Can Operations Research Bring You to the Next Level? Basics and Applications

Vincent T'kindt
LIFAT, University of Tours
Tours, France
tkindt@univ-tours.fr

Patrick Marcel
LIFO, University of Orléans
Orléans, France
Patrick.Marcel@univ-orleans.fr

ABSTRACT

We focus on some intersections between two research fields: data management and operations research (OR). More precisely, we consider how advanced OR techniques can be used to solve optimization problems occurring in databases. We recall the basics of OR, introduce a few advanced approaches and, next, present two uses cases: (i) a heuristic stemming from OR for solving the graph edit distance problem and (ii) a detailed use case on Exploratory Data Analysis (EDA) where a heuristic stemming from OR is used to cope with large instances of a database exploration optimization problem.

1 INTRODUCTION

The literature abounds with works on the solution of optimization problems occurring in data management, like query optimization, concurrency control, scheduling in Map-Reduce systems, aggregation of multiple rankings, etc. Many of these problems were shown to be $\mathcal{NP}$-hard in the strong sense, ruling out the existence of exact polynomial-time algorithms to solve them, and led to the proposition of heuristic algorithms. The topic of query optimization is probably the best illustration of such problems, as it involves the solution of many theoretical and practical problems like testing query containment, choosing a query plan, choosing a join order, choosing optimization structures to materialize, choosing the minimum set of structures to answer a set of queries, to name a few. For instance, exact algorithms, like dynamic programming, and heuristic algorithms have been proposed to pick a join order (see e.g. [11, 14]) and are limited to small instances (e.g., a few dozens of joins, a few thousand query plans). The following example illustrates another case of $\mathcal{NP}$-hardness in data warehousing.

Example: derived horizontal partitioning in data warehouses. Assume we are given a simple data warehouse modeled as a relational star schema with a single dimension table and m fact tables. We are also given a set Q of selection queries Q_j which are frequently ran on this data warehouse. The aim of horizontal partitioning is to split the fact tables and next the dimension tables so that the processing of queries in Q is as fast as possible. Each query Q_j is defined by a set of n_j predicates $i, p_{j,i,k} = A_{i,k}\theta V$ with $A_{i,k}$ an attribute of dimension table k, $\theta \in \{=, <, >, \leq, \geq\}$ and $V \in Domain(A_{i,k})$. We also assume that to each query Q_j is associated a weight w_j reflecting its frequency: more often the query is ran, higher is the weight. Finally, to limit the fragmentation of the data warehouse, we assume that a bound W on the number of partitions to generate from the tables, is given. This problem is NP-hard in the strong sense ([3]) thus ruling out the existence of an exact polynomial-time algorithm to solve it.

More recently, in Exploratory Data Analysis (EDA), the problem of generating an optimized sequence of comparison queries has been considered in [5–7]. Again, this problem is shown to be strongly $\mathcal{NP}$-hard and exact as well as efficient heuristics are proposed.

Notions of $\mathcal{NP}$-hard, exact polynomial-time algorithm and heuristics are notions stemming from operations research (OR). OR can be briefly defined as dealing with the development of advanced analytical methods to solve decision or optimization problems. It is at the crossroad of mathematics and computer science and relates to the design of relevant optimization/decision models and of efficient solution algorithms. Starting with the basics of OR, recalling the aforementioned notions and more, the originality of this tutorial is to illustrate how advanced OR approaches can be used to improve the solution of optimization problems that occur in databases.

Contributions. This paper contributes by:

(1) providing researchers in data management with elements of operations research, including a few advanced OR approaches that can be used to improve the solution of an optimization problem.
(2) illustrating on two use cases how heuristics stemming from OR can be used to cope with database optimization problems: (i) in graph databases, to compute the graph edit distance, and (ii) in Exploratory Data Analysis, to generate exploration sessions over large instances.

Outline. The paper is organized as follows. First, we present the elements of operations research insisting on elements of complexity and basic and advanced classic solution algorithms (Section 2). Next, we present a heuristic stemming from OR for solving the graph edit distance (Section 3). Finally, we detail a use case occurring in EDA (Section 4) where heuristics stemming from OR can be used to solve the problem of generating exploration sessions. We conclude in Section 5.

2 OPERATIONS RESEARCH AT A GLANCE

The main part of this tutorial provides elements of operations research, elaborating but going beyond the previous tutorial at ADBIS 2023 ([18]).

2.1 Complexity

Assume we are given an optimization problem (P) generically defined as follows:

$$\begin{gathered}\text{Minimize } f(s)\\ \text{subject to}\\ s \in \mathcal{S} \subset \mathbb{R}^n\end{gathered}$$

with S the set of solutions and $f : \mathcal{S} \mapsto \mathbb{R}$ the objective function to minimize. We will focus on problems (P) for which set $\mathcal{S}$ is a discrete subset of $\mathbb{R}^n$. For the sake of clarity, we will consider minimization problems.

To illustrate what we call an optimization problem (P) , let us consider the following classic Traveling Salesperson Problem (TSP). Given a set C of n cities, and for any cities i and j their associated distance$d_{i,j}$, the objective is to find a tour s such that: (1) any city $j \in C$ appears exactly once in s and, (2) $\sum_{i=1}^{n} d_{s[i],s[i+1]}$ is minimum, with $d_{s[n],s[n+1]} = d_{s[n],s[1]}$ and $s[k]$ refers to the city in position k in s. In the TSP, the set $\mathcal{S}$ of solutions is defined as the set of permutations over $\{1, ..., n\}$ and $f(s) = \sum_{i=1}^{n} d_{s[i],s[i+1]}$. An *instance* for this problem is a given data set and its *size* is the number of elements, i.e number n of cities.

When a problem (P) has to be solved, the first task is to determine its complexity according to complexity theory ([12]). Roughly speaking, an optimization problem (P) is either:

(1) optimally solvable in polynomial time of the *instance size* (we say it belongs to class $\mathcal{P}$), or
(2) optimally solvable but in super-polynomial time of the *instance size* (we say it belongs to class $\mathcal{NP}$-hard).

Thus, for any problem $(P) \in \mathcal{P}$ there exists an *exact* algorithm which finds an optimal solution in $O(p(n))$ time with n the input size and p a polynomial. Consequently, solving (P) requires to find such an algorithm. There exists two kinds of $\mathcal{NP}$-hard problems: weakly $\mathcal{NP}$-hard problems and strongly $\mathcal{NP}$-hard problems. The former can be solved by an exact algorithm running in *pseudo-polynomial time*, i.e. in $O(p(n, ||I||))$ with $||I||$ a measure of the magnitude of the data and p a polynomial. For instance, in TSP, we can set $||I|| = \sum_{i,j} d_{i,j}$. The hardest problems are strongly $\mathcal{NP}$-hard problems, as any exact algorithm requires an *exponential* time to compute an optimal solution, i.e. requires $O(c^n), c \in \mathbb{N}$, or $O(n!)$ time. Solving an $\mathcal{NP}$-hard problem (P) leaves us two options:

(1) Design an *exact* algorithm to compute an optimal solution. In this case, the super-polynomial running time may prevent the algorithm from solving real-life instances of (P) .
(2) Design a *heuristic* algorithm that is a polynomial-time algorithm computing a solution *as close as possible to an optimal one.*

The TSP introduced above can be shown to be strongly $\mathcal{NP}$-hard ([12]) but has been well studied in the literature and is now optimally solvable for instances with up to thousands cities.

Now, let us introduce a second very classic problem called the Knapsack problem (KP). Assume we are given a set O of n objects and, for each object i a weight w_i and a profit p_i, as well as a knapsack of total capacity W. The objective is to find selection s of objects such that $s \subseteq O$, $\sum_{i \in s} w_i \leq W$ and $f(s) = \sum_{i \in s} p_i$ is maximum. This problem is weakly $\mathcal{NP}$-hard and can be optimally solved in $O(nW)$ time ([13]). It is computationally easier to solve than TSP as instances with up to hundreds of thousands of objects can be optimally solved.

2.2 Exact algorithms

Polynomial problems are not considered hereafter as they are mainly solved by dedicated algorithms and we rather present general techniques that are relevant for $\mathcal{NP}$-hard problems.
Mathematical programming ([16]) consists in defining a mathematical model of (P) that is next solved by a black-box solver. To illustrate, let us consider KP, and let x_j, $j = 1...n$, be boolean variables with $x_j = 1$ if object j is selected in the knapsack and 0 otherwise. Thus, KP can be formulated as follows:

$$\begin{aligned} &\text{Maximize } \textstyle\sum_{j=1}^{n} p_j x_j \\ &\quad \text{subject to} \\ &\quad\quad \textstyle\sum_{j=1}^{n} w_j x_j \leq W \\ &\quad\quad x_j \in \{0;1\}, \forall j = 1..n \end{aligned}$$

Experimentally, it can solve instances with up to several hundreds of objects. Mathematical programming is very interesting as a first approach to build an exact algorithm at the cost of, only, creating a model.

Dynamic Programming is a general technique which relies on recursively decomposing a problem into sub-problems and solving them independently ([4]). Let us illustrate this on TSP: we denote by $\texttt{OPT}[\ell, C']$ the optimal solution value when a set of cities C' has to be visited after city ℓ. As all cities belong to an optimal tour, we can assume that we start, e.g., with city number 1. We have:

$$\texttt{OPT}[\ell, C'] = \min_{j \in C'} \left(\texttt{OPT}[j, C' \backslash \{j\}] + d_{\ell,j} \right)$$
and
$$\texttt{OPT}[\ell, \emptyset] = d_{\ell,1}.$$

Solving TSP for a given set of cities $C = \{1, .., n\}$ requires to compute $\texttt{OPT}[1, C \backslash \{1\}]$, which can be done in $O(n2^n)$ time and space.

To complete this big picture of exact approaches, we put a last focus on *branching algorithms* ([10], [15]) that use a search tree to explore the solutions space. Among them, branch-and-price algorithms are of a particular interest with respect to the case study developed later on in this tutorial. Often, they make use of *column generation* to drive the exploration of the search tree. For the sake of clarity, we present this technique on a slight generalization of the TSP: assume we are given n cities and two salespeople who have to visit once and only once each city. Each salesperson has his own tour and visit a subset of the cities: the goal is to find two tours visiting all the cities so that the sum of the tour lengths is minimum. Suppose we want to get a lower bound on the optimal solution. We make use of a combinatorial mathematical programming formulation relying on binary variables $x_s = 1$ if tour s is achieved by one of the salespeople and 0 otherwise. A tour s is an ordered subset of the cities. Then, solving the corresponding mathematical programming formulation reduces to determining the only two variables x_s and $x_{s'}$ equal to 1 so that each city is visited but only once and the total cost is minimum. The issue with this formulation is that, to build the mathematical programming formulation, we need to enumerate all possible tours thus leading to create $O(2^n n!)$ variables: clearly, this formulation is exponential in size and cannot be solved in practice. Column generation avoids this issue by starting with a reduced set C of tours (columns) and iteratively adding improving columns.

The *master problem*, or restricted problem, consists in solving the mathematical programming formulation for a known set of columns/variables C thus leading to a bound on the optimal solution. Next, the *pricing problem* is solved in order to generate a column/variable: if this column can lead to improve the solution of the master problem, then the process is iterated. If not, the algorithm stops with the last computed bound as being the best known approximation of the optimal solution.
Usually, branching algorithms are very effective in solving hard optimization problems but require a real expertise on the problem.

2.3 Heuristic algorithms

Such algorithms enable to provide good approximate solutions in a reasonable, polynomial, running time. Very numerous categories of algorithms can be found in the literature and we rather focus on some very basic ones which may be relevant in the context of data management.

The simplest heuristics that can be designed are called *greedy heuristics* and compute a solution by iteratively taking a decision according to a given priority rule. To illustrate this principle, let us consider a well-known heuristic for KP, running in $O(n\log(n))$ time, which, to any object j, associates a priority $\alpha_j = \frac{p_j}{w_j}$. Next, this heuristic sorts the objects by decreasing value of their priority and selects the k first ones until the knapsack is filled up. Greedy heuristics are usually basic ones which can be easily improved by more elaborated heuristics but at the cost of an increase in the computational complexity. Their main advantage is to be able to scale up to very large size instances.

Local search heuristics ([1]) are also often considered as first heuristics. They take as input an existing solution of problem (P) and improve it by means of the iterative use of *neighborhood operators*. Let us denote by s^t the solution at iteration t and by $\mathcal{N}(s^t)$ the neighborhood of s^t. To define $\mathcal{N}(s^t)$ we need to use a neighborhood operator which generates solutions "close" to s^t. Next, the local search heuristic selects $s^{t+1} \in \mathcal{N}(s^t)$ such that $f(s^{t+1})$ improves over $f(s^t)$. Ideally, we take s^{t+1} as being the best solution in $\mathcal{N}(s^t)$. This process iterates until no improving solution is found.

Notice that a lot of different local search heuristics exist like tabu search, nature-inspired heuristics or matheuristics.

Also, we mention *branching heuristics* which rely on exact branching algorithms. There are numerous variants like beam search algorithms, limited discrepancy search or branch-and-greed algorithms to quote a few. All share the common feature of pruning heuristically the search tree representing the solution space in such a way that the number of explored nodes is polynomial in the instance size. Branching heuristics are known to be efficient and not too computationally demanding.

3 A USE CASE IN GED

In this section we consider a Graph Edit Distance (GED) problem and show how it can be efficiently solved by a modern heuristic ([8, 9]). A GED problem is a graph matching problem that consists in, given two attributed graphs G and G', determining how much they are similar. For example, solving such a problem can be used to compare two patterns (images, molecules, data) and determine whether or not they are similar: each pattern is modeled by an attributed graph and the two graphs are next compared.

Problem definition. An attributed and undirected graph is a 4-tuple $G = (V; E; \mu; \xi)$ where, V is the set of vertices, E is the set of edges, such that $E \subseteq V \times V$, $\mu : V \rightarrow L_V$ (resp. $\xi : E \rightarrow L_E$) is the function that assigns attributes to a vertex (resp. an edge), with L_V (resp. L_E) the set of all possible attributes for vertices (resp. edges). Given two graphs $G = (V; E; \mu; \xi)$ and $G' = (V'; E'; \mu'; \xi')$, solving the GED problem accounts for finding the least cost complete edit path that transforms the source graph G into the target graph G'. An edit path $\lambda(G, G') = \{o_1, ...o_k\}$ consists in a series of vertices and edges operations o_j (substitution, deletion, insertion) to transform G into G' (Figure 1).

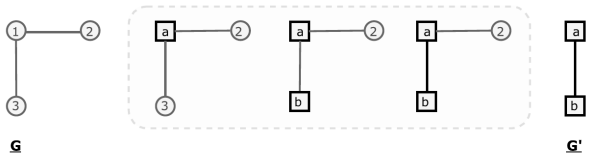

Figure 1: An edit path to transform G into G' ([8]): substitution of $v_1 \rightarrow v_a$, $v_3 \rightarrow v_b$ and $e_{1,3} \rightarrow e_{a,b}$; deletion of v_2 and $e_{1,2}$.

To each operation o_j is associated a dissimilarity cost $\ell(o_j)$ and solving the GED problem reduces to minimize:

$$d_{min}(G, G') = \min_{\lambda \in \Lambda(G,G')} \sum_{o_j \in \lambda} \ell(o_j),$$

i.e. to find the minimal dissimilarity cost for matching G and G'. This problem has been shown to $\mathcal{NP}$-hard.

Heuristic solution. It is possible to model the GED problem by means of mathematical programming that is exploited into a matheuristic framework called *local branching*. Such a modern heuristic consists of iteratively modifying the mathematical model by adding additional constraints to define neighborhoods in the solution space, which are explored using a black-box solver. It is experimentally shown to outperform all the existing standard heuristics from the literature notably on databases MUTA and PAH which contain graphs modeling molecules ([8]).

4 A USE CASE IN EDA

This section presents a particular optimization problem occurring in Exploratory Data Analysis (EDA) named the Traveling Analyst Problem (TAP, [5]). We show how column generation can inspire the design of an efficient heuristic for this problem.

$$\gamma_{a,agg(m)\rightarrow val}(\sigma_{b=val}(R)) \bowtie \gamma_{a,agg(m)\rightarrow val'}(\sigma_{b=val'}(R))$$

	sum of cases	
Continent	April	May
Africa	31598	92626
America	1104862	1404912
Asia	333821	537584
Europe	863874	608110
Oceania	2812	467

```
with comparison as
 (select t1.continent, April , May
  from
   (select month, continent, sum(cases) as April
    from covid where month = '4'
    group by month, continent) t1,
   (select month, continent, sum(cases) as May
    from covid where month = '5'
    group by month, continent) t2
  where t1.continent = t2.continent
  order by t1.continent)
select 'mean greater' as hypothesis from comparison
having avg(April)<avg(May);
```

Figure 2: Comparison query

Problem definition. The problem consists in finding sequences of interesting comparison queries [6]. *Comparison queries* (Figure 2) are extended relational queries of the form:

$$\gamma_{A,agg(M)\rightarrow val}(\sigma_{B=val}(R)) \bowtie \gamma_{A,agg(M)\rightarrow val'}(\sigma_{B=val'}(R))$$

where R is a relation with categorical attributes A, B, a numerical attribute M, agg is an aggregate function, and $val, val' \in dom(B)$. $\bowtie$ denotes the inner join operator and γ denotes the grouping / aggregation operator.

Each query $q_i \in Q_R$, with Q_R the set of all comparison queries over R, is defined by a positive time cost t_i and a positive interestingness p_i. Each pair of queries (q_i, q_j) is associated with a metric $d_{i,j}$ representing the cognitive distance of browsing from q_i to q_j. We further define $N = |Q_R|$. The TAP problem consists in finding a sequence of queries from Q_R that maximizes the total interestingness $\bar{I}$, minimize the total cost $\bar{C}$ and minimize

the total distance $\bar{D}$ between the queries of the sequence. This is a multiobjective problem that is strongly $\mathcal{NP}$-hard. To compute a Pareto optimum for these objectives it is reasonable, from the user point of view, to use the ϵ-constraint ([19]) as follows: maximize $\bar{I}$ under the constraints that $\bar{T} \leq \epsilon_t$ and $\bar{D} \leq \epsilon_d$ with ϵ_t and ϵ_d two given bounds.

Exact solution. We first present a mathematical programming formulation (MIP) for the TAP that relies on two sets of boolean variables. For each $q_i \in Q_R$, variable y_i is equal to 1 if q_i is selected in the solution; 0 otherwise. For any pair of queries $(q_i, q_j) \in {Q_R}^2$, variable $x_{i,j}$ is equal to 1 if q_i precedes q_j in the solution; 0 otherwise. The aim is to maximize the total interestingness $\bar{I} = \sum_{q_j \in Q_R} p_j y_j$ and to minimize the total processing cost $\bar{C} = \sum_{q_j \in Q_R} t_j y_j$ and the total cognitive distance $\bar{D} = \sum_{q_i, q_j \in Q_R} d_{i,j} x_{i,j}$. Finding values for the y_j's and the $x_{i,j}$'s amounts to compute a permutation π of $n \leq N$ queries.

More constraints are introduced (though not presented here), e.g., for preventing the same query to be selected twice or for ensuring that the sequence is connected. Importantly, this MIP is agnostic of the form of queries in Q_R. In other words, the particular syntax of comparison queries is not taken into account. The MIP can be solved to optimality by any black-box commercial solver for small instances (i.e., small sets of queries). Clearly, greedy heuristics like the one for KP can be used to cope with larger instances. In both cases, this requires to generate the set of all comparison queries which may be too much time consuming to handle large databases.

Coping with larger instances. We can bring the solution of the TAP to the next level by leveraging from advanced OR techniques like column generation. Here, the idea is not to get a lower bound as usual in such an approach, but to heuristically solve the problem, even for large databases. The key point is to not consider the whole set Q_R but rather to iteratively generate improving queries. This requests exploiting the form of the queries to be generated, namely comparison queries.

We present the main lines of the proposed heuristic. It consists of three main steps. First, a small initial subset $Q \subset Q_R$ of n_q queries is built by means of very simple heuristics. This set is used to initialize the iterative process: at each iteration, an *improving* comparison query q is generated and added to Q until no more *improving* queries can be generated (e.g., if a given time limit is reached). Next, the TAP is solved on the resulting set Q of queries by means of an efficient heuristic.

The key point of such an approach relates to the generation of *improving* queries. The underlying idea is to generate q that would improve the optimal solution to the TAP built on Q. Said differently, the optimal solution on $Q \cup \{q\}$ has a higher interestingness than the one on Q. We will show how, by a genuine mathematical modeling of the comparison queries, we can leverage from mathematical programming to perform such a task.

Notice that this mathematical modeling is based on specific estimations of the cost, interestingness and distances of comparison queries. These estimations are modeled as constants added to the mathematical program generating the improving queries. We will show how to devise these constants using state-of-the-art approaches for interestingness [17] and distance [2], and a simulation of the use of physical optimization structures to estimate the cost.

We will conclude with computational experiments showing that this approach can be successfully applied on relatively large databases providing even better results than traditional optimization algorithms relying on the knowledge of the whole set Q_R.

5 CONCLUSION

This paper shows how advanced operations research (OR) techniques can be used to solve optimization problems occurring in databases. After recalling the basics of OR and presenting a few advanced approaches, two use cases are presented. The first use case concerns graph databases, where a heuristic is introduced for solving the Graph Edit Distance problem. The second use case pertains to Exploratory Data Analysis (EDA) where a heuristic is used to cope with large instances of a database exploration optimization problem.

6 ACKNOWLEDGEMENTS

This work was partially supported by JUNON Program (DATA / LIFO) with financial support of Région Centre-Val de Loire, France.

REFERENCES

[1] E. Aarts and J.-K. Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.
[2] J. Aligon, M. Golfarelli, P. Marcel, S. Rizzi, and E. Turricchia. Similarity measures for OLAP sessions. *Knowl. Inf. Syst.*, 39(2):463–489, 2014.
[3] L. Bellatreche, K. Boukhalfa, and P. Richard. Data partitioning in data warehouses: hardness study, heuristics and oracle validation. In I.Y. Song, J. Eder, and T.M. Nguyen, editors, *DaWak 2008*, volume 5182 of *Lecture Notes in Computer Science*, pages 87–96. Springer, 2008.
[4] R. Bellman. *Dynamic programming*. Dover publication, 2003.
[5] A. Chanson, N. Labroche, P. Marcel, and V. T'Kindt. Comparison Queries Generation Using Mathematical Programming for Exploratory Data Analysis . *IEEE Transactions on Knowledge & Data Engineering*, 36(12):7792–7804, December 2024.
[6] A. Chanson, P. Marcel, P. Rizzi, and V. T'kindt. Automatic generation of comparison notebooks for interactive data exploration. *Proceedings of the 25th International Conference on Extening Database Technology (EDBT), ISBN 978-3-89318-085-7 on OpenProceedings.org*, 2022.
[7] A. Chanson, P. Marcel, and V. T'kindt. Matheuristics for solving the traveling analyst problem. *23rd Conference of the French Society on Operations Research and Decision Aid (ROADEF 22), Lyon*, 2022.
[8] M. Darwiche, D. Conte, R. Raveaux, and V. T'kindt. A local branching heuristic for solving a graph edit distance problem. *Computers and Operations Research*, 106:225–235, 2019.
[9] M. Darwiche, D. Conte, R. Raveaux, and V. T'kindt. Graph edit distance: Accuracy of local branching from an application point of view. *Pattern Research Letters*, 134:20–28, 2020.
[10] D. Feillet. A tutorial on column generation and branch-and-price for vehicle routing problems. *4OR*, 8:407–424, 2010.
[11] H. Garcia-Molina, J.D. Ullman, and J. Widow. *Database systems: the complete book*. Prentice Hall, 2008.
[12] M.R. Garey and D.S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. Freeman, San Francisco (USA), 1979.
[13] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004.
[14] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, November 2015.
[15] D.R. Morrison, S.H. Jacobson, J.J. Saupe, and E.C. Sewell. Branch-and-bound algorithms: a survey of recent advances in searching, branching and pruning. *Discrete optimization*, 19:79–102, 2016.
[16] G.L. Nemhauser and L.A. Wolsey. *Integer and combinatorial optimization*. Wiley, 1988.
[17] B. Tang, S. Han, M. Lung Yiu, R. Ding, and D. Zhang. Extracting top-k insights from multi-dimensional data. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1509–1524. ACM, 2017.
[18] V. T'kindt. When operations research meets databases. In Alberto Abelló, Panos Vassiliadis, Oscar Romero, and Robert Wrembel, editors, *Advances in Databases and Information Systems - 27th European Conference, ADBIS 2023, Barcelona, Spain, September 4-7, 2023, Proceedings*, volume 13985 of *Lecture Notes in Computer Science*, pages 34–41. Springer, 2023.
[19] V. T'kindt and J.-C. Billaut. *Multiobjective scheduling: theory, models and algorithms*. Springer, 2006.