -y 1 -l 1
709  -p 3 -z 1 -y 1 -l 1
537  -p 2 -z 1 -y 0 -l 1
605  -p 3 -z 1 -y 1 -l 1
```

Fig. 2.6 Parameter file (parameters.txt) for configuring ILS-GAP

first and second decades from this century, and we may highlight ParamILS (Hutter et al. 2007b, 2009), gender-based genetic algorithms (Ansótegui et al. 2009), linear genetic programming (Oltean 2005), the OPAL system (Audet et al. 2010), or the EVOCA evolutionary algorithm (Riff and Montero 2013). In addition to these, we differentiate two final classes. The first are the surrogate model-based configurators of which we detail the SPOT system (Bartz-Beielstein et al. 2005) and the currently best-performing SMAC system (Hutter et al. 2011). The second are methods that use some racing methods. The first is the F-race approach (Birattari et al. 2002), which is then developed into the irace systems (López-Ibáñez et al. 2016; Pérez Cáceres et al. 2017). Of these methods, we give some details of ParamILS (Hutter et al. 2009), SMAC (Hutter et al. 2011), and of irace (López-Ibáñez et al. 2016).

In Sect. 2.3, we have classified the advantages of the automatic algorithm configuration. We first have shown that Hutter et al. (2007a) have excellent performance for a verification procedure and the same for mixed-integer programming procedures (Hutter et al. 2010). Similar results were obtained by F-race (Birattari et al. 2002), REVAC (Nannen and Eiben 2007; Smit and Eiben 2010), and with irace (Balaprakash et al. 2010; Liao et al. 2013, 2011; Pérez Cáceres et al. 2015). Other advantages are that sometimes old algorithms are getting surprisingly good results (Franzin and Stützle 2016). When doing algorithm (re-)engineering, automatic algorithm configurations software can also be very useful (Montes de Oca et al. 2011; Hartung and Hoos 2015; Balaprakash et al. 2010, 2015). The last effort is also the most used one. We divide this into top-down and bottom-up approaches. One of the first is the SATenstein approach (KhudaBukhsh et al. 2009, 2016). There is a lot also done with single objective algorithms like in Liao et al. (2014), Aydın et al. (2017), Franzin and Stützle (2019) or also multi-objective approaches (López-Ibáñez and Stützle 2010, 2012; Dubois-Lacoste et al. 2011; Bezerra et al. 2016, 2018, 2020). This top-down approach is feasible, if the algorithm to be used is rather straightforward. As an alternative, one can use the bottom-up approach. This was done again with the SAT approach first (Fukunaga 2004, 2008). In general, it is common if we follow the hyper-heuristic settings (Burke et al. 2013, 2019) as can be seen with a few settings (Burke et al. 2012; Sabar et al. 2015). In a different approach, we used irace to do these grammar-settings. In a first approach, we used Paradiseo as algorithmic framework (López-Ibáñez et al. 2017). This system was redesigned, and currently it has the best results for a number of flowshop problems (Pagnozzi and Stützle 2019; Alfaro-Fernández et al. 2020).

Finally, we used irace to configure a small example for an ILS for the GAP. For more details on the irace package, we refer to the irace user guide.

References

Adenso-Díaz B, Laguna M (2006) Fine-tuning of algorithms using fractional experimental design and local search. Oper Res 54(1):99–114

Alfaro-Fernández P, Ruiz R, Pagnozzi F, Stützle T (2020) Automatic algorithm design for hybrid flowshop scheduling problems. Eur J Oper Res 282(3):835–845, 2020

Ansótegui C, Sellmann M, Tierney K (2009) A gender-based genetic algorithm for the automatic configuration of algorithms. In: Gent IP (ed) Principles and practice of constraint programming, CP 2009. Lecture notes in computer science, vol 5732. Springer, New York, pp 142–157

Audet C, Orban D (2006) Finding optimal algorithmic parameters using derivative-free optimization. SIAM J Optim 17(3):642–664

Audet C, Dang C-K, Orban D (2010) Algorithmic parameter optimization of the DFO method with the OPAL framework. In: Naono K, Teranishi K, Cavazos J, Suda R (eds) Software automatic tuning: from concepts to state-of-the-art results. Springer, New York, pp 255–274

Aydın D, Yavuz G, Stützle T (2017) ABC-X: a generalized, automatically configurable artificial bee colony framework. Swarm Intell 11(1):1–38

Balaprakash P, Birattari M, Stützle T, Dorigo M (2010) Estimation-based metaheuristics for the probabilistic travelling salesman problem. Comput Oper Res 37(11):1939–1951

Balaprakash P, Birattari M, Stützle T, Dorigo M (2015) Estimation-based metaheuristics for the single vehicle routing problem with stochastic demands and customers. Comput Optim Appl 61(2):463–487

Bartz-Beielstein T, Lasarczyk C, Preuss M (2005) Sequential parameter optimization. In: IEEE CEC, Piscataway, NJ, September 2005. IEEE Press, New York, pp 773–780

Bezerra LCT, López-Ibáñez M, Stützle T (2016) Automatic component-wise design of multi-objective evolutionary algorithms. IEEE Trans Evol Comput 20(3):403–417

Bezerra LCT, López-Ibáñez M, Stützle T (2018) A large-scale experimental evaluation of high-performing multi- and many-objective evolutionary algorithms. Evol Comput 26(4):621–656

Bezerra LCT, López-Ibáñez M, Stützle T (2020) Automatically designing state-of-the-art multi- and many-objective evolutionary algorithms. Evol Comput 28(2):195–226

Birattari M (2004) The problem of tuning metaheuristics as seen from a machine learning perspective. PhD thesis, IRIDIA, Université Libre de Bruxelles, Belgium, 2004

Birattari M, Stützle T, Paquete L, Varrentrapp K (2002) A racing algorithm for configuring metaheuristics. In: Langdon WB, et al (eds) Proceedings of GECCO 2002. Morgan Kaufmann Publishers, San Francisco, CA, pp 11–18

Burke EK, Hyde MR, Kendall G (2012) Grammatical evolution of local search heuristics. IEEE Trans Evol Comput 16(7):406–417

Burke EK, Gendreau M, Hyde MR, Kendall G, Ochoa G, Özcan E, Qu R (2013) Hyper-heuristics: a survey of the state of the art. J Oper Res Soc 64(12):1695–1724

Burke EK, Hyde MR, Kendall G, Ochoa G, Özcan E, Woodward JR (2019). A classification of hyper-heuristic approaches: revisited. In: Gendreau M, Potvin J-Y (eds) Handbook of metaheuristics, chap 14. Springer, New York, pp 453–477

Coy SP, Golden BL, Runger GC, Wasil EA (2001) Using experimental design to find effective parameter settings for heuristics. J Heuristics 7(1):77–97

Dubois-Lacoste J, López-Ibáñez M, Stützle T (2011) Automatic configuration of state-of-the-art multi-objective optimizers using the TP+PLS framework. In: Krasnogor N, Lanzi PL (eds) Proceedings of the GECCO 2011. ACM Press, New York, NY, pp 2019–2026

Franzin A, Stützle T (2016) Exploration of metaheuristics through automatic algorithm configuration techniques and algorithmic frameworks. In: Friedrich T, Neumann F, Sutton AM (eds) GECCO (companion). ACM Press, New York, NY, pp 1341–1347

Franzin A, Stützle T (2019) Revisiting simulated annealing: a component-based analysis. Comput Oper Res 104, 191 – 206

Fukunaga AS (2004) Evolving local search heuristics for SAT using genetic programming. In: Deb K, et al (eds.) Proceedings of the GECCO 2004, Part II. Lecture notes in computer science, vol 3103. Springer, New York, pp 483–494

Fukunaga AS (2008) Automated discovery of local search heuristics for satisfiability testing. Evol Comput 16(1):31–61

Grefenstette JJ (1986) Optimization of control parameters for genetic algorithms. IEEE Trans Syst Man Cybernet 16(1):122–128

Hartung S, Hoos HH (2015) Programming by optimisation meets parameterised algorithmics: a case study for cluster editing. In: Dhaenens C, Jourdan L, Marmion M-E (eds) LION 9. Lecture notes in computer science, vol 8994. Springer, New York, pp 43–58

Hutter F, Babić D, Hoos HH, Hu AJ (2007a) Boosting verification by automatic tuning of decision procedures. In: Proceedings of the FMCAD'07, Austin, TX, 2007a. IEEE Computer Society, Washington, DC, pp 27–34

Hutter F, Hoos HH, Stützle T (2007b) Automatic algorithm configuration based on local search. In: Holte RC, Howe A (eds) Proceedings of the AAAI '07. AAAI Press/MIT Press, Menlo Park, CA, pp 1152–1157

Hutter F, Hoos HH, Leyton-Brown K, Stützle T (2009) ParamILS: an automatic algorithm configuration framework. J Artif Intell Res 36:267–306

Hutter F, Hoos HH, Leyton-Brown K (2010) Automated configuration of mixed integer programming solvers. In: Lodi A, Milano M, Toth P (eds) Proceedings of the CPAIOR 2010. Lecture notes in computer science, vol 6140. Springer, New York, pp 186–202

Hutter F, Hoos HH, Leyton-Brown K (2011) Sequential model-based optimization for general algorithm configuration. In: Coello Coello CA (ed) LION 5. Lecture notes in computer science, vol 6683. Springer, New York, pp 507–523

KhudaBukhsh AR, Xu L, Hoos HH, Leyton-Brown K (2009) SATenstein: automatically building local search SAT solvers from components. In: Boutilier C (ed) Proceedings of the IJCAI-09. AAAI Press, Menlo Park, CA, pp 517–524

KhudaBukhsh AR, Xu L, Hoos HH, Leyton-Brown K (2016) SATenstein: automatically building local search SAT Solvers from components. Artif Intell 232:20–42

Liao T, Montes de Oca MA, Stützle T (2011) Tuning parameters across mixed dimensional instances: a performance scalability study of Sep-G-CMA-ES. In: Krasnogor N, Lanzi PL (eds) GECCO (companion). ACM Press, New York, NY, pp 703–706

Liao T, Montes de Oca MA, Stützle T (2013) Computational results for an automatically tuned CMA-ES with increasing population size on the CEC'05 benchmark set. Soft Comput 17(6):1031–1046

Liao T, Stützle T, Montes de Oca MA, Dorigo M (2014) A unified ant colony optimization algorithm for continuous optimization. Eur J Oper Res 234(3):597–609

López-Ibáñez M, Stützle T (2010) Automatic configuration of multi-objective ACO algorithms. In: Dorigo M, et al (eds) ANTS 2010. Lecture notes in computer science, vol 6234. Springer, New York, pp 95–106

López-Ibáñez M, Stützle T (2012) The automatic design of multi-objective ant colony optimization algorithms. IEEE Trans Evol Comput 16(6):861–875

López-Ibáñez M, Dubois-Lacoste J, Pérez Cáceres L, Birattari M, Stützle T (2016) The irace package: Iterated racing for automatic algorithm configuration. Oper Res Perspect 3:43–58

López-Ibáñez M, Kessaci M-E, Stützle T (2017) Automatic design of hybrid metaheuristics from algorithmic components. Technical Report TR/IRIDIA/2017-012, IRIDIA, Université Libre de Bruxelles, Belgium, 2017

Montes de Oca MA, Aydın D, Stützle T (2011) An incremental particle swarm for large-scale continuous optimization problems: an example of tuning-in-the-loop (re)design of optimization algorithms. Soft Comput 15(11):2233–2255

Montgomery DC (2012) Design and analysis of experiments, 8th edn. Wiley, New York, NY

Nannen V, Eiben AE (2006) A method for parameter calibration and relevance estimation in evolutionary algorithms. In: Cattolico M, et al (eds) Proceedings of the genetic and evolutionary computation conference, GECCO 2006, pp 183–190. ACM Press, New York, NY, pp 183–190

Nannen V, Eiben AE (2007) Relevance estimation and value calibration of evolutionary algorithm parameters. In: Veloso MM (ed) Proceedings of IJCAI-07. AAAI Press, Menlo Park, CA, pp 975–980

Oltean M (2005) Evolving evolutionary algorithms using linear genetic programming. Evol Comput 13(3):387–410

Pagnozzi F, Stützle T (2019) Automatic design of hybrid stochastic local search algorithms for permutation flowshop problems. Eur J Oper Res 276:409–421

Pérez Cáceres L, López-Ibáñez M, Stützle T (2015) Ant colony optimization on a limited budget of evaluations. Swarm Intell 9(2–3):103–124

Pérez Cáceres L, López-Ibáñez M, Hoos HH, Stützle T (2017) An experimental study of adaptive capping in irace. In: Battiti R, Kvasov DE, Sergeyev YD (eds) LION 11. Lecture notes in computer science, vol 10556. Springer, Cham, pp 235–250

Ridge E, Kudenko D (2010) Tuning an algorithm using design of experiments. In: Bartz-Beielstein T, Chiarandini M, Paquete L, Preuss M (eds) Experimental methods for the analysis of optimization algorithms. Springer, Berlin, pp 265–286

Riff M-C, Montero E (2013) A new algorithm for reducing metaheuristic design effort. In: Proceedings of CEC 2013. IEEE Press, Piscataway, NJ, pp 3283–3290

Ruiz R, MarotoC (2005) A comprehensive review and evaluation of permutation flowshop heuristics. Eur J Oper Res 165(2):479–494

Sabar NR, Ayob M, Kendall G, Qu R (2015) Automatic design of a hyper-heuristic framework with gene expression programming for combinatorial optimization problems. IEEE Trans Evol Comput 19(3):309–325

Smit SK, Eiben AE (2010) Beating the 'world champion' evolutionary algorithm via REVAC tuning. In: Ishibuchi H, et al (eds) Proceedings of CEC 2010. IEEE Press, Piscataway, NJ, pp 1–8

Yuan Z, Montes de Oca MA, Stützle T, Birattari M (2012) Continuous optimization algorithms for tuning real and integer algorithm parameters of swarm intelligence algorithms. Swarm Intell 6(1):49–75

Yuan Z, Montes de Oca MA, Stützle T, Lau HC, Birattari M (2013) An analysis of post-selection in automatic configuration. In: Blum C, Alba E (eds) Proceedings of the genetic and evolutionary computation conference, GECCO 2013. ACM Press, New York, NY, pp 1557–1564

Part II
Metaheuristic Hybrids

Chapter 3
Single Solution Metaheuristics

3.1 Introduction

Metaheuristic approaches can be classified according to different criteria, one being the number of solutions that are evolved at each stage of the algorithm: one single solution or more than one. This chapter deals with metaheuristic algorithms that evolve one single solution; they are all enhancements of a basic local search procedure. Many different approaches have been presented, which could be included here. We choose six of them, namely Simulated Annealing, Tabu Search, GRASP, Iterated Local Search, Variable Neighborhood Search, and Ejection chains, as representative of the class and as the most widely used in the literature.

Matheuristic algorithms have been used to complement each of them, along with others of the unreported ones. The techniques used to include mathematical components in the basic structure of the considered metaheuristic tend to be general and independent of the specific single solution metaheuristic of interest.

3.2 Simulated Annealing

Simulated Annealing (SA) has possibly been the first of a wave of algorithms, which included steps inspired by some natural process that happens to optimize something. In the case of SA, the natural process is (obviously) annealing, a thermal process that attains low free energy states in a solid by means of repeated heating and slow cooling phases. This process can be simulated in different ways, one being a Monte

Carlo. At the core of the simulation is a time series of states, which can be described by a Markov chain that can be loosely summarized by the following cycle:

1. Given a state i with energy E_i, perturb it to generate the subsequent state j, with energy E_j.
2. If $E_j - E_i \leq 0$ accept j, otherwise accept j with probability $exp(-(E_j - E_i)/(k_B T))$,where k_B is the Boltzmann constant and T the temperature.

If the temperature decreases slowly enough, the material can reach, at each temperature T, a thermal equilibrium characterized by a Boltzmann distribution of the energy states: $P_T(i) = exp(-E_i/(k_B T) \Big/ \sum_j exp(-E_j/(k_B T))$.

This thermodynamic process that minimizes free energy gave rise to the idea that something similar could be effective for minimizing a generalized cost function, and this resulted in the SA algorithm. SA is thus a global search algorithm, which includes a "temperature" parameter T. At "high temperatures" search is diversified, at "low temperatures" search is intensified. Temperature T starts at a high value, then it gets decreased (this mimics annealing). At each temperature, the algorithm proposes a new solution in the neighborhood of the incumbent one and accepts it, if it is an improving solution. If the new solution is worse than the incumbent one, it can accept it all the same, but with a probability of acceptance expressed as an increasing function of T and decreasing with the amount of worsening.

The general structure is therefore similar to the local search algorithm presented in Sect. 1.3.2, where an incumbent solution $\mathbf{x}$ is iteratively updated, possibly moving it to another solution $\mathbf{x}'$ in its neighborhood $\mathscr{N}(\mathbf{x})$, except that in the case of SA the moves can be made also toward worsening solutions. The worse solutions are accepted in probability, where the probability of acceptance decreases with the decrease in quality of the solution. The move acceptance formula for worsening solutions mimics the Metropolis formula, derived in turn from the numerator of the Boltzmann equation, where the energy values are replaced by the objective function values (i.e., high energy corresponds to high cost). Since the anneal permits an effective decrease of the free energy, the simulated anneal hopefully provides an effective means to decrease the solution costs.

The pseudocode of a generic SA is presented in Algorithm 6.

Line (8) implements the probabilistic acceptance formula of worsening moves.

Several elements need to be specified, besides the initial temperature value and the neighborhood function, among which are:

- the terminating condition (when to terminate the execution);
- the annealing condition (when to decrease the temperature);
- the cooling schedule (how much is the temperature to be decreased).

It has been proven that SA is guaranteed to find an optimal solution provided that the neighborhood is such that any solution is reachable from any other via a suitable chain of moves, that the initial temperature is high enough, that the cooling is slow enough, and that sufficient (possibly infinite) time is given.

Algorithm 6: Generic Simulated Annealing

```
1  function SimulatedAnnealing($T$);
   Input  : temperature $T$
   Output: A feasible solution $\mathbf{x}^*$
2  Generate a feasible solution $\mathbf{x}$; Set $\mathbf{x}^* = \mathbf{x}$;
3  Generate a feasible solution $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$;
4  if $(z(\mathbf{x}') < z(\mathbf{x}))$ then
5      Set $\mathbf{x} = \mathbf{x}'$;
6      if $(z(\mathbf{x}^*) > z(\mathbf{x}))$ then $\mathbf{x}^* = \mathbf{x}$;
7  else
8      Set $\mathbf{x} = \mathbf{x}'$ with probability $p = e^{-(z(\mathbf{x}')-z(\mathbf{x}))/(kT)}$;
9  end
10 if (annealing condition) then decrease $T$;
11 if not(terminating condition) then go to 3;
12 return $\mathbf{x}^*$;
```

Being so well-established in the optimization community, SA has been also among the first metaheuristics, which were integrated by an explicit use of a mathematical model of the problem to solve. One early example observed that a straightforward SA, on a problem of airline crew scheduling, provides quick and fairly good solutions but it is not stable, as the quality of the solutions varies greatly. Therefore the authors proposed, similarly to the method described in Chap. 9, to collect the SA solutions and include them as columns in a column generation-like master problem. They found that both improved the final solution and guarantee much more stability to the proposed results.

One problem for which SA has been repeatedly applied as a matheuristic is the timetabling problem, with opposite approaches. One approach verified the possibility to use SA to generate an initial feasible solution, which was later improved by a Very Large-Scale Neighborhood Search (VLSN, see Chap. 6), where the neighborhood was explored by solving an Integer Programming problem. Computational results show that the SA/VLSN hybrid improves over each of the constituents alone. The opposite approach, first go mathematical then metaheuristic, was also tested in a work where an initial solution is obtained by a Lagrangian relaxation based approach, then the solution is improved by a SA.

The problem of the Fixed-Cost Capacitated Multicommodity Network Design was faced as a bilevel optimization problem, where the upper-level optimization task is delegated to the SA, which specifies open and closed arcs, and the inner optimization task is solved by a Simplex method, which determines the amount of flows on the open arcs that were determined by the SA algorithm.

3.2.1 SA for the GAP, An Example

All examples from the literature are not trivial to translate to the GAP so, for illustrative purposes, we design a multiplier adjustment algorithm based on SA. To this end, we dualize the capacity constraints, obtaining formulation AGAP.

$$(AGAP) \qquad z_{AGAP} = \min \sum_{i \in I} \sum_{j \in J} c'_{ij} x_{ij} \tag{3.1}$$

$$s.t. \quad \sum_{i \in I} x_{ij} = 1, \qquad j \in J \tag{3.2}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I, j \in J \tag{3.3}$$

where the costs are initialized as $c'_{ij} = c_{ij}$, $i \in I$ and $j \in J$, and are updated as $c'_{ij} = c'_{ij} - \sum_{i \in I} \epsilon_i (Q_i - \sum_{j \in J} q_{ij} x_{ij})\}$ at each iteration, when the corresponding constraint is not satisfied. The parameters ϵ_i, $i \in I$, are user-defined parameters with a default value of 1. Actually, costs and requests should have normalized values, in order to make them comparable, but this would unnecessarily complicate this simple example, whose objective is only to show the working of a SA-based matheuristic.

The complete code of the matheuristic is presented in Algorithm 7. The general structure is the same as that of Algorithm 6, except that the neighborhood considered at step 4 is defined in the AGAP search space and that costs are updated in step 16 to drive the evolution toward feasibility.

In this code, a standard SA is thus called to solve problem AGAP, with the cost updates as per step 16 and all user-defined ϵ parameters unitary. Assume, as an actual execution trace shows, that the first few iterations all produced infeasible solutions due to capacity overloads. The overloaded servers have therefore their costs increased. Let the cost matrix become as in Eq. (3.4) and the current solution be as in Eq. (3.5), which is infeasible because of the capacities.

$$\mathbf{c}' = \begin{bmatrix} 226 & 227 & 228 & 229 & 230 & 231 & 232 & 233 \\ 237 & 239 & 241 & 243 & 245 & 247 & 249 & 251 \\ 236 & 240 & 244 & 248 & 252 & 256 & 260 & 264 \end{bmatrix} \tag{3.4}$$

$$\mathbf{x} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.5}$$

Algorithm 7: Matheuristic Simulated Annealing for the GAP

```
1  function MathSimulatedAnnealing($T$,$E$);
   Input  : temperature $T$, multipliers $E = [\epsilon_i]$
   Output: A feasible solution $\mathbf{x}^*$
2  Generate a feasible GAP solution $\mathbf{x}$; Set $\mathbf{x}^* = \mathbf{x}$;
3  Set $\mathbf{c}' = \mathbf{c}$;
4  Generate a feasible AGAP solution $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$;
5  if $(z(\mathbf{x}') < z(\mathbf{x}))$;                          // standard SA steps
6   then
7      Set $\mathbf{x} = \mathbf{x}'$;
8      if $(z(\mathbf{x}^*) > z(\mathbf{x}))$ then $\mathbf{x}^* = \mathbf{x}$;
9  else
10     Set $\mathbf{x} = \mathbf{x}'$ with probability $p = e^{-(z(\mathbf{x}')-z(\mathbf{x}))/(kT)}$;
11 end
12 if $\mathbf{x}$ is infeasible then
13     Let $L$ be a list of all overloaded servers;
14     foreach server $i \in L$ do
15         foreach $j \in J$ do
16             $c'_{ij} = c'_{ij} - \sum_{i \in I} \epsilon_i (Q_i - \sum_{k \in J} q_{ik} x_{ik})\}$;        // cost update
17         end
18     end
19 end
20 if (annealing condition) then decrease T;
21 if not(terminating condition) then go to 4;
22 return $\mathbf{x}^*$;
```

Step 4 proposes a solution which reassigns client 5 to server 2. The move is a worsening one ($c'_{15} = 230$ and $c'_{25} = 245$), but with the still initial $T = 200$ it can be accepted with probability $p = 0.86$. It is actually accepted in this trace, thereby generating solution (3.6).

$$\mathbf{x} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.6}$$

The vector of unused server capacities for solution (3.6) is $[69, 21, -26]$. The solution is still infeasible for server 3 and its costs get updated, obtaining the cost matrix of Eq. (3.7).

$$\mathbf{c}' = \begin{bmatrix} 226 & 227 & 228 & 229 & 230 & 231 & 232 & 233 \\ 237 & 239 & 241 & 243 & 245 & 247 & 249 & 251 \\ 262 & 266 & 270 & 274 & 278 & 282 & 286 & 290 \end{bmatrix} \tag{3.7}$$

A new iteration is started, this time proposing an improving move, which reassigns client 2 to server 1, generating the solution (3.8).

$$\mathbf{x} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.8}$$

This is a feasible solution, the residual capacities are [22, 21, 1], of actual cost $z_{GAP} = 334$.

3.3 Tabu Search

Tabu Search (TS) is a well-known iterative procedure, which builds on a core local search and tries to help it escape from local optima. To this end, it makes use of auxiliary memory structures aimed at preventing the algorithm to repeatedly visit the same solutions. Data on past search is in fact stored in a structure called *tabu list*, and used to limit the successive moves. This results in the possibility to accept worsening moves.

The tabu list helps to prevent cycles because recent moves are declared *tabu* and cannot be reverted unless they are proven to lead to unexplored solutions. One case where tabus can be overridden is when an otherwise forbidden move would lead to a best-so-far solution, in this case, the so-called *aspiration criterium* permits the move execution. Actually, the aspiration criterion can be tuned by the user, specifying an *aspiration level* to attain.

The memory of past search is always stored in the tabu list, acting as a short-term memory structure, and possibly also in other structures, implementing a long-term memory. Moves in the tabu list are forbidden for a number of iterations specified by a parameter called *tabu tenure*. This permits escaping from local optima and search diversification. The long term memory usually collects information about the explored regions of the search space it is used to direct search toward unexplored regions, thereby providing a strategic diversification guidance.

The interaction of the basic local search and move denials results in a balance between search intensification around current solutions and search diversification, forcing the exploration of unvisited search space regions.

A high level pseudocode of a generic TS is presented in Algorithm 8.

TS has enjoyed considerable success, both in the academy and in industry, and has given rise to many variants. Some of the matheuristic extensions are listed in the following.

A first matheuristic was applied to the so-called *m-peripatetic* vehicle routing problem, which is a special vehicle routing problem asking that each arc is used in the solution at most once for each set of *m* periods in the given time horizon. The approach used a perfect b-matching to define the candidate sets to be passed to

Algorithm 8: Generic Tabu Search

```
1 function TabuSearch($TT$);
  Input  : tabu tenure $TT$
  Output: A feasible solution $\mathbf{x}^*$
2 Generate a feasible solution $\mathbf{x}$; Set $\mathbf{x}^* = \mathbf{x}$ and $TL = \emptyset$;
3 Generate a feasible solution $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$ such that $z(\mathbf{x}') = min\{z(\hat{\mathbf{x}}), \hat{\mathbf{x}} \in \mathcal{N}(\mathbf{x}), \hat{\mathbf{x}} \notin TL$
    or $z(\hat{\mathbf{x}}) < z(\mathbf{x}^*\}$;                              // aspiration condition
4 Set $\mathbf{x} = \mathbf{x}'$, $TL = TL \cup \{\mathbf{x}\}$;
5 if ($|TL| > TT$) then remove from TL the oldest element;
6 if ($z(\mathbf{x}^*) > z(\mathbf{x})$) then set $\mathbf{x}^* = \mathbf{x}$;
7 if (tabu tenure update condition) then update $TT$;
8 if not(terminating condition) then go to 3;
9 return $\mathbf{x}^*$;
```

a granular tabu search algorithm, which is a TS where only a subset of neighbors is checked in the neighborhood of the current solution, at each move. The list that defined the neighborhood is composed of the set of unused edges that are in the solution of b-matching. Computational results executed on classic instances of VRP and TSP showed that the hybridized TS performs better than simple TS.

A more elaborate approach on the multicommodity fixed-charge network design problem makes use of MIP to initialize a solution, then tabu search explores the neighborhood in a VLNS fashion (see Chap. 6). Moreover, data is collected during TS and used to generate cuts in MIP phases to reinitialize search. The same problem has more recently been faced with another matheuristic, an iterative linear programming-based heuristic, with improved results. Also for the undirected capacitated arc routing problem, it is possible to combine TS and MIP, using MIP to try to improve each new best solution found by TS.

A combination of a cutting-plane neighborhood structure and a tabu search metaheuristic was presented for the capacitated p-median problem. In the proposed neighborhood structure, to move from the current solution to a neighbor solution, an open median is selected and closed. Then, a linear programming model is generated by relaxing binary constraints and adding new constraints. The generated linear programming solution is improved using cutting-plane inequalities. The solution to this stronger linear programming model is considered as a new neighbor solution.

3.3.1 *TS for the GAP, An Example*

We present an extension of the basic TS along the lines suggested above, where MIP is used to try to improve each new best solution found by TS. Specifically, at each new best solution we call a RINS diving heuristic (see Sect. 5.2) as a local search procedure which searches for a possibly improving local optimum.

The pseudocode of the complete algorithm is presented in Algorithm 9. In this algorithm, the tabu list TL is implemented as a $m \times n$ matrix of integers, $TL = [tl_{ij}]$, where each cell specifies the first iteration when the client corresponding to the column index can be unassigned from the server corresponding to the row index. Initially, the matrix is all filled with zeros. The neighborhood of a solution $\mathbf{x}$, $\mathcal{N}(\mathbf{x})$, is assumed to be a *opt10* neighborhood, consisting of all feasible solutions that differ from $\mathbf{x}$ only for the assignment of one single client.

Algorithm 9: Matheuristic Tabu Search for the GAP

```
1  function MathTabuSearch($TT$);
   Input  : tabu tenure $TT$
   Output: A feasible solution $\mathbf{x}^*$
2  Generate a feasible solution $\mathbf{x}$; Set $\mathbf{x}^* = \mathbf{x}$ and $TL = \emptyset$;
3  Set $iter = 0$ and $TT_0 = TT$;
4  while not(terminating condition) do
5      Generate a feasible solution $\mathbf{x}' \in \mathcal{N}(\mathbf{x})$ such that
         $z(\mathbf{x}') = min\{z(\hat{\mathbf{x}}), \hat{\mathbf{x}} \in \mathcal{N}(\mathbf{x}), \hat{\mathbf{x}} \notin TL\}$;
6      Let $j$ be the client with different assignment in $\mathbf{x}'$ and $\mathbf{x}$, and $i$ the server it is assigned
         to in $\mathbf{x}'$;
7      Set $\mathbf{x} = \mathbf{x}'$, $tl_{ij} = tl_{ij} + iter$;
8      if $(z(\mathbf{x}^*) > z(\mathbf{x}))$ then
9          $\mathbf{x} = dive(\mathbf{x})$;
10     end
11     if $(z(\mathbf{x}^*) > z(\mathbf{x}))$ then set $\mathbf{x}^* = \mathbf{x}$;
12     if (tabu tenure update condition) then
13         $TT = TT_0 * randbetween(0.7, 1.3)$;          // tabu tenure supdate
14     end
15     $iter = iter + 1$;
16 end
17 return $\mathbf{x}^*$;
```

As an example, consider the following execution trace. The solution generated at step 2 of cost 343 is the one produced by the simple constructive heuristic described in Sect. 1.3.1, that is

$$\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{3.9}$$

Solution (3.9) can be equivalently expressed as $\boldsymbol{\sigma} = (0, 0, 0, 1, 1, 2, 2, 2)$. Step 5, based on a basic *2-opt* neighborhood, identifies a solution in the neighborhood of $\mathbf{x}$ obtained by setting $x_{12} = x_{05} = 1$ and $x_{00} = x_{25} = 0$. This produces a new solution $\boldsymbol{\sigma} = (0, 2, 0, 1, 1, 0, 2, 2)$ of cost 331, which leaves an array of available capacities equal to $(23, 21, 0)$.

Assuming a tabu tenure equal to 6, the tabu list, in matrix form, becomes

$$\mathbf{TL} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.10}$$

At this point diving (see Chap. 5) is instantiated on this last solution, which improves the solution, discovering $\boldsymbol{\sigma} = (1, 3, 2, 2, 1, 1, 3, 3)$ of cost 329. Search then goes on with the following iterations of loop 4.

3.4 Iterated Local Search

Iterated Local Search (ILS) represents another approach for avoiding local search to get stuck in a local optimum. The underlying idea is implementing a sampling of the search space based on the local optima identified by any specific local search procedures. One obvious possibility to this end would be to repeat a random generation of a starting solution and then run the local optimizer on it. Experimental evidence has shown that a tighter control of the starting solutions can lead to much improved results.

In ILS, a guided core local search heuristic generates a sequence of solutions, which usually leads to better results than if one were to use repeated random trials of that same core heuristic. The core algorithm is very simple, as it only prescribes to start from a solution, find its local optimum with reference to a specific local search procedure, perturb the incumbent solution, optimize this new one, and so on.

More in detail, a generic ILS algorithm is presented in Algorithm 10.

Algorithm 10: Generic Iterated Local Search

```
1  function IteratedLocalSearch();
   Input  :
   Output: A feasible solution $\mathbf{x}^*$
2  Generate an initial feasible solution $\mathbf{x}_0$;
3  Apply a local search on $\mathbf{x}_0$ obtaining a local optimum $\mathbf{x}^\circ$;
4  repeat
5      Set $\mathbf{x}'$ = Perturbation($\mathbf{x}^\circ$, $history$);
6      Set $\mathbf{x}''$ = LocalSearch($\mathbf{x}'$);
7      Set $\mathbf{x}^\circ$ = AcceptanceCriterion($\mathbf{x}^\circ$, $\mathbf{x}''$, $history$);
8      if $(z(\mathbf{x}^*) > z(\mathbf{x}^\circ))$ then set $\mathbf{x}^* = \mathbf{x}^\circ$;
9  until termination condition;
10 return $\mathbf{x}^*$;
```

The algorithm builds upon a number of internal procedures. A feasible solution is generated at step 2, and this can be done by any constructive approach. A good constructive approach may be useful so that the starting solution optimizes the anytime behavior, though the results often turn out to be significantly independent from the starting solution.

Step 6 is likewise very general, as any local search procedure can be included, from simple ones such as *2-opt* and *3-opt* (i.e., complete explorations of changes of 2 or 3 solution components) to very involved ones such as Lin-Kernighan (for the TSP) or local branching (see Sect. 5.3).

What characterizes most of the results eventually obtained are the two remaining steps. Step 5 requires to modify the current solution into a new one, which is not too close to the original one, otherwise it would fall into the same local optimum basin of attraction, nor too far from it, otherwise the whole algorithm essentially becomes a random restart. Moreover, small - but not *too* small - perturbations are usually beneficial, as the resulting solution is not far from the corresponding local optimum, therefore the local search step is fast. It is helpful to this end to be able to control the amount of perturbation to be imposed on the instance to solve.

An interesting idea in this regard suggests to apply the same local search procedure used for local optimization also to solution perturbation. This is obtained by introducing perturbations in the problem data rather than in the solution, for example, running the optimization on a perturbed cost matrix. The cost perturbation can be made in a much more statistically principled way, therefore the perturbation phase can be better controlled.

Finally, step 7 permits to control the trade-off between intensification and diversification of search. It is in fact possible to range between accepting a new solution only if it is better than the best so far (extreme intensification, accept only if $\mathbf{x}'' < \mathbf{x}^*$) to accepting it in any case (extreme diversification, always set $\mathbf{x}^\circ = \mathbf{x}''$). Usually, intermediate choices are more effective, and these can include elements from other metaheuristics (for example, tabu lists, candidate lists, acceptance criteria, etc.) but also general directives from the global search history, for example, by forcing a random restart if there has been no improvement for too many iterations.

Coming to matheuristics, several examples of hybrids between exact methods and heuristics can be found in the context of ILS.

For example, an IP-based method was described to solve Machine Reassignment Problems (MRP), which are problems that were proposed in the Google ROADEF/EURO Challenge (2012), and require to find a reassignment of processes to machines that optimizes the usage of machine resources, possibly improving an initial given assignment. The method implements a few different versions of ILS, two of which had the perturbation phase based on an IP formulation of the problem. Essentially, the method proposes to select a subset M' of machines at random and to build a restricted MRP, that asks to reassign the processes, which were assigned to those machines but always considering only alternatives within M'. The computational results, reported on the Google ROADEF/EURO Challenge instances, demonstrate the good effectiveness of the IP perturbations, both with respect to the other possible perturbations and to the other heuristics for the problem.

Different ways for including mathematical components into ILS have been proposed, for example, for solving a cutting stock problem. The approach considers the problem of reducing the number of *cutting patterns* in the solution of a one-dimensional cutting stock problem, where a cutting pattern is a combination of items to be cut from a master surface. Since switching from a pattern to another is costly, reducing the need for such switches becomes important. The basic cutting stock problem is complicated by a constraint on the number of patterns that can be used, and they search for a solution whose quadratic deviation of the cut products from the demands is sufficiently small. This is solved by an ILS procedure, which includes a so-called Adaptive Pattern Generator (APG) phase, that—in turn—is based on Sahni's heuristic algorithm for the 0–1 knapsack problem (Sahni 1975).

3.4.1 ILS for the GAP, An Example

As an example, we will adapt the method of Lopes et al. (2015) to the case of the GAP. Keeping the general approach, we will consider our servers as the MRP machines and our clients as the MRP processes. The idea is therefore to select a subset of servers and to try to reassign among them the clients that they had assigned in the seed solution. This actually corresponds to using an instantiation of a well-known matheuristic approach, namely the *Corridor Method*, which will be presented in detail in Chap. 8, even when the approach could also be included in the set of *very large-scale neighborhood search* (VLSN, see Chap. 6).

The pseudocode of the matheuristic based on ILS for the GAP is presented in Algorithm 11. The algorithm makes use of three subroutines:

- *DataPerturbation*: the perturbation module at the core of the method. In the example, we assume to have a data perturbation subroutine, rather than a solution perturbation one. Internally, the subroutine generates a normally distributed random number in correspondence to each cost coefficient, adds the number to the coefficient, obtains a modified cost matrix $\mathbf{C}'$, and calls LocalSearch() in order to optimize the current solution using the modified cost matrix. This produces a perturbed solution.
- *LocalSearch*: simple local search identifying a local optimum. In case of data perturbation, it accepts as parameters both the initial solution to be moved to its local optimum and the cost coefficients matrix to use for optimization, which could be either the original instance cost matrix or the perturbed one, as produced by the DataPerturbation subroutine.
- *LocalSearchMIP*: MIP-based local search. In our case, it explores a k-opt neighborhood by fixing all decision variables but k in the incumbent solution and by calling a solver to optimize the resulting subproblem (see Chaps. 6 and 8 for details).

In this case, the algorithm implements a basic iterated local search, and calls a MIP restricted neighborhood exploration whenever a new best upper bound is found.

The $LocalSearch$ subroutine in this example is a *opt11*, exchange the servers to which two clients were assigned because the initial solution is already *opt10*

Algorithm 11: Matheuristic Iterated Local Search for the GAP

```
1  function IteratedLocalSearchForGAP();
   Input :
   Output: A feasible solution $\mathbf{x}^*$
2  Generate an initial feasible solution $\mathbf{x}_0$;
3  $\mathbf{x}^\circ$ = LocalSearch($\mathbf{x}_0$, $\mathbf{C}$);
4  repeat
5      Set $\mathbf{x}'$ = DataPerturbation($\mathbf{x}^\circ$,$\mathbf{C}$);
6      Set $\mathbf{x}''$ = LocalSearch($\mathbf{x}'$,$\mathbf{C}$);
7      if ($z(\mathbf{x}^*) > z(\mathbf{x}'')$) then
8          Set $\mathbf{x}^* = \mathbf{x}''$;
9          Set $\mathbf{x}^\circ$ = LocalSearchMIP($\mathbf{x}''$);
10         if ($z(\mathbf{x}^*) > z(\mathbf{x}^\circ)$) then set $\mathbf{x}^* = \mathbf{x}^\circ$;
11     end
12 until termination condition;
13 return $\mathbf{x}^*$;
```

optimal. The initial solution of cost 329, obtained after a few thousands iteration of ILS, is

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.11}$$

The *opt1-1* subroutine first identifies a swap of the assignments of clients 3 and 6, producing the solution (3.12), of cost 326 and unused capacities (28,16,0).

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.12}$$

Solution (3.12) is a local optimum for *opt11*, thus no further improvements are found. However, since the initial solution was improved, the control is passed to $LocalSearchMIP$, in this case with $k = 4$, meaning that 4 customers are free to be reassigned, therefore up to 8 decision variables can be varied. It is now possible to further improve the solution, obtaining solution (3.13) of cost 325 and with unused capacities (29,15,0).

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{3.13}$$

The variables that were allowed to change were variables 2, 3, 6, and 7, they were randomly chosen among the eight possible ones.

3.5 Variable Neighborhood Search

Variable Neighborhood Search (VNS) proposes a variation of the basic local search scheme by exploring a sequence of neighborhoods at each main search iteration (this is called a *Variable Neighborhood Descent, VND*), followed by a perturbation of the incumbent solution.

This procedure is motivated by the observation that a local minimum with respect to one neighborhood is not necessarily a local minimum for other neighborhoods. Only a global optimum is guaranteed to be a local optimum for any neighborhood function. Moreover, it is often the case that local optima of different neighborhoods are not far apart.

The core idea of varying the neighborhood used during search has been implemented in many different ways, giving rise to many variants, such as variable neighborhood descent, basic variable neighborhood search, reduced variable neighborhood search, and variable neighborhood decomposition search. Here we present a version where VNS contains at its heart a basic neighborhood sequencing procedure, the VND method. However, it should be said that several authors consider VNS to be a special case of the ILS method,[1] we keep them separate in this chapter following the amount of literature explicitly targeting VNS as such.

Notationally, let $\mathcal{N}_k$, $k = 1, \ldots, k_{max}$, be an ordered sequence of neighborhood sets, where $\mathcal{N}_k(\mathbf{x})$ is the set of feasible solutions, neighbors to solution $\mathbf{x}$ according with the k-th neighborhood function. The general structure of the VND algorithm presented as Algorithm 12, prescribes to start from a seed solution, then to apply to which is it all chosen neighborhood functions in a fixed order. Each time an improvement is found, the corresponding local optimum becomes the new incumbent solution and the process starts again from the beginning.

VNS adds to this procedure a perturbation step and an acceptance step (see Algorithm 13). The perturbation step is applied to the solution provided at the end of VND in order to re-instantiate it on another seed solution. All considerations made on perturbation in Sect. 3.4 are valid here, too, given the tight relationship between ILS and VNS. Clearly, the perturbation step must provide a new seed solution that is not contained in any of the explored neighborhoods of the solution to be perturbed. It is often implemented as the generation of a random solution in one further, larger neighborhood.

The acceptance step is again analogous to the corresponding one presented in Algorithm 10, and permits to gauge search exploration versus exploitation.

The basic scheme of VNS is conveniently simple and could require no parameter, besides the termination condition. It can be easier to analyze its working on the problem of interest than that of other methods, and possibly to design tailored extensions.

[1]For more on this, see Lourenço et al. (2002).

Algorithm 12: Generic Variable Neighborhood Descent

```
1  function VariableNeighborhoodDescent(x);
   Input  : A feasible solution x
   Output: A feasible solution $x^*$
2  Define $\mathcal{N}_k, k = 1, \ldots, k_{max}$;
3  Set $k = 1$;
4  repeat
5      Set $x'$ = FindBestNeighbor(x, $\mathcal{N}_k$);
6      if ($z(x') < z(x)$) then
7          $x = x'; x^* = x$;
8          $k = 1$;
9      else
10         $k = k + 1$;
11     end
12 until $k > k_{max}$;
13 return $x^*$;
```

Algorithm 13: Generic Variable Neighborhood Search

```
1  function VariableNeighborhoodSearch();
   Input  :
   Output: A feasible solution $x^*$
2  Generate an initial feasible solution x;
3  Set $x^* = x$;
4  Define $\mathcal{N}_k, k = 1, \ldots, k_{max}$;
5  repeat
6      Set $x'$ = VariableNeighborhoodDescent(x);    // can be any local search
7      if ($z(x^*) > z(x')$) then set $x^* = x'$;
8      Set x = Perturb($x'$);                                    // perturbation
9      Set x = Accept(x, history);                                 // acceptance
10 until termination condition;
11 return $x^*$;
```

VNS appears to be a particularly promising candidate for hybridization with mathematical programming methods.

For example, an approach was presented, which makes use of a VND procedure that combines three different neighborhoods, applied to a generalized version of the minimum spanning tree problem. The generalization consists in the fact that the nodes of the graph are partitioned into clusters and exactly one node from each cluster must be connected. The first two neighborhoods are large in the sense that they contain exponentially many candidate solutions, but efficient polynomial-time algorithms are used to identify best neighbors: one of these is named *global edge exchange* neighborhood and makes use of a dynamic programming procedure to explore the corresponding neighborhood, the second one is called *global subtree optimization* neighborhood and it is explored via MIP. The third neighborhood is not polynomial and relies on MIP to optimize selected parts within the candidate solution. Comparisons among the different neighborhoods showed that the dynamic programming component is particularly effective with respect to the other ones. Previously, the same authors proposed a different hybridization, in this case with local branching.

Another example is Relaxation Guided Variable Neighborhood Search, which includes in the usual VNS scheme a VND algorithm, where the ordering of the neighborhood structures is determined dynamically by solving their relaxations. The objective values of these relaxations are used as indicators for the potential gains of searching the corresponding neighborhoods.

Similarly, a VNS was proposed, integrating several large neighborhoods for treating a location routing problem, where the last tested neighborhoods were again based on MIP models. Numerical results show the effectiveness of the enhanced VNS, supporting the position that a MIP model is generally a worthy component to add to the VND.

Another interesting application of VLNS has been presented for solving a supplier selection problem, which is solved by a MILP local search method. Interesting results of the proposed method were obtained in dealing with a set of real-world instances when compared against a greedy procedure. The gap of the solutions obtained by the MIP local search method from the optimal solution of these real-world instances averages to 0.09%, while the average gap of the greedy heuristic is larger than 36%.

Other works adopting VNS approaches enriched by MIP to strengthen local search have been applied to very diverse problems, such as a car sequencing problem, solved by means of a VNS with an ILP component, an integrated 2-dimensional loading, and vehicle routing problem, or a video-on-demand balancing problem.

Special attention has been devoted to nurse rostering. A solution included in the VND two MIP-based problem-specific neighborhoods based on variable fixing: one fixing days and the other one fixing shifts in the incumbent solution. Another work dedicated to the same problem defined 12 problem-specific neighborhoods to be used in a variable fixing heuristic. The same approach was later used for another, related, scheduling problem, the high school timetabling problem.

Finally, something quite different. A study tries to automatically identify a neighborhood structure from a MIP model using a combination of automatic extraction of the so-called *semantic features* and automatic algorithm configuration. It starts from a knowledge base of the problem, from which one tries to identify the main components that need to be combined to obtain a feasible solution, as well as the relationships among them (these last are called by the authors *semantic features*). Automatic identification of these features leads to the desired neighborhood structure design, to be included in a VNS heuristic. Applications are reported for the Traveling Salesman Problem, the Generalized Traveling Salesman Problem, the Capacitated Vehicle Routing Problem with Time Windows, and for the Multi-Plant, Multi-Item, Multi-Period Capacitated Lot-Sizing Problem.

3.5.1 VNS for the GAP, An Example

As an example of a matheuristic rooted in VNS, we will present an approach that follows indications proposed by Lazić in her thesis (2010). The method is a specialization of a VNS variant, named Variable Neighborhood Decomposition Search (VNDS), which restricts the search process to only a subspace of the entire search space, as defined by some subset of solution attributes that can be efficiently explored. At each iteration, VNDS chooses randomly a subset of solution attributes with cardinality k, and applies a local search, possibly a VND, to the subproblem where the variables corresponding to the k attributes are fixed to the values taken in the current incumbent solution.

Following this pattern, Lazić proposes a VNDS-MIP, where the variables to be fixed at each iteration are chosen according to their distance from the corresponding linear relaxation solution values. An adapted VNDS pseudocode is presented as Algorithm 14.

Algorithm 14: Generic VNDS-MIP

```
1  function VNDS-MIP($P$, $d$, $t_{max}$);
   Input  : instance to solve, $P$, $d$, max CPU time $t_{max}$
   Output: A feasible solution x
2  Find an optimal solution $\bar{\mathbf{x}}$ of LP($P$); if $\bar{\mathbf{x}}$ is integer feasible then return $\bar{\mathbf{x}}$;
3  Generate an initial feasible solution x of $P$;
4  Set $t_{start} = cpuTime()$; $t = 0$;
5  repeat
6      Compute $\delta_j = |x_j - \bar{x_j}|$ for each $j \in J$;
7      Index the variables $x_j$, $j \in J$, so that $\delta_1 \leq \delta_2 \leq \ldots \leq \delta_p$, $p = |J|$;
8      Set $n_d = |\{j \in J : \delta_j \neq 0\}|$; $k_{step} = \lfloor n_d/d \rfloor$; $k = p - k_{step}$;
9      while $(t < t_{max})$ and $(k > 0)$ do
10         x' = $MIPSolve(P(\mathbf{x}, h\}), \mathbf{x})$, $h = \{1, 2, \ldots, k\}$;
11         if $(z(\mathbf{x}') < z(\mathbf{x})$ then
12             x = $LocalSearch(P, \mathbf{x}')$;
13             break;
14         else
15             if $(k - k_{step} > p - n_d)$ then $k_{step} = max\{\lfloor k/2 \rfloor, 1\}$;
16             Set $k = k - k_{step}$;
17             Set $t = cpuTime() - t_{start}$;
18         end
19     end
20 until $(t < t_{max})$;
21 return x;
```

The working is as follows. Initially, the LP-relaxation of the problem P is solved, obtaining a lower bound $z_{LP}(P)$ to the optimal cost $z^*(P)$ of P. If the linear solution is integer feasible, the algorithm stops and returns an optimal solution for P. Otherwise, an initial feasible solution **x** of P is generated. At each iteration of the VNDS-MIP procedure, the distances $\delta_j = |x_j - \bar{x_j}|$ between the current incumbent solution values and their corresponding LP-relaxation values are computed for each decision variable. The variables x_j, $j \in J$ are then indexed so that $0 \le \delta_1 \le \delta_2 \le \ldots \le \delta_p$ (where $p = |J|$). Then the subproblem $P(\mathbf{x}, h)$, $h = 1, 2, \ldots, k$, obtained from the original problem P, is solved by fixing the first h variables to their values in the current incumbent solution $\mathbf{x}$. If an improvement occurs, a local search procedure is performed over the whole search space and the process is repeated. If not, the number of fixed variables in the current subproblem is decreased.

This main procedure can be specified to obtain a matheuristic for the GAP. One simple possibility is to use a VND as a local search procedure at step 12. For simplicity, we implemented a VND using only two neighborhoods: an *opt01*, just moving a customer from a server to another one which can accommodate it, and an *opt11* switching the assignments of two customers. The overall algorithm is presented as Algorithm 15.

Algorithm 15: Matheuristic VNS for the GAP

```
1  function VNS-GAP($GAP$, $d$, $t_{max}$);
   Input  : Instance to solve, $GAP$, step control parameter $d$, max CPU time $t_{max}$
   Output: A feasible solution $\mathbf{x}^*$
2  Find an optimal solution $\bar{\mathbf{x}}$ of LP($GAP$); if $\bar{\mathbf{x}}$ is integer feasible then return $\bar{\mathbf{x}}$;
3  Generate an initial feasible solution $\mathbf{x}^*$ of $GAP$;
4  Set $t_{start} = cpuTime()$; $t = 0$;
5  repeat
6      Compute $\delta_{i*n+j} = |x_{ij} - \bar{x_{ij}}|$ for each $i \in I$, $j \in J$;
7      Index all variables $x_{ij}$, so that $\delta_1 \le \delta_2 \le \ldots \le \delta_p$, $p = |I| * |J|$ ;
8      Set $n_d = |\{\delta_{i*n+j} \neq 0\}|$; $k_{step} = \lfloor n_d/d \rfloor$; $k = p - k_{step}$;
9      while $(t < t_{max})$ and $(k > 0)$ do
10         $\mathbf{x}' = MIPSolve(P(\mathbf{x}^*, h\}), \mathbf{x}^*)$, $h = \{1, 2, \ldots, k\}$;
11         if $(z(\mathbf{x}') < z(\mathbf{x}^*))$ then
12             $\mathbf{x}^* = VND(\mathbf{x}')$;
13             break;
14         else
15             if $(k - k_{step} > p - n_d)$ then $k_{step} = max\{\lfloor k/2 \rfloor, 1\}$;
16             Set $k = k - k_{step}$;
17             Set $t = cpuTime() - t_{start}$;
18         end
19     end
20 until $(t < t_{max})$;
21 return $\mathbf{x}^*$;
```

The working of Algorithm 15 on the instance $example8x3$ (Fig. 1.1) is as follows.

The lower bound solution of step 2, of cost 231.45, is not integer feasible, as shown in Eq. (3.14).

$$\bar{\mathbf{x}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0.77 & 1 & 1 & 1 \\ 0 & 0.30 & 1 & 1 & 0.23 & 0 & 0 & 0 \\ 1 & 0.70 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.14}$$

The seed feasible solution used at step 3 is the simply constructed one, of cost 343, of Eq. (3.15).

$$\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{3.15}$$

Next, the delta values are computed, which in matrix format are as in Eq. (3.16).

$$\Delta = \begin{bmatrix} 1 & 1 & 1 & 0 & 0.77 & 1 & 1 & 1 \\ 0 & 0.30 & 1 & 0 & 0.77 & 0 & 0 & 0 \\ 1 & 0.70 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{3.16}$$

The ordering of the indices by ascending values generates the following vector $\mathbf{v} = [03, 08, 11, 13, 14, 15, 18, 19, 20, 09, 17, 04, 12, 00, 01, 02, 05, 06, 07, 10, 16, 21, 22, 23]$, where the elements of $\mathbf{v}$ are the row-column indices of the corresponding variable (indices starting from 0).

Suppose we have control parameters that lead to a value of $k = 4$, which means we have to fix the first 4 variables in the ordering dictated by $\mathbf{v}$. This leads to fixing the 3rd, 8th, 11th, and 13th decision variables to the values of the heuristic incumbent solution, i.e., $x_{03} = 0, x_{10} = 0, x_{13} = 1, x_{15} = 0$.[2]

Upon calling a MIP solver with those values fixed, we get the solution reported in Eq. (3.17), of cost 328.

$$\mathbf{x}' = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{3.17}$$

Since solution $\mathbf{x}'$ improves over the incumbent one, a VND is started from it. The first neighborhood, *opt1-0*, cannot find anything better, but the second neighborhood, *opt1-1*, is able to further improve this solution.

The first found improving swap is between the first two clients, and leads to the solution of Eq. (3.18) of cost 327.

$$\mathbf{x} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{3.18}$$

Finally, a swap between customers 1 and 3 is found, which leads to the solution of Eq. (3.19) of cost 325.

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{3.19}$$

[2] We remind that in the computational traces the decision variables are indexed from 0.

3.6 GRASP

GRASP, actually *Greedy Randomized Adaptive Search Procedure*, combines a constructive and a local search heuristic in one single framework. Actually, the local search phase is totally standard, what characterizes GRASP is the constructive phase.

The solution construction is in fact suggested to be *semi-greedy*, trying to combine the benefits of a greedy construction, such as that presented in Sect. 1.3.1, which include a good solution quality but no variance on the produced solution, with the benefits of a completely random solution generation, which include a high variance of the produced solution at the cost of usually bad solution quality.

GRASP implements a semi-greedy construction by means of *Restricted Candidate Lists* (RCL), which are lists produced from the standard ordering of components made by constructive heuristics (see Algorithm 1) and including only the k best ones in the RCL, but other selection criteria have also been proposed. The construction is then carried on by selecting at random at each iteration one element of the RCL. This permits to reinitialize at each iteration the local search from a different seed solution, all the while with the expectation of starting from a good quality solution.

Algorithm 16: Generic GRASP

```
1  function GRASP($k$);
   Input  : Candidate list size $k$
   Output: A feasible solution $\mathbf{x}^*$
2  $\mathbf{x}^* = NULL$;
3  repeat
4      $\mathbf{x} = \emptyset$;                // semi-greedy construction phase
5      while $\mathbf{x}$ not complete do
6          $RCL = MakeRCL(k)$;
7          $e = SelectElement(RCL)$;
8          $\mathbf{x} = \mathbf{x} \cup \{e\}$;
9      end
10     $\mathbf{x}' = local\_search(\mathbf{x})$;          // local search phase
11     if $(z(\mathbf{x}^*) > z(\mathbf{x}'))$ or $\mathbf{x}^* = NULL$ then set $\mathbf{x}^* = \mathbf{x}'$;
12 until terminating condition;
13 return $\mathbf{x}^*$;
```

Several publications described different ways according to which GRASP can be integrated with MIP. For example, an application to a transfer line balancing problem integrates GRASP and a genetic algorithm, which both use a MIP solver as a subroutine for solving subproblems arising in the search process of the metaheuristics. MIP is used

- during construction, at each step, to determine how alternative expansions in the candidate set are to be used to generate a complete solution;
- during improvement, in a problem-specific way relative to local branching.

Another work on a job shop scheduling problem integrates a branch and bound (B&B) method within a GRASP, to solve the scheduling related to one machine. The B&B is used within GRASP to solve one-machine subproblems that arise while solving the more general scheduling problem. Its results are in fact used to build the *Restricted Candidate List* RCL, where an alternative is included if the optimal one-machine scheduling has a makespan sufficiently close to the best one and sufficiently away from the worst one.

Another possibility that proved effective suggested to extend a branch and cut procedure by fixing at each node some variables via LP rounding and then applying GRASP in order to obtain a good feasible solution from different seeds.

3.6.1 GRASP for the GAP, An Example

In this example we follow the approach proposed by Dolgui et al. (2009), using a MIP formulation of the GAP at each constructive expansion step in order to determine the RCL within which to choose the next solution component. Specifically, at each construction step, we use a linear relaxation of the unsolved part of the problem in order to assess each possible extension. A major observation to consider when facing highly constrained instances, and instance *example8x3* used in this text as a running example is already partially such, is that constraints make it very likely to be unable to construct a complete feasible solution. The main concern during construction is therefore to get a complete solution rather than a low cost one. This is especially true for GRASP, where the construction phase is followed by a local search: a poor solution has every chance to be improved, provided that there is a solution to start from.

To increase the chance of achieving complete solutions, we were therefore interested in minimizing resource consumptions rather than constructing low cost partial solutions. The linear bound used to determine the RCL accordingly used the client requests as cost coefficients, all other elements being the same as in formulation GAP of Chap. 1. This results in the following formulation RGAP, where $J' \subseteq J$ is the index set of the unassigned clients and Q'_i the residual capacity at server i after discounting the requests of the clients in $J \setminus J'$.

$$(RGAP) \qquad z'_{RGAP} = \min \sum_{i \in I} \sum_{j \in J'} q_{ij} x_{ij} \tag{3.20}$$

$$s.t. \quad \sum_{i \in I} x_{ij} = 1, \qquad j \in J' \tag{3.21}$$

$$\sum_{j \in J'} q_{ij} x_{ij} \le Q'_i, \qquad i \in I \tag{3.22}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I, j \in J' \tag{3.23}$$

The pseudocode of the matheuristic is as follows, where a solution is supposed to be represented as a set of client-server pairs.

Algorithm 17: Matheuristic GRASP for the GAP

```
1  function GRASP($k$);
   Input  : Candidate list size $k$
   Output: A feasible solution $\mathbf{x}^*$
2  $\mathbf{x}^* = NULL$;
3  repeat
4      $\mathbf{x} = \emptyset$;
5      Order clients in $J$ by decreasing regrets;      // regrets see Sect. 1.3.1
6      foreach $j \in J$ do      // clients ordered by decreasing regrets
7          for $i \in I$ do
8              $z'_{RGAP}(ij) = z(\mathbf{x})$ + value of $z_{RGAP}$ upon tentatively fixing $x_{ij} = 1$;
9          end
10         Order the $z_{RGAP}(ij)$ by increasing values and let $L$ be a list of the first $k$ values
             in the ordering;
11         $RCL = \{(j : z'_{RGAP}(ij) \in L\}$;
12         $(i, e) = SelectElement(RCL)$;
13         $\mathbf{x} = \mathbf{x} \cup \{(i, e)\}$;
14     end
15     $\mathbf{x}' = local\_search(\mathbf{x})$;      // local search phase
16     if $(z(\mathbf{x}^*) > z(\mathbf{x}'))$ or $(\mathbf{x}^* = NULL)$ then set $\mathbf{x}^* = \mathbf{x}'$;
17 until terminating condition;
18 return $\mathbf{x}^*$;
```

A trace of the execution of one iteration of Algorithm 17 on the instance *example8x3* of Sect. 1, with $k = 2$, is the following.

The clients regrets are $regrets = (12, 13, 14, 15, 16, 17, 18, 19)$.

They induce an ordering at step 5 which is (8, 7, 6, 5, 4, 3, 2, 1).

At the first iteration of step 6 we have therefore $j = 8$, and the costs computed at step 8 are $z'_{RGAP}(1, 8) = 216$, $z'_{RGAP}(2, 8) = 206$, and $z'_{RGAP}(3, 8) = 244.33$. The RCL containing the indices of the servers to which the first client in the order can be assigned is, therefore, {2, 1}. Since we randomly extracted 2, we fix $x_{07} = 1$, generating a partial solution of cost $z(\mathbf{x}) = 17$.

At the second iteration of step 6 we have $j = 7$, and the costs computed at step 8 are $z'_{RGAP}(1, 7) = 236$, $z'_{RGAP}(2, 7) = 226$, and $z'_{RGAP}(3, 7) = 216$. The RCL is therefore {3, 2} and we randomly extracted 1, therefore we fix $x_{26} = 1$, generating a partial solution of cost $z(\mathbf{x}) = 101$.

At the third iteration of step 6 we have $j = 6$, and the costs computed at step 8 are $z'_{RGAP}(1, 6) = 236$, $z'_{RGAP}(2, 6) = 226$, and $z'_{RGAP}(3, 6) = 216$. The RCL is therefore {3, 2} and, since we randomly extracted 1, we fix $x_{25} = 1$, generating a partial solution of cost $z(\mathbf{x}) = 181$.

At the fourth iteration of step 6 we have $j = 5$, and the costs computed at step 8 are $z'_{RGAP}(1, 5) = 236$, $z'_{RGAP}(2, 5) = 226$, and $z'_{RGAP}(3, 5) = 216$. The RCL is therefore $\{3, 2\}$ and, since we randomly extracted 1, we fix $x_{24} = 1$, generating a partial solution of cost $z(\mathbf{x}) = 257$.

At the fifth iteration of step 6 we have $j = 4$, and the costs computed at step 8 are $z'_{RGAP}(1, 4) = 236$, $z'_{RGAP}(2, 4) = 226$, and $z'_{RGAP}(3, 4) = 216$. The RCL is therefore $\{3, 2\}$ and, since we randomly extracted 2, we fix $x_{13} = 1$, generating a partial solution of cost $z(\mathbf{x}) = 285$.

At the sixth iteration of step 6 we have $j = 3$, and the costs computed at step 8 are $z'_{RGAP}(1, 3) = 246$, $z'_{RGAP}(2, 3) = 236$, and $z'_{RGAP}(3, 3) = 226$. The RCL is therefore $\{3, 2\}$ and, since we randomly extracted 2, we fix $x_{12} = 1$, generating a partial solution of cost $z(\mathbf{x}) = 311$.

At the seventh iteration of step 6 we have $j = 2$, and the costs computed at step 8 are $z'_{RGAP}(1, 2) = 256$, $z'_{RGAP}(2, 2) = 246$, and $z'_{RGAP}(3, 2) = 236$. The RCL is, therefore, $\{3, 2\}$ and, we randomly extracted 1 but the corresponding assignment makes the partial solution infeasible. We therefore try each assignment in the ordering and we finally fix $x_{01} = 1$, generating a partial solution of cost $z(\mathbf{x}) = 322$.

At the last iteration of step 6 we have $j = 1$, and the costs computed at step 8 are $z'_{RGAP}(1, 1) = 276$, $z'_{RGAP}(2, 1) = 266$, and $z'_{RGAP}(3, 1) = 256$. The RCL is, therefore, $\{3, 2\}$ and, we randomly extracted 2 but the corresponding assignment makes the partial solution infeasible. We therefore try each assignment in the ordering and we finally fix $x_{00} = 1$, generating a solution of cost $z(\mathbf{x}) = 332$.

The solution at the end of the construction phase is the one having the least cost among those produced in all iterations, and it is therefore

$$\mathbf{x} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{3.24}$$

This solution seeds the local search phase, which is based on a *opt11* (see Sect. 1.3.2) exchange algorithm, since Algorithm 4 is often ineffective on tightly constrained instances.

A first improving swap between clients 1 and 3 leads to the solution

$$\mathbf{x} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{3.25}$$

of cost 330. A second improving swap is found between clients 2 and 4, leading to the solution

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{3.26}$$

of cost 328. Finally, a last improving swap is found between clients 4 and 5, leading to the solution

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{3.27}$$

of cost 325, which corresponds to the z_{GAP} that is proposed by the heuristic at the end of its run.

3.7 Ejection Chains

Ejection chains have been introduced as procedures "*based on the notion of generating compound sequences of moves. leading from one solution to another, by linked steps in which changes in selected elements cause other elements to be 'ejected from' their current state. position or value assignment.*" (Glover 1992a)

This definition is clearly little normative and can include most neighborhood exploration methods. In fact, it was proposed in order to generalize other search methods. The main idea behind the approach is to start with a feasible solution and to identify a set of decision variables, corresponding to assignment arcs in the case of the GAP, and let them change their state, i.e., their value. For example, in the case of a GAP solution, this could mean to undo a client-server assignment, which was in the solution, or to force one that was not included. The problem constraints will probably require that other variables undergo a state change in order to accommodate the previous changes and maintain feasibility. In the ejection chains jargon, this deriving effect is what is described as the request that some variables are "ejected from" their current state and forced into another (similarly to what is called *constraint propagation* in constraint programming contexts). Search is then implemented as a sequence of forced changes and corresponding ejection ones.

As noted, most local searches fit into this model, where part of a solution is undone in order to have the partial solution amended in a possibly improved way. For example, in the TSP, a *2-opt* local search asks to remove two arcs from the solution and reconnect the exposed end-nodes in a different way. As another example, in the case of the GAP, a *opt01* move requires to undo a client-server assignment. This induces the need to reassign the client to some other server, thereby "ejecting" the second assignment variable from its state, currently the value of 0, into a new state, the value of 1.

Such an encompassing denotation would be of little interest for research. Therefore it is customary to use it for cases when the change-correction pairs are not structured inside a well defined search pattern, such as *2-opt*, Lin-Kernighan, or the like, but when the initiating state changes are freely defined. This case is often also denoted as *relax-and-fix*. Ejection chains are often presented as a special case of very large-scale neighborhood search procedures, see Chap. 6, specifically with the denotation of *Large Neighborhood Search* , and generalized as *Adaptive Large Neighborhood Search*. It is not easy to present a general pseudocode for ejection chains, as most contributions so far have been very problem-specific. A possible general structure is presented in Algorithm 18, where $z(\mathbf{x})$ denotes the cost of solution $\mathbf{x}$.

Algorithm 18: Generic Ejection chain

```
1 function EjectionChain;
  Input  : A feasible solution $\mathbf{x}^h$
  Output: A feasible solution $\mathbf{x}^*$
2 Set $\mathbf{x}^* = \mathbf{x}^h$;
3 repeat
4     Select a subset $S$ of decision variables whose values has to be changed;
5     Set $\mathbf{x}' = \mathbf{x}^h$ except for the changed variables in $S$; // $\mathbf{x}'$ could be infeasible
        Set $\mathbf{x}$ = call fixSolution($\mathbf{x}$');
6     Set $\mathbf{x}^h$ = call acceptSolution($\mathbf{x}'$,$\mathbf{x}^h$);
7     if ($z(\mathbf{x}^h) < z(\mathbf{x}^*)$) then set $\mathbf{x}^* = \mathbf{x}^h$;
8 until terminating condition;
9 return $\mathbf{x}^*$;
```

Algorithm 18 is underspecified, due to the abstract nature of the motivating idea. Besides the terminating conditions, most steps inside the main loop have to be detailed.

Step 4 requires to identify the set S of variables, whose state will be changed. This can be done in several instance-specific ways, for example, removing the most costly values, or the most resource consuming ones or following dual-based considerations, such as regrets.

Step 5 is the most challenging and will be covered in the next subsection.

Step 6 asks to decide whether to continue from the newly found solution, in case one was found, or from the one available at the beginning of the current iteration. Typically, when only an infeasible solution is found, it is discarded, while when a feasible one is obtained, different considerations can be made.

3.7.1 Solution Fixing

The possible necessity of fixing an infeasible solution, particularly common in ejection chains when applied to tightly constrained instances, has fostered specific research, also with reference to the GAP. Some simple approaches have already been introduced in Sect. 1.4. Here we discuss some more mathematically oriented ones.

The general underlying idea is to adapt a construction algorithm so to complete a feasible, partial solution. The partial solution is, in turn, obtained by removing from the infeasible solution some parts that induce the infeasibility.

Given the GAP constraints, there are three possible causes for infeasibility:

1. *unassigned client*: a client is assigned to no server.
2. *overassigned client*: a client is assigned to more than one server.
3. *overloaded server*: the sum of the requests assigned to a server exceeds its capacity.

After checking the cause of the infeasibility, it is possible to proceed in an ejection chain fashion, removing parts of the solution until the infeasibility is eliminated and then trying to reconstruct what is missing.

Something similar was proposed by defining a feasibility measure $f(\mathbf{x}) = \sum_j |1 - \sum_i x_{ij}|$ and considering only the assignment of a solution $\mathbf{x}$. First, every job is inspected for a potential bound improvement and failing that, jobs are examined for their potential to decrease $f(\mathbf{x})$ by means of an adjustment of $\mathbf{x}$. This adjustment can lead to finding a feasible solution, or to a fixing failure.

In general, the removal phase induces a hard problem in itself: which is the least solution part to be removed, so that a feasible solution can be reconstructed? Since this problem has to be solved many times in the context of an ejection chain, a heuristic solution is in order. Among the many possibilities, Algorithm 19 proposes a quick solution, though not a particularly effective one.

The algorithm removes causes of infeasibility, preferring to remove first the assignments that are considered more undesirable according to a preference function, such as one of those already used in the construction algorithms of Sect. 1.3.1. Possible preference functions $f(i, j)$ for the assignments (i, j), $i \in I$ and $j \in J$, could, for example, be the regrets, already used in algorithms 2 and 5, which assign a higher regret (f value) to the assignments that, if removed, would result in an higher increase of assignment cost, or simpler functions, such as $f(i, j) = -c_{ij}$, $f(i, j) = -c_{ij}/q_{ij}$, $f(i, j) = -q_{ij}$, $f(i, j) = -q_{ij}/Q_i$ (in maximization form).

Algorithm 19 actually reduces the original GAP instance to a subproblem, still a GAP, where all variables in $\bar{\mathbf{x}}$ keep their values and all unassigned clients must be allocated to servers having enough residual capacity. Formally, the client set J gets partitioned by Algorithm 19 into two subsets: subset J_0, consisting of the clients that keep the assignment prescribed by the original infeasible solution, and set J', consisting of unassigned clients.

For each server i, let $\bar{Q}_i = Q_i - \sum_{j \in J_0} q_{ij}$ be its residual capacity. The subproblem to solve is as follows.

$$(RGAP) \qquad z_{RGAP} = \min \sum_{i \in I} \sum_{j \in J'} c_{ij} x_{ij} \tag{3.28}$$

$$s.t. \quad \sum_{i \in I} x_{ij} = 1, \qquad j \in J' \tag{3.29}$$

Algorithm 19: Recovering feasibility in a partial solution

```
1  function RecoverPartialFeasibility;
   Input  : An infeasible solution $\mathbf{x}'$
   Output: A feasible partial solution $\bar{\mathbf{x}}$ or $Fail$
2  Let $\bar{\mathbf{x}} = \mathbf{x}'$;
3  foreach $i \in I$ do Compute the residual capacity $\bar{Q}_i$ of server $i$ ;
4  while $\bar{\mathbf{x}}$ is infeasible do
5      if there is an unassigned client $j$ then
6          Let $\hat{I} = \{i \in I : \bar{Q}_i \geq q_{ij}\}$;          // Enough residual capacity
7          if ($\hat{I} \neq \emptyset$) then
8              Let $\hat{i} = \{i \in \hat{I}\}$ such that $f(\hat{i}, j) = max(f(i, j))$;
9              Reassign $j$ to $\hat{i}$ in $\bar{\mathbf{x}}$;
10         end
11     else if there is a multiply assigned client $j$ then
12         Let $\hat{I} = \{i \in I : x_{ij} = 1\}$;          // Servers j is assigned to
13         Let $\hat{i} = \{i \in \hat{I}\}$ such that $f(\hat{i}, j) = max(f(i, j))$;
14         Remove all assignments of $j$ to any $i \neq \hat{i}$ in $\bar{\mathbf{x}}$;
15     else if there is an overloaded server $i$ then
16         Let $\hat{J} = \{j \in J : x_{ij} = 1\}$;          // Clients assigned to i
17         Let $\hat{j} = \{j \in \hat{J}\}$ such that $f(i, \hat{j}) = max(f(i, j))$;
18         Unassign $\hat{j}$ from $i$ in $\bar{\mathbf{x}}$;
19     end
20     Update the residual capacities in $\bar{\mathbf{x}}$;
21 end
22 return $\bar{\mathbf{x}}$;
```

$$\sum_{j \in J'} q_{ij} x_{ij} \leq \bar{Q}_i, \qquad i \in I \tag{3.30}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I, j \in J' \tag{3.31}$$

We have that the problem of recovering feasibility from a partial solution is framed as a GAP itself. Since it is known that searching for a GAP feasible solution is a NP-hard problem, being NP-complete the decision problem of determining whether a GAP instance has a feasible solution or not, only fast heuristic approaches are viable as solution fixing procedures.

Besides the simple constructive or local search strategies introduced in Sect. 1, it is possible to try to obtain a feasible solution by means of a rounding strategy that proposes to start with an LP solution of the RGAP formulation and proceed rounding variables whose LP values are closest to the next integer while keeping constraint feasibility. However, reaching feasibility by this approach alone is very elusive, and improvements of this basic scheme are needed, in order to obtain some computational effectiveness.

A first approach to enhance rounding for the GAP can be found in a seminal paper on randomized rounding. Unfortunately, the approach is not well fit for general feasibility recovery. Better results can be obtained by means of algorithms proposed in Chap. 5 of this book. Another idea is to delete all variables that could not be part of an integer solution after the first rounding stage and to add a diversification component, as the one based on the Limited Discrepancy Search (LDS) paradigm.

A combinatorial fixing approach was proposed in the framework of a Lagrangian approach (see also Sect. 7.2), where a *relax-and-fix* heuristic is requested to handle all causes of infeasibility for the GAP. Algorithm 20 accordingly deals with a general case, where both assignment and capacity constraints could be unsatisfied in the current solution.

Let $\mathbf{x}' = [x'_{ij}]$, $i \in I$, $j \in J$, be the current (infeasible) solution. Algorithm 20 starts by assigning each unassigned customer i to the feasible server j which maximizes the relative desirability function, $f(i, j)$. For example, it could be assigned to the server to which it makes the least requests. Formally, for each $j \in J$ such that $\sum_{i \in I} x'_{ij} = 0$ we compute $i_{\min}$ so that $q_{i_{\min} j} = \min_i q_{ij}$. We proceed by constructing the improved solution $\bar{\mathbf{x}}$, initialized as $\mathbf{x}'$, and setting $\bar{x}_{i_{\min} j} = 1$. If no feasible server exists, an indication of failure is returned, otherwise $\bar{\mathbf{x}}$ contains a solution which is feasible for the assignments but may be infeasible for the capacities.

Let $\bar{I}$ be the set of servers for which the assignments in $\bar{\mathbf{x}}$ are feasible. All current assignments to these servers get fixed and will not be reconsidered in the following steps. Furthermore, we also fix all assignments $\bar{x}_{ij}$ that are made to servers $i \in I \setminus \bar{I}$ such that $i = i_{\min}$ with respect to client j, $j \in J$. Let J_0 be the set of all customers whose assignment has been fixed so far.

We can now compute all residual capacities: $\bar{Q}_i = Q_i - \sum_{j \in J_0} q_{ij} \bar{x}_{ij}$, $i \in I$. If any of these capacities is less than 0, the fixing routine fails, otherwise, we try to allocate all unassigned clients to feasible servers. Formally, this can be represented by means of the following formulation F.

$$(F) \qquad z_F = \max \sum_{i \in I} \sum_{j \in J'} \xi_{ij} \tag{3.32}$$

$$\text{s.t.} \sum_{i \in I} \xi_{ij} = 1, \qquad j \in J' \tag{3.33}$$

$$\sum_{j \in J'} q_{ij} \xi_{ij} \le \bar{Q}_i, \qquad i \in I \tag{3.34}$$

$$\xi_{ij} \in \{0, 1\}, \qquad i \in I, j \in J' \tag{3.35}$$

Notice that formulation F maximizes the feasible assignments, compatibly with the constraints on the residual capacities, with no concern about assignment costs. It is however still a GAP, which could be solved by Lagrangian decomposition. In order to simplify the solution procedure and make it more time efficient, the Lagrangian penalty update can be implemented by a simple ascent.

More in detail, upon relaxing constraints (3.33) the objective function becomes

$$z'_F = \sum_{i \in I} \sum_{j \in J'} (1 + \lambda_j)\xi_{ij} - \sum_{j \in J'} \lambda_j$$

where the Lagrangian penalties λ_j are associated with constraints (3.33).

It is now straightforward to decompose problem F getting $z_F = \sum_{i \in I} z_i$, where, for each $i \in I$, we solve the knapsack

$$z_i = \max \sum_{j \in J'} (1 + \lambda_j)\xi_{ij} \tag{3.36}$$

$$\text{s.t.} \sum_{j \in J'} q_{ij}\xi_{ij} \leq \bar{Q}_i \tag{3.37}$$

$$\xi_{ij} \in \{0, 1\}, \qquad j \in J' \tag{3.38}$$

The final, fixed solution, is obtained by concatenating the knapsack solutions. It could be infeasible because of the assignments: in case of unassigned clients, a failure is returned. It could be possible to improve the assignment by updating the penalty vector in input, but this is a more advanced topic which will be discussed in Sect. 7.1.

3.7.2 Ejection Chain for the GAP, An Example

We present here an example of a full ejection chain iteration, according to the pseudocodes provided in the previous sections.

Let us assume that we start from the feasible solution output by the simple constructive algorithm of Sect. 1.3.1, when the aversion function is given by the amount of resource request. The solution $\mathbf{x}_0$, of cost 343, is

$$\mathbf{x}_0 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{3.39}$$

A possible ejection move could try to reallocate some clients from the costly third server to a cheaper one. Simply deallocating one client and trying to reallocate it elsewhere would not work, as all other servers do not have enough residual capacity to accommodate any further request. One possibility that could be explored during the ejection phase could therefore be the re-allocation of the least demanding clients,

Algorithm 20: Combinatorial fixing heuristic

```
1  function fixSolution ($\mathbf{x}'$, $\boldsymbol{\lambda}$);
   Input  : An infeasible GAP solution $\mathbf{x}'$ and a penalty vector $\boldsymbol{\lambda}$
   Output: A feasible solution $\mathbf{x}^h$ or indication of failure
2  $\bar{\mathbf{x}} = \mathbf{x}'$; $\mathbf{x}^h = [0]$;
3  foreach unassigned $j \in J$ do          // compute least cost assignments
4      $i_{\min_j} = i$ such that $q_{i_{\min_j}} = \min_i q_{ij}$;
5      $\bar{x}_{i_{\min_j} j} = 1$;
6  end
7  Initialize $\bar{I}$;
8  foreach $i \in I$ do          // fix feasible and least cost assignments
9      if $i \in \bar{I}$ then
10         $x^h_{ij} = \bar{x}_{ij}, j = 1, \ldots, n$;
11     else
12         if $\left(\exists j : \bar{x}_{i_{\min_j} j} = 1\right)$ then $x^h_{ij} = \bar{x}_{ij}, j = 1, \ldots, n$;
13     end
14 end
15 Remove possible higher cost duplicated client assignments and initialize $\bar{J}$;
16 Order $I$ by increasing residual capacities $\bar{Q}_i = Q_i - \sum_{j \in \bar{J}} q_{ij} x^h_{ij}$;
17 foreach $i \in I$ do          // concatenate knapsack solutions
18     if $\bar{Q}_i < 0$ then return Fail;
19     Solve knapsack on $\bar{Q}_i$ with costs based on $\boldsymbol{\lambda}$ and get $z_i$;
20     Update $\mathbf{x}^h$ and $\bar{J}$ according to the knapsack solution;
21 end
22 if $\bar{\mathbf{x}}$ is feasible then
23     return $\mathbf{x}^h$;
24 else
25     return Fail;
26 end
```

currently assigned to the more expensive servers. This would lead to solution $\mathbf{x}_i$, which is infeasible because it requests 229 to server 1, which has a capacity of 160.

$$\mathbf{x}_i = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.40}$$

At this point Algorithm 19 intervenes, specifically at step 15 where overloaded servers are dealt with.

Partial feasibility is recovered by removing, in order of decreasing aversion, from the set of clients allocated to server 1 all those that are needed to get rid of the overload. This leads to remove first client 1, but this is not enough, then client 2. The incumbent feasible partial solution becomes $\mathbf{x}_p$.

$$\mathbf{x}_p = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.41}$$

There is no guarantee that solution $\mathbf{x}_p$ can be easily recovered, being the GAP NP-hard in the strong sense, but Algorithm 20 proposes a simple heuristic to try to.

All the assignments in $\mathbf{x}_p$ get fixed in the cycle at step 8, as by now all servers are feasible. The residual capacities are [26, 56, 27], therefore the ordering at step 16 yields the result [1, 3, 2].

First, a knapsack is tried for server 1. Assume for simplicity to have $\boldsymbol{\lambda} = [1, 1, 1]$. This knapsack clearly cannot lead to any assignment, given that we just removed the least number of clients to make it feasible.

Then a knapsack is tried for server 3. Costs in function (3.36) are [1, 1], requests in constraint (3.37) are [28, 27], and capacity is 27. As only one client fits, client 2, it enters the solution.

Then a knapsack is tried for server 2. Costs in function (3.36) are [1], requests in constraint (3.37) are [38], and capacity is 56. Therefore client 1 is feasibly assigned to server 2, obtaining the feasible solution $\mathbf{x}^h$ of cost 328.

$$\mathbf{x}^h = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.42}$$

As a last remark, we point out again that what we described here could be denoted as the effect of propagating the capacity constraints over the tentative assignments on the initial solution. But this is just another name for the same course of action.

3.8 Related Literature

Simulated Annealing was introduced by Kirkpatrick et al. (1983) based on the Monte Carlo model of Metropolis et al. (1953). The simulated annealing optimality guarantee is in Mitra et al. (1986), the airline crew scheduling application in Andersen and Funch (2008). Timetable applications are in Avella et al. (2007) and Gunawan et al. (2012), applications on the Multicommodity Fixed-Charge Network Design Problem in Yaghini et al. (2011). The multiplier adjustment, which inspired the SA example, is in Gabrielli et al. (2006).

Tabu search was originally proposed by Glover (1989, 1990), who also reviewed it in Glover and Laguna (1997). The application on the *m-peripatetic* vehicle routing problem is in Ngueveu et al. (2009), those on the Multicommodity Fixed-Charge

Network Design Problem in Chouman and Crainic (2010) and in Gendron et al. (2016), on undirected capacitated arc routing in Archetti et al. (2010), while on the capacitated p-median problem in Yaghini et al. (2013).

Iterated Local Search was presented in Lourenço et al. (2002) and generalized into Stochastic Local Search in Hoos and Stützle (2004). Hybrids between exact methods and ILS are surveyed in Lourenço et al. (2010). Data perturbation is described in Codenotti et al. (1996). The machine reassignment problems application is described in Lopes et al. (2015), and the cutting stock problem application in Umetani et al. (2003).

Variable Neighborhood Search was introduced in Mladenovic and Hansen (1997). Hu et al. (2008) present the application to the generalized minimum spanning tree and the hybrid with local branching in Hu and Raidl (2006). Relaxation Guided Variable Neighborhood Search is in Puchinger and Raidl (2008), the location—routing application in Pirkwieser and Raidl (2010), the supplier selection in Manerba and Mansini (2014). Other applications of VNS including a MIP-based local search are in Prandtstetter and Raidl (2008), Strodl et al. (2010) and Walla et al. (2009). Nurse rostering was studied in Santos et al. (2012) and Della Croce and Salassa (2014), and the high school timetabling application in Fonseca et al. (2016). The semantic feature idea is in Adamo et al. (2017). VNDS is described in Hansen et al. (2001).

GRASP is introduced in Feo and Resende (1989, 1995). The transfer line balancing application is in Dolgui et al. (2009), the job shop scheduling application in Fernandes and Lourenço (2008). The extended branch and cut approach is in Tomazic and Ljubic (2008).

Ejection chains were introduced by Glover (1992a, 1996), who applied them in several publications (Glover 1992b, Laguna et al. 1995, Hubscher and Glover 1992). Relax-and-fix is referenced in Belvaux and Wolsey (2000) Toledo et al. (2015) Roshani et al. (2017) Ferreira et al. (2010), and Adaptive Large Neighborhood Search in Ropke and Pisinger (2006).

The feasibility measure for solution fixing was proposed in Wilcox (1989). The randomized rounding idea is in Shmoys and Tardos (1993), variable deletion in Sadykov et al. (2015), limited discrepancy search in Harvey and Ginsberg (1995), and the described combinatorial fixing in Maniezzo et al. (2019).

References

Adamo T, Ghiani G, Guerriero E, Manni E (2017) Automatic instantiation of a variable neighborhood descent from a mixed integer programming model. Oper Res Perspect 4:123–135

Andersen AC, Funch C (2008) Rostering optimization for business jet airlines. Master thesis, Department of Transport, Technical University of Denmark Kongens Lyngby

Archetti C, Feillet D, Hertz A, Speranza MG (2010) The undirected capacitated arc routing problem with profits. Comput Oper Res 37(11):1860–1869

Avella P, D'Auria B, Salerno S, Vasil'ev I (2007) A computational study of local search algorithms for Italian high-school timetabling. J Heuristics, 543–556

Belvaux G, Wolsey LA (2000) bc-prod: a specialized branch-and-cut system for lot-sizing problems. Management Science 46(5):724–738

Chouman M, Crainic T (2010) MIP-tabu search hybrid framework for multicommodity capacitated fixed-charge network design. Technical report CIRRELT-2010-31

Codenotti B, Manzini G, Margara L, Resta G (1996) Perturbation: An efficient technique for the solution of very large instances of the Euclidean TSP. INFORMS J Comput 8(2):125–133

Della Croce F, Salassa F (2014) A variable neighborhood search based matheuristic for nurse rostering problems. Ann OR 218:185–199

Dolgui A, Eremeev A, Guschinskaya O (2009) MIP-based grasp and genetic algorithm for balancing transfer lines. In: Maniezzo V, Stützle T, Voß S (eds) Matheuristics. Annals of Information Systems, vol 10. Springer, Boston, MA

Feo TA, Resende MGC (1989) A probabilistic heuristic for a computationally difficult set covering problem. Oper Res Lett 8:67–71

Feo TA, Resende MGC (1995) Greedy randomized adaptive search procedures. J Global Optim 6:109–133

Fernandes S, Lourenço HRA (2008) Simple optimised search heuristic for the job shop scheduling problem. In: Cotta C, van Hemert J (eds) Recent advances in evolutionary computation for combinatorial optimization. Studies in computational intelligence, vol 153. Springer, Berlin, Heidelberg, pp 203–218

Ferreira D, Morabito R, Rangel S (2010) Relax and fix heuristics to solve one-stage one-machine lot-scheduling models for small-scale soft drink plants. Comput Oper Res 37(4):684–691

Fonseca GH, Santos HG, Carrano EG (2016) Integrating matheuristics and metaheuristics for timetabling. Comput Oper Res 74:108–117

Gabrielli R, Guidazzi A, Boschetti MA, Roffilli M, Maniezzo V (2006) Practical origin-destination traffic flow estimation. In: ODYSSEUS 2006, Third international workshop on freight transportation and logistics, Altea (Spain), pp 23–26

Gendron B, Hanafi S, Todosijević R (2016) An efficient matheuristic for the multicommodity fixed-charge network design problem. IFAC-PapersOnLine 49(12):117–120

Glover F (1989) Tabu search – part I. ORSA J Comput 1(3):190–206

Glover F (1990) Tabu search – part II. ORSA J Comput 2(1):14–32

Glover F (1992a) Ejection chains and combinatorial leverage for traveling salesman problems. Tech. rep., School of Business, University of Colorado, Boulder, CO

Glover F (1992b) New ejection chain and alternating path methods for traveling salesman problems. In: Balci, Sharda, Zenios (eds) Computer science in operations research: New developments in their interfaces, Pergamon, pp 491–507

Glover F (1996) Ejection chains, reference structures and alternating path methods for traveling salesman problems. Discrete Appl Math 65(1–3):223–253

Glover F, Laguna M (1997) Tabu search. Kluwer Academic Publishers, Boston

Gunawan A, Ming Ng K, Leng Poh K (2012) A hybridized Lagrangian relaxation and simulated annealing method for the course timetabling problem. Comput Oper Res 39(12):3074–3088

Hansen P, Mladenovic N, Perez-Britos D (2001) Variable neighborhood decomposition search. J Heuristics 7(4):335–350

Harvey WD, Ginsberg ML (1995) Limited discrepancy search. In: Proc. of IJCAI-95, Morgan Kaufmann, Montreal, Quebec, p 607–613

Hoos H, Stützle T (2004) Stochastic local search-foundations and applications. Morgan Kaufmann, San Francisco, CA, USA

Hu B, Raidl GR (2006) Variable neighborhood descent with self-adaptive neighborhood ordering. In: Proceedings of the 7th EU/ME meeting on adaptive, self-adaptive and multi-level metaheuristics

Hu B, Leitner M, Raidl GR (2008) Combining variable neighborhood search with integer linear programming for the generalized minimum spanning tree problem. J Heuristics 14(5):473–499

Hubscher R, Glover F (1992) Ejection chain methods and tabu search for clustering. Tech. rep., School of Business, University of Colorado, Boulder, CO

Kirkpatrick S, Gelatt C, Vecchi M (1983) Optimization by simulated annealing. Science 220:671–680

Laguna M, Kelly J, Gonzales-Velarde L, Glover F (1995) Tabu search for the multilevel generalized assignment problem. Eur J Oper Res 82:176–189

Lazić J (2010) New variants of variable neighbourhood search for 0-1 mixed integer programming and clustering. Tech. rep., School of Information Systems, Computing and Mathematics. Brunel University London

Lopes R, Morais VW, Noronha TF, Souza V (2015) Heuristics and matheuristics for a real-life machine reassignment problem. Int Trans Oper Res 22:77–95

Lourenço HR, Martin O, Stützle T (2002) Iterated local search. In: Glover F, Kochenberger G (eds) Handbook of metaheuristics. International series in operations research & management science. Kluwer Academic Publishers, pp 321–353

Lourenço HR, Martin O, Stützle T (2010) Iterated local search: Framework and applications. Handbook of metaheuristics, 2nd edn. International series in operations research & management science, vol 146. Kluwer Academic Publishers, pp 363–397

Manerba D, Mansini R (2014) An effective matheuristic for the capacitated total quantity discount problem. Comput Oper Res 41:1–11

Maniezzo V, Boschetti MA, Carbonaro A, Marzolla M, Strappaveccia F (2019) Client-side computational optimization. ACM Trans Math Softw 45(2):1–16

Metropolis N, Rosenbluth AW, Rosenbluth MN, Teller AH, Teller E (1953) Equation of state calculations by fast computing machines. J Chem Phys 21(6):1087–1092

Mitra D, Romeo F, Vincentelli AS (1986) Convergence and finite-time behavior of simulated annealing. Adv Appl Probab, 747–771

Mladenovic N, Hansen P (1997) Variable neighborhood search. Comput Oper Res 24(11):1097–1100

Ngueveu SU, Prins C, Wolfler R (2009) A hybrid tabu search for the m-peripatetic vehicle routing problem. In: Maniezzo V, Stützle T, Voß S (eds) Matheuristics. Annals of information systems, vol 10. Springer, Boston, MA

Pirkwieser S, Raidl GR (2010) Variable neighborhood search coupled with ILP-based very large neighborhood searches for the (periodic) location-routing problem. In: Blesa M, Blum C, Raidl G, Roli A, Sampels M (eds) Hybrid metaheuristics, HM 2010. Lecture notes in computer science, vol 6373. Springer, pp 174–189

Prandtstetter M, Raidl GR (2008) An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. Eur J Oper Res 191(3):1004–1022

Puchinger J, Raidl GR (2008) Bringing order into the neighborhoods: Relaxation guided variable neighborhood search. J Heuristics 14(5):457–472

Ropke S, Pisinger D (2006) An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation Science 40(4):455–472

Roshani A, Giglio D, Paolucci M (2017) A relax-and-fix heuristic approach for the capacitated dynamic lot sizing problem in integrated manufacturing/remanufacturing systems. IFAC-PapersOnLine 50(1):9008–9013

Sadykov R, Vanderbeck F, Pessoa A, Uchoa E (2015) Column Generation Based Heuristic for the Generalized Assignment Problem. XLVII Simpósio Brasileiro de Pesquisa Operacional, Porto de Galinhas, Brazil

Sahni S (1975) Approximates for the 0/1 knapsack problem. J Assoc Comput Mach 22(1):115–124

Santos HG, Toffolo TAM, Ribas S, Gomes RAM (2012) Integer programming techniques for the nurse rostering problem. In: Proceedings of PATAT 2012. Lecture notes in computer science. Springer, pp 256–283

Shmoys DB, Tardos E (1993) An approximation algorithm for the generalized assignment problem. Mathematical Programming 62(1-3):461–474

Strodl J, Doerner KF, Tricoire F, Hartl RF (2010) On index structures in hybrid metaheuristics for routing problems with hard feasibility checks: an application to the 2-dimensional loading vehicle routing problem. In: Blesa M, Blum C, Raidl G, Roli A, Sampels M (eds) Hybrid metaheuristics, HM 2010. Lecture notes in computer science, vol 6373. Springer, pp 160–173

Toledo CFM, da Silva A, M H, M Y B PMF, Akartunali K (2015) A relax-and-fix with fix-and-optimize heuristic applied to multi-level lot-sizing problems. J Heuristics 21(5):687–717

Tomazic A, Ljubic I (2008) A grasp algorithm for connected facility location. In: International symposium on applications and the internet. IEEE Computer Society Press, pp 257–260

Umetani S, Yagiura M, Ibaraki T (2003) One-dimensional cutting stock problem to minimize the number of different patterns. Eur J Oper Res 146(2):388–402

Walla J, Ruthmair M, Raidl GR (2009) Solving a video-server load rebalancing problem by mixed integer programming and hybrid variable neighborhood search. In: Blesa M, Blum C, Raidl G, Roli A, Sampels M (eds) Hybrid metaheuristics, HM 2010. Lecture notes in computer science, vol 6373. Springer, pp 84–99

Wilcox SP (1989) A new multiplier adjustment method for the generalized assignment problem, Working paper, General Research Corporation, Mc Lean, VI

Yaghini M, Karimi M, Rahbar M, Akhavan RA (2011) Hybrid simulated annealing and simplex method for fixed-cost capacitated multicommodity network design. Int J Appl Metaheuristic Comput 2(4):13–28

Yaghini M, Karimi M, Rahbar MA (2013) Hybrid metaheuristic approach for the capacitated p-median problem. Appl Soft Comput 13(9):3922–3930

Chapter 4
Population-Based Metaheuristics

4.1 Introduction

Metaheuristics can be roughly divided into algorithms that maintain one single incumbent solution, and algorithms that maintain a set of incumbents solutions. The first group, already discussed in Chap. 3, is usually closely related to local search and the description of its elements is usually made by means of a standard optimization terminology. The second group is more varied. Probably, the best known element of this group are Genetic Algorithms (GAs), which however were not the first algorithms of the group, not even the first algorithms using genetics for optimization. In fact, already A. Turing in 1950 envisaged the possibility to use evolution as a model for computational learning, if only to express the hope to design processes "*more expeditious than evolution*" (Turing 1950). Notwithstanding, they were the first to get significant renown, except maybe in German-speaking countries, where Evolution Strategies were already well-known.

GAs originated by the Artificial Intelligence community, rather than by the optimization one. This set the stage for the peculiar way they are usually introduced. In fact, rather than making reference to previous optimization literature and concepts, GAs—and most population based metaheuristics thereafter—have been proposed making reference to some specific natural process, whose elements are partly mimicked by the algorithm. This unusual way of proposing optimization concepts, albeit already pioneered by Simulated Annealing (see Sect. 3.2), helped much in getting attention and in widening the audience of these algorithms in the early years. It was customary and widely accepted to have new algorithms introduced by means of a metaphor, and to metaphorically use nature-related terms to refer to optimization concepts. We witnessed the proposal, besides Genetic Algorithms and Evolution Strategies, Ant Colony Optimization, and Particle Swarm Optimization before being submerged by the deluge of natural metaphors.

Currently, over 190 inspiring natural metaphors are listed in the Evolutionary Computation Bestiary website, which is maintained by Campelo (2019), with subjects ranging from “American Buffalo” to “Zombies.” Clearly, there is much wrong in this, as it has already been forcefully pointed out. Moreover, a large part of the algorithms that maintain a set of incumbent solutions, also known as population-based heuristics, derive from these natural inspirations, therefore the whole set is often confused with its large subset.

This chapter describes how three of the most influential population based metaheuristics, namely Evolutionary Computation with their two most widespread representatives, the Genetic Algorithms and the Evolution Strategies, Ant Colony Optimization (ACO) and Scatter Search, have been combined with mathematical programming components to obtain full-fledged matheuristics. We remark furthermore that several other population based heuristics have been enriched by mathematical programming components, but given the sparsity of their applications, we did not include them in this review.

It would be beneficial to refrain from using the metaphor terminology and to present optimization ideas with optimization terminology only. However, since this would mean rephrasing much of the relevant literature and since this book has only review purposes, we kept the metaphor terminology in the case of ECs, partly departed from it in the case of ACO and stuck to optimization terminology only in the case of Scatter Search.

4.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) or Evolutionary Computation (ECs), as the name easily suggests, get their inspiring metaphor from evolution. There are quite a number of EC methods available such as Genetic Algorithms, Evolution Strategies, Evolutionary Programming, and Genetic Programming. Here, we quickly say what these EC methods are and then we highlight two main ones, namely Genetic Algorithms and Evolution Strategies. In general, we refer to ECs as to methods where selective pressure drives the species living in a given environment to structures that ensure a better probability of surviving and of reproducing.

Darwin postulated that occasional random mutations of the genotype are ultimately responsible for driving the evolution of species, by conferring a survival advantage. The mechanism works in limited resource environments, where more individuals are generated than those that can survive and reproduce. Thus, there is a competition for survival. The random genotype mutations generate variants among individuals of the same species, and individuals with advantageous features are more likely to survive and reproduce, passing their traits to their offspring. Future generations will therefore include the new traits as a common component of the species genotype, ensuring better fitness in the environment they live in.

Given this theory, the idea came to use the same mechanisms in optimization. If nature bases such a fundamental course on simple processes like random mutations and reproduction, resulting in genotypes optimized for the environment they live in, why not use the same process to optimize tentative solutions with respect to the objective function we are interested in?

After abstracting the procedure as much as possible, the essential elements left were

- *selective pressure*, acting on populations of individuals bigger than the environment can support;
- *random mutations* of the individuals' genotypes;
- *sexual reproduction*, which allows for the recombination of parents' genotype into their offspring.

These three elements are taken to be the essential components of EC methods. These tenets are to be transposed into an optimization algorithm. Unfortunately, as mentioned, it became customary to use biological terms to denote optimization concepts and not the other way round. Table 4.1 presents a rough mapping of the main terms.

The metaphor suggests therefore that, just as the selective pressure ensures that a species becomes ever more fit to its environment thanks to mutations and recombinations, likewise a set of solutions can get better and better objective function values thanks to single local moves and solution recombinations. One great contribution of EC methods has been to focus attention on the updating of a whole set of solutions at each iteration, rather than of a single one.

Evolutionary Computation has just evolved to some high-level standard for a class of different types of algorithms (Fig. 4.1). We first will lead to Genetic Algorithms, which were originally built on bitstring representation, but moved to more general representation when they became more powerful. However, they were not the first one of EC methods, as, for example, the Evolution Strategies were proposed earlier.

Table 4.1 ECs nomenclature translation

Biology	Optimization
Population	Set of tentative solutions
Individual	Tentative solution
Genotype	Tentative solution
Chromosome	Tentative solution
Gene	Solution variable
Allele	Solution variable's value
Fitness	Objective function (to maximize)
Crossover	Recombination (of two solutions)
Mutation	Local move
Evolution	The search process

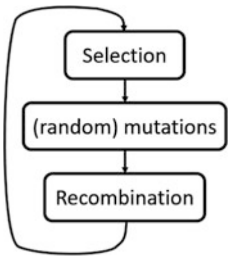


Fig. 4.1 Evolutionary algorithms main loop

4.2.1 Genetic Algorithms

Genetic Algorithms (GAs) are probably the historically most important EC methods. Initially, they were assumed to be independent of local search algorithms. Yet, when the goal was to tackle combinatorial optimization problems, it was soon clear, that the inclusions of local search algorithms were determined essential for many applications. However, to let things easy to understand, we do not discuss this here any further. Here, we focus on several cases for which the GAs optimizing steps are the same, irrespectively of the specific problem to solve. The linkage to the problem can be given only by:

- the encoding of the solutions; and
- the fitness function to maximize.

GAs work best on solutions encoded as unconstrained equal length binary strings. When possible, this encoding is preferred, though more complex solution encodings, up to trees or whole graphs, are possible and have been successfully used.

After having defined encoding and fitness function, GAs dictate to generate an initial set of tentative solutions and then to iterate, until a terminating condition, the main loop consisting essentially of three procedures: selection, recombination, and mutation.

All constituting elements of GAs, namely encoding, fitness, initialization, recombination, and mutation, vary enormously in the literature, thus the case to consider GAs as a class of algorithms, rather than an algorithm that gets adapted to different use cases. In fact, here we focus attention to the most standard mode, assume acting on a population $X = \{\mathbf{x}^1, \ldots, \mathbf{x}^i, \ldots, \mathbf{x}^m\}$ of tentative solutions, where each solution $\mathbf{x}^i \in X$ is a binary string with fitness f_i. The main GAs steps are:

- *Initialization* is usually implemented by a random generation of a number of solutions or by greedy solutions. This number is kept constant throughout search. The number of solutions in the population is usually one of the parameters of the algorithm.
- *Selection* is the next step. It can be implemented quite differently, with or without the need for an explicit computation of fitness values. The most common ones are the following.

1. *Roulette wheel selection* is a selection scheme where each surviving solution is determined by a Monte Carlo selection with repetition among all elements of X. The probability of selection of each element i is given by $p_i = f_i / \sum_{k=1}^{m} f_k$, thus it can happen that an element gets selected and appears more than once in the surviving population.
2. *Rank-based selection* is a distribution $p_1, \ldots, p_m$ of non-increasing probability values and it is defined a priori. All elements of X are ordered by decreasing fitness values, then selection goes on as in roulette wheel selection, with the probability of extraction of the k-th element in the ranking being p_k. Note that it is not necessary to have quantified fitness values, it is enough to be able to rank solutions from best to worst.
3. *Tournament selection* is a selection where each element of the surviving population is determined by randomly selecting two (or more) elements of the original population and keeping the one with higher fitness. Again, an element can be selected more than once. Note that also in this case it is not necessary to have quantified fitness values, not even a complete ranking, just comparisons do the job.

- *Recombination* is an operator that easily becomes very problem-specific when the chosen encoding permits to represent infeasible solutions. The idea is always to select two elements of the surviving population, the *parents*, and to recombine them in order to get two new *offspring* solutions. The easy case of recombination among unconstrained same-length binary arrays is usually implemented either as *one point* or *two points* crossover, see Fig. 4.2. In the first case, an index (a *cutting point*) is chosen, smaller than the array length, and all values beyond that index are swapped. In the second case, two indices are chosen, and the values between them are swapped. This operator is usually applied in probability to each pair of chosen parents, if it is not applied the two children are the same as the parents. The probability of crossover being applied is another of the standard parameters of the genetic algorithm, p_c.
- *Mutation* is originally just a random move in a predefined neighborhood of a solution. The standard implementation in the case of binary strings involves flipping with probability p_m each bit of each solution, where p_m is the third of the standard parameters of the GA. Later, the mutation has become more involved, and sometimes it was replaced with the local search on solutions.

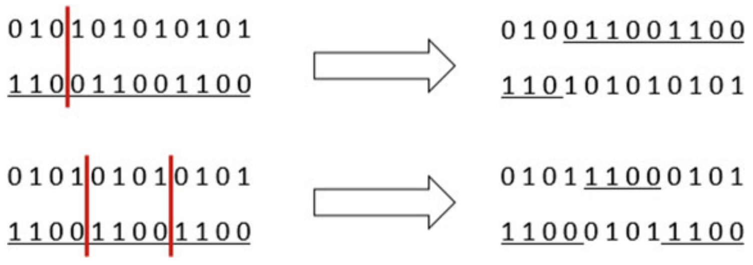


Fig. 4.2 One point and two points crossover

Clearly, the most innovative elements of GAs derive from their managing a set of solutions rather than a single one. This permits the design of optimization features that operate on sets of solutions, which are clearly inconceivable in the single solution case. While some of these features will be presented later in this section, when describing matheuristic-specific contributions, we recall here two of them, which have also been adopted in other population based algorithms.

Elitism Elitism consists of forcing a copy of the best solution from one iteration into the solution set of the next. Since the selection operator works in probability, there is otherwise no assurance that the best solutions found will survive across the iterations. While this is not a problem with respect to knowing which was the best solution found, which would hopefully be copied in a specific variable anyway, its failing to reproduce would deprive the population of possibly highly effective variable assignments (*building blocks*). Copying explicitly the best solution of one generation into the next ensures that its components are made available also for future recombinations.

Scaling In the later search stages, the quality of all solutions in the set tends to be comparable. If the selection operator is implemented as roulette wheel selection, for example, the selection probabilities will all be essentially equal. Selection becomes a purely random process, while fine-tuning would have been desired. This problem does not arise with tournament or rank-based selection, which are insensible to the magnitude of the difference in fitness among the solutions involved.

One possible solution for this problem in the case of roulette wheel selection is represented by *Dynamic Fitness Scaling*, see Fig. 4.3, where *min pop* and *max pop* stand for minimum and maximum objective function values among those taken by the individuals of the population. The technique asks to map the interval determined by the range of the objective function values into an interval corresponding to the desired range of fitness values (for example, [1, 100]). The mapping can be linear in the case of linear dynamic fitness scaling, or otherwise. If the mapping projects low objective values to low fitness values, the GA will try to maximize the objective function (left in Fig. 4.3), it will minimize the objective function if the slope is

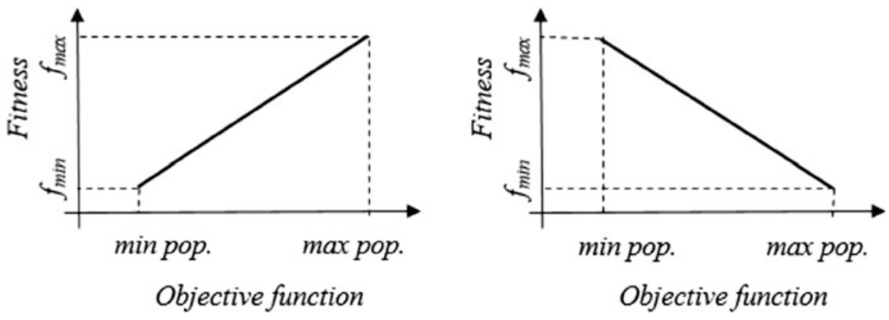


Fig. 4.3 Linear dynamic fitness scaling, for maximization (left) and minimization (right)

downward (right in Fig. 4.3). A nice property of linear dynamic fitness scaling is that intervals between objective function values will be proportionally kept also between the corresponding fitness values. A second one is that it reduces the sensitivity to the scale of the spanned values, permitting to avoid diving on much better solutions in the initial search phase and to discriminate small spans when fine-tuning in the last phase.

As mentioned, much research has been done on GAs and contributions span a really wide range of techniques and features. Among the most interesting ones, we point out that GAs are able to optimize a function even without knowing which function it is, let alone asking for its differentiability or continuity, and that they can provide a set of alternative, effective solutions. Among the cons, we can list the slowness of the process, the difficulty in fine-tuning solution and getting at least local optima (when not local searches are used), the need of adapting the basic operators when constraints are imposed on solutions, the difficulty to scale with the instance size. The integration with mathematical programming modules seems therefore highly promising.

4.2.2 Evolution Strategies

Evolution Strategies (ES) are algorithms similar in spirit to GAs, but more normative in their working. They have been proposed some time earlier than GAs, but their diffusion was initially mainly restricted to German-speaking countries, though they gained much larger visibility after the success of GAs. The general structure of the ES algorithm is the same as that of GAs, one main loop on a population of candidate solutions, which is acted upon by selection, reproduction, and mutation operators. One first main difference is that solutions were expected to be arrays of real numbers, rather than bit strings, though there are some exceptions to that rule. Peculiar to ES is the encoding within the real-valued array that represents a solution of both the data and of the control parameters of the operators that will act upon it.

Entering into some details and making use of the standard ES notation, the population is made by μ elements (solutions), each of which is composed of a numeric sequence x_i^s representing the solution and possibly another x_i^p representing the control parameters of the operators. At each iteration, a new population of λ elements is defined, with $\lambda \geq \mu$, which will then be reduced to μ by means of selection. This is symbolized by denoting the algorithm as (μ, λ)-ES. Another is the $(\mu + \lambda)$-ES, where the μ and λ are the same as before but the $+$ tells that the best μ of the $\mu + \lambda$ are kept; this means that the best μ solutions are kept and so it also holds for $\lambda > 1$.

Each ES loop in Algorithm 21 first constructs a population of λ solutions by iterating λ times a loop that:

- randomly selects ρ parent elements from the current population (the number of parents is not restricted to 2 as in GAs);

Algorithm 21: Simple (μ, λ)-ES

```
1  function EvolutionStrategies($\mu$, $\lambda$, $\rho$);
   Input  : Control parameters $\mu, \lambda, \rho$
   Output: A set of feasible solutions $X$
2  Initialize $X = \{\mathbf{x}_i = [x_i^s, x_i^p] | , 1 \leq i \leq \mu\}$;
3  repeat
4      for $i = 1, \ldots, \lambda$ do
5          $[x_i^s, x_i^p]$ = recombine(select_parents($X$, $\rho$));
6          $x_i^s$ = mutate($[x_i^s, x_i^p]$);
7          $x_i^p$ = mutate($x_i^p$);
8      end
9      $X = X \cup \{[x_i^s, x_i^p] | , 1 \leq i \leq \lambda\}$;
10     $X$ = select($\mu$, $X$);
11 until (termination condition);
```

- recombines the ρ selected parents obtaining one offspring, $\mathbf{x}_i$ (step 5);
- mutates the decision variables of the offspring, based on the control parameters, as encoded in the solution itself (step 6);
- mutates also the control parameters part of the offspring based on the current control variables values (step 7).

The parent selection has expectedly been proposed in many flavors, but it is often implemented as uniform random selection. Likewise, recombination can be done in many different ways, but it is often implemented either as a copying of each variable's value from one randomly selected parent, or by some simple function computed on the values taken by the corresponding variable in the parents (for example, taking their mean, or their maximum, etc.).

More interesting is the mutation operator. Since we work on continuous variables, simple operations such as flipping bits would be ineffective. Mutation should introduce small variations in the solution, and this is usually achieved by adding a symmetric perturbation to each variable value obtained after recombination. The perturbation is obtained by adding to the variable a number drawn from a multivariate normal distribution, $\mathcal{N}(0, \mathbf{C})$, with zero mean and covariance matrix $\mathbf{C} \in R^{n \times n}$, where n is the length of the affected solution part. The result is $\mathbf{x} + \mathcal{N}(0, \mathbf{C})$.

Different covariance matrices affect differently the expected mutation outcome. Figure 4.4 shows two possible normal distributions in 2 dimensions, overlaid by the corresponding 95% confidence ellipsoid. Each subfigure also shows 500 sample points (5% of which are likely to fall outside of the ellipsoid) along with the principal axes of the ellipsoids, as obtained from the eigenvectors of the covariance matrix, rescaled by multiplying by the square root of the corresponding eigenvalues, and by 0.95, i.e., by the confidence level degree. The left subfigure has the covariance matrix proportional to the identity, so that mutations are equally probable along any direction, while the right subfigure has the covariance matrix symmetric

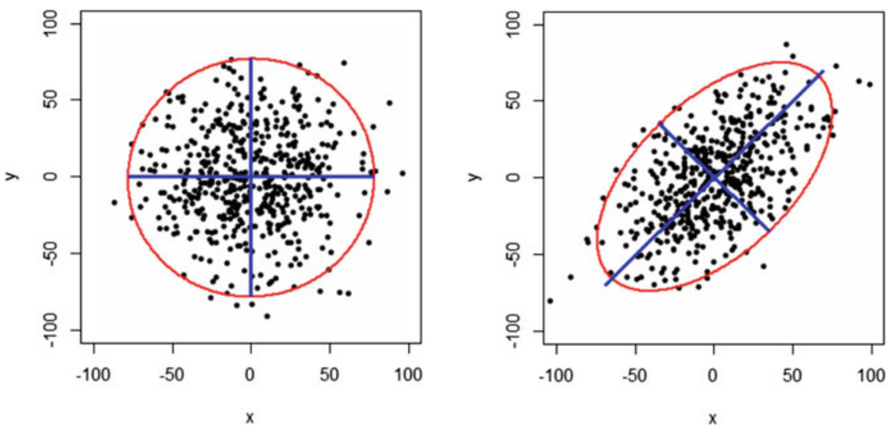


Fig. 4.4 Covariance based mutation, equal eigenvalues (left) or different eigenvalues (right)

and positive definite but not an identity, so that there are favored mutation directions. Aligning these directions with the objective function gradient can help local tuning.

The last element to describe of the algorithm is the selection at step 10. This is often simply implemented in the (μ, λ)-ES by keeping in the new population only the μ best solutions of the larger one, thus by discarding its $\lambda - \mu$ worse solutions.

4.2.3 Matheuristic Evolutionary Algorithms

Both GAs and ES are very flexible algorithmic frameworks and have been extended and integrated in many different ways. We like to mention here, though outside of the scope of this book that does not include heuristics for exact methods, the final polishing phase that is implemented as an evolutionary algorithm by exact MIP solvers, such as for example IBM Cplex. Sticking to MP methods for heuristic solving, we focus on matheuristic adaptations. The most investigated issue has been the recombination of parents for ensuring an effective evolution of the whole population.

The topic of generating the best possible offspring given two parent solutions represented by binary encoding has been treated also theoretically, obtaining some polynomial and NP-hard recombination problems. There have been two general approaches to optimized crossover:

- fixing the solution parts common to both parents and optimizing the rest;
- making the union of the parent solution components and optimizing within that set.

The first approach is a GA-adapted version of the widely used technique of variable fixing. The variable to fix are those common to both parents, which could be critical: if they are too many, it is likely that the optimization will just produce one of the two parents, while if they are too few, the search space is almost as large as that of the whole problem. In any case, only one offspring is generated from any selected pair of parents.

Use cases of the utilization of this last approach include a MIP-based recombination operator integrated within a GA for solving a supply management problem. The mathematical operator has the aim of finding the best possible combination of two given parent genotypes, simulating the mutation process with the addition of a random element. The recombination problem is modeled as a MIP problem in which all variables with zero value in both parent genotypes are fixed, except for a random subset of such variables. Reported computational results indicate the validity of this recombination when compared against the greedy-based GA. A similar MIP-recombination operator for solving a problem of balancing transfer lines with multi-spindle machines has also been implemented.

However, matheuristic contributions are not limited to an optimized crossover. A so-called dynastically optimal recombination, tries to use problem knowledge to identify the best combination of the features of the ancestors. A branch and bound method can be used within the algorithm as an operator to explore the potential of recombined solutions. Another example of the integration of an exact method within a GA was used on a problem of relay placement for wireless sensor networks. The authors proposed a decomposition process in which, at the top level, a GA finds possible relay placements and a MILP solver, at a lower level, computes a solution containing the optimal routing flow for the considered relay placement. In the area of routing, we have a study on the bus rapid transit route design problem, which is the problem of finding a set of routes and frequencies that minimize the operational and passenger costs in a bus rapid transit system. The authors propose a hybrid genetic algorithm where the evaluation of the fitness of each solution is made through the optimal solution of an LP program, which is solved by column generation.

4.2.4 Genetic Algorithm for the GAP, an Example

We provide here an example of a matheuristic GA applied to the GAP. The algorithm implements the strategy of Yagiura and Ibaraki (1996) of optimizing crossover by fixing the assignments which are common to both parents and letting MIP optimize the rest.

The pseudocode is presented in Algorithm 22, and it is necessary to specify steps 2, 6, 8, and 13.

The initialization step 2 is possibly the most precarious one, given the strictly NP-hard nature of the GAP. For instances that are not tightly constrained, initialization can be made by repetitively calling a randomized version of Algorithm 2, otherwise more sophisticated techniques are needed.

Algorithm 22: Genetic algorithm for the GAP

```
1  function GAforGAP($n$, $p_c$, $p_m$);
   Input  : Control parameters $n$, $p_c$, $p_m$
   Output: A set of feasible solutions $\mathbf{X}$
2  Initialize $X = \{\mathbf{x}_i |, 1 \leq i \leq n\}$;
3  repeat
4      $\mathbf{X}' = \emptyset$;
5      for $k = 1, \ldots, n$ do
6          $\mathbf{x}'_k$ = select_rep($X$);                              // selection
7          if $rand() \leq p_c$ then
8              $\mathbf{x}''_k$ = select_rep($X$);                         // crossover
9              $J = \{ j : \exists i \in I, x'_k[ij] = x''_k[ij] = 1\}$;
10             $\mathbf{x}'_k = MIPsolve(J)$;
11         end
12         if $rand() \leq p_m$ then
13             $\mathbf{x}'_k$ = swap_client_assignment($\mathbf{x}'_k$);   // mutation
14         end
15         $\mathbf{X}' = \mathbf{X}' \cup \{\mathbf{x}'_k\}$;
16     end
17     $X = \mathbf{X}'$;
18 until (termination condition);
```

The selection step 6 implements an extraction with repetition of one element $\mathbf{x}'_k$ from the set of solutions X. Any approach is viable, for example, roulette wheel selection with dynamic fitness scaling.

The crossover step 8 is applied with probability p_c, which is one of the input arguments of the GA procedure. If it is not applied, $\mathbf{x}'_k$ is passed on for possible mutation, otherwise a second element $\mathbf{x}''_k$ is selected, to be mated with $\mathbf{x}'_k$. The actual crossover is implemented by identifying the set J of indices of clients, which have been assigned to one same server. These assignments are fixed, then procedure *MIPsolve* is called in order to optimize the remaining partial GAP, obtained when the clients in J are removed from the client set. Note that this procedure produces one offspring for each pair of parents, which is not the standard GA practice, where often two parents produce two offspring.

Here follows part of the trace of an execution of Algorithm 22 applied to instance *example8x3* of Sect. 1.

Let function *GAforGAP* be called with arguments $n = 12$, $p_c = 0.7$ and $p_m = 1/l = 0.125$, where l is the number of bits encoding each solution. After a call to a randomized version of Algorithm 2, the population is the one presented in Table 4.2. The randomization in this case was achieved by expanding step 8 of Algorithm 2 as follows. Given the client, instead of trying to assign it to its least cost server, a candidate list was used (see GRASP, Sect. 3.6). We included in the client's candidate list its best, i.e., least-cost, *listSize* servers (in this case with $listSize = 2$), and we extracted with uniform random probability the server to assign the client to. As the resulting solution was generally infeasible, a simple fixing heuristic was finally used, trying to recover feasibility. Table 4.2 shows that this could be achieved in 9

Table 4.2 GA: initial population

Id	Sol.	Cost
1	0 1 0 1 0 2 2 2	$z = 340$
2	0 2 1 0 1 0 2 2	$z = 330$
3	1 2 0 1 1 1 2 0	$z = \infty$
4	0 1 0 1 2 2 2 0	$z = 331$
5	1 0 0 0 2 2 1 2	$z = 336$
6	1 0 0 2 1 2 2 0	$z = 328$
7	0 2 1 2 0 1 0 0	$z = \infty$
8	0 1 0 1 2 0 2 2	$z = 337$
9	1 1 0 2 0 2 0 2	$z = 328$
10	2 0 0 2 1 2 2 1	$z = \infty$
11	1 0 0 2 1 0 2 2	$z = 334$
12	0 2 1 1 0 0 2 2	$z = 329$

cases out of 12, the rows with cost $z = \infty$ correspond to infeasible solutions, which can anyway be used in the remainder of the procedure.

During the first iteration, 10 crossovers were launched, out of 12 iterations of the loop at step 5. We focus on one of these, which involved the mating of individuals 8 and 2, of costs $z = 330$ and $z = 337$, respectively. The individuals are:

$$\mathbf{x}_8 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{4.1}$$

and

$$\mathbf{x}_2 = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{4.2}$$

which agree on the assignments on positions 1, 6, 7, and 8, thus originating the following fixings:

$$\mathbf{x}_f = \begin{bmatrix} 1 & * & * & * & * & 1 & 0 & 0 \\ 0 & * & * & * & * & 0 & 0 & 0 \\ 0 & * & * & * & * & 0 & 1 & 1 \end{bmatrix} \tag{4.3}$$

The partial problem specified in Eq. (4.3) is then passed to MIP for solving, obtaining the solution in Eq. (4.4), of cost $z = 329$.

$$\mathbf{x}_f = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{4.4}$$

Table 4.3 GA population fixings

Mating	Fixes
8–12	0 * * 1 * 0 2 2
12–11	* * * * * 0 2 2
5–11	1 0 0 * * * * 2
8–6	* * 0 * * * 2 *
11–2	* * * * 1 0 2 2
6–5	1 0 0 * * 2 * *
6–2	* * * * 1 * 2 *
1–4	0 1 0 1 * 2 2 *
6–4	* * 0 * * 2 2 0

For completeness, we report in Table 4.3 all fixings that were tested in the first iteration.

4.3 Ant Colony Optimization

Ant Colony Optimization (ACO) is the name given to a class of algorithms, which generalizes a seminal proposal under the name of *Ant System*. A common trait of all ACO algorithms, as originally put forward by the Ant System, is the indirect communication among different constructive threads, aimed at improving the local choices locally made by each one of them, at each of their iterations.

The basic ACO idea derives from a biological observation, which kindled the computational model (in the late 1980s and early 1990s this design approach was not yet overused). The observation was obviously about ants, specifically about their way of identifying efficient paths from their nest to food patches.

Figure 4.5 shows what happens when a trail of ants meets a sudden obstacle, in this case, a brick suddenly put across the ant trail, as seen from above. Initially,

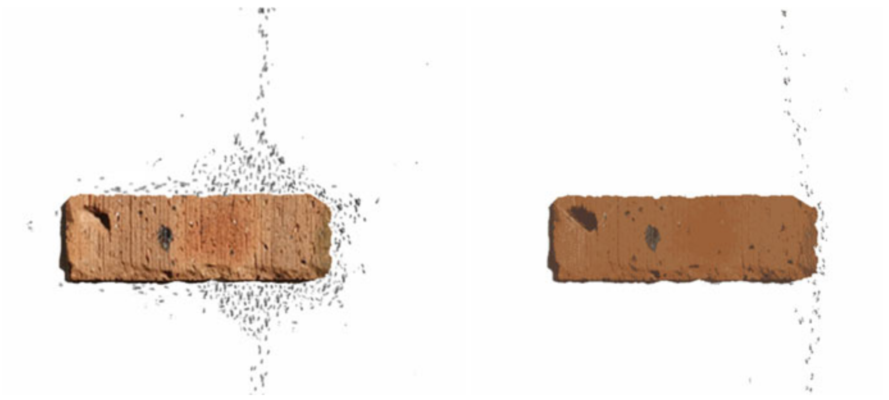


Fig. 4.5 Ants bypassing an obstacle: after a few minutes (left) and after several hours (right)

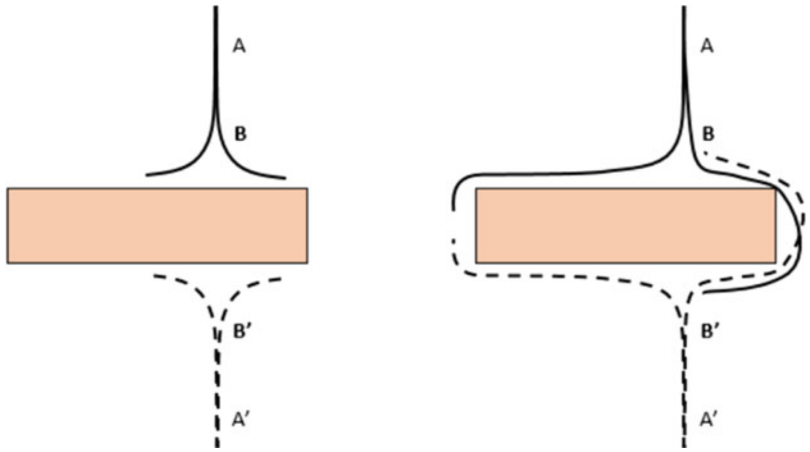


Fig. 4.6 Ants bypassing an obstacle: autocatalytic effect

the ants do not have a clue on how to circumvent the obstacle and try all ways. After a while, an efficient way round is discovered, and it gets consistently chosen afterward.

The natural process underlying this ability is based on particular molecules, *pheromones*, which are laid on the ground by wandering ants. In fact, when an ant moves, on a path connecting a food patch and its nest, it lays on the ground a trail of pheromone, which has the effect of directing any wandering ant toward the food.

If a sudden obstacle cuts the path, the ants initially have no clue about how to bypass it, and they choose at random a direction to follow, so that, for example, on either side of the obstacle half of them goes right and half of them left (Fig. 4.6 left). However, being for example the right course faster when coming from below, there will be a moment when the arriving ants see on the right both the trail left by those who previously chose that way around and the trail left by those who already walked around the obstacle (Fig. 4.6 right). The ants incoming from below will therefore find on the right twice as much pheromone than on the left, the reverse being true for those coming from above.

The choice of the path to follow is not deterministic, but it is biased by the pheromone: the more the pheromone, the higher the probability to follow it. Incoming ants will therefore choose the right course with a probability twice of that of choosing the left course. Twice as many ants are thus expected to go right and to lay their pheromone on that path, further reinforcing the difference of pheromone intensity between the shorter and the longest paths.

This gives rise to a so-called *autocatalytic*, or *positive feedback* loop, where the more ants choose the shortest way, the more inclined will the next ones be to follow it themselves. After a (relatively) short time, most of the ants will choose the shortest path, except for a few exceptions who anyway opt for the low probability path.

ACO algorithms implement a loose model of this autocatalytic process, keeping the idea that, when facing a choice, agents (ants) lay a trail, that will induce subsequent agents to repeat in probability the same choice made by the trail-laying agent. The more intense the trail, the higher the probability to make the choices it suggests.

4.3.1 ACO Metaheuristics

The Ant System (AS) drew upon the observations introduced in the previous section, defining a parallel randomized constructive heuristic where some components derive from ants behavior.

AS is a population based metaheuristic, using a pool of threads, each implementing an agent that iteratively constructs a solution of the problem to solve. Following the entomological nomenclature, the constructive agent of each thread was named an *ant*, which is said to *move* from a partial solution to another, more complete, partial solution, until completing a feasible solution.

More in general, partial problem solutions can be seen as *states*. Each ant moves from a state ι to another one ψ, corresponding to a more complete partial solution. Note that in the following ι and ψ are used to denote *states*, i.e., partial solutions, while i and j will be used to index components of solutions. For example, in the case of the GAP, ι could be a partial solution where only one half of the clients are assigned to servers, and ψ a partial solution where one more client is assigned, thus the move corresponds to the assignment of one client. In the case of the TSP, ι could be a Hamiltonian path connecting a subset of the nodes to traverse, and ψ a path where one of the exposed endpoints is connected to a previously unvisited node, thus the move would correspond to appending an arc to the path.

At each step σ, each ant computes a set of feasible expansions to its current state and moves to one of these in probability. The structure is the same as that of constructive heuristics (Sect. 1.3.1) but the ordering of the components is not deterministic. Rather, at each step, the next component to include in the solution is chosen taking into account:

- the a priori expectation, $\eta_{\iota\psi}$, of the usefulness of a particular component, which would be included in the solution by move (ι, ψ), as in standard constructive approaches. The a priori measure of expected usefulness of a move, i.e., of the corresponding component to be included, is named the *attractiveness* of the move. In several applications, its value does not change during search.
- an *a posteriori* measure, $\tau_{\iota\psi}$, of the goodness of solutions constructed using that particular move/component. The a posteriori measure of expected usefulness of a move is named the *trail* on the move. Its value gets iteratively updated during search.

Given a set (a population, in ACO jargon) P of ants, for each ant $k \in P$, the probability $p_{\iota\psi}$ of moving from state ι to state ψ depends on the combination of two values: the attractiveness of the move, as computed by some heuristic indicating the a priori desirability of that move, and the trail level of the move, which indicates how proficient it was in the past to make that particular move.

Probability of a Move
The formula for defining the probability distribution of the choice of alternative moves is often different in the different declinations of ACO algorithms. However, the best known formula comes from AS, where probabilities were computed as follows. Let $\varPsi$ be the set of feasible expansions of state ι. For each ant $k \in P$ and for each $\psi \in \varPsi$ we can compute

$$p^k_{\iota\psi} = \frac{\tau^{\alpha}_{\iota\psi}\eta^{\beta}_{\iota\psi}}{\sum_{\nu\in\varPsi}\tau^{\alpha}_{\iota\nu}\eta^{\beta}_{\iota\nu}} \tag{4.5}$$

The formula combines the attractiveness $\eta_{\iota\psi}$ of the move from state ι to state ψ with the corresponding trail $\tau_{\iota\psi}$ and normalizes this value considering all possible expansions of state ι. Two user-defined parameters are included, α and β, which gauge the relative importance of trail with respect to attractiveness.

Trail Update
Attractiveness values are input data or get computed before starting search and could never be updated during the entire search. For example, in the case of the TSP the attractiveness could be the inverse of the arc length or, in the case of the GAP, the inverse of an assignment cost. Trails on the contrary are iteratively updated during search, increasing the level of those related to moves that were part of "good" solutions, while decreasing all others. In this way, moves that contributed to good solutions increase their desirability while moves that often led to bad solutions are less likely to be chosen again.

After each iteration t of the algorithm, trails are updated using the following formula:

$$\tau_{\iota\psi}(t+1) = \rho\tau_{\iota\psi}(t) + \varDelta\tau_{\iota\psi} \tag{4.6}$$

where ρ, $0 \leq \rho \leq 1$, is a user-defined parameter called *evaporation coefficient*, and $\varDelta\tau_{\iota\psi}$ represents trail variation deriving from the sum of the contributions of all ants that used move (ι, ψ) to construct their solution. The ants' contributions must be proportional to the *quality* of the solutions achieved, i.e., the better a solution is, the higher will be the trail contributions added to the moves it used. Details on how this is actually implemented vary a lot among different ACO variants.

As a consequence of formula (4.6), trails will eventually disappear on all moves that are never chosen, making their choice ever more unlikely, and steer future choices toward moves that provided high contributions, i.e., that led to good solutions.

Ant System
Building on the elements introduced in the previous subsections, the AS algorithm is presented as Algorithm 23.

Algorithm 23: Ant system

```
1  function AntSystem($\alpha, \beta, \rho$);
   Input : Control parameters $\alpha, \beta, \rho$
   Output: A feasible solution $\mathbf{x}$
2  Initialize a population P of $m$ empty solutions: $\boldsymbol{\sigma}^k = \emptyset, k = 1, \ldots, m$;
3  Initialize $\tau_{\iota\psi}$ and $\eta_{\iota\psi}, \forall(\iota, \psi)$;
4  repeat
5      foreach ant k (currently in state $\iota$) do                    // Construction
6          repeat
7              choose in probability the state to move into using formula (4.5);
8              append the chosen component to the $k$-th ant's solution, $\boldsymbol{\sigma}^k$;
9          until (ant k completes its solution);
10     end
11     foreach ant move $(\iota, \psi)$ do                                // Trail update
12         compute $\Delta\tau_{\iota\psi}$;
13         update the trail matrix using formula (4.6);
14     end
15 until (termination condition);
```

The user-defined parameters used in algorithm AS are:

m:	number of ants in the population;
α:	relative importance of attractiveness;
β:	relative importance of trail;
ρ:	evaporation coefficient;
τ_0:	initial trail value.

The main characteristics of the Ant System algorithm can be identified in the following points.

- AS is a general purpose heuristic inspired by nature. Its structure, adding a stochastic component to a basic constructive approach, is not specific to a problem or to a class of problems, but it can be adapted to any combinatorial optimization problem.
- AS implements a parallel distributed search effort. Each ant constructs its solution without any direct communication with any other one, coordination is obtained indirectly by means of trail update. This permits us to run all constructive threads in parallel, potentially achieving very good speedups with respect to a serial implementation.
- AS combines a problem-specific constructive heuristic (attractiveness) and a general purpose adaptive problem representation, allowing to bias the heuristic indications toward features of the specific instance to solve.

AS has been generalized into Ant Colony Optimization (ACO), representing a class of algorithms, which share the main underlying idea of parallelizing search over several constructive computational threads, all based on a dynamic memory structure with information on the effectiveness of previously obtained results and in which the behavior of every single agent is (somewhat) inspired by the behavior of real ants.

Different ACO instantiations vary widely, therefore a general ACO pseudocode results to be quite abstract. Algorithm 24 presents this general ACO structure.

Algorithm 24: Ant colony optimization

```
1 function ACO_MetaHeuristic();
2 while not termination condition do
3     generateSolutions();
4     daemonActions();
5     pheromoneUpdate();
6 end
```

The algorithm iterates infinitely over three main steps. In the first step of each iteration, step 3, each ant stochastically constructs a solution by successively appending randomly chosen components to the incumbent partial solution. In the second step, step 4, problem-specific or centralized actions may be included, which cannot be performed by single ants, such as, for example, the application of local search to the constructed solutions. The third step, step 5, consists of updating the pheromone levels on each edge.

There have been dozens of different implementations of this basic structure. Among these, we remind:

- Ant Colony System (ACS), where pheromone trails are added at the end of each main iteration only to the arcs belonging to the best tour, while ants perform local pheromone trail decreases at each step, to favor the emergence of different solutions. Moreover, the minimum pheromone value is limited.
- Max-Min Ant System (MMAS), where there is a control on the maximum and minimum pheromone amounts associated with each move. In the first versions, only the global best solution or the iteration best solution is allowed to add pheromone to their trails. Moreover, trails are reinitialized when the search does not produce improving solutions for long.

4.3.2 *Matheuristic ACO*

ACO has been one of the first metaheuristics to include elements from mathematical programming. Below, we detail one such early contributions, but this has not been the only ACO-based matheuristic.

Another example from the literature can be found, in which the author proposed an ACO method for solving a symmetric traveling salesman problem. In this work, the attractiveness among pairs of customers is defined using information derived by the calculation of a Minimum Spanning Tree Problem (MSTP). MSTP and related attractiveness are computed before the beginning of ACO. The structural information given by the presence of arcs in an MSTP solution is used for calculating the attractiveness of each arc. Computational tests have been executed on some instances from TSPLIB, which showed that using MSTP information permits to obtain better quality solutions than otherwise.

A further example deals with cooperative wireless networks. In this work, the approach combines step 3 of the canonical ACO algorithm to Algorithm 24, followed by a daemon action phase 4, where a large neighborhood of the generated feasible solutions is explored by exactly solving a Mixed-Integer Program (this can be described as an ANTS implementation, see Sect. 4.3.3). The quality of the feasible solutions found through the ant-construction phase is refined by a modified Relaxation Induced Neighborhood Search (RINS, see Chap. 5).

A noteworthy contribution in this respect is *Beam-ACO*, which combines a standard ACO structure with a Beam Search (BS) component. Beam Search is described in some detail in Chap. 10, but we can sketch it by saying that it essentially consists of an incomplete tree-search exploration, where at each node expansion only a subset of the offspring is explored, all other children nodes being discarded. In Beam-ACO, artificial ants perform a probabilistic Beam Search in which the extension of partial solutions is done in the ACO fashion rather than deterministically.

ACO and BS are in fact both based on the idea of constructing candidate solutions step-by-step. However, the ways by which the two methods explore the search space are different. The policy that is used in BS algorithms for extending partial solutions is usually deterministic, while ACO algorithms explore the search space in a probabilistic way, using past search experience in order to find good areas of the search space. In Beam-ACO, each artificial ant performs a probabilistic BS. This probabilistic BS is obtained from standard BS by replacing the deterministic choice of a solution component at each construction step by a probabilistic choice based on ACO-style transition probabilities. As the transition probabilities depend on the changing pheromone values, the probabilistic beam searches that are performed by this algorithm are adaptive, a feature shared with all ACO explorations. This general working is very similar to that of a previous ACO variant, which is detailed in the following Sect. 4.3.3.

4.3.3 ANTS

ANTS, an acronym for *Approximate Nondeterministic Tree Search*, represents one of the first algorithms proposed in the literature that included Mathematical Programming (MP) elements in a metaheuristic structure. It was motivated by the observation that a general ACO algorithm (actually the Ant System) includes some elements which can otherwise be computed by more standard MP functions. It was therefore natural to substitute ad-hoc solutions with better grounded ones. The result is an ACO code, which was named ANTS, where some steps contain calls to MP functions.

The MP elements that were included in the ACO structure are the following ones.

Lower (Upper) Bounds

The attractiveness η of a move can be effectively estimated by means of lower bounds to the cost of the completion of a partial solution (upper bounds in case of maximization problems). A lower bound limits, in fact, the cost of a complete solution containing it. For each feasible move (ι, ψ), i.e., form state ι to state ψ, it is possible to compute the lower bound to the cost of a complete solution containing ψ: the lower the bound the better the move.

Since a large part of research in combinatorial optimization is devoted to the identification of tight lower bound for the different problems of interest, good lower bounds are usually available. They can prove beneficial for ACO search for different reasons.

- A tight bound gives strong indications on the effectiveness of a move.
- If the bound value becomes greater than the current upper bound, we know that the considered move leads to a partial solution that cannot possibly be completed into a solution better than the current best one. The move can therefore be discarded from further analysis.
- If the bound is computed by means of linear programming, dual cost information becomes available. It is possible to compute reduced costs for the decision variables, which in turn—when compared to an upper bound to the optimal problem solution cost—permit to a priori eliminate some variables. This results in a reduction of the number of possible moves, therefore in a reduction of the search space.
- If the bound is computed by means of linear programming, the primal values of the decision variables, as appearing in the optimal bound solution, can be used as an estimate of the contribution of each variable to good solutions. This can provide an effective way for initializing the trail values, thus eliminating the need for the user-defined parameter $\tau_{\iota\psi}(0)$, which becomes a dependent value.

Structural Simplicity

Formula (4.5) makes use of two user-defined parameters, α and β, to specify the relative importance of attractiveness versus trail. Ockham's razor suggests that *entities must not be multiplied beyond necessity*, and a single parameter could suffice to gauge the relative impact.

ANTS suggests therefore to simplify formula (4.5) into

$$p_{\iota\psi}^{k} = \frac{\alpha\tau_{\iota\psi} + (1-\alpha)\eta_{\iota\psi}}{\sum_{\nu\in\Psi} \alpha\tau_{\iota\nu} + (1-\alpha)\eta_{\iota\nu}} \tag{4.7}$$

Formula (4.7) achieves the same objective of formula (4.5), that of letting the user specify a different relative importance of trail with respect to attractiveness while eliminating one parameter β and using more computationally efficient components, multiplications instead of powers to real-valued exponents.

Stagnation Avoidance
Stagnation denotes the undesirable situation in which all agents repeatedly construct the same solutions, making impossible any further exploration in the search process. In the case of ACO, this derives from an excessive pheromone trail level on the moves of one solution, a situation that can be observed in advanced phases of the search process if parameters are not well-tuned to the instance. In particular, stagnation is strongly affected by the value of parameter ρ in the Ant System. If it is too high, stagnation might take place, while if it is too low, too little information is conveyed from previous solutions and the Ant System becomes an iterated randomized greedy procedure.

Several ACO variants intervened to face this difficulty, often by adding further parameters to constrain the acceptable trail values (see Max-Min Ant System or Ant Colony System, introduced above). ANTS takes a different path by updating trail using a different mechanism. The procedure evaluates each solution against the last k constructed ones. As soon as k solutions become available, the moving average of their costs $\bar{z}$ is computed and each new solution z_{curr} is compared to $\bar{z}$ (and then later used to compute the new moving average value). If z_{curr} is lower than $\bar{z}$, the pheromone trail level of the last solution moves is increased, otherwise, it is decreased. Formula (4.8) specifies how this is implemented for minimization problems:

$$\Delta\tau_{\iota\psi} = \overline{\tau}(0) \cdot \left(1 - \frac{z_{curr} - LB}{\bar{z} - LB}\right) \tag{4.8}$$

where $\bar{z}$ is the average of the last k solutions, $\overline{\tau}(0)$ is the average initial trail level, and LB is a lower bound to the optimal problem solution cost. This corresponds to using *dynamic scaling* for trail updating, Fig. 4.7 shows the suggested function.

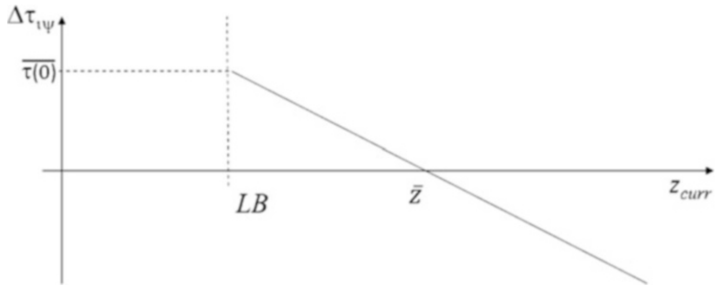


Fig. 4.7 Tau dynamic scaling

The use of this dynamic scaling procedure permits to discriminate small achievement in the latest stage of search, while avoiding to focus search only around good achievement in the earliest stages. The presented mechanism makes no use of parameter ρ and, more significantly, provides a general approach to the identification of the solution contribution both in the case of maximization and of minimization problems (upon switching lower bound LB to upper bound UB and accordingly amending the formula). However, a new parameter is introduced: k the cardinality of the moving average, whose value is anyway much less critical than that of the parameter we eliminated.

Finally, trails get updated ensuring that they do not become negative.

$$\tau_{\iota\psi} = max(0, \tau_{\iota\psi} + \Delta\tau_{\iota\psi}) \tag{4.9}$$

Actually, there would be no contraindication in using negative trail values, i.e., repulsive indications, as long as the sum of the attractiveness and the trail component of a move remain non-negative. However, we are not aware of any computational verification of this possibility.

Using the three proposed elements, the ANTS algorithm becomes the following.

Algorithm 25: ANTS algorithm

```
1  function ANTS($\alpha, k$);
   Input  : Control parameters $\alpha, k$
   Output: A feasible solution $\mathbf{x}$
2  compute a (linear) lower bound $LB$ to the problem to solve;
3  initialize $\tau_{\iota\psi}, \forall(\iota, \psi)$, with the primal variable values;
4  initialize a population P of $m$ empty solutions: $\sigma^k = \emptyset, k = 1, \ldots, m$;
5  repeat
6      foreach ant k (currently in state $\iota$) do                    // construction
7          compute $\eta_{\iota\psi}, \forall(\iota, \psi)$                  // bound to completion
8          repeat
9              choose in probability the state to move into using formula (4.7);
10             append the chosen component to the $k$-th ant's solution, $\sigma^k$;
11         until (ant k completes its solution);
12         optionally, carry the solution to its local optimum;   // highly suggested
13     end
14     foreach ant move $(\iota, \psi)$ do                     // pheromone trail update
15         compute $\Delta\tau_{\iota\psi}$ using formula (4.8);
16         update the trail matrix;
17     end
18 until (termination condition);
```

It is easy to note that the general structure of the ANTS algorithm is closely akin to that of a standard tree-search algorithm. Solutions are progressively constructed, computing a lower bound at each node (partial solution). At each construction step, for each ant, we thus have a partial solution that is expanded by branching on all possible offsprings, computing a bound for each of them, possibly expunging dominated ones, and moving to one of them on the basis of lower bound considerations. By simply adding backtracking and eliminating the Monte Carlo choice of the node to move to, we revert to a standard branch and bound procedure, this is the reason behind the use of the denotation ANTS, for Approximate Nondeterministic Tree Search.

Notice that step 7 prescribes to compute the move attractiveness as a function of the bound to the cost of the completion of the solution, after fixing the move. In case of maximization problems, attractiveness can directly be set to the bound value (i.e., the higher the bound, the more attractive the move), while in case of minimization problems an inversion is needed (the higher the bound, the less attractive the move).

As a further remark, ANTS was proposed with two alternative branching strategies. The first one is a depth-first, standard in branch and bound algorithms, where the node expanded at each level is the offspring of the incumbent one having least cost lower bound. The second strategy is a Beam Search strategy where a number of nodes are expanded at the same level before stepping deeper into the search tree. This second strategy will be better discussed in Chap. 10, and has been later thoroughly investigated in Beam-ACO.

4.3.4 ANTS for the GAP

Section 4.3.3 presented the ANTS algorithm, which can be applied to many combinatorial optimization problems. In this section, we detail how this can be done in the case of the GAP. This requires to specify, in Algorithm 25, which lower bound is to be chosen (step 2) and what is a move when building a GAP solution (step 9), all other elements being those described in this chapter, except that we made explicit at step 13 the possibility of disregarding the contributions of an ant, as soon as we discover that it is not going to produce any useful solution.

We briefly remark that the computational efficiency of the ANTS procedure is greatly affected by the pre-ordering of the construction steps. In the case of the GAP this is the order in which the clients are considered for finding the next assignment. In the algorithm we propose here, we keep the ordering found in the input data, but more effective orderings can be determined following the considerations made for construction in Chap. 1.

The following Algorithm 26 makes use of a linear lower bound and specifies a new state to be a partial solution where one further assignment is added to the originating state/partial solution. The additive nature of the objective function ensures that, whichever state an ant starts from, the cost of a move is given by the cost of the single assignment whose addition defines the move. Therefore the cost increase when moving from partial solution ι to partial solution ψ is given only by the cost of the further assignment included in solution ψ. If the move assigns client j to server i, we have that $cost(\iota, \psi) = c(i, j)$. Thus, in the case of the GAP, we can denote the moves by i and j indices, with reference to the further assignment made by the move with respect to the starting partial solution. Note that this is not true for any combinatorial optimization problem, there are cases where the cost increase depends on the whole of the involved partial solutions. Finally, for ease of presentation, we did not include the local optimization step, i.e., step 12 of Algorithm 25, though this omission greatly hampers the effectiveness of the code.

Algorithm 26: ANTS algorithm for the GAP

```
1  function ANTS($\alpha$, $k$);
   Input  : Control parameters $\alpha$, $k$
   Output: A feasible solution $\mathbf{x}$
2  Compute a (linear) lower bound $LB$ to the GAP instance;
3  Initialize $\tau_{i,j}$ $\forall(i, j)$, with the primal variable values;
4  Initialize a population P of $m$ empty solutions: $\boldsymbol{\sigma}^k = \emptyset$, $k = 1, \dots, m$;
5  repeat
6      foreach ant k (currently in state ι) do                         // Construction
7          for ($i = 0, \dots, n-1$) do
8              Compute $\eta_{ij}$, $\forall j$;                        // lower bound to completion
9              if a further feasible assignment exists then
10                 choose in probability the new assignment to make using formula (4.7);
11                 append the chosen assignement to the $k$-th ant's solution, $\boldsymbol{\sigma}^k$;
12             else
13                 fathom the ant;
14             end
15         end
16     end
17     foreach ant move ($i$, $j$) do                                  // Trail update
18         compute $\Delta\tau_{ij}$ using formula (4.8);
19         update the trail matrix;
20     end
21 until (termination condition);
```

We present a trace of the execution of this algorithm on the instance *example8x3* of Sect. 1. Let the user-defined parameters be $\alpha = 0.4$ and $k = 3$.

Step 2 requires to compute a linear lower bound, which in this case has a value $LB = 231.45$, and which induces the following initial trails, values rounded to the third decimal.

$$\boldsymbol{\tau}(0) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0.773 & 1 & 1 & 1 \\ 0 & 0.305 & 1 & 1 & 0.227 & 0 & 0 & 0 \\ 1 & 0.695 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We consider just one of the ants. Its construction goes on assigning first client 0, then client 1, and so on until client 7.[1]

[1] We remind that in the computational traces the decision variables are indexed from 0.

For client 0 we consider the 3 possible assignments, computing for each of them the linear bound that is obtained by fixing the corresponding variables in the bound solution. For example, forcing the choice of server 0 for client 0 corresponds to setting $x_{00} = 1, x_{10} = 0$ and $x_{20} = 0$, which gives a bound of $LB_{00} = 241.57$. Since the GAP is a minimization problem, we have to inverse the cost, for example setting $\eta_{00} = 1/(1 + LB_{00} - LB) = 0.0539562$. Similarly, we obtain that $\eta_{10} = 0.094213$ and $\eta_{20} = 1$. These values are combined with the trail values using formula (4.7) obtaining

$$
\begin{aligned}
\alpha\tau_{00} + (1-\alpha)\eta_{00} &= 0.4 \cdot 0 + 0.6 \cdot 0.0539 = 0.032 \\
\alpha\tau_{10} + (1-\alpha)\eta_{10} &= 0.4 \cdot 0 + 0.6 \cdot 0.0942 = 0.057 \\
\alpha\tau_{20} + (1-\alpha)\eta_{20} &= 0.4 \cdot 1 + 0.6 \cdot 1 \qquad\quad = 1
\end{aligned}
$$

A standard Monte Carlo extraction has then been performed on the array of computed values, which randomly chose to assign client 0 to server 1, $x_{10} = 1$.

Subsequent constructive iterations were as follows.

For client 1 we have $LB_{01} = 246.053$, $\eta_{02} = 0.038$, $LB_{11} = 242.494$, $\eta_{12} = 0.172$, and $LB_{21} = 236.818$, $\eta_{22} = 0.372$ and the chosen server is $i = 2$.

For client 2 we have $LB_{02} = 241.693$, $\eta_{02} = 0.0534$, $LB_{12} = 239.236$, $\eta_{12} = 0.468$, and $LB_{22} = 247.636$, $\eta_{22} = 0.035$ and the chosen server is $i = 1$.

For client 3 we have $LB_{03} = 243.123$, $\eta_{03} = 0.047$, $LB_{13} =$ *infeasible*, $\eta_{13} = 0$, and $LB_{23} = 249.636$, $\eta_{23} = 0.031$, where the assignment of client 3 to server 1 cannot lead to any feasible solution. The option is discarded, Monte Carlo is run only on the two remaining options and the chosen server is $i = 0$.

For client 4 we have $LB_{04} = 247.169$, $\eta_{04} = 0.345$, $LB_{14} =$ *infeasible*, $\eta_{14} = 0$, and $LB_{24} = 253.349$, $\eta_{24} = 0.026$. Again, the assignment of client 4 to server 1 cannot lead to any feasible solution. The option is discarded, and the chosen server is $i = 0$.

For client 5 we have $LB_{05} = 251.383$, $\eta_{05} = 0.429$, $LB_{15} =$ *infeasible*, $\eta_{15} = 0$, and $LB_{25} = 257.143$, $\eta_{25} = 0.022$, and the chosen server is $i = 0$.

For client 6 we have $LB_{06} =$ *infeasible*, $\eta_{06} = 0$, $LB_{16} =$ *infeasible*, $\eta_{16} = 0$, and $LB_{26} = 261.024$, $\eta_{26} = 0.020$. This time we have two infeasible options, therefore the forcibly chosen server is $i = 2$.

For client 7 we have $LB_{07} =$ *infeasible*, $\eta_{07} = 0$, $LB_{17} =$ *infeasible*, $\eta_{17} = 0$, and $LB_{27} = 326$, $\eta_{27} = 0.006$. Again, we have two infeasible options, therefore the forcibly chosen server is $i = 2$, which leads to the feasible solution $\boldsymbol{\sigma} = (1, 2, 1, 0, 0, 0, 2, 2)$ of cost 326.

This actual simple example already shows how the MP component implements a lookahead that helps in steering the construction away from infeasibility, an outcome otherwise very likely if the instance is tightly constrained. Effective ANTS search on tightly constrained instances can be further strengthened, as put forward in the literature. This opportunity, coupled with the intrinsic flexibility granted by the possibility to discard ants unable to complete a solution with no adaptation required in the code, make ACO, and ANTS in particular, a prominent candidate for solving this type of instances.

Another observation that comes naturally from the example is that a local search is an obvious complement of ACO, as suggested in Algorithm 25. The construction approach is very flexible, but often incapable of fine-tuning the best-found solution down to a local minimum. ACO search in the space of local optima on the contrary can lead to very good solutions.

To complete the example, we suppose that the ant for which we reported the computations, was the only one of its population completing a feasible solution. In this case, variations of trail are only due to it. The contributions are computed by formula (4.8). Fixed constants in it are $\tau(0)$, in our case of value 0.333, and LB, in our case of value 231.45. The contribution to the trail on every move, which is (i, j) couplings in our case, is thus determined by the cost of the ant solution with reference to the average cost of the last k solutions computed by the algorithm (all initialized to be equal to the cost of the first found solution at the beginning). In the case reported in this example, the average cost of the solutions computed in the previous iterations of the algorithm was 316, therefore, the contribution added by the ant to the trail of each used assignment results from formula (4.8) to be -0.039. As the ant performed worse than average, its choices are discouraged.

4.4 Scatter Search

Scatter Search (SS) algorithms, later integrated with Path Relinking, were initially proposed as a matheuristic, specifically as a heuristic based on surrogate relaxations (see Sect. 1.2.4), unfortunately with little evidence of computational effectiveness in the seminal paper. Almost two decades elapsed before the first applications were published, implementing methods, which departed from the motivating idea. Later SS was revisited as a link between Tabu Search and Genetic Algorithm.

Anyway, SS is motivated by the idea of using a systematic process for solution recombinations in order to achieve the desired characteristics or to satisfy constraints. The working is based on the use of a set R, called *Reference Set*, of solutions, or representative points. The main loop iterates a sequence of operations, where recombination operators are first applied to the points in R to generate new solutions, each of which is then used as a seed to a local search procedure. The best solutions among those obtained by this process are inserted into the reference set at the beginning of the next iteration.

The pseudocode of the general structure of the SS algorithm is presented in Algorithm 27. The notion of "good" used in the code is meant to be broad, it includes, but it is not limited to, the objective value as a criterium for evaluating the merit of newly created points. In fact, the mechanisms within scatter search are purposefully not restricted to a single uniform design, allowing the exploration of different possibilities that may prove effective in a particular implementation.

Algorithm 27: Generic scatter search

```
1 function GenericScatterSearch(numpop);
  Input  : Control parameters numpop
  Output: A set of feasible solutions X
2 repeat
3     Generate an initial set of solutions R (Reference Set);
4     Generate new solutions by applying recombination operators to the set R;
5     Modify the solutions obtained at step 4 in order to obtain feasible solutions;
6     Add a subset of good solutions among those produced in step 3 to the set R;
7 until (termination condition);
8 $X = R$
```

A “*template*” for scatter search has been published, dictating the following five modules, which imply the need to define the corresponding fundamental components.

(i) *Diversification generation (initialization)*
Generates a set of candidate solutions starting from a seed solution.

(ii) *Solution improvement (local search)*
Improves the quality of input solution. In case it was accepted that the input solution could be infeasible, this procedure also acts as a repair operator.

(iii) *Subset generation (parent selection)*
Defines subsets of solutions from the reference set which will be later recombined.

(iv) *Solutions recombination*
Constructs new solutions from subsets of previously stored ones.

(v) *Reference set update (new population)*
Defines how current reference set solutions should be replaced by newly generated ones.

When these elements are defined, the general framework algorithm becomes the one outlined in the template.

Step 1. *Initialization*: initialize the Reference Set R by means of the diversification procedure.

Step 2. *Local Search*: use a local search procedure to improve the quality of the solutions generated at Step 1, obtaining feasible enhanced solutions that are added to the reference set.

Step 3. *Subset Generation*: generate a collection of subsets of the reference set.

Step 4. *Recombination*: for each subset of reference set solutions, generate one or more new solutions.

Step 5. *Local Search*: create new enhanced trial solutions by means of local search applied to each solution produced at Step 4.

Step 6. *Reference Set Update*: update the reference set by adding all or the best solutions found in Step 5 and possibly expunging the worst present ones.

Step 7. Repeat Steps 3 to 7 until a termination condition is met.

A possible specification of this framework, leading to an operational algorithm definition, could use the components described in the following.

Diversification Procedure This procedure is used to initialize the reference set, it should generate a sequence of solutions, in such a way that each new solution maximizes its distance from the set of already generated ones, according to some metric.

This is problem-dependent. For example, the mentioned *template* proposes a *sequential diversification* procedure for (0-1) binary vectors. It is assumed that a solution $\overline{\mathbf{x}}$ complements $\mathbf{x}$ over a set J if $\overline{x}_j = 1 - x_j$, $\forall j \in J$, and $\overline{x}_j = x_j$, $\forall j \notin J$.

Step 1. Arbitrarily define an initial solution $\mathbf{x}$ and its complement $\overline{\mathbf{x}}$.

Step 2. Let n be the number of variables. Partition the set $N = \{1, \dots, n\}$ into P' and P'', each of cardinality as close as possible to $n/2$. From vector $\mathbf{x}$ define two solutions $\mathbf{x}^{P'}$ and $\mathbf{x}^{P''}$ such that $\mathbf{x}^{P'}$ complements $\mathbf{x}$ over P' and $\mathbf{x}^{P''}$ complements $\mathbf{x}$ over P''.

Step 3. Repeat the following. Define each subset generated at the last partitioning step as *key subset*. If no key subset contains more than one element, the procedure terminates. Otherwise, partition each key subset $S \subset N$ into S' and S'', each of cardinality close to $|S|/2$. If there exists a key subset S containing a single element, then set $S' = S$ and $S'' = \emptyset$.

Step 4. Let N' be the union of all subsets S' and N'' the union of all subsets S''. Generate solutions $\mathbf{x}^{N'}$ and $\mathbf{x}^{N''}$ as in Step 2 and go to Step 3.

Local Search Procedure Local search procedures are greatly problem-dependent, see Chap. 1.

Subset Generation A reported proposal generates four different types of subsets, differing in the number and quality of the elements they contain.

Subset type 1:	subsets of two elements, one of which is the best solution in the reference set.
Subset type 2:	subsets of three elements, containing the two best solutions of the reference set.
Subset type 3:	subsets of four elements, containing the three best solutions of the reference set.
Subset type 4:	subsets consisting of the best i elements of the reference set, $i = 5, \dots, |R|$.

After having constructed the subsets, the new solutions are to be generated combining only elements of the same subset type.

Recombination Procedure For each subset S obtained by the subset generation procedure, new solutions are generated as recombinations of elements of S. This operator can be problem-dependent: it is possible, in the case of integer programming, to use a linear combination in order to generate new offspring, possibly obtaining points outside of the convex hull of the parent solution set.

Reference Set Update This procedure is called after each local search step. The reference set contains the best n_{max} solutions found, ordered by decreasing quality. The new local optimum is inserted if it is of sufficient quality and if it does not duplicate an already stored solution. A hashing function should be used to facilitate the ordering/retrieval operations.

4.4.1 Path Relinking

Path Relinking is a common extension of the recombination phase of scatter search, suggesting to define the offspring by generating paths between and beyond parent solutions in neighborhood space. This generic idea presupposes that a path between two solutions in a neighborhood space generally goes through solutions that share a significant subset of attributes with the parent solutions. Paths are in fact determined by specifying decision variables whose values get modified by the moves executed in the neighborhood space and, if the endpoint solutions are both of good quality, chances are that the attributes they share contribute to the quality of the solutions.

To define a path, one starts from an *initiating solution*, then the moves progressively introduce attributes according to a *guiding solution*, reducing the distance between attributes of the initiating and guiding solutions. The roles of the initiating and guiding solutions are interchangeable, as there is no preferred way to go through the path. The details on how the moves are generated are specific to the chosen problem representation, but we remark that, once identified a path, it is usually possible to extend it beyond the two originating solutions, thus possibly generating attributes that were not present in the initial population.

Figure 4.8 shows an example of what could be expected from Path Relinking. On the background of the contour plot of the quadratic objective function, the figure shows the would-be integer solutions, two of which are the initiating and the guiding solutions. The linear path among them is partitioned into three equal length parts and the cutpoints are brought to the nearest, feasible, integer solution. In this figure, an in-path solution improves over both endpoint solutions.

4.4.2 Matheuristic Scatter Search

As mentioned, SS itself was proposed as a heuristic for optimization based on surrogate constraints, but its effectiveness at that was not demonstrated. After a long hiatus, a second proposal suggested how to integrate cutting planes in this framework, using extreme points of the feasibility polytope as seed points for a rounding procedure that can yield feasible integer solutions.

The general scheme is the same as in Algorithm 27, but it is suggested to integrate cutting planes in this framework, using extreme points of the induced

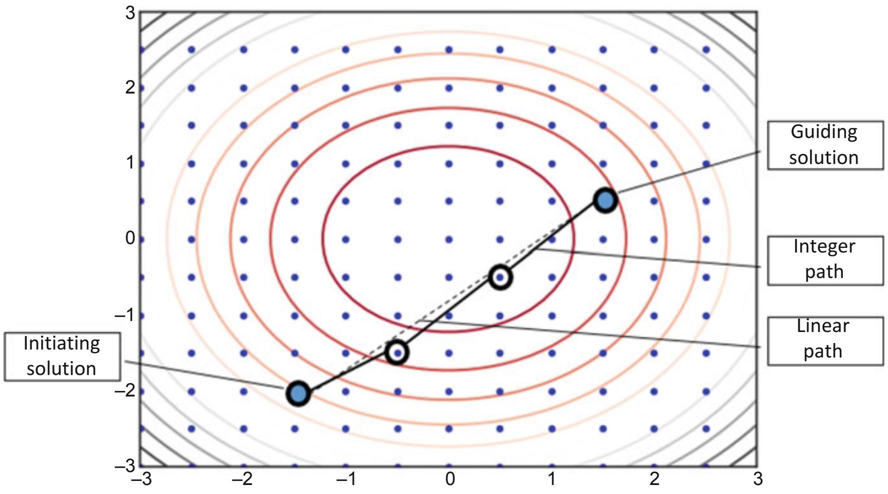


Fig. 4.8 Path Relinking, connecting two solutions and finding new ones

problem polytope as seed points for a rounding procedure that yields feasible integer solutions.

It was also pointed out that new points can also be obtained by non-convex combinations of elements of R. In this way, solutions containing information, which is not represented in the points of the current reference set, can be obtained.

We detail this general template for a generic integer program IP, defined on the variables ξ_i, $i = 1, \dots, n$. Let LP be the problem obtained from IP by relaxing the integrality constraints on variables ξ_i and let $\boldsymbol{\xi}(0)$ be an extreme point of the convex polytope defined by the LP constraints. Each extreme point adjacent to $\boldsymbol{\xi}(0)$ can be represented as $\boldsymbol{\xi}(h) = \boldsymbol{\xi}(0) + \theta_h \mathbf{d}_h$, for each $h \in NB$, that is $\{h : \boldsymbol{\xi}(h)$ is currently a nonbasic solution$\}$, where Δ_h represents the direction to follow and θ_h the step necessary to reach the new extreme point $\boldsymbol{\xi}(h)$. Each point $\boldsymbol{\xi}(h)$ obtained from $\boldsymbol{\xi}(0)$ is a new element that can be used as a seed to obtain feasible solutions for IP. In the general framework of the resulting algorithm, the same procedures introduced in Sect. 4.4 for Algorithm 27 should be specified.

Initial Generation
Can be made in any suitable way, for example by sequential diversification.

New Solutions
The solution space is explored by means of linear combinations of seed solutions, where seeds are the extreme points of the convex hull of the polytope associated with the LP-relaxation of the problem.

Local Search
While local search is strictly problem-dependent, a related activity of interest in the case of linear combination of integer solutions is the recovery of integer feasibility

Algorithm 28: Matheuristic scatter search

```
1  function ScatterSearch95();
   Input  : Control parameter $numpop$
   Output: A set of feasible solutions $X$
2  repeat
3      Generate an initial set of solutions $R$ with feasible solutions of problem IP ;
4      Solve the LP relxation of IP, let $\xi(0)$ be the optimal LP solution;
5      Set $H = \{\boldsymbol{\xi}(h) = \boldsymbol{\xi}(0) + \theta_h \mathbf{d}_h : h \in NB, \theta_h > 0\}$;  // Solution generation
6      Define the collection of sets $\Xi_1, \ldots, \Xi_k$ such that // Parent set definition
7          $\Xi_1, \ldots, \Xi_s, s < k$ contain only elements of $H$            // s can be 0
8          $\Xi_{s+1}, \ldots, \Xi_t, t < k$, contain only elements of $R$        // t can be 0
9          $\Xi_{s+t+1}, \ldots, \Xi_k$, contain both elements of $H$ and of $R$ ;
10     Construct the sets $Q(\Xi_j)$, $1 \leq j \leq k$, with points obtained from linear combination
         of points in $\Xi_j$, $1 \leq j \leq k$;                              // Recombination
11     Carry the solutions in $Q(\Xi_j)$, $1 \leq j \leq k$, to their local optimum. Update $R$ with the
         best solutions obtained;                                               // Local search
12     if no cutting plane can be identified then
13         STOP;
14     else
15         Let $t$ be a valid cutting plane for problem IP, add $t$ to LP and goto 4;
16     end
17 until (termination condition);
18 $X = R$;
```

from fractional solutions. A procedure proposed to this end is *directional rounding*, which is as follows. Let $X(0, 1)$ be a unit hypercube in the space $\mathbb{R}^n$, where $\mathbf{x} \in X(0, 1)$ stands for $0 \leq x_j \leq 1$, for all $0 \leq j \leq n$, and let $V(0, 1)$ denote the vertices of the hypercube, thus $\mathbf{x} \in V(0, 1)$ indicates $x_j \in \{0, 1\}$, for all $0 \leq j \leq n$. A directional rounding δ is a mapping from the continuous space $X(0, 1) \times X(0, 1)$ onto $V(0, 1)$ defined as follows:

$$\delta(\mathbf{x}^*, \mathbf{x}') = \left(\delta(x_1^*, x_1'), \ldots, \delta(x_n^*, x_n')\right)$$

with

$$\delta(x_j^*, x_j') = \begin{cases} 0 \text{ if } x_j' < x_j^* \\ 1 \text{ if } x_j' > x_j^* \\ 0 \text{ if } x_j' = x_j^* \text{ and } x_j' < 0.5 \\ 1 \text{ if } x_j' = x_j^* \text{ and } x_j^* \geq 0.5 \end{cases} \qquad 0 \leq x_j^*, x_j' \leq 1, x_j^*, x_j' \in \mathbb{R}^n$$

Point $\mathbf{x}^*$ is called *base point* and point $\mathbf{x}'$ *focal point*. The directional rounding between a vector $\mathbf{x}^*$ and the set $X \subset V(0, 1)$ is then defined to be

$$\delta(\mathbf{x}^*, X) = \{\delta(\mathbf{x}^*, \mathbf{x}') : \mathbf{x}' \in X\}$$

The obtained integer solutions may be still infeasible because of constraints other than the integrality ones. In this case, directional rounding is not enough and other problem-specific repair procedures are in order, such as those described in Sect. 3.7.1.

Reference Set Update
This procedure is the same as in Algorithm 27.

Cutting Planes
Each iteration of the scatter search main loop terminates with the separation of new valid inequalities. It is thus possible to add new constraints to the current LP formulation and to identify new extreme points of the redefined polytope which act as seed solutions for the subsequent iteration.

This generic IP procedure has been applied to the Covering Tour Problem, where it proved to be an effective method to get high quality heuristic solutions.

4.4.3 Scatter Search for the GAP, an Example

Scatter Search and Path Relinking have been applied quite successfully to solve the GAP. Unfortunately, the most effective SS algorithm used to optimize the GAP did not contain significant mathematical modules. Since SS itself is not a straight application of a founding idea, rather a collection of contributing elements, we propose here an example of a simplified version of the algorithm described in Alfandari et al. The algorithm implements a Path Relinking module, where pairs of parent solutions are used to identify an in-between offspring of possibly better quality: "*for each path, we select a single intermediate solution of the path to be a candidate for insertion in the population (...) Then, we improve the solution by a descent method*" (Alfandari et al. 2002). The SS algorithm for the GAP is proposed in Algorithm 29.

The set of solutions at step 2 can be generated by repeatedly applying Algorithm 1, where the ordering at its step 3 is changed at each call. It is understood that, being the GAP strongly NP-hard, there is no guarantee that the Reference Set can be constructed at all.

A very critical step is step 5, because selecting too similar solutions will lead to the reconstruction of one parent while selecting too different ones will lead to a high computational cost in the following step. Parameter *maxdiff* denotes the maximum number of assignments, which can differ in two parent solutions.

Consider the example of instance *example8x3* of Sect. 1.1. The initial generation, based on random generation of the seed ordering of Algorithm 1, can lead to the initial population presented in Table 4.4.

Table 4.4 Scatter search initial population

σ	Cost
0 1 1 2 2 0 0 2	327
1 0 1 0 2 2 2 0	329
0 1 1 0 2 2 2 0	330
0 1 2 1 0 2 0 2	328
1 0 2 0 1 2 0 2	328
1 0 1 2 0 2 2 0	326
0 1 1 2 0 2 2 0	327
1 1 0 0 2 2 2 0	328
0 0 1 1 2 2 2 0	332
0 0 2 1 1 2 0 2	331

Algorithm 29: Scatter search for the GAP

```
1  function ScatterSearchForGAP();
   Input  : Control parameters numpop,maxdiff
   Output: A set of feasible solutions $X$
2  Generate an initial set of diverse solutions $R$ (Reference Set) by means of a differently
     seeded constructive algorithm;
3  Bring each solution in $R$ to its local optimum;
4  repeat
5      Define a collection $\Xi$ of pairs of sufficiently different solutions selected among those
         of $R$;                                          // Parent set definition
6      Generate one new solution from each pair in $\Xi$ by fixing the common variables and
         optimizing the remaining subproblem;                     // Path Relinking
7      Carry each solution obtained at step 6 to its local optimum;      // Local search
8      Add a subset of good solutions among those produced in step 7 to the set $R$ to replace
         randomly selected ones;
9  until (termination condition);
10 X = $R$;
```

Running the algorithm, we generated all pairs of solutions of R, which differed in exactly *maxdiff* assignments. One such pair is made by the two last solutions of Table 4.4. The common assignments of these two solutions are (0 0 * 1 * 2 * *), where the asterisks denote a difference in the assignments. A fixing strategy can be applied, similar to a recombination operator already described in Sect. 4.2.4.

After fixing those four assignments and solving to integer optimality the remaining free subproblem, we get a solution that lies in-between the two parents, i.e., on a path connecting the two. The optimal subproblem solution is $\boldsymbol{\sigma} = (0, 0, 2, 1, 1, 2, 0, 2)$ of cost 330. It is not a globally optimal solution, but it shows that optimizing along the path can lead to better solutions than the parent ones. Clearly, upon introducing in the objective function a weighted distance measure from the endpoint solutions, it is possible to try generating more offspring along the path.

4.5 Related Literature

A strong criticism to the excessive use of metaphors in optimization has been published by Sörensen (2015).

Genetic Algorithms (GAs) were first introduced by Holland (1975), and gained greater salience after the publication of a relevant book by Goldberg (1989). Dynamic Fitness Scaling is presented for example in Colorni et al. (1997). Evolution Strategies have been presented in Rechenberg (1973), Schwefel (1981), Beyer and Schwefel (2002). The theoretical analysis of crossover is in Eremeev (2008), application for polishing is in Rothberg (2007). Mathematical programming crossover based on parent intersection is in Yagiura and Ibaraki (1996), based on parent union in Aggarwal et al. (1997). Applications of optimized crossover are in Borisovsky et al. (2009) and in Dolgui et al. (2010). Dynastically optimal recombination was proposed by Cotta and Troya (2003), the relay placement application in Flushing and Di Caro (2012) and that on transit route design in Walteros et al. (2014).

Ant System was proposed by Dorigo et al. Colorni et al. (1991), Dorigo et al. (1991), Dorigo et al. (1996), then generalized into ACO by Dorigo and Di Caro (1999) and Dorigo and Stützle (2004). Ant Colony System is due to Dorigo and Gambardella (1997), Max-Min Ant System to Stützle and Hoos (2000). The application to the TSP is in Reimann (2007), that on cooperative wireless networks in D'Andreagiovanni (2014). Beam-ACO was proposed by Blum (2005, 2008), and ANTS by Maniezzo (1999), Maniezzo and Milandri (2002).

Scatter Search was first introduced in Glover (1977), then revisited in Glover et al. (1996). The template for Scatter Search is in Glover (1997), a paper that contains most Scatter Search details reported in this section. Linear recombinations for generating offspring were suggested in Glover (1995), paving the way to Path Relinking (Glover et al. 2000). Directional rounding is again in Glover (1995). The Covering Tour Problem application is in Baldacci et al. (2005), that to the GAP in Yagiura et al. (2003), the included Path Relinking is taken from Alfandari et al. (2002).

References

Aggarwal C, Orlin J, Tai R (1997) An optimized crossover for the maximum independent set. Oper Res 45:226–234

Alfandari L, Plateau A, Tolla PA (2002) Two-phase path relinking algorithm for the generalized assignment problem. In: Proceedings of the fourth Metaheuristics international conference, pp 175–179

Baldacci R, Boschetti MA, Maniezzo V, Zamboni M (2005) Scatter search methods for the covering tour problem. In: Sharda R, Voß S, Rego C, Alidaee B (eds) Metaheuristic optimization via memory and evolution: Tabu search and scatter search. Springer, Boston, pp 59–91

Beyer HG, Schwefel HP (2002) Evolution strategies – a comprehensive introduction. Natural Comput 1(1):3–52

Blum C (2005) Beam-ACO – hybridizing ant colony optimization with beam search: an application to open shop scheduling. Comput Oper Res 32(6):1565–1591

Blum C (2008) Beam-ACO for simple assembly lime balancing. INFORMS J Comput 20(4):618–627

Borisovsky P, Dolgui A, Eremeev A (2009) Genetic algorithms for a supply management problem: MIP-recombination vs greedy decoder. Eur J Oper Res 195(3):770–779

Campelo F (2019) Evolutionary computation bestiary. https://github.com/fcampelo/EC-Bestiary

Colorni A, Dorigo M, Maniezzo V (1991) Distributed optimization by ant colonies. In: Varela F, Bourgine P (eds) Proceedings of the European conference on artificial life, ECAL'91, Paris. Elsevier Publishing, Amsterdam, pp 134–142

Colorni A, Dorigo M, Maniezzo V (1997) Metaheuristics for high-school timetabling. Comput Optim Appl 9(3):277–298

Cotta C, Troya J (2003) Embedding branch and bound within evolutionary algorithms. Appl Intell 18(2):137–153

D'Andreagiovanni FA (2014) Hybrid exact-ACO algorithm for the joint scheduling, power and cluster assignment in cooperative wireless networks. In: Di Caro G, Theraulaz G (eds) Bio-inspired models of network, information, and computing systems. Springer, Berlin, pp 3–17

Dolgui A, Eremeev A, Guschinskaya O (2010) MIP-based GRASP and genetic algorithm for balancing transfer lines. In: Matheuristics. Springer, Boston, pp 189–208

Dorigo M, Di Caro G (1999) The ant colony optimization meta-heuristic. In: Corne D, Dorigo M, Glover F (eds) New ideas in optimization. McGraw Hill, London, pp 11–32

Dorigo M, Gambardella L (1997) Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Trans Evol Comput 1(1):53–66

Dorigo M, Stützle T (2004) Ant colony optimization. MIT Press, Cambridge,

Dorigo M, Maniezzo V, Colorni A (1991) Positive feedback as a search strategy. Technical Report 91-016, Dipartimento di Elettronica, Politecnico di Milano

Dorigo M, Maniezzo V, Colorni A (1996) Ant system: optimization by a colony of cooperating agents. IEEE Trans Syst Man Cyb Part B (Cyb) 26(1):29–41

Eremeev AV (2008) On complexity of optimal recombination for binary representations of solutions. Evol Comput 16(1):127–147

Flushing EF, Di Caro GA (2012) Exploiting synergies between exact and heuristic methods in optimization: an application to the relay placement problem in wireless sensor networks. In: Di Caro G, Theraulaz G (eds) Bionetics 2012. Lecture Notes for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 134, pp 250–265

Glover F (1977) Heuristics for integer programming using surrogate constraints. Decis Sci 8:156–166

Glover F (1995) Scatter search and star paths: beyond the genetic metaphor. OR Spektrum 17:125–137

Glover F (1997) A template for scatter search and path relinking. Technical Report, School of Business, University of Colorado

Glover F, Kelly JP, Laguna M (1996) New advances and applications of combining simulation and optimization. In: Charnes J, Morrice DJ, Brunner DT, Swain JJ (eds) Proceedings of the 1996 winter simulation conference, pp 144–152

Glover F, Laguna M, Marti R (2000) Fundamentals of scatter search and path relinking. Control Cyb 29(3):653–684

Goldberg D (1989) Genetic algorithms in search, optimization and machine learning. Addison-Wesley Professional, Reading

Holland JH (1975) Adaptation in natural and artificial systems. MIT Press, Cambridge

Maniezzo V (1999) Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem. INFORMS J Comput 11(4):358–369

Maniezzo V, Milandri M (2002) An ant-based framework for very strongly constrained problems. In: Dorigo M, Di Caro G, Sampels M (eds) Proceedings of ANTS2002, from ant colonies to artificial ants: third international workshop on ant algorithms. Lecture Notes in Computer Science, vol. 2463. Springer, Berlin, pp 222–227

Rechenberg I (1973) Evolutionsstrategie. Holzmann-Froboog, Stuttgart

Reimann M (2007) Guiding ACO by problem relaxation: a case study on the symmetric TSP. In: Bartz-Beielstein T, et al (eds) Hybrid metaheuristics. HM 2007. Lecture Notes in Computer Science, vol 4771, pp 45–56

Rothberg E (2007) An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS J Comput 19(4):534–541

Schwefel HP (1981) Numerical optimization of computer models (Translation of Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie). Birkhäuser (1977). Wiley, Chichester

Sörensen K (2015) Metaheuristics - the metaphor exposed, international transactions in operational research. Special Issue: Math Model-Based Metaheuristics 22(1):3–18

Stützle T, Hoos HH (2000) Max-Min ant system. Future Gener Comput Syst 16:889–914

Turing A (1950) Computing machinery and intelligence. Mind, LIX 238:433–460

Walteros JL, Medaglia AL, Riano G (2014) Hybrid algorithm for route design on bus rapid transit systems. Transport Sci 49(1):66–84

Yagiura M, Ibaraki T (1996) The use of dynamic programming in genetic algorithms for permutation problems. Eur J Operat Res 92:387–401

Yagiura M, Ibaraki T, Glover FA (2003) Path relinking approach for the generalized assignment problem. Eur J Operat Res 169(2):548–569

Part III
Original Matheuristics

Chapter 5
Diving Heuristics

5.1 Introduction

Diving heuristics are methods that progressively include elements into a partial solution up to its possible completion, thus "jump" into a solution with no way back. While this is common to all constructive heuristics, diving ones are characterized by working on the mathematical formulation of the problem to solve. Some contributions of this type showed remarkably effective, and are included as standard components of general MIP solvers. This chapter reviews two well-known diving heuristics, namely Relaxation Induced Neighborhood Search (RINS) and Local Branching, and outlines a possible specification to the Generalized Assignment Problem.

Both methods can be used as heuristics to improve the incumbent feasible partial solution of a branch and bound node, or as standalone heuristics. We introduce them as general MIP problem of the form

$$z_{\text{MIP}} = \min \sum_{j \in J} c_j x_j \tag{5.1}$$

$$\text{s.t.} \sum_{j \in J} a_{ij} x_j \leq b_i, \qquad i \in I \tag{5.2}$$

$$x_j \in \{0, 1\}, \qquad j \in B, B \neq \emptyset \tag{5.3}$$

$$x_j \geq 0, \text{ integer}, \qquad j \in G \tag{5.4}$$

where $J = B \cup G$ is the index set of the decision variables, partitioned into B, the index set of binary decision variables, and G, possibly empty index set of general-integer nonnegative variables.

V. Maniezzo et al., *Matheuristics*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-70277-9_5

During search, at intermediate nodes of a branch tree, two solutions are maintained: the *incumbent* feasible solution (if one was found) $\mathbf{x}^h$ and the node *bound* solution $\tilde{\mathbf{x}}$. The incumbent solution is a feasible heuristic solution for the MIP, but it is not guaranteed to be optimal, while the bound solution is commonly the solution of the continuous relaxation of the MIP. Typically, the two solutions will have some variables that take the same value in both solutions, while other variables differ. RINS is a technique to bring the two solutions to agree on all variables, while Local Branching is a local search heuristic exploring a neighborhood of $\mathbf{x}^h$.

5.2 Relaxation Induced Neighborhood Search

RINS relies on the assumption that the variables instantiation in a good or optimal solution shares much with a good lower bound solution, therefore selecting an appropriate set of values for the bound variables and appropriately completing that variable-value assignment appears to be a promising path for a heuristic design. To this end, two issues are in order: which bound variables to fix and which values are to be assigned to the remaining variables.

The suggestion is to fix all variables that have common values in the bound and in the incumbent solutions, and to let the solver try to solve optimally the remaining MIP problem, the so-called *sub-MIP*, within a given node limit or with an objective cutoff.

The RINS algorithm, therefore, suggests to perform at each chosen node of the global branch-and-cut tree, or each time a partial solution needs to be completed, the following steps:

1. fix the variables that have the same values in the incumbent and in the current bound solutions;
2. set an objective cutoff based on the objective value of the current incumbent;
3. solve a *sub-MIP* on the remaining variables.

The algorithm is typically used in different nodes of a branch-and-cut (or branch and bound) search, thus the sub-MIP can take advantage of the improved formulation deriving from the added cuts. However, since search is not limited to the current subtree, the improved bounds deriving from branching may not be respected. In any case, sub-MIP search incorporates information provided both by the incumbent and by the bound solutions, which are complementary in nature, and thus it has the possibility to guide each of the two out of its narrowness thanks to the information provided by the other one.

When applied within a branch-and-cut procedure, one further advantage of RINS is that the continuous relaxation changes at every node of the tree, therefore a diversification of the starting points is structurally achieved. However, since RINS could be computationally demanding and since bounds of related nodes typically do not change by much, it is convenient to apply this procedure only periodically, after a given number of new nodes are explored.

The issue of the complexity of the heuristic procedure is faced by setting a limit on the computational resources available for optimization. The sub-MIP can in fact be quite large if too few variables were fixed, so its solution could take a time comparable with that of the original problem. A workaround is setting a limit to the number of nodes that can be expanded during the search, consequently bounding the time available for search. If a solution is found, it may become the new incumbent feasible solution, otherwise, nothing happens.

RINS fits the general matheuristics definition of mathematically based heuristics, and it is in fact closely related to other approaches described in this book. It is basically a possible implementation of the Very Large Neighborhood Search (see Chap. 6) idea, where the neighborhood is defined on the basis of the continuous relaxation of the problem and is therefore problem-independent.

5.2.1 RINS for the GAP

The application of RINS to the Generalized Assignment Problem does not require any particular adaptation of the general framework exposed in Sect. 5.2. Given a seed heuristic solution $\mathbf{x}^h$, it is necessary to compute the continuous relaxation $\tilde{\mathbf{x}}$, check for same-valued variables, and solve the sub-MIP. The pseudocode is presented as Algorithm 30.

Algorithm 30: Relaxation Induced Neighborhood Search

```
1  function RINS ($\mathbf{x}^h$,$nl$);
   Input  : A seed feasible solution $\mathbf{x}^h$ and a node limit $nl$
   Output: A feasible solution or indication of failure
2  Compute the continuous relaxation $\tilde{\mathbf{x}}$;
3  Check for variables having the value in $\mathbf{x}^h$ and $\tilde{\mathbf{x}}$;
4  Identify sub-MIP;
5  Solve sub-MIP under node limit $nl$;
6  if (found sub-MIP feasible solution $\mathbf{x}^s$) then
7      return $\mathbf{x}^s$;
8  else
9      return fail;
10 end
```

When we apply this algorithm to the example introduced in Chap. 1, Fig. 1.1, we obtain the following results.

Assume to start with the feasible solution $\sigma = \{1, 1, 1, 2, 2, 3, 3, 3\}$ of cost 343 provided by the simple constructive heuristic presented in Sect. 1.3.1 when assigning each client to its least cost available server. In binary variables, it is the solution in Eq. (5.5).

$$\mathbf{x}^h = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{5.5}$$

Solving the linear relaxation, we get a lower bound of value 224.174 given by the solution $\tilde{\mathbf{x}}$ in Eq. (5.6), where values are rounded to the third decimal.

$$\tilde{\mathbf{x}} = \begin{bmatrix} 0.000 & 0.000 & 0.000 & 0.000 & 0.773 & 1.000 & 1.000 & 1.000 \\ 0.000 & 0.305 & 1.000 & 1.000 & 0.227 & 0.000 & 0.000 & 0.000 \\ 1.000 & 0.695 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \end{bmatrix} \tag{5.6}$$

The two solutions share only one assignment, $x_{1,3} = 1$, and therefore this variable that gets fixed, obviously along with $x_{0,3} = x_{2,3} = 0$, and leaving up to the sub-MIP to fix all other variables.[1]

In this case, a solver has no difficulties in finding a feasible, optimal solution for the subproblem with fixed variables of cost 327 (not an optimal one for the whole problem), which is the one presented in Eq. (5.7).

$$\mathbf{x} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{5.7}$$

Notice that if we allowed to fix all variables having the same values in the bound (5.6) and in the initial heuristic solution (5.5), i.e., both 0's and 1's, we would have fixed $x_{1,0} = 0$, $x_{2,2} = 0$, $x_{0,3} = 0$, $x_{1,3} = 1$, $x_{2,3} = 0$, $x_{2,4} = 0$, $x_{1,5} = 0$, $x_{1,6} = 0$, $x_{1,7} = 0$, which leads to a solution of cost 329, worse than the one above.

5.3 Local Branching

Local Branching is another approach whose characteristic element can be framed as diving. Actually, it was proposed as an exact exploration strategy, albeit designed to improve the heuristic behavior of a MIP solver by means of an explicit exploration of the neighborhood of a feasible solution.

Local Branching is similar to RINS, in that it starts with an incumbent feasible solution $\mathbf{x}^h$ and defines a neighborhood specifying which variables to fix in further

[1]We remind that in the computational traces the decision variables are indexed from 0.

exploration, but it does so directly addressing a central issue, which may severely limit RINS effectiveness: how many variables are to be fixed.

When diving, in fact, fixing too few variables leads to a sub-MIP almost as complex as the original one, while fixing too many limits search to a narrow neighborhood of the current heuristic solution. Local Branching faces this trade-off by explicitly dictating the number of variables to fix at each iteration.

The basic idea is as follows. Given the two feasible solution vectors $\mathbf{x}^h$ and $\mathbf{x}$ for problem MIP (see Sect. 5.1), where the variable subset only contains binary variables, i.e., $J = B$ and $G = \emptyset$, we can compute their (Hamming) distance as $\Delta(\mathbf{x}^h, \mathbf{x}) = |\{j \in B : |x_j^h - x_j| = 1\}|$. Furthermore, let $S^h = \{j \in B : x_j^h = 1\}$ denote the *binary support* of $\mathbf{x}^h$, i.e., the subset of binary variables which take the value of 1 in the reference solution.

Local Branching defines a limited neighborhood of $\mathbf{x}^h$, to be explored by the sub-MIP, consisting only of the solutions satisfying the additional constraint $\Delta(\mathbf{x}^h, \mathbf{x}) \leq k$ using a parametric radius parameter k. If any improving solution is present in the neighborhood, the sub-MIP will spot it and let it become the new incumbent heuristic solution. Analogously to RINS, the local branching sub-MIP includes all cutting planes and variable bounds deriving from valid inequalities found during the exploration of the global branch-and-cut tree and ignores variable bounds imposed by branching and valid only on a subtree.

The request of exploring only the desired neighborhood can be enforced by adding to the formulation so-called *local branching constraints*, as follows. Given parameter k, the k-opt neighborhood $N(\mathbf{x}^h, k)$ of $\mathbf{x}^h$ is defined as the set of feasible solutions of the original MIP satisfying the additional local branching constraint:

$$\Delta(\mathbf{x}^h, \mathbf{x}) = \sum_{j \in S^h} (1 - x_j) + \sum_{j \in B \setminus S^h} x_j \leq k \tag{5.8}$$

where the two sums count the number of variables changing their value from 1 to 0 or from 0 to 1, with respect to $\mathbf{x}^h$.

Local Branching can be trivially applied also in case of models including non-binary values in the vectors $\mathbf{x}$ and $\mathbf{x}^h$, letting these make no contribution to the metric Δ used to define the neighborhood. This is effective when the MIP model contains a relevant set of binary variables. If this is not the case, it is still possible to modify the local branching constraints including the general-integer variables in the definition of the distance function.

As mentioned, Local Branching was proposed as a branching criterion within a full tree-search scheme for problem MIP, hence its name, based on the local branching constraint. The idea is simple, given a node corresponding to solution $\mathbf{x}^h$, further search can proceed partitioning the search space between solutions for which $\Delta(\mathbf{x}, \mathbf{x}^h) \leq k$ or $\Delta(\mathbf{x}, \mathbf{x}^h) \geq k + 1$. This possibility falls however outside of the scope of the current chapter.

5.3.1 Local Branching for the GAP

The pseudocode of a possible algorithm implementing local branching for the GAP is presented in Algorithm 31. When we apply this algorithm to the *example8x3* instance introduced in Chap. 1, we obtain the following results.

Algorithm 31: Local Branching

```
1  function LocalBranching ($\mathbf{x}^h$,$k$);
   Input  : A seed feasible solution $\mathbf{x}^h$ and a neighborhood radius limit $k$
   Output: A feasible solution or indication of failure
2  Add local branching cut (5.8) to MIP;
3  let x= solve(MIP);
4  if (x is feasible and $\mathbf{x} \neq \mathbf{x}^h$) then
5      if (cost(x) < cost($\mathbf{x}^h$)) then
6          $\mathbf{x}^h$ = x;
7          go to 2;
8      else
9          return $\mathbf{x}^h$;
10     end
11 else
12     return fail;
13 end
```

Assume, as we did in Sect. 5.2.1, to start with the feasible solution $\boldsymbol{\sigma} = (0, 0, 0, 1, 1, 2, 2, 2)$ of cost 343 provided by the simple constructive heuristic presented in Sect. 1.3.1, which in binary variables is represented in Eq. (5.9).

$$\mathbf{x}^h = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{5.9}$$

Local Branching dictates to first compute the local branching constraint corresponding to this solution, which in case we used $k = 4$, turns out to be the inequality of Eq. (5.10).

$$\begin{array}{lllllllll} -x_{11} & -x_{12} & -x_{13} & +x_{14} & +x_{15} & +x_{16} & +x_{17} & +x_{18} & \\ +x_{11} & +x_{12} & +x_{13} & -x_{14} & -x_{15} & +x_{16} & +x_{17} & +x_{18} & \\ +x_{11} & +x_{12} & +x_{13} & +x_{14} & +x_{15} & -x_{16} & -x_{17} & -x_{18} & \leq (4-8) \end{array} \tag{5.10}$$

This cut is then included in the formulation. Upon calling a solver, an improving solution is found, which is $\boldsymbol{\sigma} = (0, 0, 2, 1, 1, 2, 0, 2)$ of cost 331. The binary representation is in Eq. (5.11).

$$\mathbf{x}^h = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{5.11}$$

Notice how the new solution differs from the previous only in two client assignments, clients 3 and 7, therefore only the four variables $x_{0,2}$, $x_{2,2}$, $x_{0,6}$, $x_{2,6}$ were changed, according to the maximum radius parameter.

Now, this solution becomes the new reference heuristic solution and a local branching constraint is computed for it. Calling the solver, a new improving solution is found, which is $\boldsymbol{\sigma} = (1, 0, 2, 1, 0, 2, 0, 2)$ of cost 327. The binary representation is in Eq. (5.12).

$$\mathbf{x}^h = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{5.12}$$

Again, this new solution differs from the previous only in two client assignments, clients 1 and 5, therefore only the four variables $x_{0,0}$, $x_{1,0}$, $x_{0,4}$, $x_{1,4}$ were changed, according to the maximum radius parameter.

This solution now becomes the new reference heuristic solution, a local branching constraint is computed for it, and the solver can find a further improving solution (actually an optimal one), which is $\boldsymbol{\sigma} = (1, 1, 2, 0, 0, 2, 0, 2)$ of cost 325. The binary representation is in Eq. (5.13).

$$\mathbf{x}^h = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{5.13}$$

Also this new solution differs from the previous one only in two client assignments, clients 2 and 4, therefore only the four variables $x_{1,0}$, $x_{1,1}$, $x_{3,0}$, $x_{3,1}$ were changed. A new local branching constraint is computed, but no improvement is possible, therefore the local search ends here.

5.4 Feasibility Pump

The *Feasibility Pump* algorithm (FP) was proposed as a general means to get feasible solutions to initialize MIP search, and it is included as such in several general purpose MIP solvers. In the context of this book, FP is of interest both because of its ability to provide an initial feasible solution, and of its ability to bring to integer feasibility a generic, non-integer but LP feasible solution.

FP is based on the rationale that a solution is integer feasible if it is coincident with its rounding. Formally, if P is the LP polytope of the problem under study, an integer feasible solution $\mathbf{x}^*$ corresponds to a point in P such that $\mathbf{x}^* = \tilde{\mathbf{x}}^*$, where $\tilde{\mathbf{x}}^*$ denotes the rounding of $\mathbf{x}^*$, i.e., the scalar rounding of each of its constituting variables to the corresponding nearest integer.

The minimization of a distance function defined as $\Delta(\mathbf{x}^*, \tilde{\mathbf{x}})$ permits therefore to instantiate a search procedure for finding an integer feasible solution, starting from a generic LP feasible one, $\tilde{\mathbf{x}}$.

The distance function $\Delta(\mathbf{x}, \tilde{\mathbf{x}})$ can be defined for any integer variable, but in case of the GAP binary variables it can be defined as

$$\Delta(\mathbf{x}, \tilde{\mathbf{x}}) = \sum_{(i,j)\in I\times J:\tilde{x}_{ij}=0} x_{ij} + \sum_{(i,j)\in I\times J:\tilde{x}_{ij}=1} (1 - x_{ij})$$

The search of an integer feasible solution $\mathbf{x}^*$ can therefore be made by solving the linear problem $min\{\Delta(\mathbf{x}, \tilde{\mathbf{x}}) : \mathbf{x} \in P\}$. The corresponding algorithm is presented as Algorithm 32.

Algorithm 32: Feasibility Pump for the GAP

```
1  function FeasibilityPump;
   Input  : $\mathbf{x}^*$ an LP feasible solution, $maxIter$ iteration limit, $T$ perturbation parameter
   Output: A feasible solution $\bar{\mathbf{x}}$ or an indication of failure
2  Set $\tilde{\mathbf{x}} = \mathbf{x}^*$;                          // integer rounding of $\mathbf{x}^*$
3  Set $nIter = 0$;
4  while $(\Delta(\mathbf{x}^*, \tilde{\mathbf{x}}) > 0$ and $nIter < maxIter)$ do
5      $nIter = nIter + 1$;
6      $\mathbf{x}^* \in \{\mathbf{x} \in P$ such that $\Delta(\mathbf{x}, \tilde{\mathbf{x}})$ is minimzed$\}$;   // P is the GAP polytope
7      if $\Delta(\mathbf{x}^*, \tilde{\mathbf{x}}) > 0$ then
8          if $(round(x^*_{ij}) \neq \tilde{x}_{ij}$ for at least one $(i, j) \in I \times J)$ then
9              Update $\tilde{\mathbf{x}} = round(\mathbf{x}^*)$;
10         else
11             for each $(i, j) \in I \times J$ define the score $\sigma_{ij} = |x^*_{ij} - \tilde{x}_{ij}|$;
12             flip the $\tau = rand(T/2, 3T/2)$ components $\tilde{x}_{ij}$ with largest $\sigma_{ij}$;
13             if cycling is detected, restart from another LP feasible solution;
14         end
15     end
16 end
```

The steps 11–13 are a heuristic element, which can be useful for escaping from the local minima of the greedy approach implemented by the procedure. Different perturbations are possible, for example, a random selection of the variables to flip.

An application of FP on instance *example8x3* could start from the LP solution of the linear relaxation presented in Sect. 1.2.1.

$$\mathbf{x}^* = \begin{bmatrix} 0 & 0 & 0 & 0 & 0.773 & 1 & 1 & 1 \\ 0 & 0.305 & 1 & 1 & 0.227 & 0 & 0 & 0 \\ 1 & 0.695 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Using the scores computed at step 11 we incur into cycling, while a random choice of the variables to flip permits to obtain, after 150 iterations, the following feasible solution of cost 334:

$$\mathbf{x}^* = \tilde{\mathbf{x}} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

A relevant feature of FP, as mentioned, is that it is not necessary to start from the LP bound, but any LP feasible basis can do. It is therefore possible to use FP to recover feasibility for an infeasible input solution. This needs to go through a two step process, where at the first step—if needed—the infeasible solution is modified so as to become a feasible basis for the GAP polytope, and at the second step the standard FP is used. This is the procedure that was already mentioned in Sect. 3.7.1.

5.5 Related Literature

Diving heuristics were named as such in Bixby et al. (2000). RINS was proposed in Danna et al. (2005), Local Branching in Fischetti and Lodi (2003). The application to the GAP draws also from Pigatti et al. (2005). The Feasibility Pump was presented in Fischetti et al. (2005).

References

Bixby RE, Fenelon M, Gu Z, Rothberg E, Wunderling R (2000) MIP: Theory and practice – closing the gap. Kluwer Academic Publishers, pp 19–49

Danna E, Rothberg E, Pape C (2005) Exploring relaxation induced neighborhoods to improve MIP solutions. Math Program 102(1):71–90

Fischetti M, Lodi A (2003) Local branching. Math Program B 98(1–3):23–47

Fischetti M, Glover F, Lodi A (2005) The feasibility pump. Math Program 104(1):91–104

Pigatti A, Poggi de Aragão M, Uchoa E (2005) Stabilized branch-and-cut-and-price for the generalized assignment problem. Electron Notes Discrete Math 19:389–395

Chapter 6
Very Large-Scale Neighborhood Search

6.1 Introduction

Very Large-Scale Neighborhood Search, VLSN (also, at times, Very Large Neighborhood Search, VLNS) is not an algorithm or a class of algorithms, as much abstract and general as these could get, but rather a conceptual framework, which can be followed for solving combinatorial optimization problems. The approach "concentrates on neighborhood search algorithms where the size of the neighborhood is 'very large' with respect to the size of the input data", as stated by Ahuja et al. (2002). Typically, the searched neighborhoods are exponentially large, but the term has come to be used more generally to describe algorithms working on neighborhoods that are too large to exhaustively search in practice.

Given the loose definition of the topic, almost any algorithm presented in this book could be included in the VLSN class, along with many other ones, which are not based on any mathematical models. To help making the class less indistinct, thus more interesting, a categorization into three classes was proposed in the literature for methods to be classified as VLSN:

1. *variable-depth methods*, which focus on exponentially large neighborhoods and search only partially these neighborhoods using heuristics.
2. *network-flow-based improvement algorithms*, neighborhood search methods which use network flow techniques to identify improving neighbors.
3. neighborhoods for NP-hard problems induced by *subclasses or restrictions of the original problem* that are solvable in polynomial time.

This categorization is much more normative and, not without exceptions, is usually accepted in the literature to discriminate what is properly VLSN and what is, more in general, a local search over a very large, possibly exponential, neighborhood. Clearly, algorithms following these guidelines have existed for decades before the introduction of the VLNS class, which is true also for matheuristics in general, but the classification helps to point out the specific contributions.

While we refer to the relevant surveys listed at the end of the section for a thorough introduction to VLSN, we report here some contributions which can be of particular interest when designing a model-grounded heuristic for solving hard combinatorial optimization problems. This can be done when it is possible to identify the best neighbor of the incumbent solution of a core local search as a combinatorial problem itself, which can be solved efficiently, thus permitting an implicit full exploration of exponential neighborhoods.

VLNS can thus be seen as a paradigm for designing local search procedures, in which mathematical programming techniques are used to define and explore neighborhoods. Local search algorithms produce better solutions when they are allowed to explore large neighborhoods, but the exploration of the whole of large neighborhoods can be very time consuming and, hence, the time to get a local optimum can be very long. The following approaches permit to leverage mathematical programming results to achieve polynomial time explorations of exponential neighborhoods.

6.2 Variable-Depth Methods

Variable-Depth Methods (VDM) can be seen as a specialization of Variable Neighborhood Descent (VND) algorithms, thus ultimately of Iterated Local Search (see Sects. 3.4 and 3.5), where the size of the neighborhood is extended each time a local optimum is found in the incumbent neighborhood. Since neighborhoods are usually ordered by increasing complexity also in VND, the two approaches are largely overlapping.

The paradigmatic VDM is the Lin-Kernighan (LK) heuristic for the TSP (Lin and Kernighan 1973). The LK heuristic is based on progressively more complex combinations of simple local search moves. Essentially, it iterates over pairs of moves, where the first move of the pair inserts an edge in an incumbent Hamiltonian path H_1, thus closing a (sub)tour, and the second move removes the only edge different from the just inserted one that permits the reconstruction of a Hamiltonian path H_2, which will differ from H_1 for exactly 2 edges.

Figure 6.1 illustrates this basic proceeding. Starting from a tour (*a*), an edge is removed, obtaining a Hamiltonian path (*b*). Then two pairs of coupled moves are presented. First, an edge is inserted (*c*) obtaining a so-called δ-path or *stem and cycle*, which also determines the edge to be later removed to obtain a new Hamiltonian path (*d*). The second pair of coupled moves add an edge (*e*) obtaining a new δ-path and obtains another Hamiltonian path (*f*), from which a new tour is easily determined (*g*). The final tour belongs to a *3-opt* neighborhood of the initial one, where three assignment variables got substituted by another triplet, even though only *2-opt* moves were performed.

Repeating this basic block for k times permits to reach paths, which differ for $2k$ edges from an initial one (assuming cycling is avoided). A progressive increase of k corresponds to a VDM for the TSP. The LK heuristic comprises several other

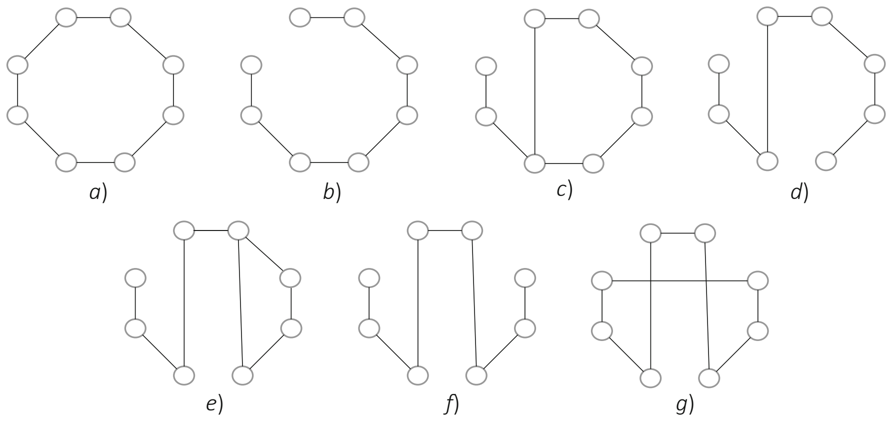


Fig. 6.1 Lin-Kernighan core moves

elements to make it as effective as it is, like backtracking or look-ahead, but this progressive enlargement of the size of the subset of edges to be removed makes it a typical example of VDM, so far not closely related to the methods of interest for this book.

Another reference, which is often made when introducing VDM, is the ejection chain algorithms (see Sect. 3.7). As we pointed out, ejection chains are another class of algorithms which was defined loosely, simply stating that they alternate steps where sets of variables change their value with steps where other variables need to be reassigned in order to recover feasibility or to try to further improve the solution quality.

This general characterization could also include the LK heuristic outlined above, along with many other local search methods presented in the literature. We refer the reader to Sect. 3.7 for a discussion of matheuristic algorithms following an ejection chain approach.

There has been a VDM specific for the GAP, where two different neighborhoods, called *shift* and *swap*, were used alternately. The shift neighborhood was defined to be the set of solutions obtainable by changing the assignment of one client, while the swap neighborhood was defined to be the set of solutions that could be obtained by exchanging the assignments of two clients. One further characteristic of this VDM is that search was allowed to proceed also in the infeasible region.

6.3 Network-Flow-Based Improvement Algorithms

Network flows are a class of problems which can be solved very efficiently by mathematical programming techniques. It is therefore promising to try to use a network flow problem as a model for the subproblem to solve at each iteration of the local search, thereby obtaining the desired polynomial time exploration of an exponential neighborhood.

Ahuja et al. (2002) classify the neighborhoods that can be explored by VLSN algorithms by means of network flow procedures into three subclasses:

1. neighborhoods defined by cycles,
2. neighborhoods defined by paths or dynamic programming,
3. neighborhoods defined by assignments and matching.

6.3.1 Neighborhoods Defined by Cycles

Cyclic exchange neighborhoods are best defined on partitioning problems, where a set of elements has to be partitioned into suitable subsets. This is, for example, the case of the GAP, which requires that the (index) set J of the clients be partitioned into m mutually disjoint subsets, one for each server in I. For example, the application of Algorithm 1 to instance *example8x3* induces a partitioning of the set $J = \{1, 2, 3, 4, 5, 6, 7, 8\}$ into the subsets $\{1, 2, 3\}$, $\{4, 5\}$ and $\{6, 7, 8\}$.

In this context, a *2-exchange* move asks to select two elements belonging to two different subsets and to swap the subsets they belong to. A *cyclic exchange* move generalizes the *2-exchange* to more than two involved subsets.

Let $J_1, J_2, \ldots, J_m$ be any feasible partition of a given set J. A cyclic-neighbor of $J_1, J_2, \ldots, J_m$ is obtained by transferring single elements among a sequence $J_{i_1}, J_{i_2}, \ldots, J_{i_k}$ of $k \leq m$ different subsets in J, in such a way that the last subset of the sequence is identical to the first, $J_{i_1} = J_{i_k}$, or $i_1 = i_k$. A 2-exchange is thus a cyclic exchange with $k = 2$.

Figure 6.2 depicts a cyclic exchange on the simple construction solution, for instance, *example8x3*, of cost 343, leading to an improved feasible solution of cost 332.

It has been shown that an improving cyclic exchange corresponds to a negative cost subset-disjoint cycle in a suitable *improvement graph*, which is defined on the

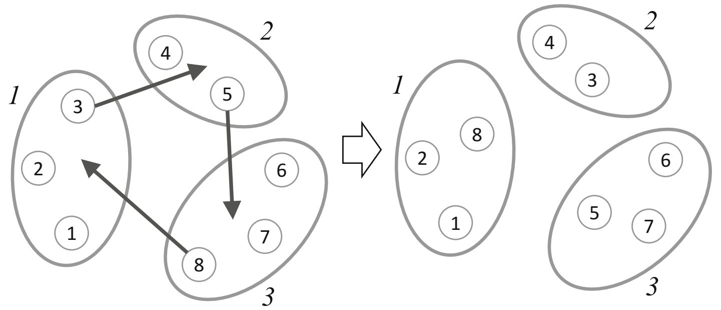
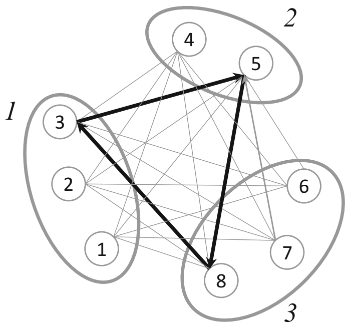


Fig. 6.2 A cyclic exchange for a solution of GAP instance example8x3



	1	2	3	4	5	6	7	8
1				-6	-8	-20	-24	-28
2				-4	-6	-16	-20	-24
3				-2	-4	-12	-16	-20
4	3	2	1			-8	-12	-16
5	4	3	2			-4	-8	-12
6	5	4	3	4	2			
7	6	5	4	6	4			
8	7	6	5	8	6			

Fig. 6.3 Improving graph, for example, 8x3

same vertex set V as the original partitioning problem instance, and which has an arc between any two vertices i and j, $i, j \in V$, if they belong to different partitions in the incumbent solution S. Let S^j denote the subset of V containing vertex j in S. The weight w_{ij} of arc (i, j) is given by the cost of the subset made up of all vertices of S^j, except that j is substituted by i, i.e., $(S^j \setminus \{j\}) \cup \{i\}$, minus the cost of S^j.

Finding a negative cost subset-disjoint cycle in such an improvement graph is NP-hard, but effective heuristics for searching the graph have been developed. Figure 6.3 shows the improvement graph for the constructive solution of instance *example8x3*, the corresponding arc weights, and the negative cost cycle corresponding to the cyclic exchange of Fig. 6.2. The weights of the cycle arcs are highlighted, and their sum is equal to the cost improvement granted by the cyclic exchange on the solution.

Different problems have been solved using cyclic exchange neighborhoods, including the vehicle routing problem, the minimum makespan machine scheduling problem, the capacitated minimum spanning tree problem, the graph coloring problem, the single source capacitated facility location, and timetabling problems.

6.3.2 Neighborhoods Defined by Paths

This class of neighborhoods consists of solutions that can be obtained by sequences, not necessarily cyclic sequences, of simple moves. They are a generalization of the swap neighborhood, obtained by aggregating an arbitrary number of so-called independent swap operations. This approach has actually given rise to two classes of matheuristics in their own right, *ejection chains* and *dynasearch*.

We already discussed ejection chains in Sect. 3.7, we refer to that section for a discussion of the topic. We point out here that the same approach advocated by ejection chains has also been proposed with the (confusing) name of *Large Neighborhood Search* in the VLNS literature and later generalized by allowing multiple destroy and repair operators, to obtain the so-called *Adaptive Large Neighborhood Search*, ALNS.

Dynasearch is an advanced local search method for improving a seed solution by building complex neighborhoods from combinations of simple search steps. The basic steps, in order to be viable for dynasearch, need to be *mutually independent*, meaning that they need not have any type of combined effect on the cost function and on the feasibility of candidate solutions. In this way, the overall effect of complex search steps can be computed as the sum of the effects of the single steps.

In this context, dynasearch proposes to find optimal combinations of mutually independent simple search steps by means of dynamic programming. The general dynasearch recursion is usually presented for permutation problems, where the independence of the search steps is guaranteed by the constraint that the moves operate on disjunct subsets of the permutation indices. The recursion equation is often presented as follows.

Let $\Pi = (\pi(j), j = 1, \ldots, n)$ be the permutation corresponding to the current solution, and let $\Delta(j)$ be the maximum total cost reduction obtained by independent moves involving only elements from position 1 to position j. Let furthermore $\delta(i, j)$ be the cost reduction resulting from moves involving positions between i and j, included.

The recursion equation is defined on the observation that the maximum cost reduction, considering moves modifying the solution up to position j, can be obtained by selecting the maximum value computed either by keeping the assignment at position j or by changing it with a combination of the best assignment up to a position $i < j$ with the best move sequence from position i up to position j, for each $j \in J$.

The recursion is initialized setting $\Delta(0) = \Delta(1) = 0$ and permits to compute $\Delta(j)$, $j = 2, \ldots, n$, in a forward evaluation using the recursive formula

$$\Delta(j) = \max\{\max_{1 \le i \le j-1} \{\Delta(i-1) + \delta(i, j)\}, \Delta(j-1)\} \tag{6.1}$$

The largest reduction in solution cost is given by $\Delta(n)$.

Dynasearch has been applied to different optimization problems, mostly problems that involve search in a space of permutations. These include the earliness-tardiness scheduling problem, the traveling salesman problem, the single machine total weighted tardiness problem, the linear ordering problem, and the dynamic berth allocation of container ships. A similar approach has been proposed by Ergun and others for vehicle routing problems. A sequence of cyclic and path exchanges over suitable improvement graphs were used for the generalized knapsack problems.

If we tried to apply dynasearch to the GAP, thus to a problem that is not defined on a search space of permutations, we should first make sure that the simple moves are independent of one another. Considering, for example, *1-opt* moves as the basic moves, i.e., reassignments of single clients, we have moves that are independent among themselves for the cost function, but not for feasibility (a move may be infeasible after a specific move but feasible after another). Feasibility therefore should not be checked on intermediate solutions, but only at the end of the recursion. Equation (6.1) must be adapted to the GAP state variables, which in this case are the

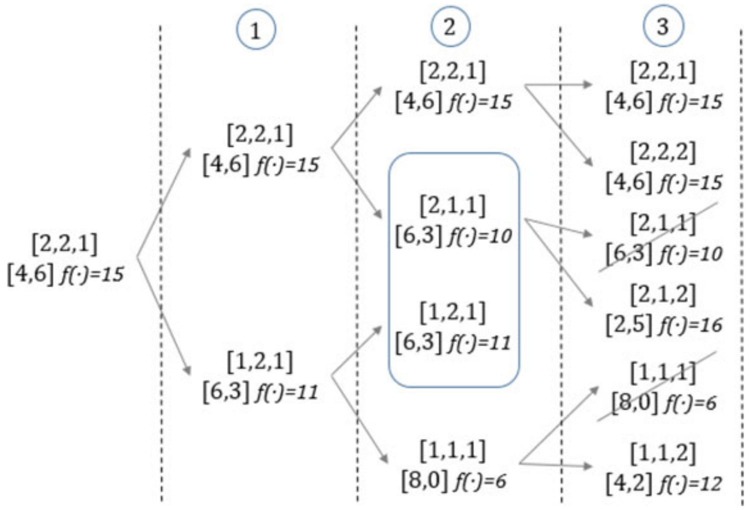


Fig. 6.4 Dynasearch recursion trace for GAP instance *tiny*

index of the client j corresponding to the recursion stage and the set X_j that contains all states at stage j. Each state is represented by the usual assignment matrix $\mathbf{x}$, from which it is possible to compute the server loads (which may be infeasible). The dynasearch equation gets the following explicit form:

$$f(j,\mathbf{x}) = \min\left\{ f(j-1,\mathbf{x}), \min_{\bar{\mathbf{x}}\in X_{j-1}} \{f(j-1,\bar{\mathbf{x}}) + gs(\mathbf{c}\odot(\mathbf{x}-\bar{\mathbf{x}}))\}\right\} \tag{6.2}$$

where $\odot$ denotes the Hadamard, or element-wise, product and function *gs()* the grand sum of all matrix elements, i.e., $gs(\mathbf{c}\odot(\mathbf{x}-\bar{\mathbf{x}})) = \mathbf{1}^\top(\mathbf{c}\odot(\mathbf{x}-\bar{\mathbf{x}}))\mathbf{1}$, or $\sum_{k\in J}\mathbf{c}^k(\mathbf{x}^k - \bar{\mathbf{x}}^k)$ using a scalar product notation. In formula (6.2) stages correspond to client indices and the cost of the sequence of moves is computed by multiplying the cost matrix $\mathbf{c}$ by the 0-1 matrices $(\mathbf{x}-\bar{\mathbf{x}})$, which denote the updated client assignments for the current state at stage j. For example, the lowest-right expansion in Fig. 6.4 goes from $\mathbf{x}_2^0 = \{1,1,1\}$ and $\mathbf{x}_2^1 = \{0,0,0\}$ to $\mathbf{x}_3^0 = \{1,1,0\}$ and $\mathbf{x}_3^1 = \{0,0,1\}$. These, multiplied by $\mathbf{c}_0 = \{1,2,3\}$ and by $\mathbf{c}_1 = \{5,7,9\}$ yield costs from $f(2,Q) = 1+2+3 = 6$ to cost $f(3,Q) = 1+2+9 = 12$.

An abstract pseudocode implementing this recursion is presented in the following Algorithm 33.

The parameter k_{max} is needed on nontrivial instances to control the computational cost of the procedure.

As an example, consider a tiny instance defined as follows, using the JSON notation introduced in Sect. 1.1:

```
{ "name":    "tiny",
  "numcli":  3,
  "numserv": 2,
  "cost":    [[1,2,3],[5,7,9]],
  "req":     [[2,2,4],[3,3,2]],
  "cap":     [4,7]
}
```

Assume that dynasearch is applied starting from solution $\boldsymbol{\sigma} = [2, 2, 1]$, which implies a cost of 15 and a server requirement of $[4, 6]$ (thus leaving residual capacities of $[0, 1]$ on the servers).

Figure 6.4 shows the search trace of dynasearch, when applied in the described settings. First, moves involving up to client 1 are considered, then up to client 2, and finally up to client 3 (equal to n in this tiny instance). State variables are defined on total requests to servers, and infeasibilities are checked only on final solutions to guarantee move independence. Note that this allows a dominance between two states (those in the box in the middle).

Compare this recursion against a classical dynamic programming recursion for the GAP. This is a constructive procedure that can be defined closely akin to that for the Multiple Knapsack Problem (MKP). It is, in fact, possible to add at each stage a client, keeping as state variables also the server residual capacities (or equivalently the loads) for the partial solution. The recursive equation is

$$f(j, Q_j) = \min_{\mathbf{x}^j}\{f(j-1, Q_{j-1} - (\mathbf{q}^j \odot \mathbf{x}^j)) + \mathbf{c}^j\mathbf{x}^j)\} \tag{6.3}$$

where $f(\cdot)$ denotes the cost function, $\mathbf{x}^j$ is a 0-1 vector representing the server assignment of client j, Q_j is the vector of server residual capacities after having assigned client j as specified in $\mathbf{x}^j$, $\mathbf{q}^j$ is the vector of requests of client j and the minimum is taken over all expansions that lead at stage j to a server availability as in Q_j.

Algorithm 33: Dynasearch for the GAP

```
1  function Dynasearch($\mathbf{x}$, $k_{max}$);
   Input  : A feasible solution $\mathbf{x}$
   Output: A feasible solution $\mathbf{x}^*$
2  Select (randomly) a subset $\mathcal{N} \subset J$ of $k_{max}$ clients;
3  Set $k = 0$;
4  for $j \in \mathcal{N}$ do
5      Set $k = k + 1$;
6      Compute all states at stage $k$ by means of Formula (6.2);
7  end
8  $\mathbf{x}^* = \mathbf{x}$;
9  foreach state $\{j_{max}, Q\}$ at stage $k_{max}$ do
10     if $(f(j_{max}, Q) < z(\mathbf{x}^*))$ then
11         $\mathbf{x}^*$ = ReconstructSolution($\{j_{max}, Q\}$);
12     end
13 end
14 return $\mathbf{x}^*$;
```

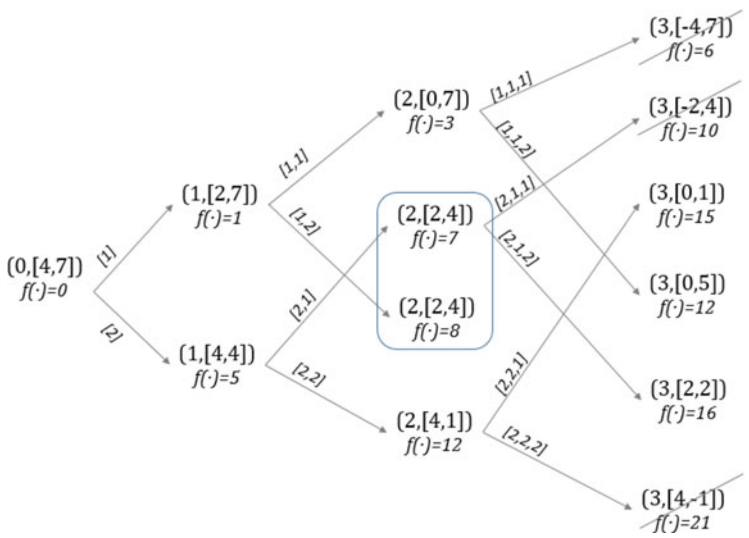


Fig. 6.5 Recursion trace for GAP instance *tiny*

Figure 6.5 depicts the trace of a dynamic programming recursion for the tiny instance. All nodes in the recursion tree report the state, as a pair (client-residual capacity vector), and the corresponding cost. All arcs are labeled by the solution corresponding to the final endnode state.

In the recursion, all states corresponding to server overloads are fathomed. Note a dominance when assigning client 2, which leads to the expansion of the least cost solution only.

Figures 6.4 and 6.5 outline the difference of the two dynamic approaches. While standard dynamic programming is constructive, going through successive partial solutions, dynasearch is a local search, working on complete solutions.

6.3.3 Neighborhoods Defined by Assignments and Matching

Matching is a problem that can be solved in polynomial time. Therefore, when it is possible to define a neighborhood by means of a matching problem, it is also possible to explore it in polynomial time despite its exponential size. One of the first neighborhoods of this kind was first presented for the TSP.

The idea for the TSP is to define a suitable improvement bipartite matching graph, based on the seed solution to be improved. This is obtained by removing k nodes from the initial tour, thus obtaining a subtour on the remaining nodes. The improvement graph then has each node to reinsert in the left layer and each arc of the subtour on the right layer. The cost of each assignment is the cost of inserting a node of the left between the endnodes of an arc on the right. This gives rise to a matching instance, which can be solved in polynomial time. Sarvanov and Doroshko analyzed the case where $k = n/2$ and n is even. Punnen generalized this to arbitrary k and n.

The same approach has been used for the distance-constrained capacitated vehicle routing problem, for the inventory routing problem, for the open vehicle routing problem, and for the problem of scheduling independent jobs on parallel machines minimizing the weighted average completion time.

A transposition of this idea to the GAP might involve a neighborhood defined over independent client reassignments, where independent means that the clients to reassign are to be assigned to different servers, and each could fit the place of any other. More formally, given a solution $\mathbf{x}$, a neighborhood can be defined by selecting a subset of clients $J' \subseteq J$ assigned to different servers in $\mathbf{x}$, such that for each $h, k \in J'$ the solution $\mathbf{x}' = [\mathbf{x}'_j]$ with $\mathbf{x}'_j = \mathbf{x}_j$, $j \neq h, k$ and $\mathbf{x}'_h = \mathbf{x}_k$, $\mathbf{x}'_k = x_h$ is feasible and $\mathbf{x}_h \neq \mathbf{x}_k$ for each $h, k \in J'$. The vector $\mathbf{x}_j$ is a binary column vector, containing only 0's except for a 1 in the row corresponding to the server to which client j is assigned (see Chap. 1).

As an example, consider Fig. 6.6 where the constructive solution produced in Sect. 1.3.1 is considered as a feasible solution of a relaxation of instance *example8x3* of Chap. 1, except that server 3 has an augmented capacity (otherwise only too simple neighborhoods are possible). Nodes corresponding to servers are on the right layers, to clients on the left layers. One client is selected for each server so that the clients are independent in the sense described above, and any single client assignment is feasible for the capacities. This leads from subfigure *a*) to *b*), with the request of solving the linear assignment problem instance of subfigure *b*), where the assignment costs are the same as those of the GAP. This is the assignment neighborhood that could be used in this case.

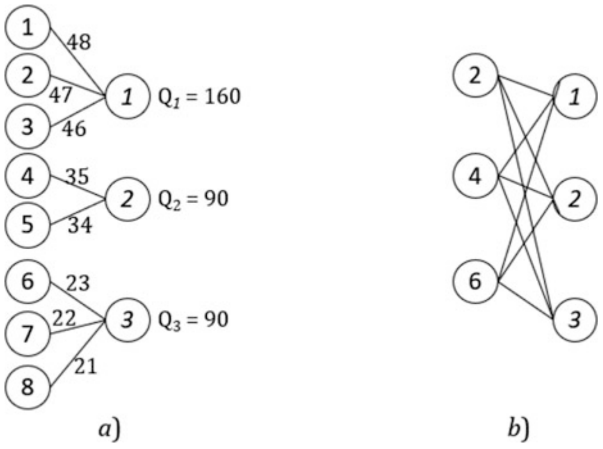


Fig. 6.6 Assignment neighborhood for GAP instance

As a final note, we point out that, for the case of the GAP, it is easy to generalize the described neighborhood to non-balanced matchings, where more than a client is chosen for each server.

6.4 Efficiently Solvable Restrictions

This is an area where recent research results have had the most impact. Ahuja et al. (2002) wrote a first survey of contributions that we eventually included in this subsection, and they titled their section "Solvable special cases" as at the time only "special cases" of NP-hard problems could be solved efficiently. Therefore, the attention was focused on cases that can be obtained from the original NP-hard problem by restricting the problem topology, or by adding constraints to the original problem. Later, it has been forcefully suggested that with the improvement of general MIP solvers and of computer hardware, a number of integer programming formulations of NP-hard problems can be efficiently solved up to big enough dimensions.

Anyway, assuming the availability of a procedure $solve(\mathscr{I}')$, capable of solving any instance $\mathscr{I}'$ of a possibly *restricted* version P' of an NP-hard problem P, a general method for turning it into a very large-scale neighborhood search technique, can be as follows.

Given a particular instance $\mathscr{I}$ of P, for every feasible solution $\mathbf{x}$ in the feasible set F of $\mathscr{I}$, there need to be a subroutine *CreateNeighborhood($\mathbf{x}$)* that creates an instance $\mathscr{I}'$ of P', such that $\mathbf{x}$ is an element of the feasible set F' of $\mathscr{I}'$ and F' is a subset of F.

The neighborhood search approach then consists of calling CreateNeighborhood($\mathbf{x}$) at each iteration, and then optimizing over $\mathscr{I}'$ using the efficient algorithm $solve(\mathscr{I}')$. Solution $\mathbf{x}$ can be replaced with the optimum of $\mathscr{I}'$ and the algorithm is iterated. Of particular interest is the case when the subroutine CreateNeighborhood runs in polynomial time, and the size of F' is exponential. Note that, following this approach, the neighborhood does not need to be created explicitly.

The original proposal of this method made reference to the so-called Halin graphs, which can provide one such restriction for the TSP, another case used a neighborhood for the single machine scheduling problem that was based on a dominance rule which could be solved in polynomial time by using a shortest processing time first heuristic.

There have not been proposals of this type for the GAP, even though it is not difficult to design some. Note that, in case the envisaged neighborhood was based on NP-hard subproblems or on MIP formulation, it would be easy to come up with an ejection chain method (relax-and-fix) such as that presented in Sect. 3.7.1, or with an algorithm that could be appropriately classified also as "Corridor Method" heuristic (see Chap. 8). This is the case, for example, of a supply chain management problem which was solved by fixing a subset of variables of the mathematical model and optimized over the remaining ones. Another example is an IP-based local search scheme for solving a Capacitated Fixed Charge Network Flow Problem, where the neighborhood was defined according to the arc-based formulation of the problem by choosing a subset of variables and including them in the formulation, fixing the value of all remaining variables, and solving the resulting restricted model with an IP solver.

6.5 VLNS for the GAP

VLSN search has been explicitly also applied to the generalized assignment problem. The small differences and the big overlaps among the various approaches in the VLSN search class make this algorithm suitable to be classified in several ways among the possibilities described in Sect. 6.1, including ejection chains, therefore also VLS, or efficient restrictions.

The idea is to implement a local search, where at each iteration some assignments of the incumbent solution are fixed. This leaves a restricted GAP instance to solve, where only the free decision variables are to be determined. Since the restricted GAP has a search space that is exponential in the number of the free variables, this approach qualifies as VLSN search.

The algorithm is a local search, iteratively updating an incumbent solution $\mathbf{x}$ by means of the best solution in the large neighborhood implicitly identified by the restricted instance. The restriction is implemented by identifying a subset $S \subseteq J$ of k clients, the *binding set*, whose assignment will not be changed and leaving free for reassignment only the clients in $S' = J \setminus S$. The restricted instance asks to solve a GAP defined only on the variables in S', the so-called *ejection set*, utilizing all servers in I with their residual capacities after the allocation of the clients in S.

The restricted instance, GAP(S, $\mathbf{x}$), defined over the variables corresponding to the clients in S' and where all clients in S are assigned as in $\mathbf{x}$, is solved obtaining an *augmented solution* $A(\mathbf{x}', \mathbf{x})$, consisting of all the assignments of the variables in S and the newly found ones for the variables in S'. Thus, keeping with the literature, $A(\mathbf{x}', \mathbf{x})$ is simply a solution where the assignments in $\mathbf{x}'$ substitute the corresponding ones in $\mathbf{x}$, leaving the other ones untouched. If the augmented solution is better than the incumbent one, it becomes the new incumbent solution and the procedure is iterated.

The last element to be defined is how to choose the clients to be included in S: a set too small induces an inefficient exploration, while a too big set makes the procedure trivial and ineffective, as little can be changed. The literature proposes several rules for this choice, based on an ordering of the decision variables by non-increasing values of the ratio c_{ij}/q_{ij}, $i \in I$ and $j \in J$, where "good" variables are assumed to be early in the assignment and worse ones later on. The selection rules are the following, where $k' = n - k$.

Rule R1: For each server, fix its best k'/n clients.
Rule R2: For each server, fix its worst k'/n clients.

Rule R3: For each server, fix $\frac{k'}{n/2}$ best and $\frac{k'}{n/2}$ worst clients.
Rule R4: For half of the servers, fix k'/n best, and for the other half of the servers, fix k'/n worst of the clients.
Rule R5: Fix the overall best k' among all possible client assignments.
Rule R6: Fix the overall worst k' among all possible client assignments.
Rule R7: Fix the best $k'/2$ and the worst $k'/2$ among all possible client assignments.
Rule R8: Controlled random fixing: fix $k'/10$ random clients in the best 10%, then fix $k'/10$ random clients in the next best 10%, etc.
Rule R9: Meta-neighborhood: generate $\lceil \frac{n}{n-k} \rceil$ different binding sets S by selecting $(n-k)$ consecutive elements from the ordering. For example, if $n = 100$ and $|S| = 20$, the following five versions of the binding set S is generated: fixing the first 20 assignments, the second 20, the third 20, the fourth 20, and the fifth 20.

The complete VLSN for the GAP is thus presented as Algorithm 34.

Algorithm 34: VLSN for the GAP

```
1  function VLS_GAP($\mathbf{x}$, $q$, $R$, $k$);
   Input  : A feasible solution $\mathbf{x}$, max inner iterations $q$, rule set $R$, ejection set size $k$
   Output: A feasible solution $\mathbf{x}^*$
2  $\mathbf{x}^* = \mathbf{x}$; $r = 1$;
3  repeat                                        // Search the neighborhood
4      while $\mathbf{x}$ improved in the last $q$ iterations do
5          Choose the binding set $S$ by rule $R_r$;          // neighborhood $N_r$
6          Set $\mathbf{x}'$ = solve GAP($S$, $\mathbf{x}$);
7          Compute the augmented solution $A(\mathbf{x}', \mathbf{x})$;
8          if ($z(A(\mathbf{x}', \mathbf{x})) < z(\mathbf{x})$) then    // update the current solutions
9              $\mathbf{x} = A(\mathbf{x}', \mathbf{x})$;
10             if ($z(\mathbf{x}) < z(\mathbf{x}^*)$) then $\mathbf{x}^*$=$\mathbf{x}$;
11         end
12     end
13     if ($r < |R|$) then
14         $r = r + 1$;
15     else
16         $\mathbf{x} = PerturbSolution(\mathbf{x})$;
17         $r = 1$;
18     end
19 until terminating condition;
20 return $\mathbf{x}^*$;
```

The algorithm proposed in the literature is actually more complex than Algorithm 34, as it includes mechanisms to foster intensification and diversification based on how often each decision variable had been fixed in a binding set, and how often it was included in a complete solution, which are omitted in Algorithm 34.

Another option, which is included in the original proposal, is the increase of the ejection set size k when moving from a neighborhood to another. This is an interesting option, which, however, falls outside of the focus of this text, where we want to present the core of each algorithmic approach, omitting all details that can lead the algorithm to the state-of-the-art research level.

We apply Algorithm 34 starting from the constructive heuristic solution of Chap. 1 and with $k = 4$, that is, allowing to change up to 4 column vectors of decision variables. Assuming to use rule R_5, the vector of ratios for the ordering of the variables is $[0.21, 0.23, 0.26, 0.80, 0.88, 3.48, 3.82, 4.19]$, which for this instance is already ordered. Therefore, the $k = 4$ assignments to fix are the first four ones.

The reduced problem is thus solved, as per step 6, and the new solution is that of Eq. (6.4), of cost 337, where the assignments of clients 5 and 8 have been modified with respect to the seed one.

$$\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{6.4}$$

This solution is a local optimum, a further cycle at step 4 does not change the ordering, thus the solution. A change of the rule defining the bounding set (step 14) or a bigger perturbation (step 16) is needed to have the possibility to further improve the solution.

6.6 Related Literature

The concept of VLSN was originally introduced by Ahuja et al. (1999, 2000), and it has been surveyed several times (Ahuja et al. 2002, 2007; Altner 2017; Altner et al. 2014; Chiarandini et al. 2008; Pisinger and Ropke 2010). The VDM specific for the GAP was proposed in Yagiura et al. (1998).

The classical reference for network flow optimization is Ahuja et al. (1993). The correspondence between improving cyclic exchange and negative cost subset-disjoint cycle is from Thompson and Psaraftis (1993). The listed applications of cyclic exchange neighborhoods can be found in Thompson and Psaraftis (1993), Gendreau et al. (2006), Frangioni et al. (2004), Ahuja et al. (1999), Chiarandini et al. (2008), Ahuja et al. (2004), and Meyers and Orlin (2006), in this order.

Large Neighborhood Search was presented in Shaw (1998) and in Pisinger and Ropke (2010) and generalized into ALNS by Ropke and Pisinger (2006). Dynasearch was presented in Congram et al. (2002). Its applications, as listed, can be found in Sourd (2006), Congram (2000), Congram et al. (2002), Nishi et al. (2020), Ergun et al. (2006), and Cunha and Ahuja (2005).

An early matching neighborhood for the TSP can be found in Sarvanov and Doroshko (1981), then the listed applications are from De Franceschi et al. (2006), Dror and Levy (1986), Salari et al. (2010), and Brueggemann and Hurink (2011).

The use of MIP solvers to deal with subproblems was put forward most explicitly by Fischetti et al. (2009). The Halin graph case is in Ahuja et al. (2002) and single machine scheduling neighborhood is from Brueggemann and Hurink (2007). The other applications are from Copado-Méndez et al. (2013) and Hewitt et al. (2010). Other examples are Schmid et al. (2010), and Roli et al. (2012).

The described GAP application is from Mitrović-Minić and Punnen (2008, 2009).

References

Ahuja RK, Magnanti TL, Orlin J (1993) Network flows: Theory, algorithms, and applications. Prentice-Hall, Upper Saddle River, NJ, USA

Ahuja RK, Orlin JB, Sharma D (1999) New neighborhood search structures for the capacitated minimum spanning tree problem. Technical Report 99-2, Department of Industrial and Systems Engineering, University of Florida

Ahuja RK, Orlin JB, Sharma D (2000) Very large-scale neighborhood search. Int Trans Oper Res 7(4-5):301–317

Ahuja RK, Ergun O, Orlin JB, Punnen APA (2002) Survey of very large-scale neighborhood search techniques. Discrete Appl Math 123:75–102

Ahuja RK, Orlin JB, Pallottino S, Scaparra MP, Scutella MG (2004) A multi-exchange heuristic for the single source capacitated facility location. Management Science 50:749–760

Ahuja RK, Ergun O, Orlin JB, Punnen AP (2007) Very large-scale neighborhood search. In: Gonzalez TF (ed) Approximation algorithms and metaheuristics. Chapman & Hall, pp 20–1–20–12

Altner DS (2017) Very large-scale neighborhood search. In: Handbook of discrete and combinatorial mathematics. Chapman and Hall/CRC

Altner DS, Ahuja RK, Ergun O, Orlin JB (2014) Very large-scale neighborhood search. In: Burke E, Kendall G (eds) Search methodologies. Springer, Boston, MA

Brueggemann T, Hurink JL (2007) Two exponential neighborhoods for single machine scheduling. OR Spectrum 29:513–533

Brueggemann T, Hurink JL (2011) Matching based very large-scale neighborhoods for parallel machine scheduling. J Heuristics 17(6):637–658

Chiarandini M, Dumitrescu I, Stützle T (2008) Very large-scale neighborhood search: Overview and case studies on coloring problems. In: Blum C, Blesa MJ, Roli A, Sampels M (eds) Hybrid metaheuristics, vol. 114. Studies in computational intelligence. Springer, pp 117–150

Congram RK (2000) Polynomially searchable exponential neighbourhoods for sequencing problems in combinatorial optimization. Ph.D. thesis, Southampton University, Faculty of Mathematical Studies, Southampton, UK

Congram RK, Potts CN, van de Velde S (2002) An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. INFORMS J Comput 14(1):52–67

Copado-Méndez P, Blum C, Guillén-Gosálbez G, Jiménez L (2013) Large neighbourhood search applied to the efficient solution of spatially explicit strategic supply chain management problems. Comput Chem Eng 49:114–126

Cunha CB, Ahuja RK (2005) Very large scale neighborhood search for the k-constrained multiple knapsack problem. J Heuristics 11:465–481

De Franceschi R, Fischetti M, Toth P (2006) A new ILP-based refinement heuristic for vehicle routing problems. Mathematical Programming 105(2-3):471–499

Dror M, Levy L (1986) A vehicle routing improvement algorithm comparison of a "greedy" and a "matching" implementation for inventory routing. Comput Oper Res 13:33–45

Ergun O, Orlin JB, Steele-Feldman A (2006) Creating very large scale neighborhoods out of smaller ones by compounding moves. J Heuristics 12(1-2):115–140

Fischetti M, Lodi A, Salvagnin D (2009) Just MIP it! In: Maniezzo V, Stützle T, Voß S (eds) Matheuristics: Hybridizing metaheuristics and mathematical programming. Annals of information systems, vol 10. Springer, pp 39–70

Frangioni A, Necciari E, Scutellá MG (2004) A multi-exchange neighborhood for minimum makespan parallel machine scheduling problems. J Comb Optim 8(2):195–220

Gendreau M, Guertin F, Potvin JY, Seguin R (2006) Neighborhood search heuristics for a dynamic vehicle dispatching problem with pick-ups and deliveries. Transp Res C Emerg Technol 14:157–174

Hewitt M, Nemhauser GL, Savelsbergh MWP (2010) Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem. INFORMS J Comput 22(2):314–325

Lin S, Kernighan B (1973) An effective heuristic algorithm for the traveling salesman problem. Operations Research 21:498–516

Meyers C, Orlin JB (2006) Very large-scale neighborhood search techniques in timetabling problems. In: Burke EK, Rudová H (eds) Proceedings of the 6th international conference on practice and theory of automated timetabling VI (PATAT'06). Springer, Berlin, Heidelberg, pp 24–39

Mitrović-Minić S, Punnen AP (2008) Very large-scale variable neighborhood search for the generalized assignment problem. J Interdisciplinary Math 11(5):653–670

Mitrović-Minić S, Punnen AP (2009) Variable intensity local search. In: Maniezzo V, Stützle T, Voß S (eds) Matheuristics: Hybridizing metaheuristics and mathematical programming. Annals of information systems, vol 10. Springer US, Boston, MA, pp 245–252

Nishi T, Okura T, Lalla-Ruiz E, Voß S (2020) A dynamic programming-based matheuristic for the dynamic berth allocation problem. Ann Oper Res 286:391–410

Pisinger D, Ropke S (2010) Large neighborhood search. In: Gendreau M, Potvin J (eds) Handbook of metaheuristics. International series in operations research & management science, vol 146. Springer, Boston, MA, pp 399–419

Roli A, Benedettini S, Stützle T, Blum C (2012) Large neighbourhood search algorithms for the founder sequence reconstruction problem. Comput Oper Res 39:213–224

Ropke S, Pisinger D (2006) An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation Science 40(4):455–472

Salari M, Toth P, Tramontani A (2010) An ILP improvement procedure for the open vehicle routing problem. Comput Oper Res 37(12):2106–2120

Sarvanov VI, Doroshko NN (1981) Approximate solution of the traveling salesman problem by a local algorithm with scanning neighborhoods of factorial cardinality in cubic time. Software: Algorithms and Programs, Mathematics Institute of the Belorussia Academy of Science, Minsk, vol 31, pp 11–13

Schmid V, Doerner KF, Hartl RF, Salazar-González JJ (2010) Hybridization of very large neighborhood search for ready-mixed concrete delivery problems. Comput Oper Res 37(3):559–574

Shaw P (1998) Using constraint programming and local search methods to solve vehicle routing problems. In: CP-98 (Fourth international conference on principles and practice of constraint programming). Lecture notes in computer science, vol 1520. Springer, pp 417-431

Sourd F (2006) Dynasearch neighborhood for the earliness-tardiness scheduling problem with release dates and setup constraints. Oper Res Lett 34(5):591–598

Thompson PM, Psaraftis HN (1993) Cyclic transfer algorithms for multivehicle routing and scheduling problems. Operations Research 41:935–946

Yagiura M, Yamaguchi T, Ibaraki T (1998) A variable depth search algorithm with branching search for the generalized assignment problem. Optim Methods Softw 10:419–441

Chapter 7
Decomposition-Based Heuristics

7.1 Introduction

Decompositions are methods derived from the "*divide et impera*" principle, dictating to break up a difficult problem into smaller ones, and to solve each of the smaller ones separately, ultimately recomposing the individual solutions to get the overall one. Decompositions have longly been applied to solve optimization problems, and they come in many different flavors, ranging from constraint programming to logical decomposition, from dynamic programming to linear decompositions, among many others.

In this text, we are interested in heuristic approaches. We present three related mathematical decomposition techniques that have been used as seeds for classes of heuristic algorithms, plainly to be included among matheuristics, namely Lagrangian, Dantzig–Wolfe, and Benders decompositions. They are all decompositions that can also be applied to linear problems and were outlined in Chap. 1 for the case of the GAP but will be introduced in the next section in some more detail. We will finally give a thorough presentation and a run trace for a Lagrangian heuristic.

7.2 Linear Decompositions

Decompositions can be applied to continuous, mixed-integer, and pure integer linear programming problems, but we only introduce here the basic formulae of decomposition in the general case of a mixed-integer problem. The discussion is for a minimization problem, but it is trivial to convert it into a maximization one. Moreover, we are interested in decomposition strategies that lend themselves to seed

V. Maniezzo et al., *Matheuristics*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-70277-9_7

new heuristic approaches, and this has been reported in the literature mainly for the case of linear problems with a specific structure. We will use this case throughout this chapter.

Decompositions of a mathematical formulation can be trivially obtained when the constraint matrix is block separable. More interesting cases arise when blocks can be identified in the constraint matrix, but they are linked by some constraints (*linking constraints*) or by some variables (*linking variables*). We will concentrate only on some specific case, as the interested reader can anyway refer to the abundant literature on general decomposition in optimization.

In the case of interest, the structure of a linear problem suitable for effective decomposition is therefore that of the following problem LP:

$$z_{LP} = \min \mathbf{c}_1\mathbf{x}^1 + \mathbf{c}_2\mathbf{y} + \mathbf{c}_3\mathbf{x}^2 \tag{7.1}$$

$$s.t.\ \mathbf{A}\mathbf{x}^1 + \mathbf{B}\mathbf{y} \geq \mathbf{b} \tag{7.2}$$

$$\mathbf{D}\mathbf{y} + \mathbf{E}\mathbf{x}^2 \geq \mathbf{d} \tag{7.3}$$

$$\mathbf{x}^1, \quad \mathbf{y}, \quad \mathbf{x}^2 \geq \mathbf{0} \tag{7.4}$$

Problem LP could be split into two subproblems, were it not for the linking variables $\mathbf{y}$ (note that in a more general case the problem could split into many subproblems). In fact, the problem becomes block separable in the variables $\mathbf{x}^1$ and $\mathbf{x}^2$ if we fix the variables $\mathbf{y}$ to some values $\bar{\mathbf{y}}$. In this case, problem LP can be solved by solving the two subproblems independently, by minimizing $z_{LP}(\mathbf{x}^1, \mathbf{x}^2, \bar{\mathbf{y}}) = z_{LP1}(\mathbf{x}^1, \bar{\mathbf{y}}) + z_{LP2}(\mathbf{x}^2, \bar{\mathbf{y}})$, where

$$\begin{aligned} z_{LP1} &= \min\ \mathbf{c}_1\mathbf{x}^1 & \qquad z_{LP2} &= \min\ \mathbf{c}_3\mathbf{x}^2 \\ &\text{s.t. } \mathbf{A}\mathbf{x}^1 \geq \mathbf{b} - \mathbf{B}\bar{\mathbf{y}} & &\text{s.t. } \mathbf{E}\mathbf{x}^2 \geq \mathbf{d} - \mathbf{D}\bar{\mathbf{y}} \\ &\mathbf{x}^1 \geq 0 & &\mathbf{x}^2 \geq 0 \end{aligned}$$

Since the problem becomes separable when $\mathbf{y}$ is fixed, a two-stage method can be used for solving problem LP. If we denote by $z_1(\bar{\mathbf{y}})$ the optimal value of the problem $min\ z_{LP1}(\mathbf{x}^1, \bar{\mathbf{y}})$—henceforth, denoted as *subproblem 1*—and similarly, by $z_2(\bar{\mathbf{y}})$ the optimal value of the problem $min\ z_{LP2}(\mathbf{x}^2, \bar{\mathbf{y}})$—henceforth, denoted as *subproblem 2*—then the original problem LP is equivalent to the problem $min_{\mathbf{y}}$ $(z_1(\mathbf{y}) + z_2(\mathbf{y}))$.

The problem of fixing the $\mathbf{y}$ variables to eventually achieve global optimality is called the *master problem*. The variables of the master problem are therefore the linking (complicating) variables of the original problem, and its objective derives from the sum of the optimal values of the subproblems.

A generic primal decomposition method solves problem LP by iteratively solving the master problem. Each iteration fixes the linking variables and proceeds solving the two subproblems independently, in order to evaluate $z_1(\mathbf{y})$ and $z_2(\mathbf{y})$, obtaining information on how to update the linking variables at the next master iteration.

Such a decomposition method is effective when there are few complicating variables, and there are efficient algorithms for solving the subproblems.

This basic decomposition method is called *primal decomposition* because the master problem and the subproblems are defined only on the primal variables. If dual problems are instead used, we deal with dual decompositions, an example of which will be explored in the following section.

We will not further detail this procedure, available in standard optimization textbooks, but we will describe it for the case of interest for matheuristic design, which so far contemplated only reports on the case where, after fixing the linking variables in the master problem, only one subproblem is left to solve, i.e., there are no $\mathbf{x}^2$ variables. However, we include in the presentation the mixed-integer linear programming (MILP) case, where the master problem variables are required to be integer. In this case, the problem to solve, called P, can be represented according to the following structure:

$$z_P = \min \mathbf{c}_1\mathbf{x} + \mathbf{c}_2\mathbf{y} \tag{7.5}$$

$$s.t.\ \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{y} \geq \mathbf{b} \tag{7.6}$$

$$\mathbf{D}\mathbf{y} \geq \mathbf{d} \tag{7.7}$$

$$\mathbf{x} \geq \mathbf{0} \tag{7.8}$$

$$\mathbf{y} \geq \mathbf{0} \text{ and integer} \tag{7.9}$$

We further assume, for ease of presentation, that the feasibility region is non-null and bounded, and that constraints (7.7) are *easy* to satisfy, while constraints (7.6) are hard, complicating constraints.

In the following subsections, we introduce three different ways to decompose problem P either by primal or by dual decomposition: Dantzig–Wolfe, Lagrangian, and Benders decompositions.

7.2.1 Dantzig–Wolfe Decomposition

Dantzig–Wolfe decomposition is an iterative procedure that successively approximates the linear relaxation of problem P by decomposing it into a sequence of smaller and/or easier *subproblems*, in accordance with the general working introduced in the previous section. The subproblems dynamically generate the columns of the *master problem* corresponding to the linear programming relaxation of P.

In the case of our example, problem P, we keep the difficult linking constraints (7.6) in the master problem, as introduced in Sect. 7.2, and demand the rest to the subproblems. Let F be the feasible region induced by constraints (7.7)–(7.9), i.e.,

$F = \{(\mathbf{x}, \mathbf{y}) : \mathbf{D}\mathbf{y} \geq \mathbf{d}, \mathbf{x} \geq \mathbf{0}, \mathbf{y} \geq \mathbf{0}$ and integer$\}$, which we assume finite and non-null, and let $\{(\mathbf{x}^t, \mathbf{y}^t) : t = 1, \ldots, T\}$ be the set of the extreme points of F.

Dantzig–Wolfe proceeds by identifying optimal (with respect to the current cost function) extreme points of F, computed as solutions of the *subproblems*, and then passing them to the *master problem* in order to check them against the relaxed constraints, i.e., the linking constraints, which are not F-defining. The master problem is formulated by replacing F with the convex combination of its extreme points. Since we have assumed that F is non-null and bounded, we do not have to consider the extreme rays.

After having computed the cost of the best combination of the so far proposed extreme points of F, taking into consideration also the relaxed constraints retained in the master, the subproblem costs are updated and are computed as reduced costs derived from the values of the dual variables associated to the relaxed constraints. The subproblem is then solved again, to see whether a new, less expensive extreme point can be found.

In the case of problem P, a possible master problem, obtained by relaxing the "difficult" linking constraints (7.6), thus keeping them in the master problem, is as follows:

$$z_{MDW} = \min \sum_{t=1}^{T} (\mathbf{c}_1\mathbf{x}^t + \mathbf{c}_2\mathbf{y}^t)\mu_t \tag{7.10}$$

$$s.t. \sum_{t=1}^{T} (\mathbf{A}\mathbf{x}^t + \mathbf{B}\mathbf{y}^t)\mu_t \geq \mathbf{b} \tag{7.11}$$

$$\sum_{t=1}^{T} \mu_t = 1 \tag{7.12}$$

$$\mu_t \geq 0, \qquad t = 1, \ldots, T \tag{7.13}$$

The corresponding subproblem is

$$z_{SDW}(\mathbf{u}, \alpha) = \min\ (\mathbf{c}_1 - \mathbf{u}\mathbf{A})\mathbf{x} + (\mathbf{c}_2 - \mathbf{u}\mathbf{B})\mathbf{y} - \alpha \tag{7.14}$$

$$s.t.\ \mathbf{D}\mathbf{y} \geq \mathbf{d} \tag{7.15}$$

$$\mathbf{x} \geq \mathbf{0} \tag{7.16}$$

$$\mathbf{y} \geq \mathbf{0} \text{ and integer} \tag{7.17}$$

where $\mathbf{u}$ and α are the dual variables corresponding to constraints (7.11) and (7.12) of the master problem, respectively.

If the optimal solution $(\mathbf{x}^*, \mathbf{y}^*)$ of the subproblem has a cost $z_{SDW}(\mathbf{u}, \alpha) < 0$, we can add into the master problem the column of cost $(\mathbf{c}_1\mathbf{x}^* + \mathbf{c}_2\mathbf{y}^*)$, having

$(\mathbf{Ax}^* + \mathbf{By}^*)$ corresponding to constraint (7.11), and a 1 coming from constraint (7.12), otherwise we have reached the optimal solution of MDW.

At each iteration of the procedure, a valid lower bound to the optimal solution value of the original problem is given by $z_{MDW} + z_{SDW}$. This lower bound is not monotonically non-decreasing; therefore, we need to maintain the best value obtained along the iterations.

7.2.2 Lagrangian Relaxation and Decomposition

A second decomposition is obtained by a *Lagrangian relaxation*, which permits to compute a lower bound to problem P by removing some difficult constraints and by dualizing them into the objective function by means of *Lagrangian penalties*.

In the example of problem P, we can again relax the difficult constraints (7.6), this time using the non-negative Lagrangian penalty vector $\boldsymbol{\lambda}$, thus obtaining the following formulation LR:

$$z_{LR}(\boldsymbol{\lambda}) = \min \mathbf{c}_1\mathbf{x} + \mathbf{c}_2\mathbf{y} + \boldsymbol{\lambda}(\mathbf{b} - \mathbf{Ax} - \mathbf{By}) \tag{7.18}$$

$$s.t.\ \mathbf{Dy} \geq \mathbf{d} \tag{7.7}$$

$$\mathbf{x} \geq \mathbf{0} \tag{7.8}$$

$$\mathbf{y} \geq \mathbf{0} \text{ and integer} \tag{7.9}$$

The value $z_{LR}(\boldsymbol{\lambda})$ is a valid lower bound to the optimal value of P, i.e., $z_{LR}(\boldsymbol{\lambda}) \leq z_P$, for every $\boldsymbol{\lambda} \geq \mathbf{0}$.

To identify the penalty vector $\boldsymbol{\lambda}$ that maximizes the lower bound $z_{LR}(\boldsymbol{\lambda})$, we solve a dualized master problem, in this case the so-called *Lagrangian Dual*, that can be formulated as follows:

$$z_{LR} = \max\{z_{LR}(\boldsymbol{\lambda}) : \boldsymbol{\lambda} \geq \mathbf{0}\} \tag{7.19}$$

which fixes a penalty vector and then asks to solve the internal *subproblem* LR. This can be rewritten as follows:

$$z_{LR}(\boldsymbol{\lambda}) = \min\ (\mathbf{c}_1 - \boldsymbol{\lambda}\mathbf{A})\mathbf{x} + (\mathbf{c}_2 - \boldsymbol{\lambda}\mathbf{B})\mathbf{y} + \boldsymbol{\lambda}\mathbf{b} \tag{7.20}$$

$$s.t.\ \mathbf{Dy} \geq \mathbf{d} \tag{7.21}$$

$$\mathbf{x} \geq \mathbf{0} \tag{7.22}$$

$$\mathbf{y} \geq \mathbf{0} \text{ and integer} \tag{7.23}$$

If the subproblem is solved to integrality, it is possible that the lower bound provided by z_{LR} is tighter than the linear relaxation of problem P.

Notice that it is possible to add to subproblem LR additional constraints that are redundant in the original formulation, but that can help convergence. Moreover, it is sometimes possible to obtain feasible dual solutions directly from the Lagrangian penalties. Approaches based on this property have been used, for example, to generate reduced problems that consider only the variables of k-least reduced cost. These approaches can make it possible to substantially reduce the number of variables to consider when solving the problem heuristically, or in some cases also to optimality. The k-least reduced cost variables can also be generated in a column generation fashion, combining Lagrangian relaxation and column generation.

Subproblem Relations
It is intriguing to note that subproblem LR is identical to subproblem SDW obtained in the Dantzig–Wolfe decomposition, if we replace $\mathbf{u}$ and α with $\boldsymbol{\lambda}$ and $-\boldsymbol{\lambda}\mathbf{b}$, respectively. Such identities among subproblems obtained by different relaxations will also be outlined in the following sections.

7.2.3 Benders Decomposition

Benders decomposition computes a lower bound to the optimal cost of the original problem again by solving a *master problem* that fixes some linking variables. Then, to improve the lower bound, it solves a *subproblem*, which adds new constraints to the master. The approach is quite coherent with the description proposed in Sect. 7.2, except that it is defined on the dual.

The original problem (7.5)–(7.9) can be rewritten splitting it into a master problem z_B:

$$z_B = \min \mathbf{c}_2\mathbf{y} + z_{SB}(\mathbf{y}) \tag{7.24}$$

$$s.t.\ \mathbf{Dy} \geq \mathbf{d} \tag{7.25}$$

$$\mathbf{y} \geq \mathbf{0} \text{ and integer} \tag{7.26}$$

and the subproblem $z_{SB}(\mathbf{y})$:

$$z_{SB}(\mathbf{y}) = \min \mathbf{c}_1\mathbf{x} \tag{7.27}$$

$$s.t.\ \mathbf{Ax} \geq \mathbf{b} - \mathbf{By} \tag{7.28}$$

$$\mathbf{x} \geq \mathbf{0} \tag{7.29}$$

The dual of the subproblem $z_{SB}(\mathbf{y})$ is

$$z_{SBD}(\mathbf{y}) = \max \mathbf{w}(\mathbf{b} - \mathbf{By}) \tag{7.30}$$

$$s.t.\ \mathbf{wA} \leq \mathbf{c}_1 \tag{7.31}$$

$$\mathbf{w} \geq \mathbf{0} \tag{7.32}$$

Upon denoting $W = \{\mathbf{w} : \mathbf{wA} \leq \mathbf{c}_1, \mathbf{w} \geq \mathbf{0}\}$ and $Y = \{\mathbf{y} : \mathbf{Dy} \geq \mathbf{d}, \mathbf{y} \geq \mathbf{0}$ and integer$\}$, we can rewrite problem B as follows:

$$z_B = \min_{\mathbf{y}\in Y} \max_{\mathbf{w}\in W} (\mathbf{c}_2 - \mathbf{wB})\mathbf{y} + \mathbf{wb}. \tag{7.33}$$

Let $\{\mathbf{w}^t, t = 1, \ldots, T\}$ be the set of the extreme points of W. Since we have assumed the feasible region finite and non-null, we can express z_B as a function of the extreme points as follows:

$$z_B = \min_{\mathbf{y}\in Y} \max_{t=1,\ldots,T} (\mathbf{c}_2 - \mathbf{w}^t\mathbf{B})\mathbf{y} + \mathbf{w}^t\mathbf{b} \tag{7.34}$$

which is equivalent to the following formulation:

$$z_{MB} = \min z \tag{7.35}$$

$$s.t.\ z \geq (\mathbf{c}_2 - \mathbf{w}^t\mathbf{B})\mathbf{y} + \mathbf{w}^t\mathbf{b}, \qquad t = 1, \ldots, T \tag{7.36}$$

$$\mathbf{y} \in Y \tag{7.37}$$

Problem MB is the *Benders' master problem* and constraints (7.36) are the so-called *Benders' cuts*. The number of Benders' cuts T is usually huge; therefore, the master problem is initially solved considering only a small number T' of Benders' cuts, i.e., $T' << T$. In order to ascertain whether the solution is already optimal or an additional cut should be added to the master, we need to solve a *subproblem* $z_{SB}(\mathbf{y})$ defined in (7.27)–(7.29).

Subproblem Relations

We have already remarked that Dantzig–Wolfe and Lagrangian decompositions give rise to almost identical subproblems. It is noteworthy to show that also for Benders decomposition we can have a subproblem similar to the two already seen ones. In fact, if we rewrite problem (7.33) as follows:

$$z_B = \max_{\mathbf{w}\in W} \min_{\mathbf{y}\in Y} (\mathbf{c}_2 - \mathbf{wB})\mathbf{y} + \mathbf{wb} \tag{7.38}$$

it is easy to note that the inner optimization $\min_{\mathbf{y}\in Y}(\mathbf{c}_2 - \mathbf{wB})\mathbf{y} + \mathbf{wb}$ is almost equivalent to the Dantzig–Wolfe and Lagrangian subproblems. The vector $\mathbf{w}$ can replace the dual variable vector $\mathbf{u}$ in the Dantzig–Wolfe decomposition or the penalty vector $\boldsymbol{\lambda}$ in the Lagrangian relaxation. Moreover, since $\mathbf{w}$ is the optimal dual solution of the subproblem $z_{SB}(\mathbf{y})$ defined in (7.30)–(7.32), due to the complementary slackness, we have in the Benders as in other decompositions that $(\mathbf{c}_1 - \mathbf{wA})\mathbf{x} = 0$.

7.3 Matheuristics Derived from Decompositions

All three decompositions, besides possibly other ones such as surrogate approaches, can be used as foundations upon which to design general heuristic cores and which could be easily adapted to a combinatorial optimization problem of interest. The literature describes in detail how this can be done in each case. Here we present only the case for Lagrangian relaxation and how it can be specified for the GAP.

7.3.1 Lagrangian Metaheuristic

The literature is rich with heuristics based on the decomposition structure outlined in Sect. 7.2. Among them, Lagrangian relaxation has a particularly good record of successful applications.

Most Lagrangian heuristics share a common general structure that can be considered a metaheuristic of its own. The structure is presented in Algorithm 35.

Algorithm 35: Core Lagrangian heuristic

```
1  function LagrHeuristic();
   Input  : Control parameters
   Output: A feasible solution $\mathbf{x}^*$ of value z*
2  Initialize the penalty vector $\boldsymbol{\lambda}$;
3  Identify an "easy" subproblem LR($\boldsymbol{\lambda}$);
4  repeat
5      solve subproblem LR($\boldsymbol{\lambda}$) obtaining the possibly infeasible solution $\mathbf{x}$;
6      check for unsatisfied constraints;
7      update penalties $\boldsymbol{\lambda}$;
8      construct problem solution $\mathbf{x^h}$ using $\mathbf{x}$ and $\boldsymbol{\lambda}$;
9      if $z(\mathbf{x^h}) < z^*$ then $\mathbf{x}^* = \mathbf{x^h}$;        // $z(\mathbf{x^h}), z^*$ are the costs of $\mathbf{x^h}, \mathbf{x}^*$
10 until end_condition;
```

The pseudocode of Algorithm 35 must be specified for any specific application. It has been observed that it is presented at the abstraction level where metaheuristics are usually introduced. In fact, this pseudocode already shows the essential ingredients of metaheuristics, which is defined as "an iterative master process that guides and modifies the operations of a subordinate heuristic," where the subordinate heuristic is at step 8.

Steps 3 and 5 are problem-dependent, much alike neighborhoods definition or crossover implementation in other contexts. Step 6 is trivial, while step 7 can be implemented by means of any state-of-the-art techniques, usually subgradient optimization or bundle methods. Interestingly, some of these techniques have been proved to converge not only to the optimal $\boldsymbol{\lambda}$, but also to the optimal $\mathbf{x}$ of the linear relaxation, thereby possibly providing a particularly "intelligent" starting point for step 8.

7.3.2 Lagrangian Matheuristics

Algorithm 35 proposes the Lagrangian heuristic as an add-on to a core procedure that computes the Lagrangian bound. This is, in fact, the way most Lagrangian heuristics have usually been proposed, and the literature describing these approaches is rich and decades-long.

The core of the heuristics is therefore the algorithm that computes the Lagrangian bound. The most used algorithms for this are the following:

- *Subgradient optimization* is a simple algorithm for minimizing a possibly nondifferentiable convex function. The method is similar to an ordinary gradient method for differentiable functions but departs from it in several important details. For example, it can use precomputed step lengths, instead of an exact or approximate line search, and, more importantly, it does not use a direction that is an actual gradient, as obtained from the differentiation of the objective function, but the direction is obtained from the violations of the relaxed constraints, violations that can define "*subgradients*." Moreover, the subgradient algorithm is not a descent method, and the function value can repeatedly increase and decrease along its iterations, as it usually does.
- *Multiplier adjustment*, interestingly introduced with an application to the GAP, increases one multiplier at a time. The method begins by initializing each multiplier λ_j to its second largest c_{ij}, so that $c_{ij} - \lambda_j > 0$ for at most one $i \in I$ and such that there is an optimal Lagrangian solution for this $\boldsymbol{\lambda}$ that satisfies $\sum_{i \in I} x_{ij} \leq 1$ for $j \in J$. If $\sum_{i \in I} x_{ij} = 1$ for all $j \in J$, then this solution is feasible and optimal. Otherwise, it is usually possible to select a client j^* for which $\sum_{i \in I} x_{ij^*} = 0$ and decrease λ_{j^*} to a value for which the new Lagrangian solution satisfies $\sum_{i \in I} x_{ij^*} = 1$ and $\sum_{i \in I} x_{ij} \leq 1$ for all other j. If the required conditions are satisfied, the multiplier λ_{j^*} is decreased and the procedure is iterated.
- *Volume algorithm* is a refinement of the subgradient approach but applied to a Dantzig–Wolfe reformulation of the problem. The algorithm attempts to find an approximate solution of the master problem by computing at each iteration a *stability center*, a *step*, and a subgradient-based *direction*. The stability center represents a point that has provided significant improvement with respect to the optimization process. In turn, the step represents how far one may move in the identified direction so that a new trial point is obtained. The stability center is updated when specific conditions are met. It is worthwhile in our context to notice that since Lagrangian penalties are related to dual variables of the problem to solve, the algorithm maintains a set of dual variables of the dualized constraints, i.e., a set of primal variables. In this case, it has been observed that when "no further improvement is found, the values (of the primal variables) should approximate an optimal solution."
- *Bundle methods* are methods where subgradient directions from past iterations are accumulated in a bundle, and a trial direction is usually obtained by solving a quadratic programming model based on the information contained in the bundle.

A line search can then be performed along the tentative direction, generating a non-null step if the function value is improved by some positive value, a null step otherwise.

Based on these cores, the literature reporting applications of algorithms, which are structured like Algorithm 35, is rich and includes contributions that expand the area of optimization in general, not being limited to the focus on heuristics. In this section, we limit to reporting about some Lagrangian heuristics proposed specifically for the GAP.

A heuristic based on the subgradient optimization of the Lagrangian relaxation of the GAP was proposed in two publications, which work on different relaxations: a Lagrangian decomposition separating two subproblems and keeping both constraint sets in the first case and a relaxation of the capacity constraints in the second one, but the heuristic is very similar in both cases. The heuristic is applied at each iteration of the subgradient algorithm and tries to construct a feasible client assignment by a greedy approach based on the ordering obtained using increasing costs $f_{ij} = c_{ij} - q_{ij}\lambda_i$, where λ_i is the incumbent penalty associated with the i-th relaxed capacity constraint. If all clients can be assigned, then the solution is feasible; otherwise, every unassigned client j is tentatively assigned to the server i to which corresponds the maximal value of f_{ij}. Next, an interchange heuristic tries to transform this assignment into a feasible one. This is done by a simple fixing heuristic, which computes a (non-negative) feasibility measure defined to be the sum of the resource shortages of the servers, and attempting to identify a better solution either by interchanging two client/server assignments or by shifting a client to another server. If a feasible assignment is obtained, the same interchange heuristic is used as a local optimization procedure.

The possibility of recovering the feasibility of the Lagrangian bound solution has been studied by including a data perturbation module, denoted as *problem space search* in the original work, where the authors relax the capacity constraints, run a subgradient, and try to restore feasibility of the bound solution by means of a greedy constructive heuristic that assigns the clients one by one to servers with enough capacity, making sure that the overall assignment is always feasible. Two different orders of assignment of the clients and the criterion used to assign are presented, but the main contribution is in the data perturbation component. In fact, since it is possible that the feasibility restoration heuristics stop without finding a feasible solution, the penalized costs are temporarily perturbed and the constructive heuristics are run on the modified costs, in the hope that the new cost configuration permits the construction of solutions that are feasible for the original instance. The perturbation is applied in each iteration of subgradient optimization for a given number of times.

Finally, it is worth mentioning here a work not directly applied to the GAP but to the *many-to-many assignment problem*, a problem with a formulation very similar to the GAP, except that the assignment constraints are replaced by knapsack constraints, i.e., clients have capacities and multiple servers can partially serve one same client. The work proposes a modified Lagrangian bound and a greedy heuristic to get a feasible Lagrangian-based solution, a heuristic that relies on the observation that in this problem all constraints are actually inequalities.

7.4 A Lagrangian Matheuristic for the GAP

Algorithm 35 is a generic framework for combinatorial optimization problems and must be detailed for the specific problem to solve, in our case the GAP. Steps 3 and 5 are, in fact, non-trivially problem-dependent, while steps 6, 7, and 8 also depend on the problem but are expected in whichever algorithm is used to solve it. Step 6 is trivial, while step 7 can be implemented by means of any state-of-the art techniques. Step 8 is where the actual primal solution is found. The end condition is usually set on the number of iterations of the *repeat-until* loop, on a maximum CPU time, on a maximum number of non-improving solutions, or on the minimal value of some internal coefficient, such as the step size coefficient α used in step 7.

In our case, these problem-dependent steps were implemented as follows.

LagrHeuristic: Step 3
Section 1.2.2 introduced the two possible Lagrangian relaxations for the GAP: relaxing assignment constraints (1.2) or capacity constraints (1.3). The second possibility leaves a trivial subproblem, which can be solved by inspection but which cannot provide bounds better than the linear relaxation of the original GAP. Relaxing the assignment constraints on the contrary leaves a combinatorial subproblem; therefore, the Lagrangian solution can be better than the linear relaxation.

In this subsection we show a detailed trace of the execution of a Lagrangian matheuristic implementing the second possibility, i.e., solving a subproblem derived from the relaxation of the assignment constraints. We copy here the formulation, already presented in Sect. 1.2.2, for ease of reference.

$$(LGAPA) \qquad z_{LGAPA} = min \sum_{i \in I} \sum_{j \in J} (c_{ij} + \lambda_j) x_{ij} - \sum_{j \in J} \lambda_j \tag{1.5}$$

$$s.t. \sum_{j \in J} q_{ij} x_{ij} \le Q_i, \qquad i \in I \tag{1.6}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in I, j \in J \tag{1.7}$$

$$\lambda_j \text{ unrestricted}, \qquad j \in J \tag{1.8}$$

LagrHeuristic: Step 5
The subproblem decomposes into $|I|$ knapsack problems, with objective function

$$z_K^i = \sum_{j \in J} (c_{ij} + \lambda_j) x_{ij}$$

where the Lagrangian penalties λ_j are associated with the assignment constraints (1.2). The subproblem solution is simply given by setting $x_{ij} = 1$ for every element j that is chosen to enter the i-th knapsack.

LagrHeuristic: Step 6
The solution can have clients assigned to multiple or to no server, and the case can be ascertained by direct inspection.

LagrHeuristic: Step 7
The quality of the bound obtained by the Lagrangian relaxations heavily depends on the specific penalty vector $\boldsymbol{\lambda}$ used. The problem of finding a penalty vector that maximizes the Lagrangian bound is the Lagrangian Dual problem. In our case, we used a standard subgradient procedure for determining the penalties. The procedure computes, at each iteration k, a new approximation $\boldsymbol{\lambda}^{k+1}$ of the Lagrangian multipliers in such a way that, for $k \to \infty$, $\boldsymbol{\lambda}^k$ is an optimal or a near optimal solution of the corresponding Lagrangian Dual.

The subgradient algorithm implements a local search in the space of the Lagrangian penalties, where the direction of the move at each iteration k is given by a subgradient $\boldsymbol{g}^k$, and the length of the move, i.e., the step size, is dictated by a scalar α^k as follows:

$$\sigma^k = \alpha^k \frac{\bar{z} - z_{LR}(\boldsymbol{\lambda}^k)}{\left\|\boldsymbol{g}^k\right\|^2} \tag{7.39}$$

where $\bar{z}$ is an overestimate of the optimal Lagrangian Dual solution value $z_{LR}(\boldsymbol{\lambda}^*)$, possibly an upper bound to the optimal problem cost, and the components of the subgradient are computed as $g_j^k = \sum_{i \in I} x_{ij} - 1$. In the literature, several different step size rules have been proposed, each one with different convergence results. We adapted in formula (7.39) the seminal rule proposed by Polyak (1969). The step size coefficient α^k is updated following a *diminishing step size* policy.

Given an incumbent solution $\boldsymbol{x}^k$, the corresponding bound is $z_{LR}(\boldsymbol{\lambda}^k) = z_{\text{LGAPA}}(\boldsymbol{\lambda}^k)$. Having these, the Lagrangian multipliers can be updated by setting

$$\boldsymbol{\lambda}^{k+1} = \boldsymbol{\lambda}^k + \sigma^k \boldsymbol{g}^k, \tag{7.40}$$

where σ^k is the length of the step along the search direction given by the subgradient.

LagrHeuristic: Step 8
The solutions provided by the inner loop of Algorithm 35 are often infeasible for the reasons mentioned in step 6. The problem of recovering feasibility from a partial solution could be framed as a GAP itself, but this would complicate the solution approach. Moreover, it is known that searching for a GAP feasible solution, or fixing a partially feasible one, is also an NP-hard problem as it is NP-complete, the decision problem of determining whether a GAP instance has a feasible solution

or not (see Chap. 1). We, therefore, used the approach of Sect. 3.7.1 as detailed in the following Sect. 7.4.1, which can handle both causes of infeasibility.

In case a feasible solution is found, the Lagrangian heuristic passes it on to a simple local search. The local optimization can be applied to every feasible solution found, and it reassigns, when feasible and convenient, the clients in decreasing regret order to the servers.

After including all elements listed in the previous section into Algorithm 35, we obtain the following Algorithm 36, which describes a Lagrangian matheuristic specialized for the GAP.

Algorithm 36: Lagrangian matheuristic for the GAP

```
1  function LagrHeuristicGAP($\alpha$);
   Input  : $\alpha$ (see formula (7.40))
   Output: A feasible solution $\mathbf{x}^*$ of value z*
2  set $z_{lb} = z_{lbBest} = -\infty$;            // iteration and best lower bounds
3  set $z_{curr} = z_{ub} = \infty$;              // iteration and best upper bounds
4  Initialize the penalty vector $\boldsymbol{\lambda}$;          // for example to a zero vector
5  Use problem LGAPA (see Sect. 1.2.2) as subproblem LR($\boldsymbol{\lambda}$);
6  repeat
7      solve subproblem LR($\boldsymbol{\lambda}$) by solving the $m$ knapsack problems it is composed by,
         obtain iteration bound $z_{lb}$ and solution $\mathbf{x}_{lb}$;
8      if $z_{lb} < z_{lbBest}$ then $z_{lbBest} = z_{lb}$;          // lower bound update
9      check for unsatisfied assignment constraints;
10     construct problem solution $\mathbf{x}_h$ using $\mathbf{x}$ and $\boldsymbol{\lambda}$ by means of Algorithm 20;
11     $z_{curr} = z(\mathbf{x}_h)$;                      // $z(\mathbf{x}_h)$ is the cost of $\mathbf{x}_h$
12     if $z_{curr} < z_{ub}$ then                     // upper bound update
13         $\mathbf{x}^* = \mathbf{x}_h$; $z_{ub} = z_{curr}$;
14     end
15     update penalties $\boldsymbol{\lambda}$ by formula (7.40);
16 until end_condition;
```

7.4.1 The Fixing Heuristic

The fixing heuristic we applied within $LagrHeuristicGAP()$ is Algorithm 20, presented in Sect. 3.7.1, which is designed to aggressively try to produce feasible solutions even starting from heavily infeasible ones, and to be computationally reasonably light. This comes at a cost, in this case of being unable to recover feasibility in several cases where a more sophisticated algorithm would be able to. However, the primary concern of algorithm $LagrHeuristicGAP()$ is to improve the bound, and this is expected to be achieved by a high number of internal iterations; therefore, it is sufficient to get feasible solutions out of only a percentage of them to ensure the effectiveness of the approach.

This fixing heuristic could be used both to repair infeasibilities deriving from the relaxation of constraints (1.2) and (1.3). We detail it in Algorithm 20 for the general case where both assignment and capacity constraints could be unsatisfied in the current Lagrangian solution, even though in the context of this section only assignments can make the solution infeasible.

7.4.2 LagrHeuristic: Computational Trace

We implemented Algorithm 36 coupled with Algorithm 20 and ran it on instance *example8x3* of Sect. 1.1. The instance is a simple one, and Algorithm 36 was able to solve it to prove optimality, though this is seldom the case with hard instances. The reported run had parameter α (see Eq. 7.39) initialized to 2.5.

The first iteration does not assign any client to any server, since any assignment would incur a positive cost. Therefore at iteration 1 we had $z_{lb} = z_{lbBest} = 0.00$, and the subgradient vector is $\mathbf{g}^1 = (1, 1, 1, 1, 1, 1, 1, 1)$, as every assignment constraint should sum to 1 but sums to 0. The subproblem solution

$$\mathbf{x}_{lb} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{7.41}$$

(null assignment) is passed on to Algorithm 20, which is very aggressive in its tentative to recover feasibility, and manages to construct the solution reported in (7.42) of cost 336; therefore, $z_{ub} = z_{curr} = 336$.

$$\mathbf{x}_{h} = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \tag{7.42}$$

Since $z_{LR}(\boldsymbol{\lambda}^1) = 0$ and there is a big gap between upper and lower bounds—say, big means that the upper bound is more than 20% over the lower bound—it is computationally convenient to modify formula (7.39) and use at the numerator of the fraction a value not much greater than the lower bound, for example, $1.2 * z_{lb} + 1$, instead of the difference among the bounds. We therefore obtain $\bar{z} = 1$, and consequently, $\sigma^1 = 2.5 * 1/8 = 0.313$. The final step of the iteration updates all multipliers starting from 0 with the product of σ^1 and the respective subgradient, which is 1 for all of them. Therefore we get $\boldsymbol{\lambda}^1 = (0.313, 0.313, 0.313, 0.313, 0.313, 0.313, 0.313, 0.313)$.

Iteration 2 is similar to iteration 1, except for the higher initial multiplier values. The Lagrangian bound is given by the sum of the penalties; therefore, $z_{lb} = z_{lbBest} = 8 * 0.313 = 2.5$. The bound solution is still the null solution; therefore, $z_{ub} = 336$ and $\alpha = 2.5$, unchanged. The subgradient vector is again $\mathbf{g}^2 = (1, 1, 1, 1, 1, 1, 1, 1)$, leading to $\boldsymbol{\lambda}^2 = (0.625, 0.625, 0.625, 0.625, 0.625, 0.625, 0.625, 0.625)$ with $\sigma^2 = 0.313$.

Penalties keep uniformly summing up until iteration 10, which is the first where the sum of the penalties exceeds the cost of an assignment (for client 1 to server 1). This leads the first row of the cost matrix, containing costs minus sum of penalties for server 1 (see Eq. 1.5), to become $\mathbf{c}^{[1]} = [-0.68, 0.32, 1.32, 2.32, 3.32, 4.32, 5.32, 6.32]$, so that the subproblem solution results to be

$$\mathbf{x}_{lb} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{7.43}$$

of cost $z_{lb} = z_{lbBest} = 84.751$ and thus subgradient vector $\mathbf{g}^{10} = (0, 1, 1, 1, 1, 1, 1, 1)$.

The bound solution is infeasible, and it is fixed obtaining again upper bound $z_{ub} = 336$. This is still too different from the lower bound to be used as $\bar{z}$, and the penalties become $\boldsymbol{\lambda}^{10} = (10.679, 16.732, 16.732, 16.732, 16.732, 16.732, 16.732, 16.732)$ with $\sigma^{10} = 6.054$.

We have to wait until iteration 44 in order to have an improvement of the upper bound. At this iteration, the subproblem yields a cost $z_{lb} = 255.808$, even though previously we got a $z_{lbBest} = 275.779$, which testifies that the bound is not subject to a monotonic increase. The bound solution is

$$\mathbf{x}_{lb} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{7.44}$$

which is converted by Algorithm 20 into the feasible solution

$$\mathbf{x}_{h} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{7.45}$$

of cost $z_{curr} = z_{ub} = 325$. At this iteration, we have $\alpha = 2.25$ and $\sigma^{44} = 8.222$. Given solution $\mathbf{x}_{lb}$, the subgradient vector at this iteration is $\mathbf{g}^{44} = (1, 1, 1, -1, 1, -2, -2, 1)$, computed on solution (7.44).

Finally, at iteration 830, the subproblem produces the solution

$$\mathbf{x}_{lb} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \tag{7.46}$$

which is still infeasible, but it has a cost $z_{lb} = z_{lbBest} = 324.327$. This guarantees that an optimal solution must have a cost of at least 325, given the integrality constraints and the fact that all costs are integer. Since we already found at iteration 44 a feasible solution of cost 325, we have the proof that the best found solution is indeed an optimal one, and we can stop the search.

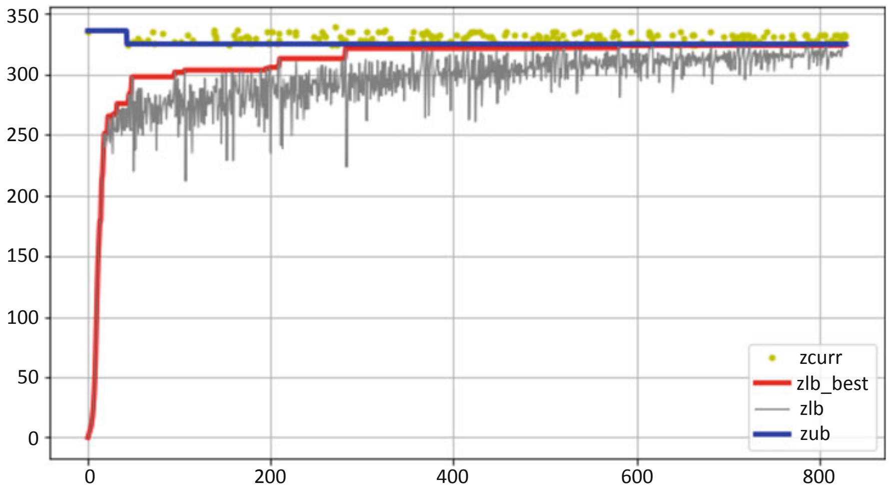


Fig. 7.1 Bound traces

To complete the presentation of this run, Fig. 7.1 shows the trace of the evolution of the upper and lower bounds. It is apparent how the Lagrangian lower bound shows a non-monotonic increase along the iterations, with decreasing variance, hence the need to store a (monotonically increasing) z_{lbBest}. The upper bound z_{curr} is not produced at each iteration, and dots show it in correspondence to the iterations where it could be found. In this run, the subproblem was never able to produce a solution feasible for the whole problem (which would have been optimal), and the fixing heuristic, though aggressive, is often unable to start from the infeasible Lagrangian solution and bring it to global feasibility.

Moreover, the z_{ub} trace shows the evolution of the cost of the best-so-far solution, which is monotonically non-increasing. At the last iteration, the value of the costs of the Lagrangian solution and of the best-so-far solution differs for less than a unit, indicating that the best so far is also an optimal solution.

Further insight can be obtained by visualizing the traces of relevant parameters along the run. Figure 7.2 shows the evolution of the values taken by each Lagrangian multiplier, and they are 8 in the case of this instance. While the evolution of every single multiplier cannot be discerned, it is interesting to notice that all of them follow similar trajectories, where at the beginning the penalties are quickly carried to values that permit to produce tight lower bounds and, afterward, they are slowly fine-tuned each one with respect to all others in order to get the tightest possible bound.

Finally, Fig. 7.3 shows the evolution of the step length $\boldsymbol{\sigma}^k$, which determines the magnitude of the multiplier update internally to the subgradient optimization algorithm. This result is obtained by formula (7.39), using the value of the α calling argument and of the subgradient vector of the current iteration.

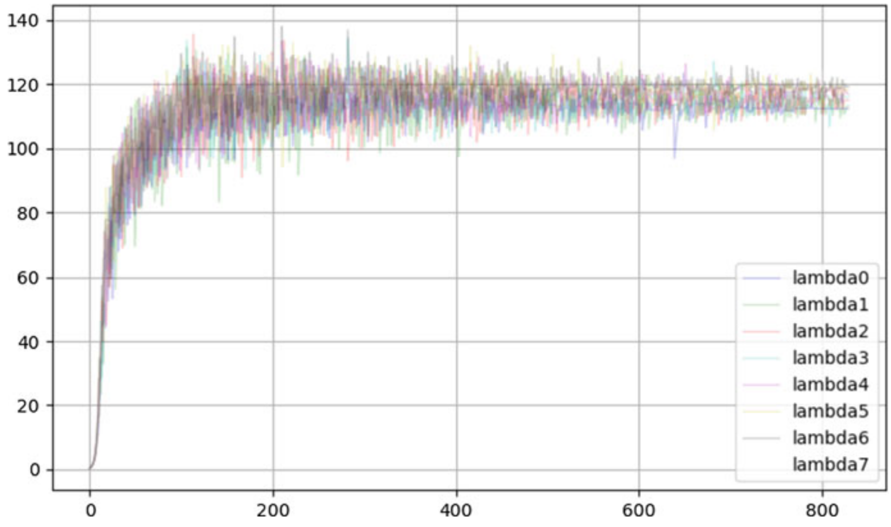


Fig. 7.2 Penalty traces

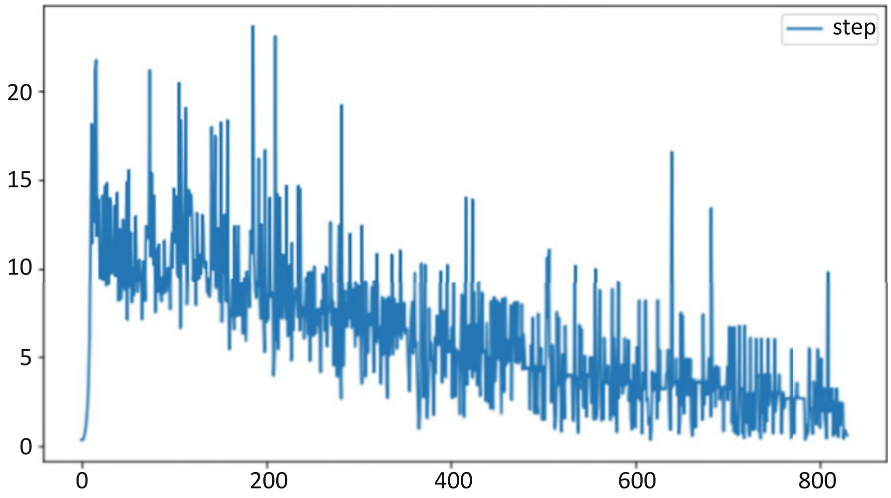


Fig. 7.3 Subgradient step length trace

Again, we can notice a very fast increase in useful values, followed by a slow, non-monotonic decrease that permits the cautious update of the multipliers and, ultimately, the identification of the optimality bound.

7.5 Related Literature

A good introduction to the topic of decompositions in mathematical programming is in Bazaraa et al. (1990). Reviews of the literature presenting decomposition-based matheuristics can be found in Boschetti and Maniezzo (2009), Boschetti et al. (2009), and Raidl (2015).

Dantzig–Wolfe decomposition was originally introduced in Dantzig and Wolfe (1960) and Benders decomposition in Benders (1962).

An excellent introduction to the whole topic of Lagrangian relaxation, and of related heuristics, can be found in Beasley (1993). Approaches obtaining dual values from Lagrangian penalties are reported in Boschetti et al. (2008), Boschetti and Maniezzo (2015), Boschetti et al. (2020), Mingozzi et al. (1999). Results on the convergence on optimal primal values of the linear relaxation were published in Barahona and Anbil (2000) and Sherali and Choi (1996).

Subgradient optimization is presented by Shor (1985), multiplier adjustment by Fisher et al. (1986), the volume algorithm by Barahona and Anbil (2000), and bundle methods by Hiriart-Urruty and Lemarechal (1993). The reported subgradient optimization of the Lagrangian relaxation of the GAP was proposed in Haddadi (1999) and in Haddadi and Ouzia (2001) and the fixing of the Lagrangian bound in Jeet and Kutanoglu (2007). The fixing used in the example was originally presented in Maniezzo et al. (2019). The work on the many-to-many assignment problem is by Litvinchev et al. (2010).

Subgradient step size rules have been reviewed in Boschetti et al. (2011) and Held et al. (1974).

References

Barahona F, Anbil R (2000) The volume algorithm: producing primal solutions with a subgradient method. Mathematical Programming 87:385–399

Bazaraa MS, Jarvis J, Sherali HD (1990) Linear programming and network flows. Wiley

Beasley JE (1993) Lagrangian relaxation. In: Modern heuristic techniques for combinatorial problems, Reeves, C.R. Blackwell Scientific Publ., pp 243–303

Benders JF (1962) Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik 4:280–322

Boschetti M, Maniezzo V, Roffilli M (2009) Decomposition techniques as metaheuristic frameworks. In: Maniezzo V, Stützle T, Voß S (eds) Matheuristics. Annals of information systems, vol 10. Springer, Boston, MA

Boschetti MA, Maniezzo V (2009) Benders decomposition, Lagrangian relaxation and metaheuristic design. J Heuristics 15(3):283–312

Boschetti MA, Maniezzo V (2015) A set covering based matheuristic for a real-world city logistics problem. Int Trans Oper Res 22(1):169–195

Boschetti MA, Mingozzi A, Ricciardelli S (2008) A dual ascent procedure for the set partitioning problem. Discrete Optimization 5(4):735–747

Boschetti MA, Maniezzo V, Roffilli M (2011) Fully distributed Lagrangian solution for a peer-to-peer overlay network design problem. INFORMS J Comput 23(1):90–104

Boschetti MA, Golfarelli M, Graziani S (2020) An exact method for shrinking pivot tables. Omega 93:10–44

Dantzig GB, Wolfe P (1960) Decomposition principle for linear programs. Operations Research 8:101–111

Fisher ML, Jaikumar R, Van Wassenhove LN (1986) A multiplier adjustment method for the generalized assignment problem. Management Science 32(9):1095–1103

Haddadi S (1999) Lagrangian decomposition based heuristic for the generalized assignment problem. Inf Syst Oper Res 37:392–402

Haddadi S, Ouzia H (2001) An effective Lagrangian heuristic for the generalized assignment problem. INFOR Inf Syst Oper Res 39:351–356

Held M, Wolfe P, Crowder HP (1974) Validation of subgradient optimization. Mathematical Programming 6(1):162–88

Hiriart-Urruty JB, Lemarechal C (1993) Convex analysis and minimization algorithms II: Advanced theory and bundle methods. A series of comprehensive studies in mathematics, vol 306

Jeet V, Kutanoglu E (2007) Lagrangian relaxation guided problem space search heuristics for generalized assignment problems. Eur J Oper Res 182(3):1039–1056

Litvinchev I, Mata M, Rangel S, Saucedo J (2010) Lagrangian heuristic for a class of the generalized assignment problems. Comput Math Appl 60(4):1115–1123

Maniezzo V, Boschetti MA, Carbonaro A, Marzolla M, Strappaveccia F (2019) Client-side computational optimization. ACM Trans Math Softw (TOMS) 45(2):1–16

Mingozzi A, Boschetti MA, Ricciardelli S, Bianco LA (1999) Set partitioning approach to the crew scheduling problem. Operations Research 47:873–888

Polyak BT (1969) Minimization of unsmooth functionals. USSR Comput Math Math Phys 9:14–29

Raidl G (2015) Decomposition based hybrid metaheuristics. Eur J Oper Res 244:66–76

Sherali HD, Choi G (1996) Recovery of primal solutions when using subgradient optimization methods to solve Lagrangian duals of linear programs. Oper Res Lett 19:105–113

Shor NZ (1985) Minimization methods for non-differentiable functions. Springer

Chapter 8
Corridor Method

8.1 Introduction

The corridor method (CM) is a general method originally proposed as a way to gain efficiency in dynamic programming (DP) search, possibly losing optimality. Later, it has been extended beyond DP to other exact optimization methods.

In its general form, CM assumes to be requested to solve a possibly NP-hard optimization problem, for which we know an exact method that could effectively solve it on not too large instances. However, real-world instances of interest are usually too big to ensure the possibility of getting an optimal solution within an acceptable time, and the direct application of the exact method becomes impractical. Here comes CM into play, whose basic idea is that of using the exact method over successive restricted portions of the solution space of the given problem. The restriction is obtained by applying exogenous constraints, which define local neighborhoods around points of interest.

CM is very general. The basic structure of the neighborhoods to explore is determined by the needs and requirements of the optimization method used for searching. For example, if the optimization method is dynamic programming (DP), then the structure of the neighborhoods will be suitable for a DP treatment and will typically constrain DP state trajectories. Otherwise, in the case of branch and bound, neighborhoods will be constructed around the incumbent solutions, and they do not need to be limited by constraints on the number of variables that can be changed as in local branching. In any case, the restrictions are obtained by adding exogenous constraints, defining neighborhoods that are typically exponentially large but designed in such a way that the chosen exact method can efficiently explore the restricted sub-instances.

The name “corridor” comes from the first application of the method, which made use of DP. In that case, exogenous constraints were used to control the state trajectory followed by DP using search, limiting its erratic possibilities. The trajectory could not change too much from its past path, and it was constrained

V. Maniezzo et al., *Matheuristics*, EURO Advanced Tutorials
on Operational Research, https://doi.org/10.1007/978-3-030-70277-9_8

in its progression as when walking along a corridor. In this metaphor, the corridor identifies the neighborhood the exact solver is forced to move along. In this way, the number of states generated from the current configuration is limited. This refers to DP, but the same idea can be applied, for example, to MIP, where the corridor can be defined around incumbent solutions, and the solver is forced to move along a trajectory connecting successive local optima solutions. In this case, the CM represents a further variation of the idea of solving to optimality a neighborhood of the incumbent solution. More in general, given the exact optimization method that will be used to solve the restricted subproblem, the algorithm designer is required to specify a way in which neighborhoods can be defined so that the chosen method can be used to efficiently solve the subproblems that will be encountered.

This core difference between CM and more standard local optimization algorithms has been remarked by denoting the standard local search as *"move-based,"* meaning that they are based on moves, i.e., small structural variations of the incumbent solution, while CM deploys neighborhoods that are *method-based*, which means that neighborhoods must be designed on the basis of the working of the exact technique that will be used for solving the subproblems.

A pseudocode of the CM is presented as Algorithm 37. This has a very general form, which does not specify the method used to solve the restricted subproblem, whose search space $\mathscr{N}(\mathbf{x})$ gets defined around an incumbent solution $\mathbf{x}$. The execution is affected by the control parameter δ_{max}, which specifies the maximum "width of the corridor," i.e., the maximum size of the subproblems passed on to solve to the exact method.

The algorithm optimizes the subproblems along a corridor constructed around the state trajectory generated by the successive incumbent feasible solutions. The best solutions found in the corridor become the new incumbent solutions at the next iteration. This process is repeated until no improving incumbent solution can be found, and the procedure either stops or somehow generates a new seeding incumbent solution, and the search process restarts.

The pseudocode presented in Algorithm 37 makes use of a further feature, the use of a dynamic corridor width. It is possible to adapt the width of the corridor following the presence of improving solutions in the examined neighborhood. If an improving solution is found in the neighborhood, the incumbent solution is updated, and a new corridor is defined around this new solution. Otherwise, the width of the corridor is widened, in the hope of finding improving solutions.

The seminal CM paper by Sniedovich and Voß did not contain any actual experiments. It mentioned good performance of simplified versions as a DP assistant, already presented in the literature in the context of reservoir control and operation problems.

The first application where CM was explicitly tested was the block relocation problem, with emphasis on application in the stacking of container terminals in a yard. In this problem, an initial collection of stacks of blocks is given, for example, containers in a port terminal together with a pickup list. Blocks have to be picked up following the given sequence. For each block to be picked, if there are other blocks above it, the pickup operation involves the relocation of the overlapping blocks

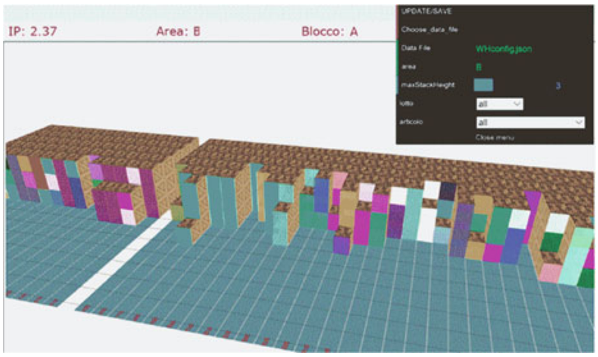
Algorithm 37: Generic corridor method

```
1  function CorridorMethod($\delta_{max}$);
   Input  : Control parameters $\delta_{max}$
   Output: A feasible solution x
2  $i = 0$;                                            // global iteration counter
3  while not(termination condition) do
4      generate incumbent feasible solution $\mathbf{x}_i$;
5      $\delta = \delta_{max}$;                        // set initial value of $\delta$
6      while $\delta > \delta_{min}$ do
7          $\mathbf{x}^* = \min \mathcal{N}(\mathbf{x}_i)$;    // apply exact method on $\mathcal{N}(\mathbf{x}_i)$
8          if $f(\mathbf{x}^*) < f(\mathbf{x}_i)$ then
9              $\mathbf{x}_i = \mathbf{x}^*$; $\delta = \delta_{max}$;    // update incumbent
10         else
11             decrease $\delta$;                     // tighten the corridor
12         end
13     end
14     $i = i + 1$;
15 end
```

into other stacks. The same problem arises in the management of block stacking warehouses, where items (e.g., pallets) are simply stacked on top of one another, with no supporting infrastructure. Figure 8.1 shows a screenshot of a warehouse management system representing a stacking warehouse. Here stacks are divided into successive substacks where only the topmost block of each first substack can be accessed. Each floor strip identifies a stack.

The blocks relocation problem requires to find the relocation sequence for each pickup operation so that the number of future relocation moves for accessing blocks of interest is minimized.

The application is based on a DP recursion that identifies all possible relocation sequences that can be generated following the known picking list. There is an exponential growth of the number of possible states, so CM is applied to the DP



Data File WHconfig.json
area B
maxStackHeight 3
lotto all
articolo all

Fig. 8.1 Stacking warehouse

recursion, where the corridor imposes limitations on the number of destination stacks that can be considered when relocating a block. This means that the corridor around the incumbent configuration is defined by imposing exogenous constraints on the solution space of the problem itself. The proposed method has been compared with the best code in the literature. The results showed that the CM approach is effective in finding optimal solutions in short computational time for small- and medium-size instances and in improving the quality of solutions for large-scale instances.

A closely related problem is warehouse pre-marshalling, where we still have a block stacking warehouse and a picking list, but here we are asked to sort the initial configuration using a minimum number of relocations so that no—or as few as possible—new relocations will be needed when blocks will have to be picked. This is in contrast to the block relocation problem, where the objective was to retrieve the blocks according to the picking list and using a minimum number of relocations. A similar application has been described for pre-marshalling, where the optimization was based on GRASP and the corridor was again defined on a subset of stacks to be considered as candidates for relocations. The approach was later extended, including a statistical estimator that can account for uncertainties in the picking lists that will be received after pre-marshalling.

Another application presented by Caserta and Voß was about DNA sequencing, a problem asking to determine the order in which sequences of nucleotides appear in an unknown fragment of the DNA. The problem has some similarities with the TSP, and, in fact, the authors propose to model it as an orienteering problem. It is thus possible to use a known MILP formulation of the orienteering problem as the representation to solve it by an exact algorithm based on a MIP solver, which typically makes use of a branch and cut solution method. The corridor is here defined by means of further cuts added to the problem formulation, cuts similar to the local branching ones described in Chap. 5 of this book. The peculiarity of the CM implemented in this work is the use of a dynamic corridor width. This method has been tested on standard benchmark instances, showing to be able to find optimal or near-optimal solutions on all of them. A similar approach was also used for solving the capacitated lot-sizing problem.

8.2 Corridor Method for the GAP

There are no applications specific to the GAP described in the CM literature. However, a closely related effort has been made for the capacitated facility location problem (CFLP). In this work, the authors base their solution approach on a Lagrangian relaxation of the CFLP capacity constraints, thus obtaining an uncapacitated facility location problem. The Lagrangian penalties were then optimized by using subgradient optimization, which identifies a set of Lagrangian multipliers and solves an uncapacitated FLP Lagrangian subproblem at each iteration (see Chap. 7).

The method requires to add a corridor constraint, which in this application is added with respect to the solution of the Lagrangian subproblem. The neighborhood around the solution imposes a maximum Hamming distance with respect to the binary variables that determine whether a facility (a server in GAP terminology) can be used or not. Upon denoting by y_i^L the binary variable declaring whether facility i is used or not in the Lagrangian subproblem solution and y_i the binary variable declaring whether facility i is used or not in the CFLP solution, $i = 1, \dots, m$, the corridor constraint is

$$\sum_{i=1}^{m} \left[y_i^L(1 - y_i) + (1 - y_i^L)y_i \right] \leq \lfloor \delta m \rfloor \tag{8.1}$$

where $0 < \delta \leq 1$ defines the width of the corridor and, consequently, the size of the neighborhood to be explored.

Constraint (8.1) defines a restricted instance of CFLP, which is then solved, not necessarily to optimality, using a MIP solver. If an improved solution is found, the current corridor is removed from the solution space, and a new corridor is defined around the new incumbent solution. These steps are repeated until a stopping criterion is met. This is a heuristic, primal phase. When it terminates, a new set of Lagrangian multipliers is obtained via subgradient optimization, and the cycle is repeated. We do not propose here the adaptation of this approach to GAP, as it implies an interplay of the corridor method with the Lagrangian decomposition approach described in Chap. 7, but we point out that the adaptation would not be too demanding.

Algorithm 38 proposes a simpler corridor approach, where the corridor specifies which servers must keep the same client assignments and which are allowed to exchange clients. An internal call to a MIP solver tries to solve to optimality the reduced problem obtained after applying the corridor constraint. The exact search is allowed to use only limited computational resources (time, tree-search nodes, memory, etc.), and therefore there is no guarantee that the reduced GAP instance will be solved to optimality or that it will be solved at all.

Algorithm 38 was applied to GAP instance *example8x3* of Sect. 1.1 with parameters $\delta_{max} = 4, \delta_{min} = 3$, and $maxres$ the maximum number of nodes which were allowed to be expanded during corridor search, set to 10000. The initial solution, obtained at step 3, was the one obtained by simple construction: $\boldsymbol{\sigma} = (0, 0, 0, 1, 1, 2, 2, 2)$ of cost 343.

The first iteration of cycle at step 5 was run with a randomly defined corridor of width $\delta = 4$, extracted from the index set of the columns (the clients), which resulted to be $C = \{0, 2, 5, 6\}$.

The corridor was enforced by fixing the variables out of the corridor to the values taken in the incumbent solution, which is in this case the first constructed one. This was obtained by fixing both the lower and upper bounds of the variables out of the corridor to the desired value. The resulting lower bounds (LBs) and upper bounds (UBs) to the variable values are presented in matrices (8.2). Note how columns

Algorithm 38: Corridor method for the GAP

```
1  function CorridorGAP($\delta_{max}$, $\delta_{min}$, $maxres$);
   Input  : Control parameters $\delta_{max}$, $\delta_{min}$, $maxres$
   Output: A feasible solution $\mathbf{x}^*$
2  $i = 1$;                                          // global iteration counter
3  set incumbent solution $\mathbf{x}_i$ to any feasible solution;
4  $\delta = \delta_{max}$;                          // set initial value of $\delta$
5  while not(termination condition) do
6      generate corridor $C$ of size $\delta$;       // C has a search space $\mathcal{N}(\mathbf{x}_i)$
7      $\mathbf{x}^* = min_{maxres}\ \mathcal{N}(\mathbf{x}_i)$; // apply constrained exact method on $\mathcal{N}(\mathbf{x}_i)$

8      if $f(\mathbf{x}^*) < f(\mathbf{x}_i)$ then
9          $\mathbf{x}_i = \mathbf{x}^*$ and go to 4;               // update incumbent
10     else
11         if $\delta > \delta_{min}$ and not($\mathcal{N}(\mathbf{x}_i)$ explored to optimality) then
12             decrease $\delta$ and go to 7;           // tighten the corridor
13         else
14             discard incumbent and go to 3;
15         end
16     end
17     $i = i + 1$;
18 end
```

0, 2, 5, and 6 have the lower bounds at 0 and the upper bounds at 1, while the complementary columns 1, 3, 4, and 7 have the same value in the corresponding positions of the two matrices.

$$LB = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad UB = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{8.2}$$

The subproblem was solved by a MIP solver, starting from the LP relaxation of the constrained subproblem. The LP solution is reported in (8.3) and has a cost of 258.565.

$$\bar{\mathbf{x}}_0 = \begin{bmatrix} 0 & 1 & 0.609 & 0 & 0 & 1 & 1 & 0 \\ 0.182 & 0 & 0.391 & 1 & 1 & 0 & 0 & 0 \\ 0.818 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.3}$$

Building on this, MIP could be solved to integer optimality, obtaining the solution (8.4) of cost 331.

$$\mathbf{x}_0 = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{8.4}$$

This solution, which can be expressed as $\boldsymbol{\sigma}_0 = (0, 0, 2, 1, 1, 2, 0, 2)$, is the new best found solution.[1]

Next comes iteration 2 of the cycle at step 5, which works on corridor $C = \{1, 4, 6, 7\}$. In this case the LP solution value is 262.455 for the solution (8.5).

$$\bar{\mathbf{x}}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0.659 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0.341 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{8.5}$$

The corresponding optimal MIP Solution has a cost of 328 and is reported in (8.6). The solution is $\boldsymbol{\sigma} = (0, 1, 2, 1, 0, 2, 0, 2)$ and corresponds to the new best found solution.

$$\mathbf{x}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{8.6}$$

At iteration 3 of the cycle at step 5, the corridor was $C = \{1, 4, 5, 6\}$. The LP solution is reported in (8.7) and has a cost of 269.182.

$$\bar{\mathbf{x}}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0.614 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0.386 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.7}$$

The corresponding optimal MIP Solution has a cost of 328 and is reported in (8.8). This is the same incumbent solution already found at the previous iteration, $\boldsymbol{\sigma} = (0, 1, 2, 1, 0, 2, 0, 2)$; no improvement was achieved.

$$\mathbf{x}_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{8.8}$$

[1]We remind that in the computational traces, the decision variables are indexed from 0.

At iteration 4 of the cycle at step 5, the corridor was $C = \{0, 1, 4, 6\}$. The LP solution is reported in (8.9) and has a cost of 320.25.

$$\bar{\mathbf{x}}_3 = \begin{bmatrix} 0.562 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0.437 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{8.9}$$

The corresponding optimal MIP Solution has a cost of 327 and is reported in (8.10). This solution, $\boldsymbol{\sigma} = (1, 0, 2, 1, 0, 2, 0, 2)$, is the new best found solution.

$$\mathbf{x}_3 = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{8.10}$$

Finally, at iteration 5 of the cycle at step 5, the corridor was $C = \{1, 3, 5, 7\}$. The LP solution is reported in (8.11) and has a cost of 237.872.

$$\bar{\mathbf{x}}_4 = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0.767 & 1 & 1 \\ 1 & 0.252 & 0 & 1 & 0 & 0.236 & 0 & 0 \\ 0 & 0.748 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{8.11}$$

The corresponding optimal MIP Solution has a cost of 325 and is reported in Eq. (8.12). Again, this solution, $\boldsymbol{\sigma} = (1, 1, 2, 0, 0, 2, 0, 2)$, is the new best found solution.

$$\mathbf{x}_4 = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \tag{8.12}$$

This simple trace shows how the corridor method can give rise to a sequence of small partial improvements that lead from an initial solution to a very different, better one. There is clearly no guarantee that improvements will be continuative nor that they will be there at all, but a good gauging of the algorithm parameters can often lead to effective search. This example deals with a very simple instance, so the more advanced features that permit to make use of incomplete exact search do not kick in, but in general, it must be accepted that the exact search does not solve to optimality the constrained subproblems; therefore, a mechanism to circumvent this limitation is needed.

8.3 Related Literature

The corridor method was originally proposed by Sniedovich and Voß (2006). The dynamic corridor width extension is taken from Caserta et al. (2010) and Caserta and Voß (2014).

Applications of the corridor method to the block relocation problem are described in Caserta and Voß (2009a), Caserta and Voß (2009c), Caserta et al. (2011), application to pre-marshalling is initially presented in Caserta and Voß (2009b), and then extended in Maniezzo et al. (2016, 2020). The application to DNA sequencing is in Caserta and Voß (2014), that to capacitated lot sizing in Caserta et al. (2010), and that to the capacitated facility location problem in Caserta and Voß (2020).

References

Caserta M, Voß S (2009a) A cooperative strategy for guiding the corridor method. In: Krasnogor N, et al (eds) Nature inspired cooperative strategies for optimization (NICSO 2008). Studies in computational intelligence, vol 236. Springer, Berlin, Heidelberg

Caserta M, Voß S (2009b) A corridor method-based algorithm for the pre-marshalling problem. In: Giacobini M, et al (eds) Applications of evolutionary computing, EvoWorkshops 2009. Lecture notes in computer science, vol 5484. Springer, Berlin, Heidelberg

Caserta M, Voß S (2009c) Corridor selection and fine tuning for the corridor method. In: Stützle T (ed) Learning and intelligent optimization, LION 2009. Lecture notes in computer science, vol 5851. Springer, Berlin, Heidelberg

Caserta M, Voß S (2014) A hybrid algorithm for the DNA sequencing problem. Discrete Appl Math 163:87–99

Caserta M, Voß S (2020) A general corridor method-based approach for capacitated facility location. Int J Prod Res 58(13):3855–3880

Caserta M, Ramirez A, Voß S (2010) A math-heuristic for the multi-level capacitated lot sizing problem with carryover. In: Chio CD, et al (eds) Applications of evolutionary computation, EvoApplications 2010. Lecture notes in computer science, vol 6025. Springer, pp 462–471

Caserta M, Voß S, Sniedovich M (2011) Applying the corridor method to a blocks relocation problem. Oper Res Spectrum 33:915–929

Maniezzo V, Boschetti M, Gutjahr W (2016) Stochastic real world warehouse premarshalling,. In: Matheuristics 2016 - proceedings of the sixth international workshop on model-based metaheuristics, Bruxelles, IRIDIA, Technical Report Series TR/IRIDIA/2016-007, pp 111–114

Maniezzo V, Boschetti M, Gutjahr W (2020) Stochastic premarshalling of block stacking warehouses. Omega. https://doi.org/10.1016/j.omega.2020.102336

Sniedovich M, Voß S (2006) The corridor method. A dynamic programming inspired metaheuristic. Control Cybern 35(3):551–578

Chapter 9
Kernel Search

9.1 Introduction

Kernel search (KS) is a purely matheuristic method, which leverages MILP solvers to obtain heuristic, or possibly optimal, solutions of instances encoded as (mixed-) integer linear programming problems. KS was in fact first presented as a method to solve MILP problems defined on binary variables modeling items selection, together with other integer or continuous variables related to the selected items. The first reported applications of this sort addressed portfolio selection and multidimensional knapsack. Later contributions proved the possibility to effectively apply the method also to problems that do not involve a selection stage.

The central idea of KS is to use some method, for example, the LP-relaxation, to identify a subset (a *kernel*) of promising decision variables and then to partition the remaining ones into *buckets*. The buckets are concatenated one at a time to the kernel to check whether improving solutions can be found.

More in detail, KS goes through two phases: initialization and expansion. In the initialization phase, the MILP formulation $\mathscr{F}$ of the instance to solve is first used to identify a promising subset of variables, which could enter the optimal solution. These variables are selected as the initial kernel Λ of the algorithm. One way to identify them could be by means of the LP-relaxation of $\mathscr{F}$, but other methods could be used as well. The variables not in the kernel are partitioned into $nbuck$ subsets, the buckets, where $nbuck$ could be a control parameter or a value identified by a suitable clustering algorithm. Let $\mathscr{F}(\Lambda)$ be formulation $\mathscr{F}$ restricted only to the variables in Λ. The best found solution $\mathbf{x}^*$, of cost z^*, is initialized by calling a MILP solver on $\mathscr{F}(\Lambda)$ and allowing it to restricted computational resources (time, memory, or whatever).

Figure 9.1 depicts this initialization phase. Initially, all variables are considered, and a few of them, represented as black cells, are selected to enter the initial kernel. The variables are then sorted according to some criterion, keeping the kernel variables in the first positions, and a MILP subproblem is solved on the

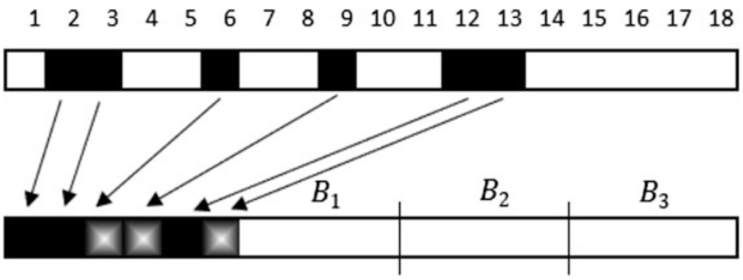


Fig. 9.1 Initial kernel

kernel variables only. The solution variables are the shaded gray ones in the figure. Moreover, the out-of-kernel variables are partitioned into *nbuck* buckets. In the figure $nbuck = 3$ and the buckets are B_1, B_2, and B_3.

The *expansion phase* follows, where a sequence of MILP subproblems is solved. Each i-th subproblem is restricted to a subset Λ_i of the problem variables, which includes the current kernel and a successive bucket B_i. The subproblems are further constrained to include in the solution at least one variable of the current bucket B_i and to provide a solution of cost better than that of the so far best found solution. In case such a solution is found, it becomes the new best found one, its cost the new z^*, and its nonzero variables are included in the kernel Λ. Note that in this version of KS the size of the kernel is non-decreasing: variables can be added to it but are never discarded.

We remark that both here and in the initialization phase, the subproblems are solved under limited computational resource constraints. Subproblems are, in fact, typically NP-hard; therefore, their solution to prove optimality could be too demanding. The increasing power of commercial MILP solvers actually enables in several cases to confidently go for optimality, but in general, one should concede to heuristic solutions.

Figure 9.2 shows the result of the test of the first bucket, B_1. The bucket variables are included as decision variables in the subproblem to solve; they become gray in the figure. The MILP solver is thus called to solve a subproblem over the kernel and first bucket variables, and with the additional constraint that the objective function must improve the best so far, if any. In the figure, one bucket variable enters the solution and is therefore included in the kernel.

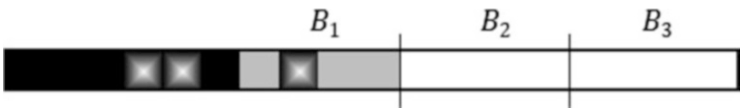


Fig. 9.2 First bucket

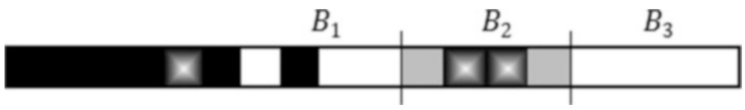


Fig. 9.3 Second bucket

Figure 9.3 shows the assessment of the second kernel. Its variables become gray and are included in the subproblem to solve, together with the kernel ones, which now include also one B_1 variable. Again, the MILP solver is constrained to find an improving solution. In this case, it can find one using two B_2 variables, which are therefore included into the kernel. The B_1 variable is not any more part of the best found solution, but it is kept in the kernel anyway.

Finally, Fig. 9.4 represents the subproblem testing the third bucket variables. In this case, its variables keep on being all gray as no one of them enters the solution. The best found solution is the one found when solving the subproblem on the second bucket. This could be an optimal solution, but also a heuristic one, as nothing forbids the possibility to have a better solution, for example, consisting of the variables of B_3 and the white variables of B_1. However, with the used procedure, we have guaranteed the maximum allowed computational resources to the search while keeping a reasonable possibility to provide an optimal solution.

A pseudocode of a basic KS is presented in Algorithm 39, solving the formulation $\mathscr{F}$ of a (minimization) problem $\mathscr{P}$.

The initialization phase is in steps 3 to 7, and it implements the construction of an initial kernel and of the sequence of buckets. Step 3 is usually applied to the continuous linear relaxation of $\mathscr{P}$, but other methods that provide quantitative information on the relative merits of each decision variable could be used, for example, Lagrangian relaxation. The information obtained in this step is, in fact, used in the next step to sort all variables by means of some criterion that tries to keep in the first positions the variables that are more likely to belong to the optimal integer solution. In step 5, the kernel Λ is initialized by selecting the first $nkinit$ items of the ordered set, while the remaining variables are partitioned into buckets $B_i, i = 1, \ldots, nbuck$. The number of variables included in each bucket does not need to be the same for all of them (obviously, in case the number of variables to partition was not a multiple of $nbuck$).

Finally, step 7 solves the formulation $\mathscr{F}(\Lambda)$, which is the formulation $\mathscr{F}$ restricted to the kernel variables of the input instance, and possibly obtains a first heuristic solution. In this example, the limited computational resource is the

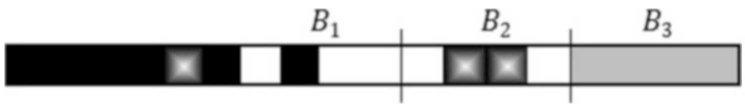


Fig. 9.4 Third bucket

maximum CPU time, and step 6 computes its maximum value that can be allocated to the solution process. This value is used here but also in all MILP solution steps in the subsequent expansion phase.

Algorithm 39: Basic kernel search

```
1  function Kernel Search($nbuck$, $nkinit$, $t_{max}$);
   Input  : Control parameters $nbuck$, $nkinit$, $t_{max}$
   Output: A feasible solution $\mathbf{x}^*$ of value z*
2  Let $\mathscr{F}$ be the MILP formulation of the instance to solve;
3  Solve a relaxation of $\mathscr{F}$;                    // needs a MILP formulation
4  Sort the variables according to a suitable criterion;   // initialization start
5  Build the initial kernel $\Lambda$ and a sequence of $nbuck$ buckets;
6  Let $t = t_{max}/nbuck$;                          // example of resource constraint
7  Solve $\mathscr{F}(\Lambda)$ with a time limit $t$, let $\mathbf{x}^*$ and z* be the best solution and its cost;
8  for $i$ = 1 to $nbuck$ do                                  // expansion phase
9      Construct the set $\Lambda_i = \Lambda \bigcup B_i$;          // $B_i$ is the current bucket
10     Add to $\mathscr{F}(\Lambda_i)$ the constraint $\sum_{j \in B_i} x_j \geq 1$;
11     Add to $\mathscr{F}(\Lambda_i)$ the constraint $z < z^*$;     // cut-off value for max cost
12     Solve $\mathscr{F}(\Lambda_i)$ with a time limit $t$;
13     if feasible solution found then
14         Let $\mathbf{x}^*$ and z* be the best solution and its cost;
15         Add to $\Lambda$ the variables which belong to $B_i$ and have been selected in $\mathbf{x}^*$;
16     end
17 end
```

The cycle at step 8 implements the expansion phase, which is devoted to the enlargement of the initial kernel Λ by means of the successive analysis of the identified buckets. At each iteration i, the variables in bucket B_i are concatenated to the kernel Λ, obtaining the new subset $\Lambda_i = \Lambda \bigcup B_i$. The subproblem $\mathscr{F}(\Lambda_i)$ is then solved with computational resources limited as per step 6 and after the addition of two further constraints. The first constraint ensures that at least one variable of the current bucket B_i enters the solution, and the second guarantees that the final solution improves over the best one found so far (the first constraint is implied by the second if the problem is solved to optimality and a solution is found). If a solution meeting these two constraints is found, the variables $\bar{\Lambda}_i$ selected from the current bucket B_i are permanently added to the kernel $\Lambda = \Lambda \bigcup \bar{\Lambda}_i$. When the last bucket has been analyzed, the procedure stops, returning $\mathbf{x}^*$ with value z* as the best found solution.

In order to completely define the basic kernel search, we need to specify:

1. the relaxation used at step 3 (usually an LP-relaxation);
2. the criterion used to sort the items in step 4;
3. the number of variables initially selected to construct the kernel Λ;
4. the way to cluster variables into buckets.

Different implementations of these elements give rise to different KS algorithms. Furthermore, the basic KS presented has been enriched in the literature by a number of refinements that make it more effective on a wider range of problems.

In the basic KS, the sequence of buckets is examined only once. A simple extension, an iterative variant, suggests restarting the extension phase when the sequence of buckets has been completely analyzed if some new variables were added to the kernel. Let q be the index of the last bucket from which at least one variable was selected to enter the kernel. The extension phase is restarted by setting $nbuck = q - 1$. If any new variable is selected in this phase, $nbuck$ is reset to its original value and the extension phase is started again. The iterative KS stops when no new item is added to set Λ for a whole expansion phase. Clearly, also the computational resources available to the iterative KS must be adapted to permit those that are not completely used in the first call of the extension phase to be passed on to the subsequent ones.

The original portfolio optimization application was extended in a further work to index tracking, a problem where a fund manager is expected to create a portfolio of assets whose performance replicates, as close as possible, that of a financial market index chosen as a benchmark. In this context, a KS working on binary and other continuous variables was proposed, where a limited number of variables are explored and variables can be removed from the kernel. Later, the application was extended to multiobjective optimization, working on the bi-objective enhanced index tracking problem, where two competing objectives, i.e., the expected excess return of the portfolio over the benchmark and the tracking error, are taken into account.

The possibility of applications got expanded after dealing with Binary Integer Linear Programming (BILP). A first significant application in this area tackles a problem with only one set of binary variables. Then, KS was used to solve MILP problems where a continuous variable is associated with each binary variable, such as portfolio selection and index tracking, even with a large number of continuous variables associated with each binary variable. Finally, KS has been applied to a general binary ILP problem, namely the single source capacitated facility location problem. Besides the generalization of the addressed problem, this work proposes two KS variants implementing hard variable fixing (see Chap. 5). In both of them, the selection of the variables to be fixed is guided by the information provided by the optimal solution of the linear relaxation. The two proposed variants make, in fact, use of the information retrieved from the optimal solution of the linear relaxation to fix some binary variables either to 1 or to 0, and this is done on all restricted problems of the bucket sequence.

Other recent contributions and applications include the Time-Dependent Rural Postman Problem, by using a specific arc-path formulation of the problem, in order to formulate the restricted subproblems solved by means of a MILP solver, and a supply chain optimization paper proposing a specific MILP model and embedding it into KS.

9.2 Kernel Search for the GAP

There is no application of KS to the GAP reported in the literature so far, let alone one simple enough that could fit in this section. The closest reported application is that on single source capacitated facility location, which is a problem quite similar to the GAP, except that not all servers need to be used. Moreover, using a server incurs in a fixed cost. Unfortunately, the request to choose the server subset entails the need of using a specific set of binary decision variables, and the work on the single source capacitated facility location is centered exactly on this further set of variables, so it cannot be directly translated to the GAP.

Fortunately, however, KS, and specifically the basic KS reported in Algorithm 39 are normative enough to permit a direct application of the basic pseudocode to our problem. All what is needed is the specification of the points listed in Sect. 9.1. The result is presented in the following Algorithm 40.

Algorithm 40: Kernel search for the GAP

```
1  function Kernel Search($nbuck$, $t_{max}$);
   Input  : Control parameters $nbuck$, $t_{max}$
   Output: A feasible solution $\mathbf{x}^*$ of value z*
2  Let $\mathscr{F}$ be the MILP formulation of the instance to solve;
3  Solve an LP-relaxation of $\mathscr{F}$;            // needs a MILP formulation
4  Sort the variables by decreasing reduced costs;      // initialization start
5  Build the initial kernel $\Lambda$ and a sequence of $nbuck$ buckets;
6  Let $t = t_{max}/nbuck$;
7  Solve $\mathscr{F}(\Lambda)$ with a time limit $t$, let $\mathbf{x}^*$ and z* be the best solution and its cost;
8  for $i$ = $1$ to $nbuck$ do                          // expansion phase
9      Construct the set $\Lambda_i = \Lambda \bigcup B_i$;      // $B_i$ is the current bucket
10     Add to $\mathscr{F}(\Lambda_i)$ the constraint $\sum_{j \in B_i} x_j \geq 1$;
11     Add to $\mathscr{F}(\Lambda_i)$ the constraint $z < z^*$;   // cut-off value for max cost;
12     Solve $\mathscr{F}(\Lambda_i)$ with a time limit $t$ by means of a MIP solver;
13     if feasible solution found then
14         let $\mathbf{x}^*$ and z* be the best solution and its cost;
15         add to $\Lambda$ the variables which belong to $B_i$ and have been selected in $\mathbf{x}^*$;
16     end
17 end
```

Algorithm 40 specifies Algorithm 39 on the following points.

1. The relaxation used at step 3 is actually the LP-relaxation of formulation GAP (see Chap. 1).
2. The criterion used to sort the items in step 4 is to use first the variables that enter the optimal solution of the LP-relaxation, and then all other variables are ordered by decreasing reduced costs.
3. The variables initially selected to construct the kernel are all the nonzero ones of the solution of the LP-relaxation, let $nkinit$ be their number.

4. The way to cluster variables into buckets is simply to consider the variables in the order, filling all *nbuck* buckets to their full capacity except possibly the last.

A trace of the execution of this algorithm on instance *example8x3* of Sect. 1.1, with parameter $nbuck = 3$, is reported in the following. The remaining two control parameters were defined within the code, as $nkinit$ was set equal to the number of nonzero variables of the optimal LP-relaxation solution and t_{max} was unbounded, as all subproblems were solved to optimality, when feasible.

Being $nbuck = 3$, the loop 8 is repeated for 3 iterations, starting from the initial solution. The initialization phase first solves the LP-relaxation of the problem, obtaining the solution reported in Sect. 1.2.1 of cost 231.45, which is the linear lower bound to the cost of the optimal solution of this instance.

The reduced costs of the 24 variables of formulation GAP are as follows:[1]

$$\begin{array}{llllll}
c'_0 = 4.47 & c'_1 = 3.35 & c'_2 = 2.23 & c'_3 = 1.12 & c'_4 = 0.00^* & c'_5 = -1.12^* \\
c'_6 = -2.23^* & c'_7 = -3.35^* & c'_8 = 0.00 & c'_9 = 0.00^* & c'_{10} = 0.00^* & c'_{11} = 0.00^* \\
c'_{12} = 0.00^* & c'_{13} = 0.00 & c'_{14} = 0.00 & c'_{15} = 0.00 & c'_{16} = -3.08^* & c'_{17} = 0.00^* \\
c'_{18} = 3.08 & c'_{19} = 6.16 & c'_{20} = 9.24 & c'_{21} = 12.32 & c'_{22} = 15.40 & c'_{23} = 18.49
\end{array}$$

The starred variables are the nonzero ones. There are negative reduced cost values among them because the variables are bounded to take values in [0, 1] rather than subject to specific constraints. All nonzero variables are included in the solution, independently on whether they were basic variables or not.

When the variables are ordered in step 4, we find first the positive reduced cost ones, then the zero valued ones, both basic and nonbasic in lexical order (the sorting algorithm is stable), and then the negative reduced cost ones. Actually, the index order becomes 23, 22, 21, 20, 19, 0, 1, 18, 2, 3, 4*, 8, 9*, 10*, 11*, 12*, 13, 14, 15, 17*, 5*, 6*, 16*, and 7*, where again the nonzero variables are starred. This is the ordering that will be used for the rest of the run. Note that this ordering is somewhat arbitrary, given the computation based on bounded variables. More elaborated choices could be studied.

The nonzero variables are 10 and therefore $nkinit = 10$, and the nonkernel variables are 14 and are clustered into buckets as follows:

$$\begin{aligned}
B_1 &= \{\ 23,\ 22,\ 21,\ 20,\ 19,\ \ 0\} \\
B_2 &= \{\ \ 1,\ 18,\ \ 2,\ \ 3,\ \ 8,\ 13\} \\
B_3 &= \{\ 14,\ 15\}
\end{aligned}$$

The initial kernel contains all nonzero variables, that is, $\Lambda = \{4, 5, 6, 7, 9, 10, 11, 12, 16, 17\}$. Step 7 tries to get a first feasible solution using these variables only but does not succeed, so initially $\mathbf{x}^*$ is not initialized and $z^* = \infty$.

[1]We remind that in the computational traces the decision variables are indexed from 0.

We enter therefore the first loop iteration, where the first bucket is appended to the current kernel, obtaining $\Lambda_1 = \{0, 4, 5, 6, 7, 9, 10, 11, 12, 16, 17, 19, 20, 21, 22, 23\}$. In this case, the MIP solver is able to find a feasible integer solution of cost 327, which includes the nonkernel variables 0, 19, 20, and 23. These variables are added to the kernel, and the best found solution is initialized, as well as $z^* = 327$.

At the second iteration of the expansion loop , the second bucket is concatenated to the expanded kernel, yielding $\Lambda_2 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16, 17, 18, 19, 20, 23\}$. Moreover, the constraint $z < 327$ is included in the model to ensure the output of only an improving solution, if any. The MIP solver is capable of finding a feasible optimal solution of cost 325, which is saved in the new best found solution, together with its cost $z^* = 325$. The bucket variables that are included in the solution and therefore enter the kernel are 2 and 8.

Finally, the last iteration of the expansion loop starts with the last bucket added to kernel, that is $\Lambda_3 = \{0, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 19, 20, 23\}$. The MIP solver cannot find an improving solution (the current best found one is actually optimal for this instance) and the execution terminates.

9.3 Related literature

Kernel search was introduced in Angelelli et al. (2007, 2010). The portfolio selection application is described in Angelelli et al. (2007, 2012), and the multi-dimensional knapsack one in Angelelli et al. (2010). Further applications included portfolio optimization (Angelelli et al. 2012) and index tracking (Guastaroba and Speranza 2012a). The multiobjective optimization extension is in Filippi et al. (2016).

Binary Integer Linear Programming (BILP) applications were described first in Angelelli et al. (2010), then in Angelelli et al. (2012) for portfolio selection, and in Guastaroba and Speranza (2012a,b) for index tracking.

The general binary ILP application is in Guastaroba and Speranza (2014), the Time-Dependent Rural Postman Problem in Zanotti et al. (2019), and the supply chain optimization application in Zhang et al. (2019).

References

Angelelli E, Mansini R, Speranza MG (2007) Kernel search: a heuristic framework for MILP problems with binary variables. Technical report, Department of Electronics for Automation, University of Brescia, R.T.2007-04-56

Angelelli E, Mansini R, Speranza MG (2010) Kernel search: a general heuristic for the multi-dimensional knapsack problem. Comput Oper Res 37(11):2017–2026

Angelelli E, Mansini R, Speranza MG (2012) Kernel search: a new heuristic framework for portfolio selection. Comput Optim Appl 51(1):345–361

Filippi C, Guastaroba G, Speranza MG (2016) A heuristic framework for the bi-objective enhanced index tracking problem. Omega 65(C):122–137

Guastaroba G, Speranza MG (2012a) Kernel search: an application to the index tracking problem. Eur J Oper Res 217(1):54–68

Guastaroba G, Speranza MG (2012b) Kernel search for the capacitated facility location problem. J Heuristics 18(6):877–917

Guastaroba G, Speranza MG (2014) A heuristic for BILP problems: The single source capacitated facility location problem. Eur J Oper Res 238(2):438–450

Zanotti R, Mansini R, Ghiani G, Guerriero E (2019) A Kernel search approach for the time-dependent rural postman problem. In: WARP3, 3rd International workshop on arc routing problems, Pizzo (Calabria, Italy)

Zhang Y, Chu F, Che A, Yu Y, Feng X (2019) Novel model and kernel search heuristic for multi-period closed-loop food supply chain planning with returnable transport items. Int J Prod Res 57(23):7439–7456. Taylor & Francis

Chapter 10
Fore-and-Back

10.1 Introduction

Fore-and-Back has been presented in the literature as *Forward&Backward* and also as *F&B*, but the algorithm presented in this chapter differs in some details from the previously published ones. It is an extension of *beam search* (BS) that can improve its effectiveness. Fore-and-Back, when run with no limits on computational resources, becomes an exact solution method. However, by design, it is mainly concerned with heuristic solving, trying to quickly get high quality solutions with little attention paid to optimality proofs.

A significant characteristic of this method is that, despite being a primal only method, it is able to compute bounds to the cost of completing partial solutions, therefore to discard partial solutions from expansion and ultimately to reduce the search space.

Beam search (BS) is a variant of standard tree search that limits the number of offsprings that are expanded at each iteration. BS core ideas were originally introduced in artificial intelligence contexts and only later transposed to optimization. The first problems for which BS was used were scheduling problems, but BS has since proved successful also on many other different combinatorial optimization problems.

BS does not complete the search that would normally be carried out by branch and bound algorithms; therefore, it is an approximate method and a matheuristic of its own. BS has, in fact, been proposed as an effective heuristic methodology, and as such it has been enhanced and hybridized with other heuristics, for example, ant colony optimization. Other matheuristics closely related to BS have been proposed. One is the pilot method, which consists of a partial enumeration strategy, where the possible expansions of each partial solution are evaluated by means of a pilot heuristic. Another one is the Filter&Fan method, which starts with a feasible solution and builds a search tree, where branches correspond to submoves in the neighborhood of the solution and where each node corresponds to a solution

V. Maniezzo et al., *Matheuristics*, EURO Advanced Tutorials on Operational Research, https://doi.org/10.1007/978-3-030-70277-9_10

obtained as a result of the sequence of submoves associated with the root-node path. In this algorithm, the initial candidate list of moves is filtered at each tree level by evaluating each move in the list with respect to all the solutions at that level. The best moves at each level are included in the candidate list of the next level, and the corresponding solutions are the nodes of the successive level.

The characterizing idea of BS is to allow the extension of partial solutions into a limited number of offsprings. This is similar to what we have already seen for diving (Chap. 5), for VLSNS (Chap. 6) and for the Corridor Method (Chap. 8), but here the focus is on the result, the number of offsprings, and not on the method to limit their number. At each BS iteration, the algorithm extends a partial solution from a set **T**, the *beam*, generating a possibly limited number of offsprings. Each offspring is either a complete solution, or it is inserted into the set **T** itself, in case it is a partial solution worth of further analysis.

At the end of the expansions, BS selects from **T** up to δ (a parameter called the *beam width*) solutions. The selection is based on some criterion for ranking the expected usefulness of an expansion, for example, on the basis of bounds to the cost of the completions.

More in detail, BS proceeds as follows. At the first step, the beam **T** is initialized with an empty solution. Then, the algorithm iterates a basic procedure in which a set of promising nodes at a given level of the search tree is expanded to generate their δ offspring, in which all become members of the set of the unexpanded nodes of a subsequent tree level. A node can be considered to be promising in accordance with its completion bound cost. When a level has been expanded, two strategies are possible: either expand the subsequent level or expand the nodes of lowest cost completion bound. Intermediate strategies are possible, where expansions proceed depth-first in order to quickly complete good solutions but, when the last level is expanded, it is possible to backtrack to the nodes with the lowest completion bound, which can be high in the tree hierarchy. This process is iterated until a termination condition is met (heuristic) or until all unexpanded nodes have a completion bound cost that is not smaller than the current upper bound, which is, therefore, the optimal cost (exact).

10.2 Fore-and-Back

Algorithm Fore-and-Back builds on this general BS approach, alternating searches following opposite expansion directions, and storing in memory previous partial results that can be used as a lookahead to complete partial solutions when search is performed in the opposite direction. The algorithm works therefore best when the problem suggests a natural direction of partial solution expansions, which can also be reversed.

Actually, there is a vast class of combinatorial optimization problems that fit this schema. These are problems that exhibit a regular substructure that can be decomposed into n subproblems that are linked together by a set of coupling constraints. These problems can often be modeled by defining, for each k-th subproblem, a set S_k containing all the feasible solutions for the k-th subproblem. The resulting problem consists of choosing, from each set S_k, a single component in such a way that the set of selected components satisfies all constraints. This is, for example, the case for the GAP, where subproblems could refer to the assignments of single clients and the capacity constraints act as linking constraints, or vice versa (subproblems defined on capacities and linking constraints on assignments).

Fore-and-Back exploits this structure, molding around it an iterative heuristic algorithm that adopts a memory-based look-ahead strategy exploiting the knowledge gained in its past search history. Algorithm Fore-and-Back iterates a partial exploration of the solution space by generating a sequence of beam search-like search trees of two types, called *forward* and *backward* trees. Each node at level h of the trees represents a partial solution containing h components. At each iteration t, the algorithm generates a forward tree $\mathbf{F}^t$ if t is odd, or a backward tree $\mathbf{B}^t$ if t is even. In generating a tree, each partial solution X is extended to a feasible solution using the partial solutions generated at the previous iteration in the complementary tree, and the cost of the resulting solution is used to bound the quality of the best complete solution that can be obtained from X.

10.2.1 Search Trees

During search, Fore-and-Back alternatively builds *forward trees* and *backward trees*.

A forward tree is an n-level tree, if the number of subproblems is n, where each level $h = 1, ..., n$ is associated with a component set S_h and each node at level h corresponds to a partial solution containing one component from each set $S_1, S_2, \ldots, S_h$. Components are considered to be solution elements, for example, one component could be the assignment of a client to a server in the case of the GAP, or an arc to be included in the route in the case of the TSP.

Conversely, in a backward tree, each level h is associated with a set S_{n-h+1}, in case these labels were numbered according to the forward exploration, so that a node at level h represents a partial solution containing one component of each set $S_n, S_{n-1}, \ldots, S_{n-h+1}$.

A list, denoted by L_h^t, is associated with each level h of the tree built at iteration t. The list contains δ nodes generated but not expanded at level h, where δ is an input control parameter equivalent to the likewise denoted parameter of beam search. All nodes so far expanded at level h in all trees at odd iterations are kept in set E^h for forward trees, while those expanded at level h in all trees at even iterations are kept in $\overline{E}^h$ for backward trees. The nodes in the lists L_h^t, $h = 1, \ldots, n$, represent the memory of iteration t, which will be used to guide the exploration of the current tree and of the tree that will be explored in the following iteration $t + 1$.

The core idea of Fore-and-Back is to evaluate the completion cost of partial solutions stored at level h of the tree by means of the partial solutions stored in

L_{n-h}^{t-1} in case of forward trees, or, analogously, the completion of partial solutions in L_{n-h}^{t} by means of solutions in L_h^{t-1} in case of backward trees.

As an example, suppose we are building the forward tree associated with an odd iteration t. Consider two partial solutions, $X \in L_h^t$ and $\overline{X} \in L_{h+1}^{t-1}$. Since t is odd, X contains one component of each set $S_1, S_2, ..., S_h$, while $\overline{X}$ contains one component of each set $S_n, S_{n-1}, ..., S_{h+1}$. These two solutions can be combined to obtain a (not necessarily feasible) complete solution $X \cup \overline{X}$ of cost $c\big(X \cup \overline{X}\big)$. Clearly, if the resulting solution $X \cup \overline{X}$ satisfies all constraints, the associated cost represents a valid upper bound to the optimal problem solution cost (assuming, here and in the rest of the chapter, to deal with a minimization problem).

In general, at each iteration t, algorithm Fore-and-Back builds the associated tree and computes the cost $c(X)$ of each node X. The cost of an inner node is derived from the cost of completing the partial solution X and from the penalty assigned to the level of infeasibility that the completion shows. Algebraically,

$$c(X) = \min_{\overline{X} \in L_{h+1}^{t-1}} \left\{ c\big(X \cup \overline{X}\big) + c_{inf}\big(X \cup \overline{X}\big) \right\}, \tag{10.1}$$

where $c_{inf}(X \cup \overline{X})$ is an arbitrary function whose value is related to the degree of infeasibility of $X \cup \overline{X}$ and that is equal to 0 if $X \cup \overline{X}$ is a feasible solution. It is important to effectively balance the push toward feasibility against the quest for good solutions, a balance that is problem-dependent when not even instance-dependent.

During the first iteration, there is no backward tree to match partial solutions against. The lists L_h^0 are empty at each level; therefore, expression (10.1) gives the cost of the partial solution X.

10.2.2 *Fore-and-Back Pseudocode*

To simplify the code and avoid all duplications needed to account for forward or backward directions, we will denote by **T** alternatively the forward tree **F** or the backward tree **B**, depending on the parity of the iteration counter. Consequently, T_h denotes the level h of the tree **T**, be it forward or backward.

The algorithm is controlled by three user-defined parameters: δ, the number of nodes expanded at each level of both forward and backward trees; *maxn*, the maximum total number of nodes to expand; and *maxtnodes*, the limit of the number of nodes of each tree.

In order to expand level h of a tree at iteration t, the algorithm computes the value $c(X)$ for each node $X \in T_h$ and builds the set $L_h^t \subseteq T_h$. The list is defined by ordering T_h by increasing cost values and keeping in L_h^t only its δ nodes having the smallest cost, which will be further expanded.

A distinguishing feature of Fore-and-Back is that at each level h of the tree built at iteration t, we also store the cost $\hat{c}_h^t$, which is the least cost of nodes in

T_h but not in L_h^t. This is the least cost we are expected to pay in case we want to complete a complementary partial solution without using the partial solutions that led to defining L_h^t.

In case a feasible solution is found, we possibly update a variable z_{best}, which keeps the cost of the best solution achieved so far: it is initialized to ∞ at the beginning of the algorithm and takes on progressively lower values during search.

In forward trees, each node X included in L_h^t is expanded to create a new node $X \cup \{s\}$ for each component s of the set S_{h+1} of the subproblem associated with level $h + 1$, provided that the following conditions hold:

(1) $X \cup \{s\}$ does not violate any constraint.
(2) $c(X \cup \{s\}) + \hat{c}_{h+2}^{t-1} < z_{best}$, in case $X \cup \{s\}$ cannot be feasibly completed with a partial solution in L_{h+2}^{t-1} or in $\overline{E}^{h+2}$ (pruning).

These conditions also hold, with opportune indices update, for backward trees. In the pseudocode, we make use of two dummy levels, L_0^t for forward trees and L_{n+1}^t for backward trees, in order to initialize their computation.

Node expansions continue in this fashion until the last level is reached ($h = n$) or until a maximum number of nodes have been expended in the current tree. This last condition is typically met in problems, like the GAP, where partial solutions cannot be feasibly expanded in any way, and therefore, backtracks are in order. Backtracking can be made either in a depth-first way, expanding the last generated unexpanded nodes, or jumping to the stored node of least expected cost $c(X)$.

Algorithm Fore-and-Back terminates after the expansion of $maxn$ nodes, or after two consecutive iterations where the value of z_{best} does not improve. This last is a condition that no further improvements would be possible if infeasibilities are properly accounted for.

Figures 10.1 and 10.2 show an example of possible forward and backward trees for the first iteration of the algorithm, expanding $\delta = 2$ nodes per level.

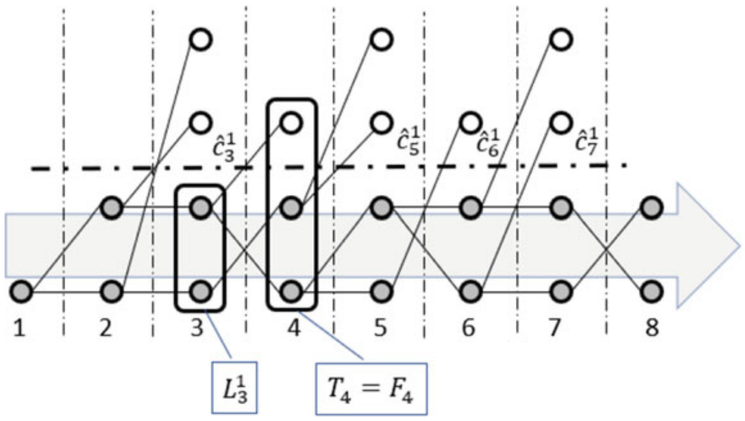


Fig. 10.1 Forward tree, initial ($t = 1$)

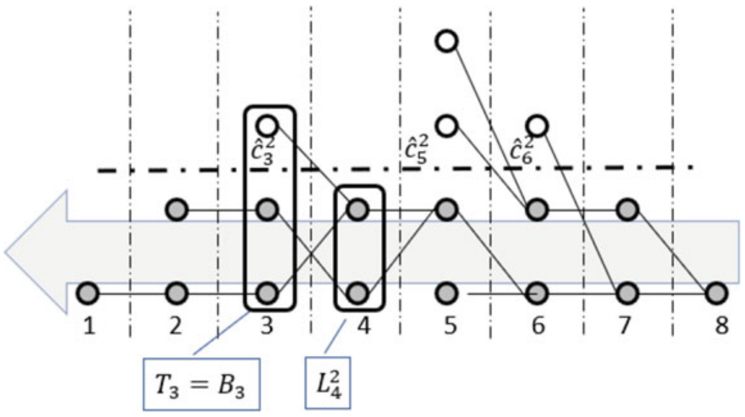


Fig. 10.2 Backward tree ($t = 2$)

10.3 Fore-and-Back for the GAP

Algorithm 41 can be directly applied to the GAP, as this problem enjoys the structural property required in Sect. 10.2. In the following run, in fact, we assumed the GAP to be composed of subproblems defined by client assignments, one subproblem for each client. The linking constraints are given by the capacity constraints, much alike the decomposition used in the example of Sect. 7.4. Algorithm 41 is applied to instance *example8x3* of Sect. 1.1 and run with parameters $\delta = 2$, $maxn = 5000$, and $maxtnodes = maxn/10$.

Initially, step 2, variables, and structures are initialized, and a standard beam search is run. We saw in Chap. 1 that GAP is strongly NP-hard and that the number of infeasible solutions that can be expressed by the decision variables of formulation GAP (see Sect. 1.1) exceeds by far the number of feasible solutions (Table 1.1). This implies that unguided search is likely to produce partial solutions that cannot be feasibly expanded. Figure 10.3 shows the complete forward tree that was explored during the first forward run. The figure shows all generated nodes and the resulting tree topology, except that nodes corresponding to complete solutions are not stored in the code, being immediately pruned if dominated, and are consequently not shown in the figure. The root node is the dummy empty node.

The condition that actively terminated the first search was on the number of tree nodes, $maxtnodes$, and search went through 41 backtrackings. No feasible solution could be found. The bounds for unmatched completions were $\hat{c}_1^1 = 88$, $\hat{c}_2^1 = 172$, $\hat{c}_3^1 = 252$, $\hat{c}_4^1 = 282$, $\hat{c}_5^1 = 221$, $\hat{c}_6^1 = 229$, and $\hat{c}_7^1 = 257$.[1]

[1]We remind that in the computational traces the decision variables are indexed from 0.

Algorithm 41: Algorithm Fore-and-Back

```
function Fore-and-Back($\delta$, $maxn$, $maxtnodes$);
Input : $\delta$, beam width, $maxn$, max total num of nodes, $maxtnodes$, max num of nodes
        per tree
Output: A feasible solution $\mathbf{x}^*$ of value $z_{best}$
initialize $t = 0$, $noimpr = 0$, $nNodes = 0$, $z_{best} = \infty$;
while $nNodes \leq maxn$ do      // Alternate forward and backward trees
    $t = t + 1$; $noimpr = noimpr + 1$;
    let $L_0^t = L_{n+1}^t = \{\emptyset\}$, $ntNodes$=0;
    foreach $level\ h = 1, \ldots, n$ do            // generate the node set $T_h$
        set $T_h = \{\emptyset\}$;
        if $t$ is odd then
            set $k = h$; $k1 = h - 1$;
        else
            set $k = n - h + 1$; $k1 = n - h + 2$;
        end
        foreach $node\ X \in L_{k-1}^t$ do
            foreach $component\ s \in S_k$ do
                let $X' = X \cup \{s\}$;
                if $X'$ meets conditions $i)$ and $ii)$ then
                    set $T_k = T_k \cup X'$;
                    $ntNodes = ntNodes + 1$;
                    $nNodes = nNodes + 1$;
                    if $h = n$ and $c(X') < z_{best}$ then      // feasible solution
                        set $z_{best} = c(X')$, $noimpr = 0$ and $\mathbf{x}^* = X'$;
                    end
                end
            end
            set $E^k = E^k \cup X$;                               // node expanded
        end
        foreach $node\ X \in T_k$ do              // extract the subset $L_k^t \in T_k$
            if $|T_k| \leq \delta$ then
                set $L_k^t = T_k$;
            else
                let $L_k^t$ contain only the $\delta$ least cost partial solutions of $T_k$;
            end
        end
        if $ntNodes > maxtnodes$ then break;
    end
    if $noimpr = 2$ then// best sol. not improved for two iterations
        stop;
    end
end
```

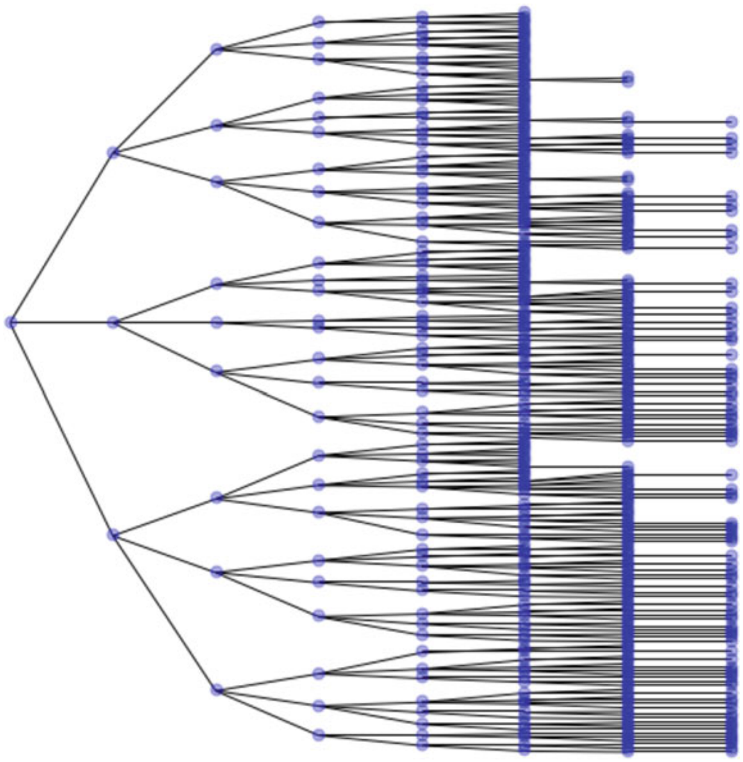


Fig. 10.3 Initial forward tree for instance *example8x3*

The stored partial solutions, along with the bounds, proved extremely useful to guide search for the backward tree. Figure 10.4 shows the first backward tree. This time the active terminating condition was the completion of the loop at step 6, as a complete feasible solution could be constructed (of the two longer paths of Fig. 10.4 one could not be expanded including the last client). However, feasible solutions could be found before that, upon matching partial backward solutions with partial solutions stored by the forward search.

The first feasible matching, thus the first feasible solution, was found at level 4 of the backward tree, where the partial solution $\overline{X} = (\_, \_, \_, \_, 2, 2, 2, 1)$ of cost 276 could be matched with the forward partial solution $X = (0, 0, 0, 1, \_, \_, \_, \_)$ of cost 61, thus producing a feasible solution of cost 337.

This upper bound was repeatedly improved during the backward search. After obtaining higher cost matchings, at level 4 the forward partial solution $X = (0, 0, 0, 2, \_, \_, \_, \_)$ of cost 105 could eventually be matched against the backward partial solution $\overline{X} = (\_, \_, \_, \_, 1, 2, 2, 1)$ of cost 230, thus producing a feasible solution of cost 335.

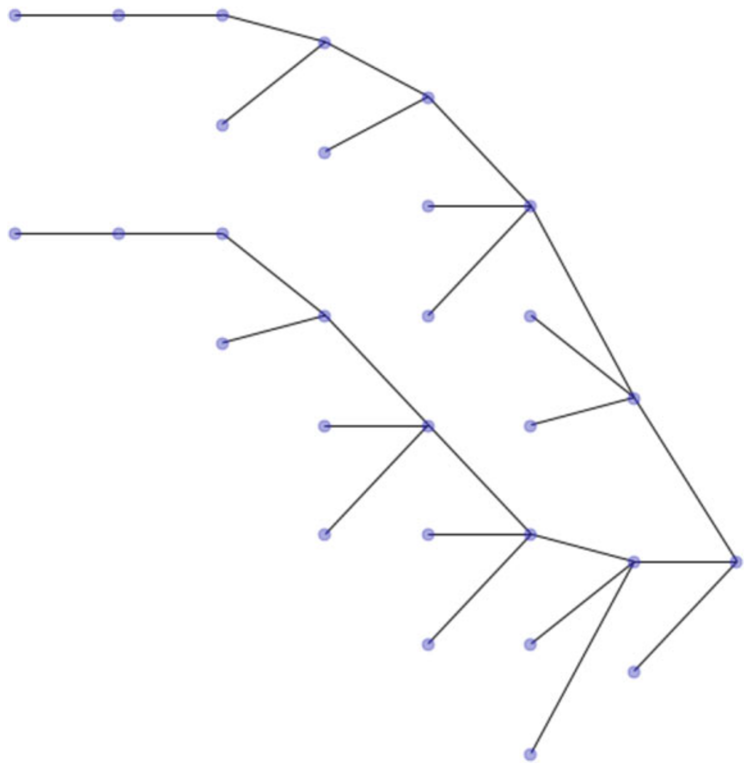


Fig. 10.4 First backward tree for instance *example8x3*

Then, an improved matching of cost 334 was found, and then one of cost 330, until the forward partial solution $X = (1, 0, 0, 2, \_, \_, \_, \_)$ of cost 117 could be matched with the backward partial solution $\overline{X} = (\_, \_, \_, \_, 1, 2, 2, 0)$ of cost 211, thus producing a feasible solution of cost 328. At the end of the backward run, the algorithm expanded 29 nodes, which permitted to identify 36 feasible solutions (one by constructions, the other ones by matching unexpanded offspring with stored partial solutions).

The best found solution of cost 328 is still not optimal. An optimal solution could be found in the subsequent forward run, with $t = 2$, where first an improving solution of cost 327 was found, then one of cost 326, and finally, after backtracking at level 1, the forward partial solution $X = (1, \_, \_, \_, \_, \_, \_, \_)$ of cost 22 could be matched with the backward partial solution $\overline{X} = (\_, 1, 0, 2, 0, 2, 2, 0)$ of cost 303, thus producing a feasible solution of cost 325, which is optimal.

Having no awareness of the reached optimality, search goes on until a termination condition is met. In this case, the condition refers to the number of iterations without best solution improvements and lets search terminate after 3 main loop iterations.

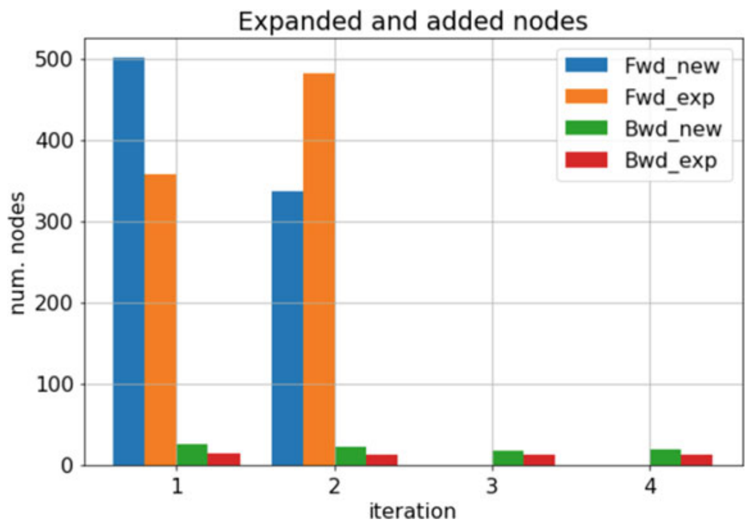


Fig. 10.5 Number of expanded nodes

Figures 10.5 and 10.6 report about the number of nodes explored during search. The left figure shows the number of node expansions at each iteration, along with the number of nodes that were actually stored into memory. It is possible that the number of stored nodes is higher than the number of expansions, as at iteration 1 because when expanding a node more than an offspring is generated. It is also possible that the number of expansions is higher than the number of stored nodes, as at iteration 2, because many infeasible offsprings get generated, and these are not stored into memory.

Data is presented separately for forward and backward trees. It is apparent how the first forward tree, which cannot use completion bounds for pruning, generates a number of nodes much higher than the following ones, which can make use of search memory.

Figure 10.6 presents aggregate data on the total number of stored nodes and on the total number of open nodes per iteration. Again, one can see how most nodes are generated during the first two iterations, mainly during the first, and that the termination condition stops the search when there would still be open nodes to expand, which could, however, not lead to improving solutions (though the algorithm is unaware of this).

Coming to feasible solutions obtained by matching partial solutions, Fig. 10.7 shows, at each level of the search tree, the number of feasible solutions that could be obtained. The distribution is heavily skewed to the right, as a result of the much higher number of nodes produced by forward search, which permitted the early pruning of backward solutions, when they were generated. No matchings were achieved at level 7.

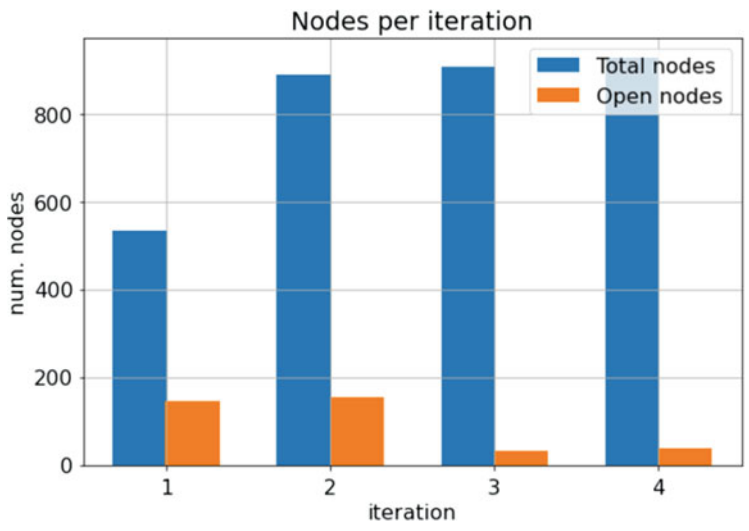


Fig. 10.6 Number of nodes per iteration

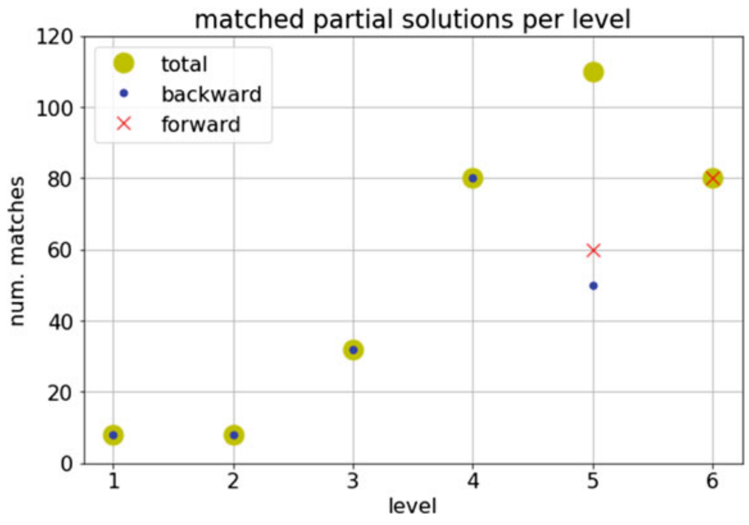


Fig. 10.7 Number of matched solutions

Finally, Fig. 10.8 shows, separately for forward and for backward trees, the profile of the completion bounds at each level h of the trees. These are the $\hat{c}_h^2$ and $\hat{c}_h^3$ bounds that could be read at the end of the run, $h = 1, \ldots, n$.

The blue forward bound profile increases monotonically up to level $n-1$ (the last level has no bound as it corresponds to complete solutions), thanks to the sufficient number of nodes that were generated.

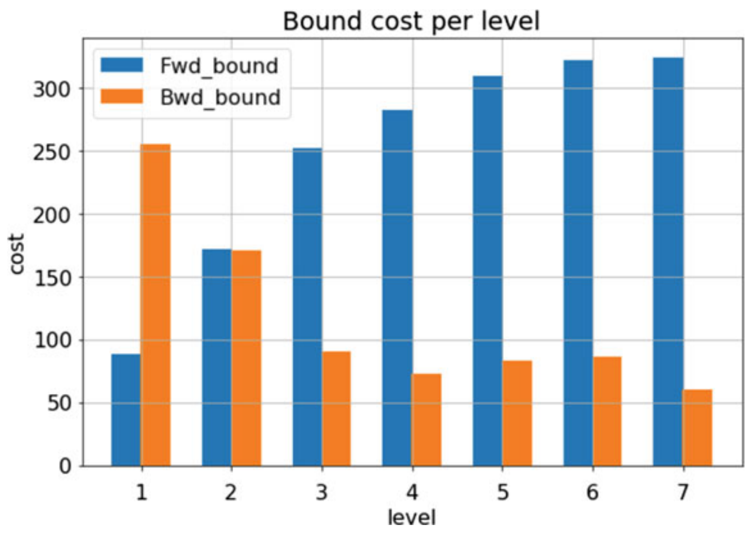


Fig. 10.8 Completion bound cost per level

The orange backward bound is counter-intuitively non-monotonic and very low for the middle levels. This comes from the fact that comparatively few nodes were generated for backward trees; therefore, data refers only to the best solutions, which were produced in a depth-first fashion. For example, nodes at level 4 derived from the expansion of the best nodes at level 5, and the node that caused a comparatively high bound at level 5 was not yet expanded, therefore did not cause a monotonic increase of the backward completion bound.

10.4 Related Literature

Fore-and-Back was initially presented in Bartolini et al. (2008) and Bartolini and Mingozzi (2009).

AI introduction on beam search context can be found in Lowerre (1976), Reddy (1977). Scheduling applications of beam search are described in Ow and Morton (1988) and Pinedo (1995). Extensions and hybrids can be found, for example, in Della Croce et al. (2004), Blum (2005), Blum (2008), and Maniezzo (1999).

The pilot method was proposed in Duin and Voß (1999) and the Filter and Fan in Glover (1998) and Greistorfer and Rego (2006).

References

Bartolini E, Mingozzi A (2009) Algorithms for the non-bifurcated network design problem. J Heuristics 15(3):259–281

Bartolini E, Maniezzo V, Mingozzi A (2008) An adaptive memory-based approach based on partial enumeration. In: Maniezzo V, Battiti R, Watson JP (eds) LION 2, LNCS 5313. Springer, pp 12–24

Blum C (2005) Beam-ACO - Hybridizing ant colony optimization with beam search: an application to open shop scheduling. Comput Oper Res 32(6):1565–1591

Blum C (2008) Beam-ACO for simple assembly line balancing. INFORMS J Comput 20(4):618–627

Della Croce F, Ghirardi M, Tadei R (2004) Recovering beam search: Enhancing the beam search approach for combinatorial optimization problems. J Heuristics 10(1):89–104

Duin C, Voß S (1999) The pilot method: A strategy for heuristic repetition with application problem in graphs. Networks 34:181–191

Glover F (1998) A template for scatter search and path relinking. In: Ronald E, Schoenauer M, Snyers D, Hao JK, Lutton E (eds) Artificial evolution. Lecture notes in computer science, vol 1363, pp 3–51

Greistorfer P, Rego C (2006) A simple filter-and-fan approach to the facility location problem. Comput Oper Res 33:2590–2601

Lowerre B (1976) The HARPY speech recognition system. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA

Maniezzo V (1999) Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem. INFORMS J Comput 11(4):358–69

Ow P, Morton T (1988) Filtered beam search in scheduling. Int J Prod Res 26:297–307

Pinedo M (1995) Scheduling: Theory algorithms, and systems. Prentice-Hall

Reddy D (1977) Speech understanding systems: A summary of results of the five-year research effort. Tech. rep., Department of Computer Science, Carnegie-Mellon University

Index