Optimization Methods and Software

ISSN: (Print) (Online) Journal homepage: https://www.tandfonline.com/loi/goms20

Improving dynamic programming for travelling salesman with precedence constraints: parallel Morin–Marsten bounding

Yaroslav. V. Salii & Andrey S. Sheka

To cite this article: Yaroslav. V. Salii & Andrey S. Sheka (2020): Improving dynamic programming for travelling salesman with precedence constraints: parallel Morin–Marsten bounding, Optimization Methods and Software, DOI: 10.1080/10556788.2020.1817447

To link to this article: https://doi.org/10.1080/10556788.2020.1817447

Published online: 14 Sep 2020.

OPTIMIZATION METHODS & SOFTWARE
https://doi.org/10.1080/10556788.2020.1817447

Taylor & Francis
Taylor & Francis Group

Improving dynamic programming for travelling salesman with precedence constraints: parallel Morin–Marsten bounding

Yaroslav. V. Salii [a,b] and Andrey S. Sheka [b,c]

[a]McGill University, Montreal, Canada; [b]Krasovskii Institute of Mathematics and Mechanics UB RAS; [c]Ural Federal University, Yekaterinburg, Russia

ABSTRACT

The precedence constrained traveling salesman (TSP-PC), also known as sequential ordering problem (SOP), consists of finding an optimal tour that satisfies the namesake constraints. Mixed integer-linear programming works well with the 'lightly constrained' TSP-PCs, close to asymmetric TSP, as well as the with the 'heavily constrained' (Gouveia, Ruthmair, 2015). Dynamic programming (DP) works well with the *heavily constrained* (Salii, 2019). However, judging by the open TSPLIB SOP instances, the worst for any method are the 'medium'.

We implement a *parallel* Morin–Marsten *branch-and-bound* scheme for DP (DPBB). We show how the lower bound heuristic parameterizes DPBB's worst-case complexity and DPBB 'inherits' the *abstract travel cost aggregation* feature of the DP, permitting its direct use with both the conventional and *bottleneck* TSP-PC.

The scheme was tested on TSPLIB instances, with best known upper bounds (TSP-PC), or those found by *restricted* DP (Bottleneck TSP-PC), and lower bounds from a greedy-type heuristic. Our OpenMP-based parallel implementation achieved 20-fold speedup for larger instances. We close the long-standing `kro124p.4.sop`[1] (conventional TSP-PC) and both `kro124p.4.sop` and `ry48p.2.sop` (Bottleneck TSP-PC).

ARTICLE HISTORY
Received 29 October 2019
Accepted 25 August 2020

KEYWORDS
Precedence constraints; travelling salesman; sequential ordering problem; dynamic programming; brand-and-bound; parallelism

2010 MATHEMATICS SUBJECT CLASSIFICATIONS
90-08; 06A06; 90C39

Introduction

The Travelling Salesman Problem with Precedence Constraints (TSP-PC), or Sequential Ordering Problem (SOP) [18], amounts to visiting every city from the set $1..n$, $n \in \mathbb{N}$, exactly once so as to minimize the total *travel cost* while satisfying *precedence constraints* $P = (1..n, <_P)$ where $a <_P b$ is interpreted as 'city a *must* be visited before city b'.

The main benchmarks for TSP-PC are the SOP instances from TSPLIB [45], which range from 7 to 378 cities, and the ones with more cities from SOPLIB [41], which

CONTACT Y. V. Salii yvs314@gmail.com Krasovskii Institute of Mathematics and Mechanics UB RAS, ul. S. Kovalevskoi 16, 620108 Yekaterinburg, Russia Ural Federal University, pr. Lenina 51, 620083 Yekaterinburg, Russia

Supplemental data for this article can be accessed here. https://doi.org/10.1080/10556788.2020.1817447

range from 100 to 700 cities. Note that city quantity alone is *insufficient* to characterize a given instance as 'hard' or 'easy': no optimal solution is known for the relatively 'small' 48-city `ry48p.2.sop` (TSPLIB), yet ostensibly much larger 700-city instances `R.700.1000.[30,60]` (SOPLIB) have been solved to optimality [21]. Each instance's specific precedence constraints play a major part [46] by limiting the search space, expressed as e.g. the number of feasible solutions or the number of dynamic programming states.

State-of-the-art exact solutions for both libraries were obtained through branch-and-cut in [21]; upper bounds for SOPLIB instances were improved upon in [54] through an ant colony algorithm supplemented with simulated annealing and then in [35] by tree search. Other recently used approaches include branch-and-bound [28,43], a hybrid approach involving multivalued decision diagrams and constraint programming [17,30], and dynamic programming [11,46].

A long-standing TSPLIB SOP instance `ry48p.3.sop` was recently solved to optimality [46] through dynamic programming (DP); this instance was pointed out as the 'smallest' open TSPLIB instance in view of time and space complexity estimates [46, Section 4]; its solution took 120 GB of RAM and circa $2\frac{1}{2}$ hours.

However, the same complexity estimates [46, Supp. 2] ruled out direct applications of DP to close the harder problems, it ought to be hybridized and parallelized or abandoned. Note that it is *space* (memory), not time that is the bottleneck for DP, e.g. a test DP run for the open `kro124p.4.sop` exhausted the available 256 GB of fast memory in circa 5 hours[2] [46]. In fact, it takes 396 GB to solve it on a single computer, with new and improved hash table implementation from [32] saving half the memory compared with [46, Table 1], or 47 hours and a total of 1.876 TB RAM on 18 computation nodes, with an average of 106 GB per node [11], through the distributed scheme [15].

Although the distributed scheme [15] can combat the lack of memory on a single node to an extent, the limit to this is clear-cut: some of the larger TSPLIB problems (100+ cities) have *lower* bounds on space complexity (projected memory usage) as high as 2^{60} [46, Supp. 1, Table A.2], yet no supercomputer is presently expected to have a ZETTABYTE to spare.

In 1976, one hybrid method was introduced [42], a branch-and-bound scheme making use of a pre-computed instance's *upper bound* and a set of *lower bounds* on DP states to eliminate ('fathom') those proven to take no part in any optimal solution; we refer to it as DPBB. It has not been used to tackle TSP-PC since the 1990s [6,39] (the latter paper also described handling *time windows* in addition to precedence constraints), however, it was relatively recently applied in [7] to a problem of sequencing in- and outbound trucks at a *cross docking* terminal. Finally, an approach similar to DPBB was efficiently applied to TSP with Time Windows (TSP-TW) in [3] and to a similar problem with a different, time-dependent objective function in [57].

Scholium: TSP-TW should not be conflated with TSP-PC. In TSP-TW, each city $i \in 1..n$ may only be visited during its *time window* $[a_i, b_i] \subset \mathbb{R}$, see the details in e.g. [3]. Although the time windows *induce* a partial order on the cities, which is used e.g. to eliminate the infeasible states during preprocessing, it does not constitute a reduction of TSP-TW to TSP-PC or the vice versa: ($\Rightarrow$) clearly, a partial order does not *uniquely* define the time

windows, and ($\Leftarrow$) it is unclear if each kind of partial order can be represented by some time windows, although the authors are not aware of specific counterexamples.

Algebraic abstraction for DP in TSP-like problems. One special feature of DP is a way of relaxing the conventional *sum* of travel costs to an abstract operation, which has to be nondecreasing [52] or nondecreasing and associative [46, Theorem 3], not unlike a generalization of the Chu–Liu–Edmonds minimum spanning arborescence algorithm [20] or the algebraic framework designed for minimum spanning tree problems [25].

This 'abstract aggregation' [46,52] allows for direct application of essentially the same procedure to both conventional min-sum TSP-PC and the minmax Bottleneck TSP (see the general reference in [24, Ch. 15] and newer results [33]) with precedence constraints (BTSP-PC), while retaining all the complexity bounds [46, Proposition 1,2].

One can reason of applications of Bottleneck TSP-PC in the same vein as BTSP compared to TSP: where TSP minimizes the *total time* required to visit all cities, BTSP is the minimum *cycle time*, how frequently could one complete the tours assuming there are as many salesman as there are the cities; likewise, if TSP is the total energy expenditure of an electric vehicle, then the BTSP solution is the minimum energy capacity necessary to complete the tour assuming the vehicle recharges at every city.

There are few studies of BTSP-PC or its generalizations, probably due to it being complicated to handle both the bottleneck (minimax) objective function and precedence constraints. The authors are only aware of the works by A. G. Chentsov and his coauthors, e.g. [16,48,53], which should also be noted for their treatment of Generalized[3] BTSP-PC.

Contribution.

(1) We introduce *abstract travel cost aggregation* and *precedence constraints* into the Morin–Marsten branch-and-bound scheme for dynamic programming in TSP, thereby obtaining a framework for solving any TSP-PC-like problem with the travel cost aggregation that is *nondecreasing*, *associative* and *commutative*. We also establish its worst-case *time* (Proposition 3) and *space* (Proposition 2) complexities.
(2) We make explicit the time–space tradeoff in choosing whether to *memoize* or not the routing decisions in DP for TSP-PC: memoization saves at most $\mathcal{O}(n^2)$ time (see Proposition 1), where n is the number of cities, but uses additional space proportional to the number of DP states, which is typically exponential in n.
(3) Using the DPBB framework for min-sum TSP-PC with the *best known* upper bounds [21], we get exact solutions to 22 of 41 TSPLIB instances, in particular, we close the long-standing `kro124p.4.sop`[4]; in the Bottleneck TSP-PC statement, with the upper bounds obtained by Restricted DP [46, Section 3.3], we close both `kro124p.4.sop` and `ry48p.2.sop`, among the 22 of 41 TSPLIB instances.
(4) We adapt a shared-memory, OpenMP-based parallelization scheme of [48] for DPBB, which greatly reduces run times – up to 17–35-fold, depending on instance, on 36 processor cores; when used for DP, it consistently reduces run times 8-fold.

Paper structure. Section 2 introduces a rigorous statement of *abstract aggregation* TSP-PC and the DPBB framework building on 'non-bounded' DP [46] for the abstract aggregation statement. It then describes the worst-case complexity for the DPBB framework.

Section 1 describes a shared-memory parallelization scheme for DP and DPBB based on [48, Section 5]. Section 2 describes how do DP and DPBB scale as the number of processor cores increases (Section 3.1), what good were 4 TB RAM for backward and forward statements of *abstract aggregation* DPBB (Section 3.2), and how does DP compare to DPBB time- and memory-wise (Section 3.3). The Supplement lists the upper bounds obtained by a form of *beam search* heuristic, Restricted DP [46, Section 3.3], as applied to the BTSP-PC statement of TSPLIB SOP instances.

1. Dynamic programming for TSP-PC and the Morin–Marsten branch-and-bound scheme

The purpose of this section is to sketch the main ideas of the Morin–Marsten Branch-and-Bound Scheme for DP [42], also called Bounded DP [7]. To distinguish it from the 'ordinary' DP [12,46], the latter will be occasionally called 'non-bounded'.

Dynamic programming for TSP-PC is described in the notation of [46], however, we omit its elements reflecting time- or (past) sequence-dependent travel cost functions to save on book-keeping. Equality by definition is denoted $\triangleq$. *Ordered tuples* are written in parentheses, e.g. $(a_1, a_2, \ldots, a_{\text{last}})$. *Integer interval* from $a \in \mathbb{Z}$ to $b \in \mathbb{Z}$ is denoted $a..b$.

1.1. Problem statement

The agent must visit the cities $1..n$, $n \in \mathbb{N}$, starting at a separate, special city 0 (the depot) and finishing at another special city $\mathbb{t} = n + 1$ (the terminal). We do not explicitly introduce any graphs, but, where convenient, write *arc* (a, b) as a shorthand for 'travel from city a to city b'.

For two distinct cities a and b, the *cost* of travelling from a to b is denoted $\mathfrak{c}(a, b)$; if an arc (a, b) *contradicts* the precedence constraints, e.g. $b <_P a$, its cost is undefined. In TSPLIB and SOPLIB, the travel costs are nonnegative integers.

1.1.1. Precedence constraints

Precedence constraints are expressed by a partially ordered set[5] $P = (1..n, <_P)$; where the context allows, we do not distinguish between the order P and its relation $<_P$. For any two cities $a, b \in 1..n$, the relation $a <_P b$ means that a *must* be visited before b in any feasible solution. Note that special cities 0 and $\mathbb{t}$ are not included in P since their positions are fixed. We find it easier to deal with these as fringe cases.

One formal way of measuring how 'strict' are the given precedence constraints is the *density* $\rho(P)$ introduced in [38] and designed to reflect how close is the given partial order P to the extreme, a *linear* order where there is nothing to sequence anymore. It is defined a fraction of the maximum possible number of comparable cities that is already present in P,

$$\rho(P) \triangleq \frac{|\leq_P|}{n(n-1)/2}.$$

To the best of the authors' knowledge, the densities of TSPLIB SOP instances were first computed in 2017, in [49]; for convenience, these are listed in Table 6 in this paper.

Another way of measuring this *strictness* is the *width* parameter $w(P)$, defined as the maximum number of cities incomparable in P – the *maximum antichain cardinality*; for

details on width and antichains, see [10, Section 1.3.2]. The closer the width w is to the number of cities n, the less strict are the precedence constraints: in the 'unconstrained' (asymmetric) TSP, $w = n$, whereas if $w = 1$, then the cities' order is already fixed by the precedence constraints.

The relation between the *width* w and the complexity of DP for precedence-constrained scheduling was first observed in [55]; these results were adapted to DP for TSP-PC in [46,49]. For TSPLIB SOP, the widths were first computed in [49]; for convenience, we list them in Table 6. In addition, the column #w in Table 6 and other tables in this paper is the instance's number in their list sorted by increasing w; the greater this number, the harder is the instance for DP and derived methods due to increasing lower bound on space complexity.

Feasible solutions

A *solution* of a TSP-PC instance may be viewed as a *permutation* α of its cities $1..n$, where α_i denotes the city located at ith place in α and α_a^{-1} denotes the index of city a in α. A permutation is called *feasible* if it satisfies precedence constraints P as follows: $\forall a, b \in 1..n\ (a <_P b) \Rightarrow \alpha_b^{-1} < \alpha_a^{-1}$; denote by $\mathbb{A}$ the set of all such permutations. Naturally, not all of the $n!$ permutations are feasible. However, it is #P-hard to compute how many are in the general case: a feasible solution maps bijectively onto a *linear order* on $1..n$, that is, a *linear extension* of P, and it is #P-hard to find their number [8].

TSP-PC with Abstract Travel Cost Aggregation

As the agent visits the cities, the total travel cost is tracked through the *aggregation operation* $\oplus$: for a solution α, the *objective function* in TSP-PC is the total cost, $\mathfrak{C}[\alpha] = \mathfrak{c}(0, \alpha_1) \oplus \mathfrak{c}(\alpha_1, \alpha_2) \oplus \cdots \oplus \mathfrak{c}(\alpha_{n-1}, \alpha_n) \oplus \mathfrak{c}(\alpha_n, \mathbb{t})$, thus the whole problem is

$$\textbf{Find } \alpha^* \in \mathbb{A} \textbf{ such that } \alpha^* = \operatorname*{argmin}_{\alpha \in \mathbb{A}} \mathfrak{C}[\alpha]. \qquad \text{(TSP-PC)}$$

If we set $a \oplus b \triangleq a + b$, we get the conventional TSP-PC (SOP). Setting $a \oplus b \triangleq \max\{a, b\}$, as in the *idempotent analysis* [31], yields the Bottleneck TSP-PC. For DP to be valid, the aggregation operation $\oplus$ must be *associative*, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ and *nondecreasing*, see [46, Theorem 3]. In addition to $+$ and $\max\{a, b\}$, we could also use multiplication, e.g. $a \oplus b \triangleq ab$, however, the authors are unaware of the applications that might require such a model.

To avoid further complications to the problem nomenclature, we retain the name TSP-PC for the problem with *abstract aggregation* throughout this paper, however, we require this operation $\oplus$ to be not only *associative* and *non-decreasing* but also *commutative* since the latter is required by the lower bound we use in the branch-and-bound scheme for DP in Section 1.3.2.

1.2. 'Non-bounded' dynamic programming for TSP with precedence constraints

Dynamic programming breaks down an instance of TSP-PC into a family of smaller interconnected subproblems called DP *states* and then computes each state's *value* based on those 'directly preceding' it; in TSP-like problems, these states can be viewed as 'sets with an

interface' [58]. In *forward* DP following Held and Karp [26], a state (K, x) represents a 'sub-instance' of starting at 0, optimally walking through the cities in K, $K \subset 1..n$, and finishing at city x. In *backward* DP following Bellman [5], a state (x, K) represents a sub-instance *starting* at x, optimally visiting the cities in K, and finishing at $\mathbb{t} = n + 1$.

For TSP-PC, the *forward* DP was first formulated in [34] and the *backward* in [14]; the latter statement could also treat the generalized version, with clusters instead of cities. Both statements (i) have the same complexity[6] and (ii) produce the same optima, though not necessarily the same solutions .[7] In DPBB, the property (i) is not preserved, and the time and space complexity may thus vary, as observed in Section 2.

Scholium: A special feature of DP in TSP-PC is the fact that not all $K \subset 1..n$ need be considered, which reduces the number of states – the *space* complexity – depending on the specific precedence constraints P. This is possible because certain states may only result in *infeasible* routes and thus are safe to omit, which is indirectly established by DP validity theorems such as [52, Theorem 1] or [46, Theorem 3].

Throughout this paper, we default to the *forward* DP, where a state (K, x) represents starting at 0, walking through K, and finishing at x. In view of precedence constraints, it is enough to consider as *task sets* K just the *order ideals* of P,

$$\mathcal{I} \triangleq \{I \subseteq 1..n \mid \forall\, i \in I\, \forall\, j \in 1..n\, (j <_P i) \Rightarrow (j \in I)\}.$$

In *backward* DP, a dual notion of *order filters* is used; for more detail on order ideals and filters, see e.g. [10, § 1.4.2].

Unless there are no precedence constraints at all ($<_P = \varnothing$), there are *strictly fewer* order ideals than subsets of $1..n$. In particular, a single constraint, e.g. $<_P = \{(a, b)\}$ (visit city a before city b, do not restrict the other), gives a 25% reduction, $|\mathcal{I}| = \frac{3}{4}|\mathcal{P}(1..n)|$, which may be derived from the formulas for series-parallel partial orders [55, Section 3.1] or by a direct combinatorial argument. However, in general it is #P-hard to enumerate $\mathcal{I}$ [44].

In addition to restricting the *task sets* to the *order ideals* of P, in a state (I, x), the *interface* x must also satisfy the precedence constraints: recall that it is the position of the agent after visiting the task set I and it must be feasible to visit it *after* the cities in I. This is handled through the *feasible extension operator*

$$\mathbf{E}[I] \triangleq \{m \in 1..n \setminus I \mid I \cup \{m\} \in \mathcal{I}\} = \mathrm{Min}[1..n \setminus I] \quad \text{[49, Theorem 1]},$$

where $\mathrm{Min}[K]$ picks the $<_P$-*minimal* elements, see e.g. [51, Definitions 2.1.5, 2.3.2], and the (forward) DP states are defined and 'valued' as follows:

$$\mathcal{S} \triangleq \{(I, x) \in \mathcal{I} \times 1..n \mid (I \in \mathcal{I} \setminus \{1..n\}) \wedge (x \in \mathbf{E}[I])\} \cup \{(1..n, \mathbb{t})\},$$

$$v(I, x) \triangleq \begin{cases} \mathfrak{c}(0, x), & I = \varnothing; \\ \min_{\alpha \in \mathbb{A}_I} \{\mathfrak{c}(0, \alpha_1) \oplus \mathfrak{c}(\alpha_1, \alpha_2) \oplus \cdots \oplus \mathfrak{c}(\alpha_{|I|}, x)\}, & \text{otherwise}; \end{cases} \tag{1}$$

here $\mathbb{A}_I$ is the set of solutions to the subproblem (I, x), the *feasible* permutations of I; effectively, these are the prefixes of certain feasible routes over the whole $1..n$.

The Bellman function $\mathrm{BF}(I, x)$ then provides the means of computing the value of a state (I, x) through the *values* of the states it *covers*, much quicker than it would take to sift

through all the feasible permutations. To define it in the precedence-constrained setting, we need a way to get, from a state (I, x), to those *covered* by it without encountering *infeasible* states. This is resolved through the *feasible entry point* operator

$$\mathbf{I}[I] \triangleq \{m \in I \,|\, I \setminus \{m\} \in \mathcal{I}\} = \mathrm{Max}[I] \quad [49, \text{Theorem 2}],$$

where $\mathrm{Max}[K]$ picks the $<_P$-*maximal* elements, see e.g. [51, Definitions 2.1.5, 2.3.2].

Thus the Bellman function is defined as

$$\mathrm{BF}(I, x) \triangleq \min_{m \in \mathbf{I}[I]} \left\{ v(I \setminus \{m\}, m) \oplus \mathfrak{c}(m, x) \right\};$$

and the *validity of the dynamic programming* is nothing but the fact that a state's *Bellman function* actually **matches** its *value* – except for those in $\mathcal{S}_0$, which serve as the initial conditions. The following theorem was proved in [46] using the techniques established in [52, Theorem 1].

Theorem (Forward DP Validity for TSP-PC with Abstract Travel Cost Aggregation): *If $\oplus$ is nondecreasing in its first argument and left-associative, then, for all $(I, x) \in \mathcal{S} \setminus \mathcal{S}_0$, we have*

$$v(I, x) = \mathrm{BF}(I, x) = \min_{m \in \mathbf{I}[I]} \left\{ v(I \setminus \{m\}, m) \oplus \mathfrak{c}(m, x) \right\}. \tag{2}$$

The DP and DPBB algorithms proceed cardinality-wise, and it is convenient to partition both the order ideals set $\mathcal{I}$ and the state set $\mathcal{S}$ by cardinality.

A subscripted ideals set, e.g. $\mathcal{I}_k$, contains all cardinality k ideals; this notation carries over to states, thus, $\mathcal{S}_k$ are all states associated with cardinality k ideals; call $\mathcal{S}_k$ the kth state space *layer*. The solution then proceeds from the initial conditions in $\mathcal{S}_0$ all the way through to the complete problem $\mathcal{S}_n = \{(1..n, \mathbb{t})\}$.

Algorithm 1 Dynamic programming for abstract aggregation TSP-PC

1: INIT: $\mathcal{I}_0 = \{\varnothing\}$, $\mathcal{S}_0 = \{(\varnothing, x), x \in \mathbf{E}[\varnothing]\}$,
2: $\forall (\varnothing, x) \in \mathcal{S}_0$ $v(\varnothing, x) = \mathfrak{c}(0, x)$, $\mathcal{I}_1 = \mathbf{E}[\varnothing]$.
3: **for all** $k \in 1..n-1$ **do**
4: **for all** $I \in \mathcal{I}_k$ **do**
5: COMPUTE $\mathbf{E}[I]$
6: **for all** $x \in \mathbf{E}[I]$ **do**
7: ADD $I \cup \{x\}$ TO $\mathcal{I}_{k+1}$
8: ADD (I, x) TO $\mathcal{S}_k$
9: COMPUTE $v(I, x)$
10: COMPUTE $v(1..n, \mathbb{t})$
11: RECOVER SOLUTION BY v.

After v is completely computed, one must *recover* a solution from the 'graph' of v because the DP procedure in Algorithm 1 does not *memoize* (store) routing decisions made when computing v (Step 9); this is done to save memory and was, apparently, first proposed

in [26]. This recovery starts with finding a (not necessarily unique) city z_n that achieves the minimum in $v(1..n, \mathbb{t})$; then, it searches for the next z_{n-1} that achieves the minimum in $v(1..n \setminus \{z_n\}, z_n)$, and so forth until it finds $z_1 = \operatorname{argmin}_{x \in \mathbf{I}[1..n \setminus \{z_2,\dots,z_n\}]} \{v(\varnothing, x) \oplus \mathfrak{c}(x, z_2)\}$; see the description and sample computation in [46, Supp. C].

In [46, Proposition 2], a trivial upper bound on this search's complexity was used – *recomputing* the whole v ($\mathcal{O}(|\mathcal{I}| \cdot wn)$) – though empirically it never took this long, barely a few milliseconds' worth. Indeed, it is actually polynomial in n and w, *independently* of $|\mathcal{I}|$.

Proposition 1: *Solution recovery takes $\mathcal{O}(nw)$, where n is the number of cities and w is the width*[8] *of precedence constraints $P = (1..n, <_P)$.*

Proof: The search is conducted n times. When searching for next z_{n-i}, the search space consists of the states *covered* by the incumbent $(1..n \setminus \{z_n, \dots, z_{n-i+1}\}, z_{n-i+1})$, that is, $(I, y) \in \mathcal{S} \mid y \in \mathbf{I}[1..n \setminus \{z_n, \dots, z_{n-i+1}\}]$, of which there are at most w since $\mathbf{I}$ produces the set of *maximal* elements [49, Theorem 2], which is an antichain *by definition*. ■

In view of the above, *memoizing* routing decisions in TSP-PC or its generalizations as proposed in e.g. [34] or [40, Ch. 8] is not advisable vis-a-vis post factum recovery since memoization adds a constant factor to space complexity (memory usage): saving $\mathcal{O}(nw)$ time, we lose $\mathcal{O}(|\mathcal{I}| \cdot w)$ space; the latter is typically *exponential* in n and it is thus important to save even a small fraction of that cost.

1.2.1. Backward dynamic programming

Backward DP for TSP-PC, following [5], requires a set of definitions *dual* in the order-theoretic sense to the forward as described in the previous section. Start with *order filters*

$$\mathcal{F} \triangleq \{F \subseteq 1..n \mid \forall\, i \in F\, \forall\, j \in 1..n\, (j >_P i) \Rightarrow (j \in I)\},$$

and then replace every structure and operator with its *dual*, which we denote by subscripting the latter with $_\mathrm{b}$, as follows:

$$\begin{aligned}
\mathbf{E}_\mathrm{b}[F] &\triangleq \{m \in 1..n \setminus F \mid F \cup \{m\} \in \mathcal{F}\} = \operatorname{Max}[1..n \setminus F],\\
\mathbf{I}_\mathrm{b}[F] &\triangleq \{m \in F \mid F \setminus \{m\} \in \mathcal{F}\} = \operatorname{Min}[F],\\
\mathcal{S}_\mathrm{b} &\triangleq \{(x, F) \in 1..n \times \mathcal{F} \mid (F \in \mathcal{F} \setminus \{1..n\}) \wedge (x \in \mathbf{E}_\mathrm{b}[F])\} \cup \{(0, 1..n)\},\\
v(x, F) &= \begin{cases} \mathfrak{c}(x, \mathbb{t}), & F = \varnothing;\\ \min_{m \in \mathbf{I}_\mathrm{b}[F]} \{\mathfrak{c}(x, m) \oplus v(m, F \setminus \{m\})\}, & \text{otherwise;} \end{cases} \qquad (\mathrm{BF}_\mathrm{b})
\end{aligned}$$

Recall that the *backward* state (x, F) denotes the situation of the agent residing at x and designing to optimally walk through the cities in F and then visit the terminal $\mathbb{t}$. For the *backward* DP to be valid, the operation $\oplus$ has to be nondecreasing in its *second* argument and *right*-associative (cf. Theorem 1).

Thus, to have the option of freely choosing between backward and forward statements, the aggregation $\oplus$ has to be nondecreasing in *both* its arguments and *associative*; these properties are satisfied by $+$ and $\max\{a, b\}$. A possible use case for the *forward*-only statement would be $a \oplus b \triangleq a \div b$ or $a \oplus b \triangleq a - b$, which are only *left*-nondecreasing, however, the authors are not aware of the applications requiring this.

1.3. Morin–Marsten bounding

The branch-and-bound scheme for DP in TSP-PC attempts to eliminate some DP states that no optimal solution may be based upon, in view of a pre-computed *upper bound* and the *lower bounds* computed for each state considered. Like 'non-bounded' DP, it avails of precedence constraints to reduce the state space; in contrast with typical branch-and-bound methods that have *exhaustive search* through all $n!$ solutions as a worst case,[9] the DPBB's worst case is retaining all the DP states, the number of which is reduced in view of precedence constraints.

Let V^{UB} be an *upper bound* on a TSP-PC instance, $V^{\mathrm{UB}} \geq v(1..n, \mathbb{t})$, and suppose we know how to compute, for a state $(I, x) \in \mathcal{S}$, a *lower bound* of the cost of travelling through the *remaining* cities $1..n \setminus (I \cup \{x\})$, starting from x and finishing at $\mathbb{t}$; denote this value $\mathrm{LB}(I, x)$.

This information can be used to test if the state (I, x) is required to find an optimal solution: if $v(I, x) \oplus \mathrm{LB}(I, x) \geq V^{\mathrm{UB}}$, then it is not [42, Proposition 1.2]; this expression is known as a *fathoming criterion*, and such a state (I, x) is said to be *fathomed*. If all states are fathomed at some point (the next $\mathcal{S}_k$ is empty), then the procedure can be stopped, and V^{UB} declared the *optimal* value [42, Corollary 1.1].

Algorithm 2 Bounded dynamic programming for abstract aggregation TSP-PC

```
1:  Init: I_0 = {∅}, S_0 = {(∅, x), x ∈ E[∅]},
2:  ∀(∅, x) ∈ S_0 v(∅, x) = c(0, x), I_1 = E[∅].
3:  for all k ∈ 1..n − 1 do
4:      if S_k = {∅} then
5:          Stop: 'All states fathomed at layer k, thus V^UB is optimal'.
6:      else
7:          for all I ∈ I_k do
8:              Compute E[I]
9:              for all x ∈ E[I] do
10:                 Compute v(I, x)
11:                 if v(I, x) ⊕ LB(I, x) < V^UB then
12:                     Add I ∪ {x} to I_{k+1}
13:                     Add (I, x) to S_k
14: Compute v(1..n, t)                  ▷ Assuming we did not terminate at step 4
15: Recover solution by v.              ▷ Assuming we did not terminate at step 4
```

Note that denoting the DPBB states $\mathcal{S}_1, \ldots, \mathcal{S}_n$ and the corresponding order ideals $\mathcal{I}_1, \ldots, \mathcal{I}_n$ constitutes an *abuse of notation*: there is no guarantee that all the states as defined in (1) are present; in fact, the *raison d'être* of DPBB is to eliminate as many as possible and terminate the algorithm early.

The principal differences between DP and DPBB are as follows:

(i) the fathoming condition 11–13 in Algorithm 2, which prevents the extension of *fathomed* states;

(ii) the stopping condition at lines 4–5 in Algorithm 2 catching the case when *all* states are fathomed at some point, which means it is impossible to improve upon the upper bound V^{UB};
(iii) if Algorithm 2 is terminated at lines 4–5, no solution is produced, the procedure only proves that the upper bound V^{UB} *may not be improved*; this allows an unorthodox use of DPBB: it can prove that some given number, supplied in lieu of the upper bound, is a *lower bound*.

1.3.1. Worst-case complexity

Assuming *no* states are fathomed, due to poor upper or lower bounds – clearly, the worst case for DPBB – the space complexity matches that of non-bounded DP, $|\mathcal{S}| \leq w|\mathcal{I}|$ [46, Proposition 1], and time complexity is $\mathcal{O}(|\mathcal{I}| \cdot w\mathrm{C}_{\mathrm{LB}})$, where C_{LB} is the complexity of the lower bound used in the DPBB scheme and assuming it is over $\mathcal{O}(n)$; otherwise, it is $\mathcal{O}(|\mathcal{I}| \cdot nw)$ like for the non-bounded DP [46, Proposition 2].

Proposition 2: *The **worst-case** space complexity of DPBB matches that of non-bounded DP, which is at most $|\mathcal{I}| \cdot w$ (see [46, Proposition 1]), where w is the width of $<_P$.*

Proposition 3: *Assuming C_{LB} is the time complexity of the lower bound used in the DPBB framework, the latter's **worst-case** time complexity is bounded by $\mathcal{O}(|\mathcal{I}| \cdot w \cdot \max\{n; \mathrm{C}_{\mathrm{LB}}\})$, where w is the width of $P = (1..n, <_P)$.*

Proof: Recall that the non-bounded DP's time complexity is *dominated*[10] by the computation of states' values [46, Proposition 2]. Let us repeat the latter's argument and factor in C_{LB}.

There are again at most w states per each ideal I, and for each $(I, x) \in \mathcal{S}_{|I|}$ of them, the value has to be computed over at most w[11] covered states $\{(I \setminus \{m\}, m) \in \mathcal{S}_{|I|-1} \mid m \in \mathbf{I}[I]\}$; we also have to bear in mind the cost of calling $\mathbf{I}[I]$ used to *identify* the covered states ($\mathcal{O}(n)$, [46, Lemma 1]), thus the cost per ideal is at most $(w \cdot w + n) \leq (2nw) = \mathcal{O}(nw)$. In addition to the value, we also have to compute the lower bound, thus, the total cost per ideal becomes at most $(w \cdot (w + \mathrm{C}_{\mathrm{LB}}) + n) = \mathcal{O}(|\mathcal{I}| \cdot w \cdot \max\{n; \mathrm{C}_{\mathrm{LB}}\})$. ■

For DPBB, two conflicting properties are wanted of a good lower bound heuristic (LBH): it must be *fast*, computationally cheap, yet *strong* enough to eliminate at least some states at some point.

State-of-the-art lower bounds for TSP-PC [21] are based on mixed integer-linear programming (MILP); however, we decided it will be too complicated, at least, in the software integration sense, to marry a linear programming solver with a DP model, hence the attempt to first consider a fast graph-based LBH.

1.3.2. Node-greedy graph – one primitive lower bound

Formally, for a state $(I, x) \in \mathcal{S}$, an LBH must produce a value that is a lower bound on the cost of optimally traversing the cities in $F = 1..n \setminus (I \cup \{x\})$ starting from x and then finishing at the terminal $\mathbb{t}$ such that the precedence constraints P are satisfied.

An obvious lower bound on that is *minimum spanning arborescence* (directed tree) over $F \cup \{\mathbb{t}\}$ with root x, which can be obtained by variants of Edmonds–Chu–Liu

algorithm, e.g. [19]; its *bottleneck* variant is proposed in [9], and an 'abstract travel cost aggregation' version, which provides a common specification for any *commutative* and *associative* operation, is presented in [20].

Vying for speed, we only take the first step of this algorithm as LBH, with some corrections for precedence constraints; this step is called *node greedy solution* [36, §. 5.2.2] and, in contrast with the whole algorithm, it may produce cycles, but it is still a valid lower bound, although a weaker one than the complete algorithm. We denote it NG and compute it as follows: for $(I, x) \in \mathcal{S} \setminus \{(1..n, \mathbb{t})\}$, set $F = 1..n \setminus (I \cup \{x\})$. Then,

$$\mathrm{NG}(x, F) = \bigoplus_{b \in F \setminus \mathrm{Min}[F]} \min\{\mathfrak{c}(a, b) : a \in F\} \oplus \bigoplus_{b \in \mathrm{Min}[F]} \min\{\mathfrak{c}(a, b) : a \in \{x\} \cup F\} \tag{3}$$

$$\oplus \min\{\mathfrak{c}(a, \mathbb{t}) : a \in \mathrm{Max}[F]\}, \tag{4}$$

where $\mathrm{Min}[K]$ and $\mathrm{Max}[K]$ denote, respectively, the minimal and maximal elements of K with respect to $<_P$, see e.g. [51, Definitions 2.1.5, 2.3.2]:

$$\mathrm{Min}[K] \triangleq \{m \in K \mid \forall k \in K \setminus \{m\}\ \neg(k <_p m)\},$$

$$\mathrm{Max}[K] \triangleq \{m \in K \mid \forall k \in K \setminus \{m\}\ \neg(k >_p m)\}.$$

Special accommodations for precedence constraints include: (i) never travel against precedence constraints, i.e. eliminate the arcs *dual* to $<_P$, $(a, b) : (a >_P b)$; (ii) eliminate 'transitive shortcuts', arcs not in $<_P$'s *transitively irreducible kernel* [50, Definition 3.2.4]; and (iii) travel from *root* x only to *minimal* elements of K, travel to *terminal* $\mathbb{t}$ only from *maximal* elements of K. Items (i) and (ii) are a part of standard preprocessing for TSP-PC, see e.g. [2, Proposition 2.2],[21, § 2.2]; item (iii) is our addition.

Proposition 4: *The time complexity of the node greedy heuristic is at most* $\mathcal{O}(n^2)$.

Proof: For a state (I, x), $|I| = k$, every arc must end in one of $n-k$ cities, and may start at one of the other $n-k$, hence, one must exhaustively check at most $(n-k)^2$ arcs, and then aggregate the costs of the $(n-k)$ minimal ones. In addition, by [46, Lemma 1], it takes $\mathcal{O}(n-k-1)$ to compute $\mathrm{Min}[F]$ in (4), which does not change the order of time complexity. ■

Scholium: A similar lower bound, ignoring the use of $\mathrm{Min}[F]$, can be computed in *constant* time $\mathcal{O}(1)$ [35, Section 2.4] if the aggregation is *invertible*, like $+$ in conventional TSP-PC. However, since the technique involves *subtracting* the cost of the minimum arc, it cannot be directly applied to BTSP-PC since the $a \oplus b = \max\{a, b\}$ operation is *not invertible*.

2. Shared-memory parallelism for dynamic programming in TSP-PC

Various flavours of OpenMP-based shared-memory parallelism have been used to improve the performance of DP for TSP-PC and its generalizations since at least 2011, see e.g. [22], however, few proper scaling tests have been carried out; other parallelization attempts and approaches to DP for TSP-PC include the *parallel rollout* algorithm [23] and A. G. Chentsov's *independent computations* scheme [11,13,15], a *distributed* approach.

Our attempt at improving performance through OpenMP continues the `tasks`-based approach first described in [48] for DP in Bottleneck Generalized TSP-PC, with improvements as described below in Sections 2.1.1 and 2.1.2. Its pseudocode description as adapted for TSP-PC is presented as Algorithm 3. In this listing, we omit the initialization (cf. steps 1–2 in Algorithm 1) and solution recovery (cf. steps 10-11 in Algorithm 1).

We describe this parallelization scheme for the DP case to show that it is generally applicable to all DP-derived methods and also to focus more on the parallelization details. The amendments necessary for DPBB are listed at the very end of this section.

The basis for OpenMP-style parallelism is stereotypically a *loop* the iterations of which can be computed concurrently. In our case, this loop is extending each new layer l's order ideals $\mathcal{I}_l$, thereby (a) generating next-tier, cardinality $l+1$, order ideals; (b) generating the states $\mathcal{S}_l$ connected with $\mathcal{I}_l$ and (c) computing the values of these states, that is, the loop starting at step 4 in Algorithm 1 (resp., step 7 in the DPBB Algorithm 2).

However, our principal data structure ('`store`' in Algorithm 3), where we keep the order ideals, states, and their values, ordered cardinality-wise, is a *nested hash table*, which cannot be iterated through by the common OpenMP `parallel for` because it does not guarantee *constant-time* $\mathcal{O}(1)$ access to its elements. To get around this, we used the `task` construct, introduced in OpenMP 3.0. The `store` data structure is described in Section 2.1.1.

Algorithm 3 OpenMP `task`-based parallelization

```
1:  for all l ∈ 1..n − 1 do
2:      #pragma omp parallel default (shared)
3:      #pragma omp single nowait
4:      for all I ∈ I_l do
5:          #pragma omp task untied firstprivate(I)
6:          Compute E[I]
7:          for all x ∈ E[I] do
8:              Put I ∪ {x} into I_{l+1}        ▷ stored, until layer l is complete,
9:                                              ▷ in a temporary thread-safe container
10:             Compute v(I, x)
11:             store[l][I][x] := v(I, x)       ▷ no races ⇒ can write immediately
12:     Move I_{l+1} into permanent storage     ▷ into store[l+1]
```

In plain language, here is what Algorithm 3 does:

At step 1, the processing of a new state layer $\mathcal{S}_l$ starts.

Step 2 executes the `omp parallel default (shared)` directive, creating a team of *parallel execution threads* that have *shared* access to all variables and data structures (the `default (shared)` clause) available at the time of creation, including (i) `store[`$l-1$`]`, which contains the preceding states $\mathcal{S}_{l-1}$ necessary to compute the values of the current states $\mathcal{S}_l$ and (ii) `store[`l`]`, which contains the current, cardinality l, order ideals $\mathcal{I}_l$. This 'parallel zone' lasts throughout step 11.

Step 3 commands one and only one single thread (`omp single nowait`) to start iterating through $\mathcal{I}_l$ (step 4). This thread is going to package each iteration of this loop, i.e.

the processing at lines 6–11 for each $I \in \mathcal{I}_l$, as a *task* to be independently executed by the other threads in the team. The `nowait` clause lets the other threads start on the *tasks* as soon as these are generated, without *waiting* for the 'generator' thread to finish creating them.

Step 5: The `omp task untied firstprivate(`I`)` directive commands the designated thread to generate an OpenMP `task` containing steps 6–11, one for each $I \in \mathcal{I}_l$. The `untied` clause allows the *other* threads to execute these tasks. The `firstprivate(`I`)` clause directs the generator thread to make a separate copy of I for each task. In absence of this clause, it would have been *shared* by all the threads instead, which defeats the idea of this parallel scheme where each thread processes the steps 6–11 *independently*, given its own $I \in \mathcal{I}_l$.

Steps 6–11: The *tasks* executed in parallel. Each thread gets its own $I \in \mathcal{I}_l$, for which it computes $\mathbf{E}[I]$ (step 6) and, for each city x that *feasibly extends* I (step 7), generates the next-tier, cardinality $l+1$ order ideal $I \cup \{x\}$, which is stored in a temporary container with *thread-safe insertion*[12] (step 8) and computes the *value* of the associated state $(I, x) \in \mathcal{S}_l$. In contrast with the generation of next-tier order ideals $\mathcal{I}_{l+1}$, no race conditions arise when writing the state's values with `store[`l`][`I`][`x`]` $:= v(I, x)$ because (a) each time, each thread processes a *unique* $I \in \mathcal{I}_l$, (b) they are already stored in `store[`l`][`I`]` and (c) for different $I, J \in \mathcal{I}_l$, the *interior* hash tables that associate $x \in \mathbf{E}[I]$ with I and $y \in \mathbf{E}[J]$ with J are processed independently.

Step 12: The `omp parallel` region is over, and the thread team is 'disbanded', until the next layer, $l+1$. The next-tier order ideals $\mathcal{I}_{l+1}$ are moved from the temporary thread-safe container into the permanent storage `store[`$l+1$`]`

DPBB Modifications to Algorithm 3. Only two amendments are necessary: (1) add the check at lines 4–5 from Algorithm 2 before starting the parallel execution (`omp parallel`) and (2) replace lines 7–11 with lines 9–13 from Algorithm 2, retaining the use of the temporary thread-safe container for *order ideals*.

2.1. Implementation improvements

There were two principal improvements to the software implementation as compared with [46,48].

2.1.1. New hash map saves 50% memory

The principal data structure we use to store state layers $\mathcal{S}_l$, $l \in 1..n$, as originally described in [48], is a *nested hash table*, where the *outermost* table maps, to an order ideal $I \in \mathcal{I}_l$, the *innermost* table, which holds the pairs of the form $(x \in \mathbf{E}[I], v(I, x))$, the values for all states corresponding to I; thus, to access the value $v(I, x)$, one would call the *subscript operator* `[]` three times: `store[`l`][`I`][`x`]`.

In [48], this data structure was implemented in C++ as

`std::vector<std::unordered_map<uint32_t, std::unordered_map<uint16_t, float> > >` where `unordered_map` was the C++ Standard Template Library's *hash table* implementation, and `uintXX_t` was the XX-bit long *unsigned integer*; these integer types encoded, respectively, the order ideal, as a bit mask, and the 'interface' city. The *floating-point* number stored the state's value.

Later, in [46], this data structure was amended as follows: `std::vector<std::unordered_map<std::bitset, std::map<uint16_t, uint32_t>>>`, notably, (a) order ideals were encoded with `std::bitset` instead of 32-bit unsigned integers `uint32_t`, to accommodate the TSPLIB instances, which have as much as 378 cities (`rbg378a.sop`); (b) per the TSPLIB custom, the travel costs were *integer*, not floating-point; (c) associative array `std::map` was used instead of the hash table for the *innermost* container, in an attempt to provide more consistent memory footprint.

In this paper, we replaced the STL hash table implementations with `flat_hash_map` [32] from Abseil C++ Library.[13] The principal data structure of the implementation we use in this paper is as follows:

```
std::vector< <flat_hash_map<std::bitset, flat_hash_map
<uint16_t,t_cost> > >
```

Compared with [46], the new data structure uses *half* as much memory: in particular, `ry48p.3.sop` took circa 120 GB to solve by DP in [46], whereas our present implementation solves it in under 62 GB. Computation speed does not seem to have been affected by this new data structure.

2.1.2. Container with concurrent insertion simplifies the code

The OpenMP-based scheme as published in [48] required an `omp critical` directive to prevent race conditions when writing newly computed order *ideals* (cf. line 8 in Algorithm 3) into the shared data structure. In our updated implementation, instead of this directive, we use a container supporting concurrent insertion, `parallel_hash_map` by Gregory Popovich[14]; after each layer l is computed, the contents of this *thread-safe* container (the order ideals of the next cardinality $l+1$) are moved to the principal data structure, see line 12 in Algorithm 3.

This effectively shifts the responsibility for ensuring there are no *race conditions* from the 'user' designing the parallel scheme to the library's author, which, at the very least, improves the readability of the actual code.

3. Experiment

We used the following two experimental setups:

First setup, AWS, ran an Amazon AWS `x1e.32xlarge`[15] machine, powered by four Intel Xeon E7 8880 v3 processors, presented as up to 128 virtual CPUs, with 3904 GB RAM. These machines ran Deep Learning AMI 23.1 images[16] based on Ubuntu 16.04, which functioned as a *host* operating system for Docker 19.03 containers that would house our solver. The Docker containers were based on Ubuntu 18.04, which was supplemented with Python 3.6 to run experiment scripts; finally, our DP solver's C++ code was compiled with the aid of gcc 7.4.0, which was the default for Ubuntu 18.04. For more information on Docker, refer to the official documentation at https://docs.docker.com/.

This somewhat complicated setup was necessary to make use of Amazon AWS machines and harness these nearly 4 TB RAM that they offer.

Second setup, Uran-apollo, works off the same machines as that of [46], the `apollo` nodes of the Uran supercomputer at Krasovskii IMM UB RAS, powered by dual Intel Xeon E5−2697 v4 CPUs with 256 GB RAM under CentOS Linux 7 x64, and the software kit that matches the AWS setup.

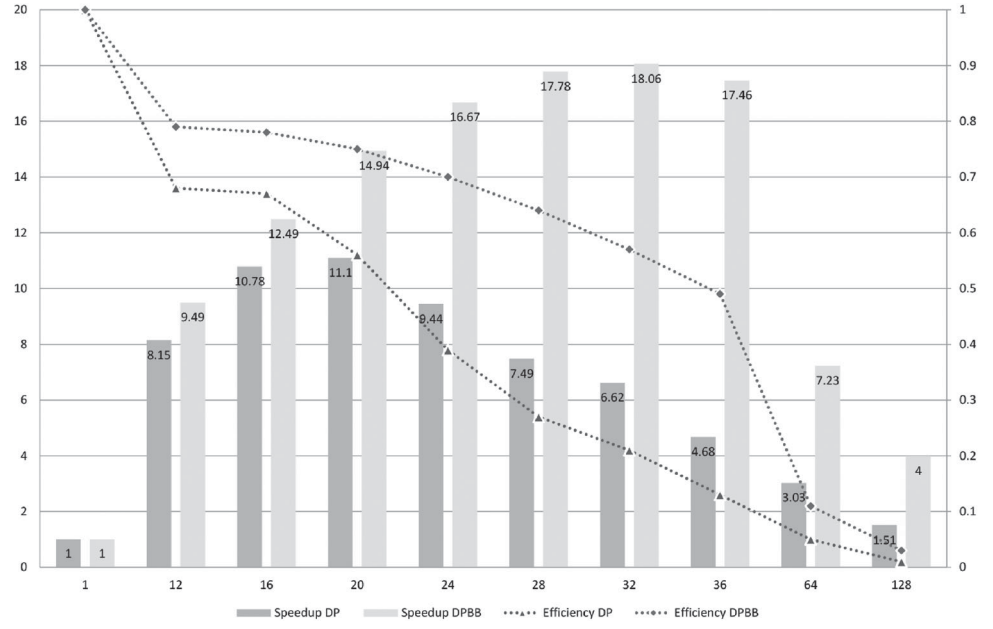

Figure 1. Speedup and efficiency of forward DP and DPBB for `p43.3.sop`, 1–128 threads (AWS).

In all experiments, we recorded the run time, measured through C++ `<chrono>` library's `steady_clock::now()` method with 1-*millisecond* precision[17] and memory usage, measured as reported by the host Linux operating system in the `VmRSS` line in `/proc/self/status`.

One of the objectives of this paper is to see what rudimentary DPBB can offer provided with more memory than it can use in reasonable time (the hard limit time was 48 hours), and how does its performance on *larger* TSP-PC instances compare with non-bounded DP. To do this efficiently, we first tested how well does Algorithm 3 scale as the number of cores increase, for DP and DPBB cases, on several larger instances.

3.1. Scaling tests

In describing scaling and parallel speedup, we follow [4, Section 3.2]: the *speedup* S_p, achieved by a parallel algorithm running on p cores, is defined as $S_p \triangleq t_1/t_p$, where t_p is the time to solve a problem on p cores. Linear speedup, $S_p = p$, is viewed as the ideal. *Efficiency*, or the fraction of linear speedup attained, is obtained by normalizing the speedup, $E_p \triangleq S_p/p$, and the linear speedup is attained at $E_p = 1$.

We conducted three scaling tests. The first one was on AWS, and its results are presented in Figure 1. Its objective was to determine the number of threads (one thread per core) to request for 'record-breaking' attempts. Recall that AWS provides up to 128 virtual CPUs; we tested `p43.3.sop`, representative of the larger instances still tractable by DP, with 1, 12, 16, 24, 28, 32, 36, 64, and 128 OpenMP threads. We only ran scaling tests on AWS for *forward* DP and DPBB in conventional min-sum TSP-PC; the results may safely be extended to other statements since their theoretical time complexity is the same.

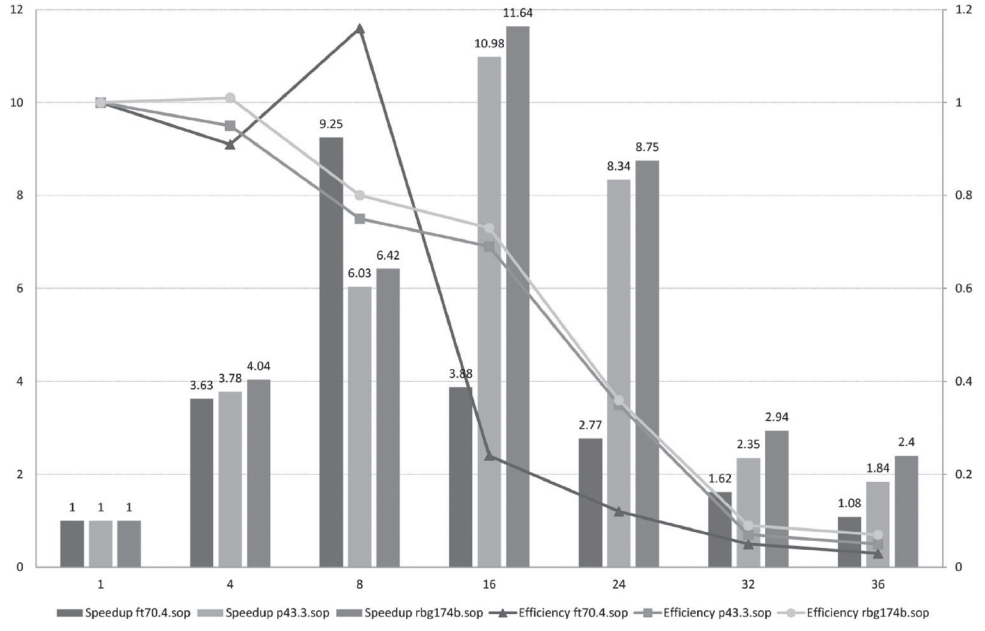

Figure 2. Speedup and efficiency of forward DP, 1–36 threads (Uran).

The second one was conducted on the Uran-apollo setup, on the larger instances, `ft70.4.sop`, `rbg174b.sop` and `p43.3.sop`, which we selected based on their memory usage; for DP, it was circa 0.5 GB, 1.5 GB, and 22 GB, respectively, and somewhat less than that for DPBB (respectively, 4%, 8% and 26% less).

See Figures 2 and 3 for the comparison of *forward* DP and DPBB scaling and efficiency on `p43.3.sop`. Although we tested scaling on the mentioned instances for both forward and backward DP and DPBB in standard and bottleneck statements, we found no qualitative differences in scaling, thus, to save space, we only put forth the figures for forward DP and DPBB for TSP-PC. However, the advertised 35-fold speedup was achieved for *backward* DPBB on `rbg174b.sop` in TSP-PC statement; the forward case exhibited 31.21-fold speedup, see Figure 3.

The last scaling test was conducted after the principal experiments (see Sections 3.2 and 3.3) to determine the *smallest run time* of forward DP and DPBB achievable on 1–36 cores available on Uran-apollo nodes. It was only conducted for conventional TSP-PC statement.

Table 1 lists the best run times in seconds (T:DPBB and T:DP), maximum speedups (S:DPBB and S:DP) and how many processor cores were used to achieve it. Run times and scaling are grey for the instances that run above the memory limit. The instances that took less than 20 seconds to solve by serial DPBB are omitted from this table since they are easy enough for serial non-bounded DP.

3.2. Gleefully wasting 4 terabyte worth of RAM: The AWS DPBB exploratory study

We used the copious amounts of RAM at AWS to conduct an *exploration study* in what a DPBB with a rudimentary lower bound could yield; naturally, this study would

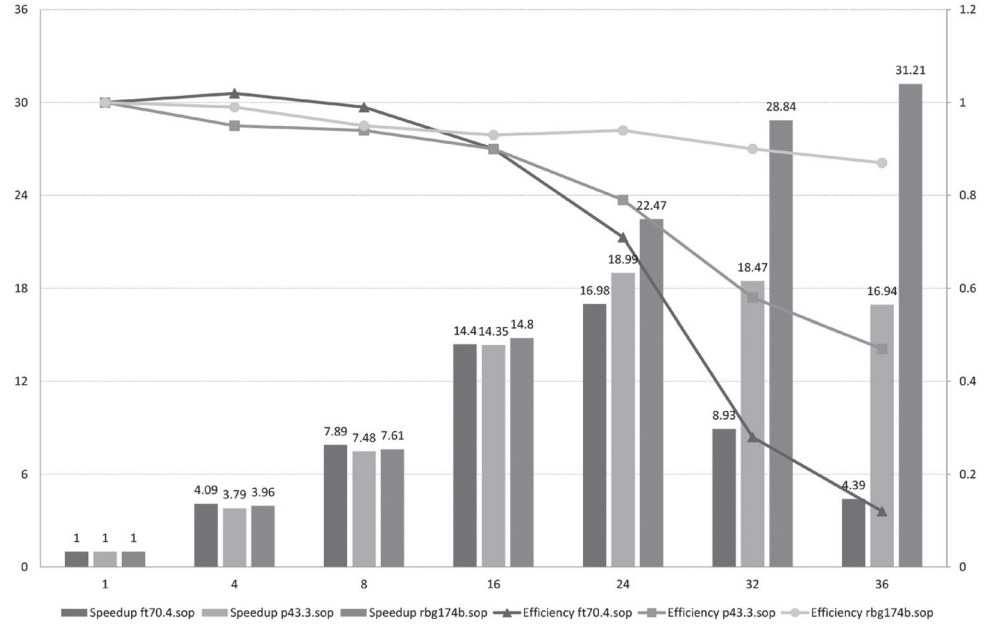

Figure 3. Speedup and efficiency of forward DPBB, 1–36 threads (Uran).

Table 1. Best run times, speedups and how many cores were used to achieve it (the number after '–' in the S columns) for *forward* DP and DPBB on larger TSPLIB instances, Uran-apollo. The grey run times and speedups denote the runs that terminated due to *memory limit*.

Instance	T:DPBB	S:DPBB	T:DP	S:DP
ft70.4.sop	8.59	15.69–24	5.07	5.34–8
rbg174b.sop	25.97	17.8–36	13.24	9.56–16
rbg253a.sop	92.64	22.19–36	20.57	9.52–24
p43.3.sop	182.72	20.27–32	181.59	11.88–20
ry48p.3.sop	508.57	22.26–32	508.24	12.72–20
rbg048a.sop	1769.55	24.06–36	853.52	12.82–28
ESC47.sop	2718.23	18.03–36	263.23	10.93–36
prob42.sop	2871.38	18.4–36	1017.11	12.57–28
kro124p.4.sop	4044.96	26.66–36	1962.31	13.75–20

not have been possible without some prior parallelization efforts, as described in Section 1.

Having only limited time on AWS, we sorted the TSPLIB instances in the order of increasing precedence constraints' *width*[18] (reflected in *#w* column) and ran, for each one, both backward and forward procedures. The lower bound heuristic was the node greedy, see Section 1.3.2. For TSP-PC, we used the well-known best upper bounds from [21]. For BTSP-PC, the *upper bounds* were obtained through Restricted DP [37] as adapted to TSP-PC with *abstract aggregation* in [46, Section 3.3], supplemented with precedence-aware preprocessing [2, Proposition 2.2]; for completeness, we list those in the Supplement to this paper, see Tables A.1 and A.2.

The hard time limit was 48 hours for each instance, however, many computations were stopped prior to that, either due to hitting the memory limit or not getting halfway through

Table 2. Backward and forward DPBB on TSPLIB SOP instances in conventional min-sum statement, on a 4-terabyte machine (AWS).

Instance	#*w*	Value	MEM:F	TIME:F	MEM:B	TIME:B	T:ECN	M:ECN
br17.10.sop	5	55	4.750 MB	0.128	4.672 MB	0.071	(B) 45%	(B) 2%
br17.12.sop	3	55	4.520 MB	0.112	4.426 MB	0.184	(F) 39%	(B) 2%
ESC07.sop	1	2125	4.258 MB	0.128	8.195 MB	0.129	(F) 1%	(F) 48%
ESC11.sop	2	2075	4.258 MB	0.059	4.152 MB	0.089	(F) 34%	(B) 2%
ESC12.sop	4	1675	4.180 MB	0.06	4.160 MB	0.096	(F) 38%	(B) 0%
ESC25.sop	12	1681	9.273 MB	0.4	329.883 MB	6.533	(F) 94%	(F) 97%
ft53.3.sop	18	10262	437.890 GB	8601.451	629.222 GB	11395.83	(F) 25%	(F) 30%
ft53.4.sop	9	14425	45.430 MB	0.88	44.746 MB	0.604	(B) 31%	(B) 2%
ft70.4.sop	11	53530	554.023 MB	7.407	559.258 MB	10.68	(F) 31%	(F) 1%
kro124p.4.sop	15	**76103**	**186.214 GB**	**6019.743**	**337.642 GB**	**13850.77**	(F) 57%	(F) 45%
p43.2.sop	19	28480	1.701 TB	30634.787	1.590 TB	29865.24	(B) 3%	(B) 7%
p43.3.sop	14	28835	15.859 GB	229.203	17.659 GB	258.85	(F) 11%	(F) 10%
p43.4.sop	8	83005	11.602 MB	0.309	9.180 MB	0.268	(B) 13%	(B) 21%
prob42.sop	25	243	176.590 GB	4007.834	141.517 GB	3162.722	(B) 21%	(B) 20%
rbg048a.sop	22	351	2.653 TB	64768.886	0.22 TB	4592.45	(B) 93%	(B) 92%
rbg050c.sop	21	467	1.116 TB	24323.695	0.911 TB	17063.86	(B) 30%	(B) 18%
rbg109a.sop	7	1038	7.836 MB	0.162	7.172 MB	0.177	(F) 8%	(B) 8%
rbg150a.sop	10	1750	11.270 MB	0.375	11.812 MB	0.405	(F) 7%	(F) 5%
rbg174b.sop	16	2033	1.334 GB	36.679	1.454 GB	127.498	(F) 71%	(F) 8%
rbg253a.sop	17	2950	1.505 GB	129.904	1.518 GB	168.571	(F) 23%	(F) 1%
ry48p.3.sop	13	19894	39.730 GB	673.094	37.756 GB	618.671	(B) 8%	(B) 5%
ry48p.4.sop	6	31446	18.344 MB	0.514	23.688 MB	0.414	(B) 19%	(F) 23%

Table 3. Backward and forward DPBB on TSPLIB SOP instances in *bottleneck* (minimax) statement, on a 4-Terabyte machine (AWS).

Instance	#*w*	Value	MEM:F	TIME:F	MEM:B	TIME:B	T:ECN	M:ECN
br17.10.sop	5	8	4.684 MB	0.081	4.684 MB	0.093	(F) 13%	(B) 0%
br17.12.sop	3	8	4.426 MB	0.086	8.418 MB	0.106	(F) 19%	(F) 47%
ESC07.sop	1	1000	4.316 MB	0.046	4.258 MB	0.122	(F) 62%	(B) 1%
ESC11.sop	2	419	4.238 MB	0.13	4.160 MB	0.09	(B) 31%	(B) 2%
ESC12.sop	4	222	4.152 MB	0.05	4.035 MB	0.118	(F) 58%	(B) 3%
ESC25.sop	12	181	7.367 MB	0.18	24.559 MB	1.078	(F) 83%	(F) 70%
ft53.3.sop	18	977	297.146 GB	5070.184	151.774 GB	3004.532	(B) 41%	(B) 49%
ft53.4.sop	9	978	40.254 MB	0.877	9.410 MB	0.185	(B) 79%	(B) 77%
ft70.4.sop	11	1532	288.96 MB	6.755	399.523 MB	7.365	(F) 8%	(F) 28%
kro124p.4.sop	15	**1508**	166.130 GB	4529.431	**243.210 GB**	**10615.53**	(F) 57%	(F) 32%
p43.2.sop	19	25070	–	–	2.308 TB	40482.08	N/A	N/A
p43.3.sop	14	25040	10.726 GB	144.89	10.138 GB	141.108	(B) 3%	(B) 5%
p43.4.sop	8	25070	11.566 MB	0.275	9.871 MB	0.451	(F) 39%	(B) 15%
rbg048a.sop	22	24	3.192 TB	68615.216	1.517 TB	26739.43	(B) 61%	(B) 52%
rbg050c.sop	21	23	1.106 TB	23829.154	0.329 TB	5853.241	(B) 75%	(B) 70%
rbg109a.sop	7	27	7.723 MB	0.294	7.496 MB	0.285	(B) 3%	(B) 3%
rbg150a.sop	10	28	11.293 MB	0.332	11.594 MB	0.5	(F) 34%	(F) 3%
rbg174a.sop	16	28	1.456 GB	39.494	1.454 GB	126.261	(F) 69%	(B) 0%
rbg253a.sop	17	28	1.507 GB	131.466	1.517 GB	168.248	(F) 22%	(F) 1%
ry48p.2.sop	20	**588**	62.069 GB	1522.184	**255.604 GB**	**5498.469**	(F) 72%	(F) 76%
ry48p.3.sop	13	**658**	61.961 MB	1.066	**88.895 MB**	**1.454**	(F) 27%	(F) 30%
ry48p.4.sop	6	1235	7.734 MB	0.295	15.355 MB	0.263	(B) 11%	(F) 50%

an instance[19] in the first several hours of computation. Some of the harder (i.e. of greater width) instances had to be left out of consideration due to limits on computation time at AWS.

The results for TSP-PC and BTSP-PC are presented in Tables 2 and 3. The #*w* column shows the instance's number in the width-sorted list (greater is harder).

These tables also show whether the *forward* or *backward* statement was the best in terms of run time and memory, which is marked with *light grey* background of the corresponding table cell and either (B) or (F) in the T:ECN and M:ECN columns, which display the advantage of the *better* DPBB statement for a given instance, e.g.

$$\mathrm{M:ECN} \triangleq \frac{\max\{\mathrm{MEM_B};\mathrm{MEM_F}\} - \min\{\mathrm{MEM_B};\mathrm{MEM_F}\}}{\max\{\mathrm{MEM_B};\mathrm{MEM_F}\}},$$

where 'B' stands for backward DPBB and 'F' stands for forward DPBB.

Although the *faster* method is not always the *leaner*, the difference in memory usage in these 'mismatch' cases is minuscule, several MBs at most, which we attribute to the works and vagaries of specific data structures' overhead, garbage collection, timing of operating system calls, and so forth. Thus, in case of a mismatch, we deem it is safe to assume that the *best* method is the *fastest*.

TSP-PC results. We solved all instances #*w* 1–19, and also #*w* 21, 22 and 25, in total, 22 of 41 TSPLIB instances, which is up from 16 of 41 solved by non-bounded DP in [46]. In particular, we closed `kro124p.4.sop` (# 15), which took nearly 2 hours (on 28 cores) and 186 GB RAM. As expected, the best known [21] upper bound 76,103 proved optimal.

Forward DPBB was better in 13 cases and backward in the remaining 9. The run time relation was between 1.01 and 16.33, and the memory relation was between 1 and 35.57. Spectacular absolute difference was observed on `rbg048a.sop`, #*w* 22, which was solved in circa 4600 seconds and 226 GB by the backward statement but took about 65,000 seconds and 2.6 TB in the forward case.

BTSP-PC results. We solved all instances numbered #*w* 1–22, in total, 22 of 41 TSPLIB instances. In particular, we closed `kro124p.4.sop` (#*w* 15), which took nearly 2 hours and 166 GB RAM and `ry48p.2.sop` (#*w* 20), which took less than 30 minutes and about 62 GB. In the TSP-PC statement, `ry48p.2.sop` hit the 4 TB memory limit at layer 16 in 17 hours (forward DPBB) and at layer 18 in 86 hours (backward DPBB).

Forward DPBB was better in 13 cases, and backward in the remaining 9. The run time relation ran between 1.03 and 5.99, and the memory relation was between 1 and 4.28. Spectacular absolute difference was again observed on `rbg048a.sop`, which was solved in circa 27,000 seconds and 1.52 TB by the backward statement but took about 69,000 seconds and 3.2 TB in the forward case.

To the best of our knowledge, bottleneck objective functions have never been considered for TSPLIB SOP instances, so all these 22 optimal solutions are 'first ever'. The non-SOP (without precedence constraints) bottleneck statement of asymmetric TSPLIB instances was considered in e.g. [33].

3.3. Comparing run time and memory usage for forward and backward DP and DPBB: the Uran-apollo experiment

Table 4 (5) contrasts backward and forward DP running at 16 threads with DPBB running at 36 on the Uran-apollo setup for the TSP-PC (BTSP-PC) statements of the instances that were known to be computable within 256 GB from the exploratory study (Section 3.2).

The numbers of parallel threads for DP and DPBB were chosen based on the scaling of `rbg174b.sop` and do not necessarily reflect the *peak performance* of either method. The prime motivation was to significantly improve the run times compared with the older

Table 4. Time and memory usage of forward and backward DP at 16 cores and DPBB at 36 cores for TSP-PC, Uran-apollo. Asterisk * denotes termination due to memory limit.

		Memory usage				Time in seconds			
Instance	Value	DP_b^{16}	DP_f^{16}	$DPBB_b^{36}$	$DPBB_f^{36}$	DP_b^{16}	DP_f^{16}	$DPBB_b^{36}$	$DPBB_f^{36}$
br17.10.sop	55	2.922 MB	3.012 MB	**2.918 MB**	2.922 MB	**0.1**	**0.1**	0.2	0.19
br17.12.sop	55	**2.398 MB**	2.496 MB	2.660 MB	2.648 MB	**0.07**	**0.07**	0.14	0.13
ESC07.sop	2125	**2.152 MB**	2.160 MB	2.434 MB	2.438 MB	**0.02**	**0.02**	0.04	0.04
ESC11.sop	2075	2.242 MB	2.414 MB	**2.398 MB**	4.391 MB	**0.04**	0.05	0.08	0.06
ESC12.sop	1675	**2.395 MB**	2.402 MB	2.398 MB	2.406 MB	0.05	**0.04**	0.1	0.08
ESC25.sop	1681	1.114 GB	1.113 GB	0.318 GB	**0.007 GB**	12.44	**11.51**	20.76	0.69
ft53.4.sop	14425	43.465 MB	43.453 MB	**41.020 MB**	43.215 MB	**1.13**	**1.13**	3.81	3.44
ft70.4.sop	53530	564.117 MB	563.621 MB	555.863 MB	**542.840 MB**	**9**	9.43	33.23	32.24
kro124p.4.sop	76103	*ML–L30	*ML–L71	*ML–L29	**186.208 GB**	*2286.62	*2079.88	*6512.89	**4014.13**
p43.3.sop	28835	21.524 GB	21.524 GB	17.657 GB	**15.865 GB**	186.43	**183.48**	268.08	222.99
p43.4.sop	83005	11.359 MB	11.805 MB	**6.605 MB**	8.254 MB	**0.37**	**0.37**	0.65	0.79
prob.42.sop	243	*ML–L10	*ML–L11	**141.562 GB**	176.585 GB	*840.71	*1546.43	**2303.04**	2891.54
rbg048a.sop	351	*ML–L23	*ML–L12	**226.265 GB**	*ML–L12	*789.57	*1256.6	**3893.31**	*1784.71
rbg109a.sop	1038	**4.723 MB**	4.859 MB	5.070 MB	4.984 MB	**0.31**	**0.31**	0.49	0.45
rbg150a.sop	1750	**7.031 MB**	**7.031 MB**	9.973 MB	7.859 MB	0.46	**0.43**	0.7	0.96
rbg174b.sop	2033	1.447 GB	1.451 GB	1.453 GB	**1.335 GB**	**12.61**	13.86	88.38	25.97
rbg253a.sop	2950	1.511 GB	**1.501 GB**	1.516 GB	1.507 GB	**14**	14.77	118.88	92.45
ry48p.3.sop	19894	61.125 GB	61.122 GB	37.739 GB	**39.730 GB**	**452.44**	453.26	537.49	533.32
ry48p.4.sop	31446	18.738 MB	18.441 MB	17.973 MB	**17.516 MB**	0.58	**0.55**	1.76	1.76

Table 5. Time and memory usage of forward and backward DP at 16 cores and DPBB at 36 cores for Bottleneck TSP-PC, Uran-apollo. Asterisk * denotes termination due to memory limit.

		Memory usage				Time in seconds			
Instance	Value	DP_b^{16}	DP_f^{16}	$DPBB_b^{36}$	$DPBB_f^{36}$	DP_b^{16}	DP_f^{16}	$DPBB_b^{36}$	$DPBB_f^{36}$
br17.10.sop	8	2.926 MB	3.008 MB	**2.668 MB**	2.926 MB	**0.1**	**0.1**	0.17	0.19
br17.12.sop	8	**2.395 MB**	2.500 MB	2.676 MB	2.664 MB	**0.07**	**0.07**	0.14	0.14
ESC07.sop	1000	**2.156 MB**	**2.152 MB**	2.438 MB	2.438 MB	**0.02**	**0.02**	0.03	0.03
ESC11.sop	419	2.406 MB	**2.391 MB**	2.406 MB	2.402 MB	**0.04**	**0.04**	0.05	0.05
ESC12.sop	222	2.395 MB	**2.254 MB**	2.395 MB	2.406 MB	**0.04**	0.05	0.07	0.07
ESC25.sop	181	1.115 GB	1.113 GB	0.021 GB	**0.005 GB**	12.42	10.26	1.57	**0.51**
ft53.3.sop	977	*ML–L19	*ML–L25	**151.715 GB**	*ML–L32	*1589.56	*1858.9	**1893.8**	2519.58
ft53.4.sop	978	43.680 MB	43.938 MB	**7.105 MB**	38.242 MB	1.13	1.1	**0.96**	3.24
ft70.4.sop	1532	564.969 MB	563.375 MB	398.031 MB	**288.590 MB**	10.15	**9.51**	26.45	19.14
kro124p.4.sop	1508	*ML–L30	*ML–L71	243.152 GB	**166.108 GB**	*2367.29	*2085.6	6838.78	**2997.4**
p43.3.sop	25,040	21.525 GB	21.528 GB	**10.131 GB**	10.718 GB	188.26	178.89	**156.53**	168.44
p43.4.sop	25,070	11.629 MB	11.211 MB	**7.535 MB**	7.879 MB	**0.36**	**0.36**	0.74	0.73
rbg109a.sop	27	**4.754 MB**	4.852 MB	5.117 MB	4.969 MB	0.32	**0.28**	0.47	0.46
rbg150a.sop	28	**7.062 MB**	7.211 MB	9.254 MB	7.551 MB	**0.44**	0.47	0.64	0.74
rbg174b.sop	28	**1.447 GB**	1.450 GB	1.453 GB	1.452 GB	**11.95**	21.89	88.64	27.1
rbg253a.sop	28	1.511 GB	**1.500 GB**	1.516 GB	1.507 GB	**13.88**	14.49	118.84	92.97
ry48p.2.sop	588	*ML–L11	*ML–L11	*ML–L31	**62.062 GB**	*1263.58	*1372.63	*4306.78	**1282.26**
ry48p.3.sop	658	61.126 GB	61.125 GB	0.075 GB	**0.055 GB**	455.55	459.22	5.14	**3.27**
ry48p.4.sop	1235	18.512 MB	18.543 MB	10.887 MB	**5.648 MB**	0.64	0.61	1.32	**0.72**

serial implementation [46, Table 1], which has been achieved, cf. `ry48p.3.sop`, which took 7805 seconds in [46], but only requires 453 seconds (62 GB, 16 cores, DP) or 537 seconds (40 GB, 36 cores, DPBB). Let us again note that scaling is dependent on instance (see Section 3.1), and the bigger ones do not always scale much better after reaching a circa 20-fold advantage over 1-core run.

The objective of this experiment was to empirically compare the memory usage of DP and DPBB 'parameterized' with the best known upper bound and the *node greedy* lower

Table 6. Dynamic programming with and without bounding on TSPLIB SOP, compared with best known *exact* solutions [21, Table 6]; DP_f^1 are from [46].

						Time in seconds					
Instance	Value	n	w	#w	ρ	BK	DP_b^{16}	DP_f^{16}	$DPBB_b^{36}$	$DPBB_f^{36}$	DP_f^1
br17.10.sop	55	16	10	5	0.12	**0**	0.1	0.1	0.2	0.19	**0.08**
br17.12.sop	55	16	9	3	0.18	**0**	0.07	0.07	0.14	0.13	**0.05**
ESC07.sop	2125	7	5	1	0.33	**0**	0.02	0.02	0.04	0.04	**0.01**
ESC11.sop	2075	11	9	2	0.09	**0**	0.04	0.05	0.08	0.06	**0.02**
ESC12.sop	1675	12	10	4	0.17	**0**	0.05	0.04	0.1	0.08	**0.03**
ESC25.sop	1681	25	19	12	0.04	**0**	12.44	11.51	20.76	0.69	91
ESC47.sop	1288	47	41	30	0.03	**2**	–	–	–	–	–
ESC63.sop	62	63	53	35	0.12	**0**	–	–	–	–	–
ESC78.sop	18230	78	32	23	0.09	**1**	–	–	–	–	–
ft53.1.sop	7531	52	42	31	0.01	**91**	–	–	–	–	–
ft53.2.sop	8026	52	34	26	0.02	**29,842**	–	–	–	–	–
ft53.3.sop	10262	52	24	18	0.16	**8629**	–	–	–	–	–
ft53.4.sop	14425	52	13	9	0.57	2	**1.13**	**1.13**	3.81	3.44	2.04
ft70.1.sop	39313	69	55	36	0.01	**17**	–	–	–	–	–
ft70.2.sop†	*40419	69	44	33	0.02	–	–	–	–	–	–
ft70.3.sop	42535	69	35	27	0.09		–	–	–	–	–
ft70.4.sop	53530	69	16	11	0.56	249	**9**	9.43	33.23	32.24	31
kro124p.1.sop†	*39420	99	78	41	0.01	–	–	–	–	–	–
kro124p.2.sop†	*41336	99	65	40	0.01	–	–	–	–	–	–
kro124p.3.sop†	*49499	99	43	32	0.05	–	–	–	–	–	–
kro124p.4.sop	**76103**	**99**	**22**	**15**	**0.48**	–	–	–	–	**4014.13**	–
p43.1.sop	28140	42	36	28	0.01	**2**	–	–	–	–	–
p43.2.sop	28480	42	26	19	0.04	**279**	–	–	–	–	–
p43.3.sop	28835	42	21	14	0.11	**177**	186.43	183.48	268.08	222.99	2549
p43.4.sop	83005	42	13	8	0.58	11	**0.37**	**0.37**	0.65	0.79	0.66
prob.42.sop	243	40	34	25	0.01	**6**	–	–	2303.04	2891.54	–
prob.100.sop†	*1163	98	57	39	0.02	–	–	–	–	–	–
rbg048a.sop	351	48	32	22	0.40	**0**	–	–	3893.31	–	–
rbg050c.sop	467	50	31	21	0.41	**1**	–	–	–	–	–
rbg109a.sop	1038	109	12	7	0.91	1	**0.31**	**0.31**	0.49	0.45	0.35
rbg150a.sop	1750	150	13	10	0.92	2	0.46	**0.43**	0.7	0.96	0.72
rbg174b.sop	2033	174	22	16	0.93	**6**	12.61	13.86	88.38	25.97	165
rbg253a.sop	2950	253	22	17	0.95	16	**14**	14.77	118.88	92.45	178
rbg323a.sop	3140	323	47	34	0.93	**159**	–	–	–	–	–
rbg341a.sop	2568	341	33	24	0.94	**70,997**	–	–	–	–	–
rbg358a.sop	2545	358	55	37	0.88	**791**	–	–	–	–	–
rbg378a.sop†	*2816	378	55	38	0.89	–	–	–	–	–	–
ry48p.1.sop	15805	47	37	29	0.01	**12,483**	–	–	–	–	–
ry48p.2.sop†	*16666	47	29	20	0.02	–	–	–	–	–	–
ry48p.3.sop	**19894**	**47**	**19**	**13**	**0.12**	7805	**452.44**	453.26	537.49	533.32	7805
ry48p.4.sop	31446	47	12	6	0.55	92	0.58	**0.55**	1.76	1.76	0.7

Note: **Boldface** run times are the best known. Columns n, w and ρ describe the 'size' of instances. Column #w is the instance's number in their list sorted by increasing w.

bound (see Section 1.3.2). Tables 4 and 5 display its results, where the better run times and memory usages are set in **boldface**.

TSP-PC results. Bounded DP at 36 threads was only quicker than DP at 16 threads when it saved more than 99% memory – in the forward procedure for `ESC25.sop`. However, non-bounded DP hit the memory limit for `kro124p.4.sop`, `prob42.sop` and `rbg048a.sop`, which were all successfully solved by DPBB within the allotted 256 GB. Let us also note that, on Uran-apollo at 36 threads, the best computation time for **kro124p.4.sop**, 4014 seconds, went down by more than 33% from 6020 seconds observed

on AWS at 28 threads, which was probably due to this instance being able to use more processor cores with efficiency.

DPBB saved an appreciable amount of memory, both in relative (18% or more) and absolute (1 or more GB) terms for 3 instances tractable by both methods,

We have been able to solve 19 of 41 TSPLIB instances, which is up from 16 of 41 solved through non-bounded DP in [46]. In particular, we closed through DPBB the long-standing `kro124p.4.sop` (#*w* 15), in a little above 1 hour and 186.2 GB RAM.

BTSP-PC results. Bounded DP at 36 threads was quicker than ordinary DP at 16 threads whenever it saved more than 50% memory, on `ESC25.sop`, `p43.3.sop`, `ry48p.3.sop` (forward and backward) and `ft53.3.sop` (forward). However, non-bounded DP hit memory limit for **kro124p.4.sop**, `prob42.sop`, **ry48p.2.sop** and `rbg048a.sop`, which were all successfully solved by DPBB within the allotted 256 GB.

DPBB saved an appreciable amount of memory, both in relative (18% or more) and absolute (1 or more GB) terms, for 4 instances tractable by both methods.

We have been able to close 19 of 41 TSPLIB instances. In particular, we solved `kro124p.4.sop` (#*w* 15), which took 50 minutes and 166.1 GB RAM and `ry48p.2.sop` (#*w* 20), which took a little over 20 minutes and 62 GB.

Forward DPBB was better than the backward DPBB in 13 cases, and backward in the remaining 9. The run time relation ran between 1.03 and 5.99, and the memory relation was between 1 and 4.28. The most spectacular absolute difference was again observed on `rbg048a.sop`, #*w* 22, which was solved in circa 27,000 seconds and 1,52 TB by the backward statement but took about 69,000 seconds and 3.2 TB in the forward case.

To the best of our knowledge, bottleneck objective functions have never been considered for TSPLIB SOP instances, so all mentioned 22 optimal solutions are vacuously 'first ever'. The non-SOP (without precedence constraints) bottleneck statement of asymmetric TSPLIB instances was considered in e.g. [33].

3.4. Rough practical guide for larger precedence-constrained instances

In this section, we endeavour to recommend the DP/DPBB computation parameters for solving larger, i.e. 'medium-constrained' instances of TSP-PC and related problems, based on the final scaling tests as presented in Table 1.

Note that for smaller 'DP-tractable' instances described in [46, Section 5], not larger than e.g. `ft70.4.sop` (e.g. instances #*w* 1–11), we recommend running non-bounded DP at 4–8 cores at most and not bothering with DPBB at all. The direction of DP is not important.

Thus we recommend running DPBB on at most 36 cores (24 cores ensure CPU time is not wasted much) in both forward and backward directions, recall (Section 3.2) that the difference may be quite significant. DP's run time continues to improve for up to 20 cores (16 to ensure little waste of CPU time), but on larger instances it will most probably hit the memory limit.

Decent upper bounds may be produced by Restricted DP [37,46]; in particular, for BTSP-PC, the upper bounds obtained by Restricted DP with the parameter $H = 10^5$ for the 22 of 41 instances we were able to solve proved optimal. Recall [46] that for Restricted DP, direction *matters*, so it is advisable to run both forward and backward and choose the better bound.

Conclusion

We devised an *abstract aggregation* framework for Morin–Marsten Branch-and-Bound Scheme for Dynamic Programming (DPBB) and tested its OpenMP `tasks`-based shared-memory parallel implementation on conventional and *bottleneck* statements of Travelling Salesman with Precedence Constraints on instances from TSPLIB. The scheme proved better than non-bounded dynamic programming at tackling the larger instances, in particular, due to decreased memory usage, we have been able to close the long-standing `kro124p.4.sop`.

In bottleneck statement, we close 22 of 41 TSPLIB instances, including `kro124p.4.sop` and `ry48p.2.sop`.

The scheme proved to scale reasonably well, efficiently using as many as 36 processor cores to achieve a 15–35 reduction in computation time compared with its *serial* implementation. While not always absolutely faster than 'non-bounded' DP, in many cases, DPBB offered considerable savings in memory usage. The improvement on 'non-bounded' DP was more pronounced in the *bottleneck* case, see Tables 3 and 5, as was the difference between backward and forward statements.

The results for conventional, min-sum TSP-PC are summarized in Table 6, which compares the run times between 16-thread DP and 36-thread DPBB in forward and backward statements, our previous (serial) DP implementation [46], and the best known run times of other *exact* solutions from [21, Table 6]. The instances for which the optimum is not known are affixed with the dagger † symbol, and the **Value** column contains, instead of optima, the best known *upper bounds*, which is denoted by asterisk *.

This table also describes the parameters influencing the 'specific' complexity of instances, where n is the number of cities, w is the instance's *width*, $\#w$ is the instance's number in their list sorted by increasing w (the greater $\#w$ the harder, relatively, is the instance for DP), and ρ is the instance's *density*; for a discussion of these, see Section 1.1.1.

Overall, through DP and DPBB, we were able to improve the best known run times on 8 of 41 TSPLIB instances, all of them rather heavily constrained. For two of these, `ry48p.3.sop` and `kro124p.4.sop`, the only optimal solutions known were produced by DP or DPBB.

There is much room for improvement in the considered branch-and-bound scheme for dynamic programming. Speaking of improving the method – saving time and memory – one could try a *strong* lower bound rather than the 'fast' one considered here, for example, state-of-the-art bounds from [21].

Concerning abstraction of the aggregation operation, it will be interesting to find out whether there is a *lower bound heuristic* that could efficiently handle time- or past-sequence-dependent travel costs such as considered in [1,30,46,52] within this framework.

In addition to using improved lower bounds, various means of distributing the DP computations may be used to decrease the overall run time and further improve scalability such as (a) 'independent computation' distributed DP scheme [15], (b) parallel rollout [23], (c) divide-and-conquer approach for DP [34,47].

Items (a) and (c) may further improve DP states fathoming since they lead to a second layer of decomposition, thus, smaller problems, for which both lower and upper bound heuristics are expected to give results closer to the optimum.

We believe that a scheme coupling DPBB with the distributed scheme [15], using Restricted DP [46, Section 3.3] to get upper bounds on 'subproblems' arising during distribution and the node greedy, or better lower bound heuristic, should be much more efficient than either single-node DPBB or distributed non-bounded DP, and may be able to close at the very least `ry48p.2.sop` in the conventional TSP-PC statement.

Notes

1. Closed contemporaneously and independently by A. M. Grigoryev and A. G. Chentsov by a distributed computation scheme for DP [11], on 18 computational nodes, which took 47 hours and a total of 1.876 TB RAM, with an average of 106 GB per node (A. M. Grigoryev, *personal communication*).
2. On a single processor core.
3. See, e.g. [24, Ch. 13] and [27] for state of the art.
4. Closed independently by A. M. Grigoryev and A. G. Chentsov [11].
5. Equivalent formalizations are *acyclic* digraph, e.g. [18], and well-founded relation [15].
6. A bijection exists between the *forward* and *backward* states.
7. The DP validity theorems establish the fact that an optimum solution as obtained by DP is in fact an optimal solution in the sense of the problem itself, which has no inherent direction.
8. Maximum antichain cardinality [10, Definition 1.30]; for *nonempty* precedence constraints, $w < n$.
9. The authors are not aware of branch-and-bound solutions the search space of which are only *feasible* solutions except for [29], where the precedence constraints were not general.
10. Assuming the states are generated in *at most* $\mathcal{O}(nw)$ per ideal (satisfied by Algorithm 1); the best known are $\mathcal{O}(\log n)$ per ideal or less [10, Section A.2.2].
11. Since $\mathbf{I}[I] = \mathrm{Max}[I]$ [49, Theorem 2] and maximal elements form an antichain.
12. Special data structure, the design of which prevents *race conditions* when multiple threads attempt to write to it concurrently, see Section 2.1.2.
13. See https://abseil.io/about/.
14. See https://github.com/greg7mdp/parallel-hashmap.
15. See the official description at https://aws.amazon.com/ru/ec2/instance-types/x1e/.
16. See https://aws.amazon.com/marketplace/pp/B077GCH38C.
17. Note that it is *wall-clock*, not CPU time.
18. See the full list in [46, Table A.2, Supp. 2]. For relation between *width* and the complexity of DP for TSP-PC, see [46, Propositions 1,2]; note that *width* is central to upper and lower bounds on the number of order ideals [56, Equations (4.1),(4.2),], which lead to bounds on DP states.
19. Informally, we say computation is *halfway through* an n-city instance if it is at layer $\lceil n/2 \rceil$.

Acknowledgements

The authors gratefully acknowledge using the 'Uran' supercomputer, courtesy of the Krasovskii Institute.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Notes on contributors

Yaroslav. V. Salii is a Postdoctoral Researcher with McGill University, on sabbatical from his Researcher position with Krasovskii Institute of Mathematics and Mechanics. His scientific interests

include very large-scale network control, through graphons and mean-field games, as well as precedence-constrained problems in operations research and their connections with order theory.

Andrey S. Sheka is a Researcher in the Laboratory of Complex Systems Analysis of the Computational Systems Department at the N.N. Krasovskii Institute of Mathematics and Mechanics. He has authored over 30 publications on data analysis, artificial intelligence, and computational optimization.

ORCID

Yaroslav. V. Salii http://orcid.org/0000-0002-8617-5228
Andrey S. Sheka http://orcid.org/0000-0001-5763-351X

References

[1] H. Abeledo, R. Fukasawa, A. Pessoa and E. Uchoa, *The time dependent traveling salesman problem: polyhedra and algorithm*, Math. Program. Comput. 5 (2013), pp. 27–55.
[2] E. Balas, M. Fischetti and W.R. Pulleyblank, *The precedence-constrained asymmetric traveling salesman polytope*, Math. Program. 68 (1995), pp. 241–265.
[3] R. Baldacci, A. Mingozzi and R. Roberti, *New state–space relaxations for solving the traveling salesman problem with time windows*, INFORMS. J. Comput. 24 (2012), pp. 356–371.
[4] R.S. Barr and B.L. Hickman, *Reporting computational experiments with parallel algorithms: issues, measures, and experts' opinions*, ORSA J. Comput. 5 (1993), pp. 2–18.
[5] R. Bellman, *Dynamic programming treatment of the travelling salesman problem*, J. ACM 9 (1962), pp. 61–63.
[6] L. Bianco, A. Mingozzi, S. Ricciardelli and M. Spadoni, *Exact and heuristic procedures for the traveling salesman problem with precedence constraints, based on dynamic programming*, INFOR: Inform. Syst. Oper. Res. 32 (1994), pp. 19–32.
[7] N. Boysen, M. Fliedner and A. Scholl, *Scheduling inbound and outbound trucks at cross docking terminals*, OR Spectrum 32 (2010), pp. 135–161.
[8] G. Brightwell and P. Winkler, *Counting linear extensions*, Order 8 (1991), pp. 225–242.
[9] P.M. Camerini, *The min–max spanning tree problem and some extensions*, Inf. Process. Lett. 7 (1978), pp. 10–14. min–max directed spanning tree (arborescence).
[10] N. Caspard, B. Leclerc and B. Monjardet, *Finite Ordered Sets: Concepts, Results and Uses*, Encyclopedia of Mathematics and Its Applications Vol. 144, Cambridge University Press, Cambridge, 2012.
[11] A.G. Chentsov, A.M. Grigoryev and A.A. Chentsov, *Procedures of local optimization in routing problems with constraints*, in *Mathematical Optimization Theory and Operations Research, 18th International Conference, MOTOR 2019, Ekaterinburg, Russia, July 8–12, 2019, Revised Selected Papers*, I. Bykadorov, V.A. Strusevich, and T. Tchemisova, eds., Communications in Computer and Information Science, Springer, Vol. 1090. 2019.
[12] A.G. Chentsov, *Extremal problems of routing and scheduling: a theoretical approach [in Russian]*, Regular and Chaotic Dynamics, Izhevsk, 2008.
[13] A.G. Chentsov, *On a parallel procedure for constructing the Bellman function in the generalized problem of courier with internal jobs*, Autom. Remote Control 73 (2012), pp. 532–546.
[14] A.G. Chentsov and P.A. Chentsov, *A precedence constrained routing problem (precedence-constrained TSP): dynamic programming [in Russian]*, Bull. USTU (2004), pp. 148–151.
[15] A.G. Chentsov and A.M. Grigoryev, *A scheme of independent calculations in a precedence constrained routing problem*, in *International Conference on Discrete Optimization and Operations Research*. Springer, Vladivostok, Russia, September 19–23, 2016.
[16] A.G. Chentsov and Y.V. Salii, *A model of "nonadditive" routing problem where the costs depend on the set of pending tasks*, Vestnik YuUrGU. Ser. Mat. Model. Progr. 8 (2015), pp. 24–45.
[17] A.A. Cire and W.J. van Hoeve, *Multivalued decision diagrams for sequencing problems*, Oper. Res. 61 (2013), pp. 1411–1428.

[18] L.F. Escudero, *An inexact algorithm for the sequential ordering problem*, Eur. J. Oper. Res. 37 (1988), pp. 236–249.
[19] H.N. Gabow, Z. Galil, T. Spencer and R.E. Tarjan, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica 6 (1986), pp. 109–122.
[20] L. Georgiadis, *Arborescence optimization problems solvable by Edmonds' algorithm*, Theor. Comput. Sci. 301 (2003), pp. 427–437.
[21] L. Gouveia and M. Ruthmair, *Load-dependent and precedence-based models for pickup and delivery problems*, Comput. Oper. Res. 63 (2015), pp. 56–71.
[22] A.M. Grigoriev, E.E. Ivanko and A.G. Chentsov, *Dynamic programming in a generalized courier problem with inner tasks: elements of a parallel structure*, Model. Anal. Inform. Sist. 18 (2011), pp. 101–124.
[23] F. Guerriero and M. Mancini, *A cooperative parallel rollout algorithm for the sequential ordering problem*, Parallel. Comput. 29 (2003), pp. 663–677.
[24] G. Gutin and A.P. Punnen, *The Traveling Salesman Problem and Its Variations*, Combinatorial Optimization Vol. 12, Kluwer Academic Publishers, Dordrecht, 2002.
[25] W. Guttmann, *An algebraic framework for minimum spanning tree problems*, Theor. Comput. Sci. 744 (2018), pp. 37–55.
[26] M. Held and R.M. Karp, *A dynamic programming approach to sequencing problems*, J. Soc. Ind. Appl. Math. 10 (1962), pp. 196–210.
[27] K. Helsgaun, *Solving the equality generalized traveling salesman problem using the Lin–Kernighan–Helsgaun algorithm*, Math. Program. Comput. 7 (2015), pp. 269–287.
[28] J. Jamal, G. Shobaki, V. Papapanagiotou, L.M. Gambardella and R. Montemanni, *Solving the Sequential Ordering Problem using Branch and Bound*, in *Computational Intelligence (SSCI), 2017 IEEE Symposium Series on*. IEEE, Honolulu, HI, 2017, pp. 1–9.
[29] B. Kalantari, A.V. Hill and S.R. Arora, *An algorithm for the traveling salesman problem with pickup and delivery customers*, Eur. J. Oper. Res. 22 (1985), pp. 377–386.
[30] J. Kinable, A.A. Cire and W.J. van Hoeve, *Hybrid optimization methods for time-dependent sequencing problems*, Eur. J. Oper. Res. 259 (2017), pp. 887–897.
[31] V.N. Kolokoltsov and V.P. Maslov, *Idempotent Analysis and Its Applications*, Mathematics and Its Applications Vol. 401, Springer Science & Business Media, Dordrecht, 1997.
[32] M. Kulukundis, *Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step*, in CPPcon. Standard C++ Foundation, 2017. Conference Talk Recording. Accessed 06.09.2019. Available at https://www.youtube.com/watch?v = ncHmEUmJZf4.
[33] J. LaRusic and A.P. Punnen, *The asymmetric bottleneck traveling salesman problem: algorithms, complexity and empirical analysis*, Comput. Oper. Res. 43 (2014), pp. 20–35.
[34] E.L. Lawler, *Efficient implementation of dynamic programming algorithms for sequencing problems*, Tech. Rep. BW 106/79, Stichting Mathematisch Centrum, 1979.
[35] L. Libralesso, A.M. Bouhassoun, H. Cambazard and V. Jost, *Tree search algorithms for the sequential ordering problem*, (2019). Available at https://arxiv.org/abs/1911.12427.
[36] T.L. Magnanti and L.A. Wolsey, *Optimal trees*, in *Network Models*, M.O. Ball, T.L. Magnanti, C.L. Monma, and G.L. Nemhauser, eds., Handbooks in Operations Research and Management Science Vol. 7, chap. 9, Elsevier, Amsterdam, 1995, pp. 503–615.
[37] C. Malandraki and R.B. Dial, *A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem*, Eur. J. Oper. Res. 90 (1996), pp. 45–55.
[38] A.A. Mastor, *An experimental investigation and comparative evaluation of production line balancing techniques*, Manage. Sci. 16 (1970), pp. 728–746.
[39] A. Mingozzi, L. Bianco and S. Ricciardelli, *Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints*, Oper. Res. 45 (1997), pp. 365–377.
[40] M. Minoux, *Programmation Mathèmatique. Théorie et Algorithmes*, 2nd ed., Lavoisier, Lassay-les-Châteaux, 2008.
[41] R. Montemanni, D.H. Smith and L.M. Gambardella, *A heuristic manipulation technique for the sequential ordering problem*, Comput. Oper. Res. 35 (2008), pp. 3931–3944.

[42] T.L. Morin and R.E. Marsten, *Branch-and-bound strategies for dynamic programming*, Oper. Res. 24 (1976), pp. 611–627.
[43] V. Papapanagiotou, J. Jamal, R. Montemanni, G. Shobaki and L.M. Gambardella, *A comparison of two exact algorithms for the sequential ordering problem*, in *Systems, Process and Control (ICSPC), 2015 IEEE Conference on*. IEEE, Bandar Sunwa, 2015, pp. 73–78.
[44] J.S. Provan and M.O. Ball, *The complexity of counting cuts and of computing the probability that a graph is connected*, SIAM J. Comput. 12 (1983), pp. 777–788.
[45] G. Reinelt, *TSPLIB–a traveling salesman problem library*, ORSA J. Comput. 3 (1991), pp. 376–384. See https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.
[46] Y. Salii, *Revisiting dynamic programming for precedence-constrained traveling salesman problem and its time-dependent generalization*, Eur. J. Oper. Res. 272 (2019), pp. 32–42. https://doi.org/10.1016/j.ejor.2018.06.003.
[47] Y.V. Salii, *On Forward and Backward Dynamic Programming for Precedence Constrained Routing Problems and The Algorithms for Generation of Feasible Subproblems [in Russian]*, Proceedings of the XV All-Russian Conference on Mathematical Programming and Applications. N.N. Krasovskii Institute of Mathematics and Mechanics, UrB RAS; B.N. Yeltsin Ural Federal University, Bulletin of the Mathematical Programming Association Vol. 13, 2015, pp. 168–169.
[48] Y.V. Salii, *Restricted Dynamic Programming Heuristic for Precedence Constrained Bottleneck Generalized TSP*, Proceedings of the 1st Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists, A. Sozykin, E. Akimova, and D. Ustalov, eds., CEUR Workshop Proceedings, Yekaterinburg, Vol. 1513. 2015, pp. 85–108. Available at http://ceur-ws.org/Vol-1513/#paper-10.
[49] Y.V. Salii, *Order-Theoretic Characteristics and Dynamic Programming for Precedence Constrained Traveling Salesman Problem*, Proceedings of the Fourth Russian Finnish Symposium on Discrete Mathematics, J. Karhumäki, Y. Matiyasevich, and A. Saarela, eds., TUCS Lecture Notes Vol. 26. Turku Centre for Computer Science, Turku, 2017, pp. 152–164. Available at http://urn.fi/URN:ISBN:978-952-12-3547-4.
[50] G. Schmidt and T. Ströhlein, *Relations and graphs: discrete mathematics for computer scientists*, *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, Berlin, 1993.
[51] B.S.W. Schröder, *Ordered Sets*, Birkhäuser, Boston, 2003.
[52] A.N. Sesekin, A.G. Chentsov and A.A. Chentsov, *Routing with an abstract function of travel cost aggregation [in Russian]*, Trudy Inst. Mat. i Mekh. UrO RAN 16 (2010), pp. 240–264.
[53] A. Sesekin, A. Chentsov and A. Chentsov, *On a bottleneck routing problem*, Proc. Steklov Inst. Math. 272 (2011), pp. 165–185.
[54] R. Skinderowicz, *An improved ant colony system for the sequential ordering problem*, Comput. Oper. Res. 86 (2017), pp. 1–17.
[55] G. Steiner, *On the complexity of dynamic programming for sequencing problems with precedence constraints*, Ann. Oper. Res. 26 (1990), pp. 103–123.
[56] G. Steiner, *On estimating the number of order ideals in partial orders, with some applications*, J. Stat. Plan. Inference. 34 (1993), pp. 281–290.
[57] C. Tilk and S. Irnich, *Dynamic programming for the minimum tour duration problem*, Transportation Sci. 51 (2017), pp. 549–565.
[58] C.H. zu Siederdissen, S.J. Prohaska and P.F. Stadler, *Dynamic programming for set data types*, in *Brazilian Symposium on Bioinformatics*. Springer, Belo Horizonte, 2014, pp. 57–64.