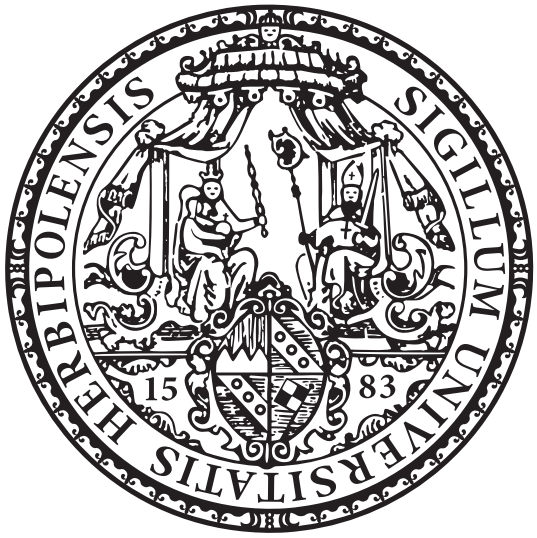
Practical Course Report

Bounded State-Space-Search for the Traveling-Salesman-Problem with Pickup and Deliveries (TSPPD)

Lukas Kröger

Date of Submission: 2026-06-15
Advisor: Prof. Dr. Marie Schmidt



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

1. Overview

The research objective is evaluating an iterative solution strategy for specific types of vehicle routing problems. We propose a solving scheme related to *state space search* that progressively explores a *state space* graph while taking into account lower and upper bounds for the remaining costs. Out of the possible applications of this type of state space search, the Traveling-Salesman-Problem with Pickups and Deliveries (TSPPD) is chosen for this report.

2. Problem description

2.1. Introduction

The TSPPD is a generalization of the Traveling-Salesman-Problem (TSP) and a special case of the Traveling-Salesman-Problem with precedence constraints. The TSPPD, just like the TSP, is restricted to a single uncapacitated vehicle. To distinguish it from a (state space) graph defined later, we will consider the TSPPD to be defined on a graph of *locations*. Foremost, a single depot location is given, which is segregated into a depot departure location 0^+ (where the transporting vehicle starts its tour from) and a depot arrival location 0^- (where the vehicle ends its tour). All other locations are associated with requests $\mathcal{R}$ for an item to be transported. In general, each of the $n = |\mathcal{R}|$ items $i \in \mathcal{R}$ has a dedicated pickup i^+ and a dedicated delivery location i^-. Geographically, different locations can coincide – in particular, the depot departure and arrival location.

Now, we are searching for the shortest (Hamiltonian) tour covering all locations while respecting the problem-specific precedence constraints. That is, that each pickup location i^+ is visited by the vehicle before the corresponding delivery location i^- for $i \in \mathcal{R}$.

2.2. Input Instance

2.2.1. Requests

The most fundamental input construct to the TSPPD is the set of requests $\mathcal{R}$. Requests in the TSPPD typically correspond to a physical object/item. Each request $i \in \mathcal{R}$ has a dedicated pickup location i^+ and a dedicated delivery location i^-. For simplicity, in typical instances used for benchmarking solution algorithms, locations are most often randomly spread over a bounded two-dimensional Euclidean plane with the pairwise distance between locations given by the Euclidean distance.

2.2.2. Locations

Each location $l \in \mathcal{L}$ is characterized by a geographical position on the one hand and a *location type* on the other hand:

1. Departure at depot 0^+
2. Arrival at depot 0^-

3. Pickup location i^+ corresponding to a request $i \in \mathcal{R}$

4. Delivery location i^- corresponding to a request $i \in \mathcal{R}$

Like in the TSP, all locations are interconnected, i.e. the graph induced by the locations and arcs is complete. We will use $\mathtt{d}(l_1, l_2)$ as the distance and cost to get directly from location l_1 to l_2.

When we specifically refer to *Location*-Graph that is induced by these very locations, we will refer to it as $LG = (\mathcal{L}, \mathcal{L}^2)$.

2.3. (Partial) Tours

Definition 2.1. *A* (partial) tour t *is an ordered sequence of location identifiers* $l \in \mathcal{L}$ *that represents visiting the locations in some clearly specified order:*

$$t := (l_1, l_1, \ldots, l_k) \tag{1}$$

A partial tour is called complete *if it starts* and *ends at the depot while serving* all *locations* $l \in \mathcal{L}$*, with the first and last element being* 0^+ *and* 0^-*, respectively.*

Additionally, any (partial) tour has to satisfy:

1. *Any location* $l \in \mathcal{L}$ *will occur at most once in this list.*

2. $l_1 = 0^+$ *and/or* $l_k = 0^-$

3. *Precedence constraints for any complete tour that includes the (partial) tour: Each pickup* i^+ *happens before its corresponding delivery* i^-*. That means, any such complete tour needs to be of the form* $(\ldots, i^+, \ldots, i^-, \ldots)$ *for all* $i \in \mathcal{R}$*.*

Moreover, a partial tour t^p is a *prefix* of another, longer (partial) tour t, iff t^p is fully contained in t as a continuous sequence starting at t_1, i.e. iff $|t| > |t^p|$ and $t^p_i = t_i$ for $i \leq |t^p|$.

The cost c^t of a tour $t = (l_1, l_2, \ldots, l_n)$ is defined as $\sum_{i=1}^{n-1} \mathtt{d}(l_i, l_{i+1})$.

If a complete tour t is of minimum cost, such that there exists no other tour t' with $c^{t'} < c^t$, we may call it t^{OPT}. The formal objective of the TSPPD is finding t^{OPT}.

3. Status Quo on Exactly Solving the TSPPD

3.1. Bellman-Held-Karp

Even though not covered in the literature, it would be possible to transpose the Bellman-Held-Karp algorithm [Bel62] (a dynamic programming algorithm originally designed for the TSP) to the TSPPD (Bellman [Bel62], Held et al. [HK62]). This would divide the TSPPD into subproblems of only computing optimal solutions subsets of the location set and a last location. These subgraphs and -problems can be seen as *states* or *state nodes* and will be relevant for this report to. Here, we can make advantage of the precedence

constraints and only allow certain subsets of the location set. In detail, Section 4 will explain this.

As Section 4 will show, such a modified Bellman-Held-Karp would require continuous storage of a dynamic programming table of size $|\mathcal{L}| - \cdot 3^{|\mathcal{R}|-1} + 2$. This table could then be filled with $O(n^2 \cdot 3^n)$ many steps until the optimal solution value for the complete problem. The solution itself can be reconsctructed using an additional table of size $n \cdot 3^n$ storing the optimal predecessor of each state.

3.2. ILP-Based Approach

Integer linear programs (ILP) for routing problems typically use binary decision variables $x_e \in x$, such that x would be a vector $\in \mathbb{Z}_2^{|L^2|}$ associated with the edges e in our location graph L^2. For finding the optimal assignment with reasonable efficiency, a binary Branch-and-Bound tree can be constructed having nodes for each decision variable. It considers one decision variable (node) at a time, where evaluates the assignments $x_e = 0$ and $x_e = 1$. Each of these choices for x_e then forms an individual branch in the (binary) Branch-and-Bound tree. These branches are supposed to be evaluated in terms of lower and upper bounds of the objective function value, with x_e fixed. This recursively happens by considering another edge e' with again the choices $x_{e'} = 0$ and $x_{e'} = 1$. So, nodes are associated with some partial assignment of previously fixed decision variables. For this reason, every node in the Branch-and-Bound tree already provides some lower bound on the final costs with a complete assignment containing the (partial) assignment of decision variables at the node — given by the costs of the edges of this partial assignment.

A suitable order of evaluation of decision variables then enables branches to be *pruned* based on suboptimal lower bounds such that not all of the $2^{|\mathcal{L}^2|}$ combination of choices for the decision variables need to be tried to find the true optimal solution and prove its optimality.

A more sophisticated method is Branch-and-Cut, which is most notably done for the TSPPD by Dumitrescu et al. [DRCL]. In these, (non-integer) relaxations of the ILP are iteratively formulated and computed while gradually forcing the relaxed LP to assign integer values to the decision variable using so called *cuts*. Cuts introduce equations into the linear program that are satisfied by every valid integer solution to the problem, but *not* by the current optimal solution for the relaxed linear program. Because of this characterisitic, cuts are also often called valid inequalities.

3.3. State-Space-Based Approach

As opposed to that, in the approach to be presented in this report, a type of branching is used that is untypical in Branch-and-Bound algorithms. Branches will emerge in the process of exploring partial tours in the form of candidate successor locations of the last location of a partial tour. However, these branches are not destined to be separated forever (see Corollary 5.1), as the underlying decision graph will not be a tree. This prohibits "pruning" in the traditional sense, but in return, we can work with a graph

significantly smaller than a Branch-and-Bound tree. Another similarity to Branch-and-Bound techniques is that we are gradually narrowing down the value of the optimal solution for our problem and subproblems as well.

4. State-Space-Graph

4.1. Introduction

The *state space graph* $G = (V, A)$ was originally motivated by the success of state space (or event-based) formulations for the Dial-a-Ride-Problem (e.g. Gaul et al [GKS21]). On the other hand, the state space graph captures how the Bellman-Held-Karp-Algorithm models the TSP. Lastly, it is an instance of the concept of *state spaces*, a term that happens to be used most prominently in the foundations of (classical) artificial intelligence research, most famously by Russel and Norvig [RN16].

Definition 4.1. State nodes v *are tuples of a location and a set of (already) visited locations*

$$v := (l_v, L_v) \qquad \text{with } l \in \mathcal{L}, L_v \subseteq \mathcal{L} \tag{2}$$

such that there exists some partial tour $t := (l_0, l_1, \ldots, l_v)$ *and* $L_v = \{l_0, l_1, \ldots, l_v\}$ *When we are interested in the location associated with an state node* v*, we will refer to it as* l_v*.* L_v *will refer to the set of visited locations associated with* v*.*

Since for the depot departure node $(0^+, \{0^+\})$ and the depot arrival node $(0^-, \mathcal{L})$ the set of visited locations is fixed, we will refer to them as 0^+ and 0^-, respectively.

Also, it is often important to subsume the *unvisited* locations given some state node v, which will be done with $\overline{L_v}$. Often, we will also require the set of all unvisited locations at v, *except* the depot arrival location $\overline{L_v} \setminus \{0^-\}$.

Computationally, it is possible to uniquely map tours $t \in T$ to state nodes. For this, each request can be thought of to be in one of three states: not picked up, picked up, picked up and dropped off:

$$\mathcal{R} \times \{0, 1, 2\}^n \to V \tag{3}$$

It has to be noted that this function is not injective, as different tours can be mapped to the same state node, provided the same set of locations has been visited and the last element of the tour coincides. In particular, the state node corresponding to the final arrival at the depot 0^- with all locations served $L_v = \mathcal{L}$ subsumes all possible solutions for our TSPPD instance.

Fine-grained layered sets As can be seen in Figure 1, the nodes spread over several *layers*. We will identify the index of the layer as *depth* $d \in \{1, 2, \ldots, |\mathcal{L}|\}$, defined by the number of locations $|L_v|$ already visited at some node $v \in V^d$. For each $d \in \{1, \ldots, |\mathcal{L}|\}$, there exists a specific subset of the vertices V^d that includes all state nodes with $|L_v| = d$. In total,

$$V = \bigcup_{1 \leq d \leq |\mathcal{L}|} V^d$$

Arcs $a \in A$ Arcs (u, v) are pairs of state nodes and are closely related to the edges (l_1, l_2) in the location graph L. However, the state nodes u, v in an arc (u, v) need to obey certain restrictions to form an arc $a \in A$ as per Definition 4.2.

Definition 4.2. *The set of arcs A contains exactly the arcs $((l_u, L_u), (l_v, L_v)) \in V^2$ where the following holds:*

1. *l_v is not (already) visited at node u: $l_v \notin L_u$*
2. *v has the same set of visited locations as u plus l_v: $L_v = L_u \cup \{l_v\}$*
3. *Precedence constraints are satisfied: if l_v is a delivery location of the form i^-, the corresponding pickup location must be already visited, i.e. $i^+ \in L_u$.*

All arcs $a \in A$ inherit the costs and travel times from the edges $\mathcal{L}^2$ of the location graph $L = (\mathcal{L}, \mathcal{L}^2)$. As a shorter alternative to $\mathtt{d}(l_u, l_v)$ we will however use $\mathtt{d}(u, v)$ that is defined for all arcs $(u, v) \in A$.

4.2. Successors $w \in \sigma_v$ and Predecessors $u \in \pi_v$ of State Nodes v

For compactness, we will use an adaption of the usual δ-notation for subsuming outgoing and ingoing arcs in directed graphs.

Definition 4.3. *σ_v is the set of possible* successors *of a state node v. These are state nodes w with $\exists(v, w) \in A$. On the contrary, π_v is the set of possible* predecessors *of a state node u. These are state nodes u with $\exists(u, v) \in A$.*

Observation 4.1. *Successors and predecessors are symmetric relationsships: if v is a successor of a node u, u is a predecessor of v and vice versa.*

Lemma 4.1. *The successors of some state node u all share the same set of predecessors. Likewise, the predecessors of some node all have the same set of successors.*

$$\forall u \in V \colon \forall v, v' \in \sigma_u : \pi_v = \pi_{v'}$$

$$\forall v \in V \colon \forall u, u' \in \pi_v : \sigma_u = \sigma_{u'}$$

We prove the two statements separately, beginning with the first:

Proof. $\pi_v = \pi_{v'}$ Due to substitutability of v by v', we prove $\pi_v \subseteq \pi_{v'}$ and $\pi_v \supseteq \pi_{v'}$ in one step:

Given some state node u, suppose there would be nodes $v, v' \in \sigma_u, v \neq v'$ where there is a predecessor $x \in \pi_v$ of v that is not a predecessor of v', i.e. $x \notin \pi_{v'}$. Obviously, $x \neq u$.

Now for x to *not* be part of $\pi_{v'}$, any of the three conditions given in Definition 4.2 has to be violated regarding $l_{v'}, L_{v'}, L_x$. Fortunately enough, we can show that all of these are guaranteed easily. Namely, we just have to take a look at L_x and apply the conditions of the arc definition for the arcs::

- Existence of (x, v) requires that $l_v \notin L_x, L_v = L_x \cup \{l_v\}$ and that predence constraints are satisfied
- Existence of (u, v) requires that $l_v \notin L_u, L_v = L_u \cup \{l_v\}$ and that predence constraints are satisfied

So it has to hold $L_u = L_x$.

Now given that $(u, v') \in A$ and thus none of the conditions are violated w.r.t. $l'_v, L_{v'}, L_u$, we can immediately see that the conditions cannot be violated w.r.t. $l'_v, L_{v'}, L_x$, because the conditions rely on the same objects. Hence, such a node $x \notin \pi_{v'}$ cannot exist. □

Proof. $\sigma_u = \sigma_{u'}$ Again, we prove $\sigma_u \subseteq \sigma_{u'}$ together with $\sigma_u \supseteq \sigma_{u'}$ in one step: Given some state node v, suppose there would be some nodes $u, u' \in \pi_v, u \neq u'$ with another node $x \in \sigma_u, x \notin \sigma_{u'}$.

Now for x to be *not* part of $\sigma_{u'}$, any of these three conditions given in Definition 4.2 has to be violated regarding $l_x, L_x, L_{u'}$. Applying the conditions from Definition 4.2 in the same way as before for the existing arcs (u', v) and (u, v), we can infer $L_u = L_{u'}$. So if $(u, x) \in A$, it must be $(u', x) \in A$ as well, as both rely on the exact same objects. Thus, such an arc $x \notin \sigma_{u'}$ cannot exist. □

Lemma 4.2. *Any node $v \in V_d$ with $d < 4$ has at most one predecessor and incoming arc and any node $v \in V_{d'}$ with $d' > |\mathcal{L}| - 2$ has at most one successor and outgoing arc.*

We are going to fully prove only the first part of the Lemma. The second part can be proven symmetrically.

Proof. We will show this by simply looking at all values for $d \in \{1, 2, 3\}$ separately.

1. For $d = 1$, this is trivial, as 0^+ is the only node at this depth and is has no incoming arc.
2. At $d = 2$, v must be of the following form

   $$v = (l, \{l\}), \exists i \in \mathcal{R} \colon l = i^+$$

   because only one location has been visited, namely some pickup location. It is obvious that it only has the depot departure 0^+ as predecessor.
3. At $d = 3$, we must have visited exactly two locations:

   $$v = (l', \{l, l'\}) \text{ with } \exists i \in \mathcal{R} \colon l = i^+ \text{ and } (\exists i' \neq i \in \mathcal{R} \colon l' = i'^+ \text{ or } l' = i^-)$$

   In words: either l' is a pickup location as well, or l' is the corresponding delivery location to l. In either case, there can only be one predecessor to such a node, namely $u = (l, \{l\})$. It cannot be $(l', \{l'\})$, since l' is visited at state node v.

A proof for second part of the lemma could be formulated using the exact same technique, with the only difference the we consider the nodes $\overline{L_v}$ at a state node v that have *not* yet been visited instead. □

4.3. Cardinality of V

Unsurprisingly, Theorem 4.1 highlights that a dynamic programming approach to the TSPPD saving properties of all state nodes requires exponential space.

Theorem 4.1. *With n requests, the number of state nodes $|V|$ is $\in \Theta(n \cdot 3^n)$*

Proof. Given the structure of an state node (l_v, L_v), a total of $|\mathcal{L}|$ locations can coocurr with some subsets of $L_v \subseteq \mathcal{L}$. These subsets have to satisfy the condition that for any delivery location i^- they also contain the corresponding pickup location i^+, otherwise the precedence constraint would have been violated. To capture these specific subsets, one could use a ternary encoding mapping each request to a ternary digit to signify if the request is unserved, picked up, picked up and delivered. This would result in a total of 3^n different subsets of $\mathcal{L}$ that can be referenced by valid state nodes.

In addition, the current location l_v determines and fixes the state of the corresponding request. A special case are the depot nodes 0^+ and 0^- that each can only cooccur with a specific set of visited locations, $\{0^+\}$ and $\mathcal{L}$ itself, as the depot can only be left without any other location visited so far and reached only with all locations already visited. Thus, the number of state nodes is $(|\mathcal{L}| - 2) \cdot 3^{|\mathcal{R}|-1} + 2 \in \Theta(n \cdot 3^n)$. □

4.4. Definite Costs for State Node Pairs

Similar to $\mathtt{d}(u, v)$, which is only defined for $u \in \pi_v, v \in \sigma_u$, we have a more general cost property $\mathtt{c}(u, v)$ that can apply to any two nodes u, v.

Definition 4.4. *Let $t := (l_0, l_1, \ldots, l_k)$ be some partial tour and $v_i = \{l_i, L_i\}$ with $L_i = \{l_0, l_1, \ldots, l_i\}$. Then $\mathtt{c}(v_i, v_j)$ with positive $i < j \leq k$ is the exact cost of the cheapest partial tour from l_i to l_j that visits the same set of locations as t between l_i and l_j, but in arbitrary order while still respecting precedence constraints.*

In particular, $\mathtt{c}(0^+, 0^-)$ is the definite cost of t^{OPT}.

Many of the previous definitions and propositions are also visually observable in figure 1. It shows that the state space graph consists of $|\mathcal{L}|-1$ layers with growing sizes towards the left center layer at depth $\lfloor \frac{|\mathcal{L}|}{2} \rfloor$ and shrinking layer sizes starting from the right center layer at depth $\lceil \frac{|\mathcal{L}|}{2} \rceil$. The d-th layer corresponds to the set V_d. Besides, the illustration demonstrates the perfect two-fold symmetry of this type of graph. For a clearer picture, edge directions are not indicated here, but in general, all edges go from left to right — making this a directed acyclic graph (DAG). While it apparently is not a tree, it resembles a contraction of the solution tree of the TSPPD and exhibits some tree-like properties we are going to take advantage of.

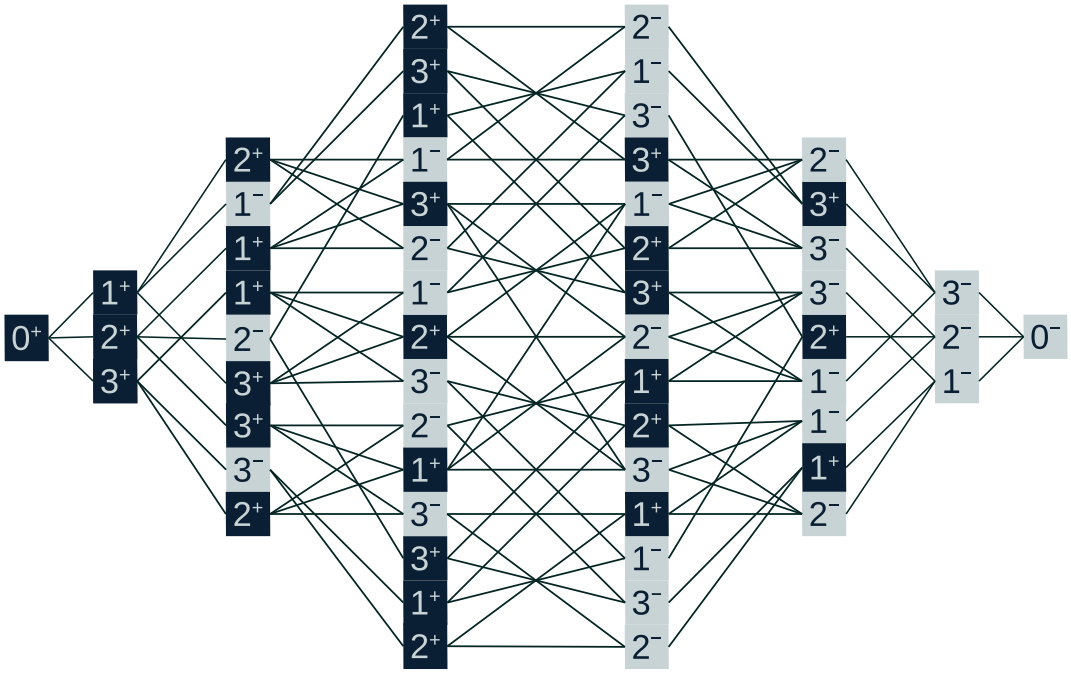


Fig. 1: A visualization of the state space graph for $n = 3$. Due to lack of space, state nodes v are only labeled by their location l_v and can be identified by the paths leading to them. For example, the upmost node in the left center column labeled 2^+ represents the state node $(2^+, \{1^+, 1^-, 2^+\})$. Pickup nodes are in a dark blue tone, dropoff nodes in a light blue tone.

5. State-Space-Search

Now that we have introduced the concept of the state space for the TSPPD, we will focus on how you can traverse the state space to find an exact solution to the TSPPD. We will first deal with the memory aspect (which nodes we will store).

5.1. Cached Properties of State Nodes

For the cost property from Definition 4.4 we now impose a lower bound property $\texttt{LB}(u, v)$ and an upper bound property $\texttt{UB}(u, v)$. In this report, we are only interested in cost properties where at least one of the two nodes is a depot node. Because of this, we use more convenient notations: $\texttt{c}^+(u) = \texttt{c}(0^+, u), \texttt{c}^-(u) = \texttt{c}(0^-, u)$ and likewise, $\texttt{LB}^+(u) = \texttt{LB}(0^+, u), \texttt{LB}^-(u) = \texttt{LB}(0^-, u)$ as well as $\texttt{UB}^+(u) = \texttt{UB}(0^+, u), \texttt{UB}^-(u) = \texttt{UB}(0^-, u)$.

- $\texttt{LB}^+(v)$: The current lower bound for the cost of getting from the depot departure 0^+ to v (cost-so-far), only relevant for the backward-search
- $\texttt{UB}^+(v)$: The current upper bound for the cost of getting from the depot departure 0^+ to v, only relevant for the backward-search

- $\texttt{LB}^-(v)$: The current lower bound for the cost of getting from v back to the depot arrival node 0^- (cost-remaining), only relevant for the forward-search
- $\texttt{UB}^-(v)$: The current upper bound for the cost of getting from v back to the depot arrival node 0^-, only relevant for the forward-search

These properties are subject to change during the executiong of the algorithm, but the fundamental trait of these properties is that at any time $\texttt{LB}(u, v) \leq \texttt{c}(u, v) \leq \texttt{UB}(u, v)$.[1]

The lower bounds of nodes added to V^{map} are initialized with a *suitable* value when an state node is created and monotonically increased, whereas the upper bounds will be initialized with ∞ and monotonically reduced — with the exception of $\texttt{UB}(0^-, 0^-) = \texttt{UB}(0^+, 0^+) = 0$. This exception is necessary and sufficient for the termination of Algorithm 1, which will be proven in Section 5.10. While Section 5.11 deals with reasonable methods for lower bound initialization, for simplicity, we will initialize all lower bounds with the value zero.

5.2. Storage of State Nodes

Whether the set of visited locations should actually be stored as a ternary number as Theorem 4.1 suggests, is subjected to a tradeoff between memory and performance implications. For practical implementations, the potential memory savings of the more compact ternary encodings compared to a binary encoding has to be weighed against the clear performance advantages of processing bit strings.

Our aim will be to not explicitly store all state nodes v. For this reason, after the search direction is fixed, an array of hash-maps V^{map} is used that stores two of the totally four bound values for each state nodes in the data structure, indexed by locations $l \in \mathcal{L}$ foremost.

$$V^{\text{map}}[l] : \{0, 1\}^{|\mathcal{L}|} \to \mathbb{Q}^2 \tag{4}$$

Unlike in the Bellman-Held-Karp algorithm, it is not necessary to store a *parent* location that points to the previous location to establish t^{OPT} itself. The optimal tour can instead be reconstructed employing upper bound values after the algorithm has terminated (see section Section 5.7).

5.3. Main Objectives of State Space Search

For the actual search routine, we need to address the issue which nodes have to be explored to find the optimal solution $t^{\text{OPT}} \in T$ with costs $\texttt{c}(0^+, 0^-)$. We use an iterative approach that traverses our graph in a path-based way. Thereby, we have the following two goals:

1. Finding t^{OPT} as fast as possible by aligning $\texttt{UB}(0^+, 0^-)$ with $\texttt{c}(0^+, 0^-)$

[1] For brevity, there is no additional argument used indicating the time the value of one those properties is retrieved. However, it is still important to note that all of these four properties are transient as long as the gap between lower and upper bound is not closed. We implicitly always refer to the bounds at a specific time τ, when we compare bounds with each other at any passage of this publication.

2. Proving optimality of t^{OPT} as fast as possible once we have found it by aligning $\mathtt{LB}(0^+, 0^-)$ with $\mathtt{c}(0^+, 0^-)$

These two goals can indeed be in conflict with each other, which is a common challenge in optimization and exact algorithms. Branch-and-Bound or Branch-and-Cut methods for exactly solving ILPs often focus on proving optimality which can slow down the process of finding better solutions.

5.4. Search Paths

A central concept for the state space search is the *search path* $P = (v_0, v_i, \ldots, v_k)$. It captures a valid sequence of state nodes – valid in the sense that

1. for forward seach any state node v_{i+1} in the search path is a valid successor of the v_i, starting with $v_0 = 0^+$
2. for backward seach any state node v_{i+1} in the search path is a valid predecessor of the v_i, starting with $v_0 = 0^-$

This concept is not new, as it is closely related to the partial tours $(0^+, l_1, .., l_v) \in T$ described in 2.3. The only difference is that the search path is capturing state nodes instead of pure locations. The equivalence is founded in the following: Any location $l_i \in T$ together with the prefix of l_i in the search path — $(l_0, l_1, \ldots, l_{i-1})$ — will define a state node (l_i, L_i) with $L_i = \{l_0, l_1, \ldots, l_{i-1}\}$. From another perspective, it also means that our algorithm repeatedly follows many distinct valid tours.

An important figure for the more elaborate search routine presented in Section 6.4 will be the cost $\mathtt{c}(P)$ of the current search path P. This cost can be set in relation with our previously defined bounds. If we are executing a forward-oriented search starting at 0^+, $\mathtt{c}(P)$ is an upper bound for $\mathtt{c}(0^+, v_k) = \mathtt{c}^+(v_k)$. If we are executing a backward-oriented search starting at 0^-, $\mathtt{c}(P)$ is an upper bound for $\mathtt{c}(v_k, 0^-) = \mathtt{c}^-(v_k)$.

As an accessory, Corollary 5.1shows some fundamental property of search paths that holds for the TSPPD, but also for the TSP. Two search paths can share share some prefix, diverge at some index and "meet again". This is also visible in Figure 1.

Corollary 5.1. *Let $P = (v_0, v_1, \ldots, v_i, v_{i+1}, \ldots, v_j, \ldots, v_n)$ be a search path and $P' = (v_0, v_1, \ldots, v_i, v'_{i+1}, \ldots, v_j, \ldots, v_n)$ be a search path diverging from P at the index $i+1$, such that $v_{i+1} \neq v'_{i+1}$. Then $j \geq i+3$.*

Proof. For the proof, let $l_i = l_{v_i}, v_i \in P$, $l'_i = l_{v'_i}, v'_i \in P'$, $L_i = L_{v_i}, v_i \in P$, $L'_i = L_{v'_i}, v'_i \in P'$.

Nodes $v_{i+1} \neq v'_{i+1}$ diverge iff $l_{i+1} \neq l'_{i+1}$. Now suppose that $v_{i+2} = v'_{i+2}$. That would require $l_{i+2} = l'_{i+2}$ and $L_{i+2} = L'_{i+2}$. However, if $l_{i+1} \neq l'_{i+1}$ and $l_{i+2} = l'_{i+2}$, then $l'_{i+1} \notin L_{i+2}$ and $l_{i+1} \in L'_{i+2}$.

On the other hand, one can construct according search paths P and P', such that $v_{i+3} = v'_{i+3}$. For this, we can set $l_{i+1} = l'_{i+2}$, $l_{i+2} = l'_{i+1}$ and $l_{i+3} = l'_{i+3}$. Then, we obviously have $L_{i+3} = L'_{i+3}$. □

Using search paths, we can also give an alternative, equivalent definition of the definite cost.

Definition 5.1. *$c(u, v)$ is computable for all $u, v \in V$ for which there there exists a search path P of the form $(\dots, u, \dots, v, \dots)$. It indicates the minimal possible cost for the sequence of arcs between u and v in any search path.*

5.5. Progressing the Search Path and Lower Bound Lifting

5.5.1. Progressing Rule

Initially, any path starts with the depot departure 0^+ or arrival 0^- with the cost of the search path being initialized with 0. Then, the further progress of the search path is determined by the *progression rule*. It determines which successor node is most attractive as the next node in the search path.

This lays the foundation to the following progression rule for finding the next state node $v \in \sigma_u$ to explore when currently working on state node u:

The progression rule constitutes that we always continue the search with the successor σ_u^* of u that minimizes $\mathtt{d}(u, v) + \mathtt{LB}^-(v)$. The formal definition Definition 5.2

Definition 5.2. *Let P be the current search path and u the most recently added node to P. Then only some node v that is in the set*

$$\sigma_u^* = \arg\min_{v \in \sigma_u} (\boldsymbol{d}(u, v) + \boldsymbol{LB}^-(v))$$

is eligible for being the added to P next for the forward search. On the other hand, the corresponding set for the backward search is:

$$\pi_v^* = \arg\min_{u \in \pi_v} (\boldsymbol{LB}^+(u) + \boldsymbol{d}(u, v))$$

Same as for the LB and UB-properties of event nodes, we implicitly handle the time τ at which σ_u^* has a certain value.

5.5.2. Lower Bound Successor-to-Depot Lift (minS2D)

The lower bound of the remaining costs (independent of search direction) of a state node can at any point in search be increased to the the minimum of the lower bound of all successor nodes including the cost to get to the respective successor node:

$$\forall u \in V : \mathtt{LB}^-(u) \leftarrow \min_{v \in \sigma_u} (\mathtt{d}(u, v) + \mathtt{LB}^-(v)) \tag{5}$$

This relationship is directly implied by the fact that any optimal (partial) tour $\mathtt{c}^-(u)$ from u to 0^- has at least (and even exactly) the costs $\min_{v \in \sigma_v}\{\mathtt{d}(u, v) + \mathtt{c}^-(v)\}$. This fact is closely related to the main idea behind the famous Held-Karp algorithm. Using the predecessor and successors of nodes in our state space-model, the relationship is almost trivial: $t^{\mathrm{OPT}}(v, 0^-)$ must contain exactly one node $v \in \sigma_u$.

The corresponding construct for the backward search is:

$$\forall u \in V : \mathtt{LB}^+(v) \leftarrow \min_{u \in \pi_v} (\mathtt{LB}^+(v) + \mathtt{d}(u, v)) \tag{6}$$

While the assignment might seem trivial, it is worth noting that progression rule and `minS2D`-lift and go hand in hand in the lower bound evolution of state nodes during the search progress:

The `minS2D`-lift indirectly introduces a key aspect of the evolution of lower bound of nodes $v \in V^{\text{map}}$. $\mathtt{d}(u, v^*) + \mathtt{LB}^-(v^*)$ with $v^* \in \sigma_u^*$ is imposing an upper bound on $\mathtt{LB}^-(u)$. Increasing $\mathtt{LB}^-(v')$ for any other successor $v' \notin \sigma_u^*$ will not help increasing $\mathtt{LB}^-(u)$ — at least not at the current time τ. We can also say that any $v^*\sigma_u^*$ together with the associated value $\mathtt{d}(u, v^*) + \mathtt{LB}^-(v^*)$ currently *limits* u and the associated value $\mathtt{LB}^-(u)$. On the other hand, the progression rule actively selects the a node $v^* \in \sigma_u^*$ for further exploration – if we are lucky, with the consequence of increasing $\mathtt{LB}^-(v^*)$. Symmetric relationships hold for the backward search.

Furthermore, at this point, we introduce the concept of $\widehat{\mathtt{LB}}(u, v)$ that serves as an incumbent value for $\mathtt{LB}(u, v)$ as long as not all successor nodes $v \in \sigma_u$ (or predecessor nodes $v \in \pi_u$ for backward search) have been evaluated, i.e. the minimum operator has not been completely applied yet. Hence, we will always have $\widehat{\mathtt{LB}}^-(u) \geq \mathtt{LB}^-(u)$.

5.5.3. Search Path Discontinuation

On top of that, we will apply *discontinuation-predicates* for the search paths. A discontinuation-predicate is a predicate of some state node u. If the last added node in the search path has such a predicate, further extension of the search path is stopped and a new search path is initiated.

Our first discontinuation-predicate will be *tightness-discontinuation* (`TD`). It is given by the following rule with u being the node added to the search path P most recently.

$$\mathtt{LB}^-(u) = \mathtt{UB}^-(u) \implies \mathtt{TD}(u)$$

Whenever this applies, any further extension of P from here would not provide any additional information. For the backward search direction, tightness-discontinuation is defined congruently:

$$\mathtt{LB}^+(u) = \mathtt{UB}^+(u) \implies \mathtt{TD}^-(u)$$

5.6. Search Algorithm

5.6.1. Outer Search Routine

Algorithm 1 shows the intentionally simplistic structure of our *outer* search routine. The outer search routine itself uses the *inner* search routine for establishing search path that is described in Section 5.6.2. Each iteration of the inner search routine yields a search path of variable length. The length of one search path is determined by the time.

Algorithm 1: Outer search routine

```
1 Function SolveTSPPD:
2     V^map ← HashMap  with key v ∈ V and values:
3               LB^−(v) and UB^−(v) (if searchForward)
4               or LB^+(v) and UB^+(v) (else)
5     while LB(0^+, 0^−) < UB(0^+, 0^−) ∧ time limit not reached do
6         if searchForward then
7             GenerateSearchPath⟨Forward⟩
8         else
9             GenerateSearchPath⟨Backward⟩
```

5.6.2. Inner Search Routine

To start with, we will define an initial, straightforward version of the routine generating search paths in 2. We will extended the inner search routine in Section 6.6 to improve runtime and memory requirements in the benchmarks. Furthermore, the initialization of the lower bound of states nodes added to V^{map} is not supposed to be fully defined here. The function $\textsc{InitLB}^-(v)$ or $\textsc{InitLB}^+(v)$ can represent any function that could yield a valid lower bound $\texttt{LB}^-(v)$ or $\texttt{LB}^+(v)$ for state node v, respectively. We will discuss this in Section 5.11.

5.6.3. Directionality

The search regime allows searching in reverse direction via the *backward-oriented search*. For this documentation, we will lay the focus on the forward-oriented-search, as it is easier to follow and behaves the same way as the backward-oriented-search, but when required with reversed parameters in function calls and logic w.r.t. processing nodes.

For choosing the search direction appropriately, there are multiple approaches. For the remainder of this report, we decided for the following: The decision is done based on the initial values lower bounds for cost-so-far or cost-remaining, respectively, for the successors of 0^+ and the predecessors for 0^-. We procede with the forward search, iff $\min_{v \in \sigma_{0^+}}(\texttt{d}(0^+, v) + \texttt{LB}^-(v))$ yields a value that is at least as large as $\min_{v \in \pi_{0^-}} \texttt{LB}^+(v) + \texttt{d}(v, 0^-)$.

5.7. Establishing $t^{\texttt{OPT}}$

A nearby strategy for retrieval of a possibly cheap (not necessarily optimal) solution to the TSPPD – a complete tour – after termination of the outer search in forward direction would be the following: Performing a single iteration of a variation of Algorithm 2 that always procedes with some node $v \in \arg\min_{v \in \sigma_u}(\texttt{UB}^+(u) + \texttt{d}(v, u))$ after the outer search has terminated. For this, it would only be required that at least one outer search iteration

Algorithm 2: Inner search routine (Forward)

1 **Function** GENERATESEARCHPATH⟨Forward⟩**:**
2 $P \leftarrow []$
3 $u \leftarrow 0^+$
4 P.PUSH(u)
5 **while** $\neg$TD(u) **do**
6 $\widehat{\texttt{LB}}^-(u) \leftarrow \infty$
7 **foreach** $v \in \sigma_u$ **do**
`/* Check if node has already been seen during search */`
8 **if** $v \notin V^{\text{map}}[l_v]$ **then**
9 $\texttt{LB}^-(v) \leftarrow$ INITLB$^-(v)$
10 $V^{\text{map}}[l_v]$.INSERT(v)
11 $\widehat{\texttt{LB}}^-(u) \leftarrow \min\{\widehat{\texttt{LB}}^-(u), \texttt{d}(u,v) + \texttt{LB}^-(v)\}$
12 $\texttt{UB}^-(u) \leftarrow \min\{\texttt{UB}^-(u), \texttt{d}(u,v) + \texttt{UB}^-(v)\}$
13 $\texttt{UB}^-(0^+) \leftarrow \min\{\texttt{UB}^-(0^+), \texttt{c}(P) + \texttt{UB}^-(u)\}$
14 $\texttt{LB}^-(u) \leftarrow \widehat{\texttt{LB}}^-(u)$
15 $\sigma_u^* \leftarrow \arg\min_{v \in \sigma_u}(\texttt{d}(u,v) + \texttt{LB}^-(v))$
16 $u \leftarrow$ some node $v^* \in \sigma_u^*$
17 P.PUSH(u)

has been performed. We already pointed out that this solution may not be optimal (in particular, when the outer search terminated due to the time limit).

One might assume the following for nodes $v \in T, c(T) = \texttt{UB}^-(0^+)$ that are part of a valid tour:

$$\texttt{UB}^+(v) + \texttt{UB}^-(v) = \texttt{UB}^-(0^+) \tag{7}$$

However, it is actually not even clear whether the tour resulting from this strategy would be of cost $\texttt{UB}(0^+, 0^-)$ or whether all nodes v part of such a tour will have a value $\texttt{UB}^-(v)$ less than ∞.

More precisely, $\texttt{UB}^-(0^+)$ is updated in every single iteration of the inner search routine as opposed to $\texttt{UB}^-(v)$. Also, $\texttt{UB}^+(v)$ is not being tracked by the forward search and hence we can only work with $c(P_v) \geq \texttt{UB}^+(v)$ being the cost of the search path $P_v = (0^+, \ldots, v)$ leading v in our reconstruction iteration of the inner search routine.

So, up to this point, only the following holds for certainty for all $v \in T$ with $c(T) = \texttt{UB}^-(0^+)$:

$$\texttt{c}(P_v) + \texttt{UB}^-(v) \geq \texttt{UB}^-(0^+) \tag{8}$$

On the other hand, there is at least one node $u \in T$ where equality holds:

$$\texttt{c}(P_u) + \texttt{UB}^-(u) = \texttt{UB}^-(0^+) \tag{9}$$

That is, because $\texttt{UB}^-(0^+)$ was assigned its final value in some inner search routine iteration in line 13 given the node $u \in T$ and the search path P. Saving a current snapshot

Algorithm 3: Inner search routine (Backward)

Function GENERATESEARCHPATH⟨Backward⟩**:**
- $P \leftarrow []$
- $u \leftarrow 0^-$
- P.PUSH(u)
- **while** $\neg$TD$^-(u)$ **do**
  - $\widehat{\texttt{LB}}^+(u) \leftarrow \infty$
  - **foreach** $v \in \pi_u$ **do**
    - **if** $v \notin V^{\text{map}}[l_v]$ **then**
      - $\texttt{LB}^+(v) \leftarrow$ INITLB$^+(v)$
      - $V^{\text{map}}[l_v]$.INSERT(v)
    - $\widehat{\texttt{LB}}^+(u) \leftarrow \min\{\widehat{\texttt{LB}}^-(u), \texttt{LB}^+(v) + \texttt{d}(v,u)\}$
    - $\texttt{UB}^+(u) \leftarrow \min\{\texttt{UB}^-(u), \texttt{UB}^+(v) + \texttt{d}(v,u)\}$
    - $\texttt{UB}^+(0^-) \leftarrow \min\{\texttt{UB}^+(0^-), \texttt{c}(P) + \texttt{UB}^+(u)\}$
  - $\texttt{LB}^+(u) \leftarrow \widehat{\texttt{LB}}^+(u)$
  - $\pi^*_u \leftarrow \arg\min_{v \in \pi_u}(\texttt{d}(v,u) + \texttt{LB}^+(v))$
  - $u \leftarrow$ some node $v^* \in \pi^*_u$
  - P.PUSH(u)

of the node u and P_u as u^{INC} and P_u^{INC} whenever $\texttt{UB}^-(0^+)$ is decreased takes only $O(n)$ additional space in total. With P_u we are already given the tour T up to node u and only have to reconstruct the remaining state nodes from u to 0^-. For this, we can use the function inner search routine in the modified form Algorithm 4 of proceding with nodes $v \in \arg\min_{v \in \pi_u}(\texttt{UB}^+(u) + \texttt{d}(v,u))$.

Algorithm 4: Reconstruction of t^{OPT}

Function RECONSTRUCTOPTIMALTOUR⟨Forward⟩**:**
- $t^{\text{OPT}} \leftarrow P_u^{\text{INC}}$
- $u \leftarrow u^{\text{INC}}$
- P.PUSH(u)
- **while** $\neg$TD(u) **do**
  - $u \leftarrow$ some node $\in \arg\min_{v \in \sigma_u}(\texttt{d}(u,v) + \texttt{UB}^-(v))$ P.PUSH(u)

5.8. Analysis

The analysis focuses on the forward-oriented search. All proofs can be applied symmetrically for the backward-oriented search with only the notation being alternated.

5.9. Proof of Termination and Runtime

Lemma 5.1. *Every iteration of Algorithm 2 tightens the bounds of exactly one additional state node u, resulting in $\mathtt{LB}^-(u) = \mathtt{UB}^-(u)$.*

Lemma 5.1 shows that any node v (a successor of u) that gets added to a search path with $\mathtt{TD}(v^{\blacksquare})$ subsequently applying will not be added to any search path again. Beyond that, it implies that no search path can be generated that has an already generated search path as a prefix.

Proof. The first implication of the Lemma is: at least one node will be tightened during an arbitrary iteration. Suppose there is some search path $P = (\ldots, u, v^{\blacksquare})$ that will not be extended after $v^{\blacksquare}$. This happens, iff $\mathtt{TD}(v^{\blacksquare})$ with $\mathtt{LB}^-(v^{\blacksquare}) = \mathtt{UB}^-(v^{\blacksquare})$ applied. In the iteration before, in line 14, $\mathtt{LB}^-(u)$ would have been increased in the following manner:

$$\mathtt{LB}^-(u) \leftarrow \mathtt{d}(u, v^{\blacksquare}) + \mathtt{LB}^-(v^{\blacksquare}) = \mathtt{d}(u, v) + \mathtt{UB}^-(v^{\blacksquare})$$

The corresponding upper bound $\mathtt{UB}^-(u)$, would have been decreased (via the update in line 12) to a value that is not larger than

$$\mathtt{d}(u, v) + \mathtt{UB}^-(v) = \mathtt{d}(u, v^{\blacksquare}) + \mathtt{LB}^-(u)$$

In turn $\mathtt{LB}^-(u) = \mathtt{c}^-(u) = \mathtt{UB}^-(u)$ – marks a discontinuation implication for a future inner search path involving u. Moreover, TD (currently) is the only measure able to stop an inner search iteration. In consequence, for at least one additional node, lower bound and upper bound will be aligned to the true remaining costs per iteration.

The second implication of the Lemma, *at most* one additional node will be tightened within an iteration, can be proven by contradiction:

Let us suppose some node t other than u will be tightened. Only nodes in the search path could possibly tightened by the algorithm. Hence, some node t occurring earlier than u in the search path would have been tightened, such that decreasing $\mathtt{UB}^-(t)$ or increasing $\mathtt{LB}^-(t)$ would have made them equal. This would assume:

$$\min_{u \in \sigma_t} \big(\mathtt{d}(t, u) + \mathtt{LB}^-(u)\big) = \min_{u \in \sigma_t} \big(\mathtt{d}(t, u) + \mathtt{UB}^-(u)\big)$$

Since $\mathtt{LB}^-(u) \leq \mathtt{UB}^-(u)$ for all state node u, at least one successor u of t must already be cost-remaining-tight, i.e. $\mathtt{LB}^-(u) = \mathtt{UB}^-(u)$. Such a successor would consequently be chosen by the progression rule and $\mathtt{TD}(u)$ would apply immediately afterwards. Hence, u can only be v and t can only be u. □

Lemma 5.1 and the number of state nodes $|V| = |\mathcal{L}| \cdot 3^{|\mathcal{R}|-1} + 2$ result in Lemma 5.2.

Lemma 5.2. *The outer search routine in Algorithm 1 terminates after $O(|\mathcal{L}| \cdot 3^{|\mathcal{R}|})$ iterations.*

For the proof of total runtime, we just need Lemma 5.2 as the final ingredient.

Lemma 5.3. *The inner search routine (Algorithm 2) has a runtime $O(n^2)$.*

Proof. The number of iterations of the while-loop is bounded by the maximum length of a search path, which is $|\mathcal{L}| = |\mathcal{L}|$. A single iteration is bounded by the number of locations (other than l_u), which is $|\mathcal{L}|$. □

Now, Lemma 5.2 combined with Lemma 5.3 gives us the worst-case runtime of Algorithm 1:

Theorem 5.1. *Algorithm 1 has a runtime* $\in O(n^3 \cdot 3^n)$.

5.10. Proof of Optimality

Lemma 5.4. *The outer search terminates having found the optimal solution value* $\mathtt{c}(t^{OPT})$*, when no time limit is set.*

The proof handles the complexity that search paths of different costs can update $UB^-(0^+)$ to different values while considering the same state node v that is already cost-remaining-tight with $\mathtt{c}^-(v)$ already known. It shows that eventually, a search path with $\mathtt{c}(P) + \mathtt{c}^-(v)$

Proof. Suppose there is some search path $P = (\ldots, u, v, w)$ not being extendend beyond w because of $\mathtt{TD}(w)$, which implies $\mathtt{LB}^-(v) = \mathtt{c}^-(v) = \mathtt{UB}^-(v)$.

Let $P' = (\ldots, u')$, $u' \in \pi_v$ be another search path that considers v as the minimizing successor of u', i.e. $v \in \sigma^*_{u'}$:

$$v \in \sigma^*_{u'} \iff v \in \arg\min_{v' \in \sigma_{u'}} (\mathtt{d}(u', v') + \mathtt{LB}^-(v'))$$

The broader implication of this minimizing successor is that there is no tour from u' to 0^- over another successor v' with smaller *definite* costs:

$$\begin{aligned} \implies \forall v' \in \sigma_{u'} \colon \mathtt{d}(u', v') + \mathtt{c}^-(v') &\geq \mathtt{d}(u', v') + \mathtt{LB}^-(v') && (10)\\ &\geq \mathtt{d}(u', v) + \mathtt{LB}^-(v) && (11)\\ &= \mathtt{d}(u', v) + \mathtt{UB}^-(v) && (12) \end{aligned}$$

This particulary applies, if at some point, the role of u' is taken by the depot departure 0^+. Hence, finding the optimal complete tour is guaranteed. □

5.11. Initialization of Lower Bounds for State Nodes

For simplicity, this chapter primarily focuses on the computation of initial lower bounds associated with the forward-oriented search – namely $\mathtt{LB}^-(u)$ – for any state node u. All described procedures can be applied symmetrically when the search orientation is backward. Since our state space-graph has special characteristics in the left-most and right-most partition V_d with $d \leq 4$ or $d \geq |\mathcal{L}| - 3$, it was decided to completely separate the lower and upper bound computation for these cases. These edge cases have the benefit that corresponding upper bounds can be set right away as well.

5.11.1. Edge Cases with $d \geq |\mathcal{L}| - 3, d \leq 4$

For $d = |\mathcal{L}|$, we have remaining costs of $\texttt{LB}^-(0^-) = \texttt{c}(0^+, 0^-) = \texttt{UB}^-(0^+) = 0$.

The situation is similar at $d = |\mathcal{L}| - 1$, except that we are not at the depot arrival 0^- yet. Any node $u \in V^{|\mathcal{L}|-1}$ will thus only have one outgoing arc $a \in \sigma_v$ to the depot arrival node 0^- and $\sigma_u = \{a\}$. So $\texttt{LB}^-(u) = \texttt{c}^-(u) = \texttt{LB}^-(u) = \texttt{d}(l_u, 0^-)$.

When $d = |\mathcal{L}| - 2$, we have exactly two unvisited locations: 0^- and some l_v with $v \in \sigma_u$ and $v \in \pi_{0^-}$. However, there still exists only one partial tour serving the residual locations, namely $t = (l_u, l_v, 0^-)$. Thus, we can fix $\texttt{LB}^-(u) = \texttt{c}^-(v) = \texttt{UB}^-(u) = \texttt{d}(l_u, l_v) + \texttt{d}(l_v, 0^-)$ in this case as well.

When $d = |\mathcal{L}| - 3$, , we have exactly three unvisited locations: 0^-, some l_v with $v \in \sigma_u$ and some l_w with $w \in \sigma_v, w \in \pi_{0^-}$. There are at most two partial tours from u to 0^- in such a case – precedence constraints could even only allow one partial tour (when v is a pickup node). Hence, we will directly set $\texttt{c}^-(v)$ to the cost of the cheapest of these at most two possible partial tours.

Symmetrically, the previous cases apply w.r.t to $\texttt{LB}^+(v)$. Here, we can set $\texttt{UB}^+(v) = \texttt{LB}^+(v)$ for all nodes $v \in V^d$ with $d \leq 4$.

5.11.2. Lower Bound for the General Case $4 < d < |\mathcal{L}| - 3$

Research Background For the cost of the optimal tour in the traveling salesman problem, the cost of a minimum spanning tree (MST) is a frequently used lower bound. Held and Karp [HK70] were able to prove a slightly tighter lower bound: Consider a graph $G = (V, E)$, then compute an MST T for the Graph $G' = G \setminus \{v\}$ with an arbitrarily chosen vertex $v \in V(G)$. Then connect v with the two closest nodes $v \in \arg\min \texttt{d}(u, v), w \in \arg\min_{v \neq w} \texttt{d}(u, w)$ and compute $cost(T) + \texttt{d}(u, v) + \texttt{d}(u, w)$. You can see this results in a subgraph with higher costs than a simple MST, as:

1. The degree of v inside this subgraph is forced to be 2 (in an MST, it could be as large as $|V(G)| - 1$)
2. It contains an additional edge compared to a tree

Adaption for State-Space-Search on the TSPPD For the following, we will refer to the cost of the MST for the graph induced by the locations L as $c^{\text{MST}}(L)$

1. When we consider a node $v \in \sigma_u$ as the successor to our current node $u \in V_d, d < |\mathcal{L}| - 3$ in the search path, we first compute an MST of $\overline{L_v} \setminus \{0^-\}$, thus isolating the locations l_v and 0^- (or l_v and 0^+ for backward search).
2. We then connect each l_v and 0^- (or 0^+) with their closest adjacent location inside the MST $\overline{L_v} \setminus \{0^-\}$ and return the cost of the two additional edges together with the cost of the previously computed MST. However, an additional constraints is respected here: We connect v only to locations that produce a valid successor nodes to v, e.g. we do not allow connecting l_v to an exit location of a yet unserved request. Similarly, we allow 0^- to only connect to delivery locations in general (and for backward search 0^+ to only connect to pickup locations).

Summing this up, we can use the following relationship for initializing lower bounds $\texttt{LB}^-(v)$ of state nodes u.

Theorem 5.2.

$$\forall u \in V^d, d < |\mathcal{L}| - 3 : \texttt{c}^-(u) \geq \min_{v \in \sigma_u} \texttt{d}(u,v) + c^{MST}(\overline{L_u} \setminus \{0^-\}) + \min_{w \in \pi_{0^-}} \texttt{d}(w, 0^-) \quad (13)$$

Proof. We can show this by decomposition of the accumulated costs of the equation:

1. The current location l_v is succeeded by some location $l^\sigma \in \overline{L_v} \setminus \{0^-\}$ in any tour from v to 0^- (and in particular, the optimal tour) with $\texttt{d}(l_v, l^\sigma) > \min_{w \in \sigma(v)} \texttt{d}(l_v, l_w)$.
2. It is trivial to show that any minimum spanning tree of $\overline{L_v} \setminus \{0^-\}$ is a lower bound for $\texttt{c}(w,t)$ using any $w \in \sigma_v, t \in \pi_{0^-}$ — since such an MST uses exactly the cheapest $|\mathcal{L}| - d + 1$ edges available in this very graph.
3. The depot arrival location 0^- is preceded by some location $l^\pi \in \overline{L_v} \setminus \{0^-\}$ in any tour from v to 0^- with $\texttt{d}(l^\pi, 0^-) > \min_{w \in \pi(0^-)} \texttt{d}(l_w, 0^-)$.

The MST serves as a necessary intermediate component, as the current location l_v and the depot location 0^- can only be directly connected, if $\overline{L_{v'}} \setminus \{0^-\} = \emptyset$ is empty and thus $d = |\mathcal{L}| - 2$ (which was already handled in 5.11.1). □

Using the valid successors $\sigma_{v'}$ and predecessors π_{0^-}, it also implicitly assured that the edge connecting v' to the MST is valid w.r.t to the problem-induced precedence constraints. Likewise, 0^- will only be connected to some delivery location in the MST.

5.12. Runtime Impact

From an algorithm analysis perspective, a convenient property of our more elaborate lower bound initialization in comparison to initialization to 0 as in Algorithm 2 is that it does not asymptotically influence our assumed worst-case runtime at all. For each state node v, we are facing $0(n^2)$ time for computing a minimum spanning tree and $O(n^2)$ time for "connecting" it to the locations l_v and 0^- (or 0^+ for the backward search). In turn, the (theoretic) worst-case of having to add all of the $|\mathcal{L}| \cdot 3^{n-1}$ nodes into V^{map} contributes to our runtime with an additional $O(n^3 \cdot 3^n)$. This is exactly congruent with the total runtime of the outer search routine we have determined in Section 5.9.

6. Improving the State Space Search

Again, for simplicity, the explanations will cover the behavior forward-oriented search with the backward-oriented search behavior being equivalent with alternated notation.

Breaking-predicates We will now introduce a concept that can help us shorten the runtime of Algorithm 2. A *breaking-predicate* is a predicate that leads to stopping the foreach-loop in the inner search routine early if true. Every relationship here also applies symmetrically when the search orientation is backward. In fact, Section 6.6.2 will show how these predicates are defined for the backward-oriented search.

Sorted location-neighborhood list As a precondition to the following breaking-predicates, we need a sorted list of neighboring non-depot locations for each location by distance. We call this *location-neighborhood list* $\overline{\mathcal{L}}(l)$ [2], $l_i \neq l \in \mathcal{L}$ with $\mathtt{d}(l, l_i) \leq \mathtt{d}(l, l_{i+1}) \forall i \leq |\mathcal{L}|$. It can be computed readily in advance for all locations individually with a runtime of $O(n^2 \log n)$ using standard search algorithms like QUICKSORT or MERGESORT.

6.1. Preliminary Lower Bounds for Successors of the Most Recent State Node in Search Path

For the search algorithm to be shown later, we do not only use dedicated lower bounds for single state nodes. Based on the lower bound proven before, it is possible to make use of a *preliminary lower bound* for getting from an arbitrary $v \in \sigma_u$ to 0^-, which we will call $\mathtt{LB}^-(\overline{L_u})$.

$$\mathtt{LB}^-(\overline{L_u}) = c^{\mathrm{MST}}(\overline{L_u} \setminus \{0^-\}) + \min_{w \in \pi_{0^-}} \mathtt{d}(l_w, 0^-) \tag{14}$$

Likewise, for the backward-directed search, we will use $\mathtt{LB}(0^-, \pi_u)$ for the lower bound of getting from 0^+ to u via any of its predecessors. This lower bound is generally looser than the specific lower bounds for each successor node computed separately, but it can be used to eliminate some successors from consideration.

The takeaway is Corollary 6.1

Corollary 6.1.

$$\forall v \in \sigma_u, u \in V\colon \mathtt{c}^-(v) \geq \mathtt{LB}^-(v) \geq \mathtt{LB}^-(\overline{L_u}) \tag{15}$$

Finally, we can reuse the preliminary lower bound $\mathtt{LB}^-(\overline{L_u})$ for the computation of lower bounds of state nodes u' that have $L_v = L_{v'}$. On the other hand, when we initiate $\mathtt{LB}^-(u)$ for a state node u, we can reuse a part of the initialization equation. This holds, because the preliminary lower bound $\mathtt{LB}^-(\overline{L_u})$ overlaps with this initialization:

$$\mathtt{LB}^-(u) \leftarrow \min_{v \in \sigma_u} \mathtt{d}(u, v) + c^{\mathrm{MST}}(\overline{L_u} \setminus \{0^-\}) + \min_{w \in \pi_{0^-}} \mathtt{d}(w, 0^-) = \min_{v \in \sigma_u} \mathtt{d}(u, v) + \mathtt{LB}^-(\overline{L_u}) \tag{16}$$

Consequently, it may prove beneficial to cache these preliminary lower bounds separately from the state nodes which are cached in V^{map}. As opposed to V^{map}, this data structure uses location subsets $L \subset \mathcal{L}$ as keys, but still rational numbers as values.

Tightening of preliminary lower bounds Whenever we know the actual minimum lower bound of all successors of u $\min_{v \in \sigma_u} \mathtt{LB}^-(v)$, we can increase $\mathtt{LB}^-(\overline{L_u})$ to this value. In particular, this applies, when we have evaluated all successor nodes $v \in \sigma_u$ in an inner search iteration.

Correspondingly, the cost of any cheapeast arc to a valid successor node u together with a lower bound for a cheapest tour covering all locations $\overline{L_v}$ is still a valid lower bound for $\mathtt{c}^-(u)$.

[2] Not to be confused with $\overline{\mathcal{L}_v}$ – the unvisited locations at an state node v

Lemma 6.1.

$$\forall v' \in V\colon \mathtt{c}^-(u) \geq \min_{v\in\sigma_u, v'\in V, \overline{L_v}=\overline{L'_v}} (\mathtt{d}(u,v) + \mathtt{LB}^-(v')) \tag{17}$$

This can be used to improve the effectiveness of breaking-predicates in future outer search iterations that involve state nodes u' with $L'_u = L_u$ having the same set of successors as u (see Lemma 4.1).

6.2. Breaking based on $\widehat{\mathtt{LB}}^-(u)$ (MSB)

To take advantage of the preliminary lower bound, we define a breaking-predicate called *Minimizing-successor-breaking*, which is denoted by $\mathtt{MSB}(u,v)$ with u being the last node in the search path and v beeing a successor of u being evaluated currently.

Let us now recall our progression rule from Section 5.5.1. It consists in finding $\sigma^*_u \in \arg\min_{v\in\sigma_u}(\mathtt{d}(u,v) + \mathtt{LB}^-(v))$ to be the next state node in the search path following u.

Definition 6.1. *$\mathit{MSB}(u,v)$ is a breaking-predicate referencing the most recently added node to the search path u and some successor v of u. It is implied by the distance from u to v and the preliminary lower bound of u's successors being already larger than the currently best known value for getting from u to 0^- via some successor.*

$$\forall v \in \sigma_u, u = P_{|P|}\colon \mathtt{d}(u,v) + \mathtt{LB}^-(\overline{L_u}) \geq \widehat{\mathtt{LB}}^-(u) \implies \mathit{MSB}(u,v) \tag{18}$$

Corollary 6.2. *Let*

$$v^\bullet \in \arg\min\{\mathtt{d}(u,v) | v \in \sigma_u, \mathit{MSB}(u,v)\} \tag{19}$$

If MSB applies for a successor of some most recently added node $u = P_{|P|}$ in the search path, either v is not one of the successors minimizing the progression rule or there is more than one such successor.

$$\forall v \in \sigma_u, \mathit{MSB}(u,v)\colon v \notin \sigma^*_u \vee |\sigma^*_u| > 1 \tag{20}$$

An equivalent statement: Whenever MSB applies to a successor of some node u in the search path and we are scanning the successors in the order of their distance to u, we have already found at least one member of σ^*_u.

Proof. Foremost, we can state that the set $\arg\min\{\mathtt{d}(u,v) + \mathtt{LB}^-(\overline{L_u}) | v \in \sigma_u, \mathtt{MSB}(u,v)\}$ is in fact equal to $\arg\min\{\mathtt{d}(u,v) | v \in \sigma_u, \mathtt{MSB}(u,v)\}$, because $\mathtt{LB}^-(\overline{L_u})$ is a general preliminary lower bound to all sucessors of u.

Combining Corollary 6.1 with the definition of $\mathtt{MSB}(u,v)$, we can infer the following for all v that are at least as far away from u as $v^\bullet$:

$$\begin{aligned}
\mathtt{d}(u,v) + \mathtt{LB}^-(v) &\geq \mathtt{d}(u,v) + \mathtt{LB}^-(\overline{L_u}) && (21)\\
&\geq \mathtt{d}(u,v^\bullet) + \mathtt{LB}^-(\overline{L_u}) && (22)\\
&\geq \widehat{\mathtt{LB}}^-(u) \geq \mathtt{LB}^-(u) && (23)
\end{aligned}$$

So either $\mathtt{d}(u,v) + \mathtt{LB}^-(v)$ is strictly larger than $\widehat{\mathtt{LB}}^-(u)$ or we have more than one successor minimizing the progression rule. □

Now, scanning the locations l_v that are possible successor locations to u in the order of their distance from l_u, we can finish the foreach-loop of the inner search routine as soon as the condition holds for some node $v^\bullet$. Given that all successor nodes $v \in \sigma_u$ evaluated later have a larger distance $\mathtt{d}(u,v) \geq \mathtt{d}(u,v^\bullet)$ from u, the rule would also apply to all these later evaluated successor nodes.

In other words, with

$$v^\bullet \in \arg\min\{\mathtt{d}(u,v) + \mathtt{LB}^-(\overline{L_u})|v \in \sigma_u, \mathtt{MSB}(u,v)\} \tag{24}$$

$$= \arg\min\{\mathtt{d}(u,v))|v \in \sigma_u, \mathtt{MSB}(u,v)\} \tag{25}$$

we ensure that for any successor v^* we have found that currently minimizes the progression among all successors of u evaluated in this inner search iteration, i.e.

$$v^* \in \arg\min\{\mathtt{d}(u,v) + \mathtt{LB}^-(v)|v \in \sigma_u, \mathtt{d}(u,v) \leq \mathtt{d}(u,v^\bullet)\} \tag{26}$$

will also minimize the progression rule among all (including non-evaluated) successors. Hence, can will stop the evaluation of successors at this point.

Besides potentially saving compute time, this predicate may also reduce memory consumption because we do not even have to insert the affected nodes into V^{map} — at least not during this iteration of the inner search routine.

6.3. Improving the Discontinuation-Predicate

Before introducing another breaking-predicate, we will replace tightness-discontinuation (TD) with a stricter discontinuation-predicate.

Definition 6.2. *Complete-tour-discontinuation $\mathit{CTD}(u)$ is a predicate implied by the cost of the current search path and the lower bound of the costs of getting from the most recent node $u = P_{|P|}$ in the search path to 0^- being larger than the current upper bound for the costs of a complete tour.*

$$\mathtt{c}(P) + \mathit{LB}^-(u) \geq \mathit{UB}^-(0^+) \implies \mathit{CTD}(u)$$

The predicate is motivated by our goal of finding the optimal solution value t^{OPT} as early as possible. When the corresponding condition is true, it is clear that the current search path cannot be extended to a path that improves the current value of $\mathtt{UB}^-(0^+)$. In other words, unless t^{OPT} has already been found and we are currently "following" t^{OPT}, the current search path cannot be a prefix to t^{OPT}, whenever CTD applies.

Let us also recall our goal of aligning $\mathtt{LB}^-(0^+)$ with $\mathtt{UB}^-(0^+)$ as fast as possible. In regards to this goal, we can make the following proposition Theorem 6.1.

Theorem 6.1. *Let $P = (0^+, v_1, \ldots, v_k)$ be a search path with $\mathit{CTD}(v_k)$. Extending P beyond v_k will not enable an additional increase of $\mathit{LB}^-(0^+)$ until a path $P' = (0^+, v'_1, \ldots, v'_{k-1}, v_k)$ with $\mathtt{c}(P') + \mathit{LB}(v_k) < \mathit{UB}^-(0^+)$ is found.*

This following proof is very comprehensive and relies heavily on the concept of *limitation* introduced in Section 5.5.1. It is also an inductive proof: if case *Item* $2(a)ii$ applies, you can apply the proof to an alternative search path P'. The intuition is: We are looking at an *arbitrary* search path P' generated in a future iteration: If P' is a prefix of P, it will „propagate“ the lower bound leading to the discontinuation-predicate being true. If it generates a different search path P' still including v_k, it depends on the exact costs of it. If P' does not even v_k, v_k is no limiting successor and an additional increase of its remaining-cost is not beneficial anyway.

Proof. We identify two possible scenarios:

1. Base case: Suppose that in some future iteration the search path $P' = P \setminus \{v_k\}$ (i.e. a prefix of P) has emerged and v_k would still limit $\texttt{LB}^-(v_{k-1})$. Then $\texttt{c}(P') = \texttt{c}(P) - \texttt{d}(v_{k-1}, v_k)$ and $\texttt{LB}^-(v_{k-1}) = \texttt{d}(v_{k-1}, v_k) + \texttt{LB}^-(v_k)$. Then

   $$\texttt{c}(P') + \texttt{LB}(v_{k-1}) = \texttt{c}(P) - \texttt{d}(v_{k-1}, v_k) + \texttt{d}(v_{k-1}, v_k) + \texttt{LB}^-(v_k) = \texttt{c}(P) + \texttt{LB}^-(v_k)$$

   In consequence, $\texttt{CTD}(v_{k-1})$ will now apply. This specific effect could now propagate backwards to predecessors of v_{k-1}. Only if it propagates all the way back to 0^+, it could possible lead to a direct increase of $\texttt{LB}^-(0^+)$. However, if it propagated to 0^+, that would mean

   $$\texttt{c}(P) + \texttt{LB}^-(0^+) \geq \texttt{UB}^-(0^+) \equiv \texttt{LB}^-(0^+) \geq \texttt{UB}^-(0^+)$$

   and our outer search would be finished.

2. It could be the case that the search path P' generated in a future iteration diverges from P at some index $i < k - 1$. This happens if there is a different minimizing successors $v'_{i+1} \in \sigma^*_{v_i}, v'_{i+1} \neq v_{i+1}$ in such a future iteration. That implies v_{i+1} (that is not an element of this future search path P') does not limit $\texttt{LB}^-(v_i)$ anymore, because the progression rule always chooses a limiting successor in $\sigma^*_{v_i}$. Hence, a further increase (in the original iteration generating P) of some $\texttt{LB}^-(v_{i+1})$ does not provide a foundation to further increase $\texttt{LB}^-(v_i)$ as long as $v_{i+1} \notin \sigma^*_{v_i}$.

   For the remaining elements of P' following the diverging node v_i, there are now two subcases:

   a) The original v_k is included in P', such that is is of the form $(0^+, v'_1, \ldots, v'_{k-1}, v_k)$ despite divergence from P at an earlier index $i < k$. This can happen, if the search paths “meet again” at v_k, but followed a different partial tour in the meantime (also see Corollary 5.1). While P' and P can have very different costs, the only important distinction is the following:

      i. If it is *cheap enough*, i.e.:

         $$\texttt{c}(P') + \texttt{LB}(v_k) < \texttt{UB}^-(0^+)$$

         $\texttt{CTD}(v_k)$ will not apply anymore and the “until”-criterion of our proof has arised.

ii. If it is not cheap enough, we can inductively "restart the proof", now using the new search path P' as a reference.

b) v_k is not part of such a future search path P': Then, having increased $\texttt{LB}^-(v_k)$ by having extended the original search path P regardless of $\texttt{CTD}(v_k)$ being true, could not possibly help increasing any $\texttt{LB}^-(v_i)$ for a node $v_i \in P', i < k-1$.

□

You can see that this new discontinuation-predicate is implied by TD, whenever tightness-discontinuation applies, complete-tour-discontinuation applies as well.

$$\texttt{LB}^-(u) \geq \texttt{UB}^-(u) \tag{27}$$

$$\texttt{c}(P) + \texttt{LB}^-(u) \geq \texttt{c}(P) + \texttt{UB}^-(u) \geq \texttt{UB}^-(0^+) \tag{28}$$

For the backward direction, $\texttt{CTD}^-$ is defined as follows:

$$\texttt{CTD}^-(u) \leftarrow \texttt{LB}^+(u) + \texttt{UB}^-(u) \geq \texttt{UB}^-(0^+) \tag{29}$$

6.4. Breaking-Predicate Based on $\texttt{UB}^-(0^+)$ (CTB)

We now introduce a breaking-predicate based on the new discontinuation-predicate, which we will refer to as complete-tour-breaking-predicate (CTB).

Definition 6.3. *Complete-tour-discontinuation $\mathit{CTB}(u, v)$ is a predicate referencing the most recently added node to the search path $u = P_{|P|}$ and some successor $v \in \sigma_u$ of u. It is implied, if the cost of the search path together with the distance from u to v and the preliminary lower bound of u's successors is larger than the current upper bound for a complete tour:*

$$\texttt{c}(P) + \mathit{d}(u, v) + \mathit{LB}^-(\overline{L_u}) \geq \mathit{UB}^-(0^+) \implies \mathit{CTB}(u, v) \tag{30}$$

Note that if we evaluate $\texttt{MSB}(u, v)$ before $\texttt{CTB}(u, v)$ $\texttt{MSB}(u, v)$ was not satisfied, the value retrieved by minimum operator is lower bounded by:

$$\texttt{LB}^-(u) \geq \texttt{d}(u, v) + \texttt{LB}^-(\overline{L_u}) \tag{31}$$

Corollary 6.3 is related to Corollary 6.2 while residing in the context of our new breaking-predicate CTB. However, it additionally incorporates a possible discontinuation via CTD.

Corollary 6.3. *Let $v^\bullet$ be the successor node with minimum location distance to some node $u = P_{|P|}$ in the search path where the breaking-predicate CTB applies:*

$$v^\bullet \in \arg\min\{\mathit{d}(u, v) | v \in \sigma_u, \mathit{CTB}(u, v)\} \tag{32}$$

If CTB applies for a successor $v^\bullet$ in the search path, then either $v^\bullet$ does not minimize the progression rule ($v^\bullet \notin \sigma_u^*$) or the discontinuation-predicate CTD will apply for all successors minimizing the progression rule:

$$\forall v \in \sigma_u, \mathit{CTB}(u, v) \colon v \notin \sigma_u^* \vee \quad \forall v^* \in \sigma_u^* \colon \mathit{CTD}(v^*) \tag{33}$$

Proof. In the simplest case we have already considered a node $v \in \sigma_u^*$ and we already know the true minimizing successor:

$$\mathtt{LB}^-(u) \geq \widehat{\mathtt{LB}}^-(u) \overset{v \in \sigma_u^* \text{ seen}}{=} \mathtt{LB}^-(u)$$

If such a v has not been considered so far, we know that the not yet computed $\mathtt{LB}^-(v)$ must be at least $\mathtt{d}(u, v^\bullet) + \mathtt{LB}^-(\overline{L_u})$, as all successors v' considered after $v^\bullet$ have $\mathtt{d}(u, v') \geq \mathtt{d}(u, v^\bullet)$. So, the preliminary lower bound $\mathtt{LB}(\sigma_u)$ applies to them as well. Hence, in such a scenario, we can infer

$$\mathtt{LB}^-(u) \overset{\text{no } v \in \sigma_u^* \text{ seen}}{\geq} \mathtt{d}(u, v^\bullet) + \mathtt{LB}^-(\overline{L_u})$$

□

The main takeaway of this is that, if we scan the successors of some rear node in the search path u in the order of their distances to u, we can finish evaluating successor nodes as soons as one successor v of u is found where $\mathtt{CTB}(u, v)$ resolves to true.

Another takeaway is that if the discontinuation predicate does not apply for $v^* \in \sigma_u^*$, then the minimizing will be d

For the backward direction, $\mathtt{CTB}^-$ is defined as follows:

$$\mathtt{CTB}^-(u, v) \leftarrow \mathtt{LB}(0^+, \pi_u) + \mathtt{d}(v, u) + \mathtt{c}(P) \geq \mathtt{UB}^-(0^+) \tag{34}$$

6.5. On the Computation of Upper Bounds

Let's recapitulate from Section 5.11 that particular subsets of nodes are sufficiently close to the depot nodes 0^+ and 0^- enabling to close the gap of one cost property immediately. This is the case for $\mathtt{UB}^+(v)$ for nodes $v \in V^d$ with $d \leq 3$ and for $\mathtt{UB}^-(v)$ for nodes $v \in V^d$ with $d \geq |\mathcal{L}| - 3$ Thus $\mathtt{UB}^-(v)$ can be immediately set to the previously computed value for $\mathtt{LB}^-(v)$.

Apart from that, the search algorithm will always follow feasible tours and thus for the forward direction, when currently at state node u and evaluating possible successors $v \in \sigma_u$, we can potentially reduce $\mathtt{UB}^+(v)$ to $\mathtt{UB}^+(u) + \mathtt{d}(l_u, l_v)$. On the other hand $\mathtt{UB}^-(u)$ can potentially be reduced to $\mathtt{d}(l_u, l_v) + \mathtt{UB}^-(v)$.

Our search algorithm pragmatically relies on this mechanism of updating upper bounds on the fly during search. It would be possible to "propagate" reductions of $\mathtt{UB}^+(v)$ to successor nodes of $w \in \sigma_v$ by subtracting $\mathtt{d}(l_v, l_w)$ and to the same to the successors of w. But applying that to all successors recursively would require exponential runtime. The same goes for "propagating" reductions of $\mathtt{UB}^-(u)$. One feasible solution would be to choose selected successors, e.g. only one particular $w \in \sigma_v$ with minimal $\mathtt{UB}^-(w)$ and then one particular successor of $t \in \sigma_w$ with minimal $\mathtt{UB}^-(t)$. This may be able to decrease the time to find the optimal tour, but will not help much in closing the gap between $\mathtt{LB}^-(0^+)$ and $\mathtt{UB}^-(0^+)$.

Finally, what is important to be aware of is that any upper bound of the form $\mathtt{UB}^-(v)$ is always associated with (at least) one tour from v to 0^- with exactly cost $\mathtt{UB}^-(v)$.

6.6. Improved Search Routine

6.6.1. Forward-Oriented Search

Algorithm 7 shows the final version of our inner search routine in forward orientation with all the previously defined breaking-predicates. To keep it short, the informal condition $l \in L$ *is valid successor location for* $u \in V$ was used that excludes possible delivery locations $i^- \in \overline{\mathcal{L}}(l_u)$ that correspond to yet unvisited pickup locations i^+, i.e. $i^+ \notin L_v$.

Algorithm 5: Improved inner search routine in forward orientation

Function GENERATESEARCHPATH⟨Forward⟩**:**
- $P \leftarrow []$
- $u \leftarrow 0^+$
- P.PUSH(u)
- **while** $\neg$CTD(u) **do**
  - $\widehat{\mathtt{LB}}^-(u) \leftarrow \infty$
  - **foreach** $l \in \overline{\mathcal{L}}(l_u)$ **do**
    - **if** l not a valid successor location for u **then**
      - **continue**
    - $v \leftarrow (l, L_u \cup \{l\})$
    - **if** MSB(u, v) **then**
      - **break**
    - **if** CTB(u, v) **then**
      - $\widehat{\mathtt{LB}}^-(u) \leftarrow \min\{\widehat{\mathtt{LB}}^-(u), \mathtt{d}(u,v) + \mathtt{LB}^-(\overline{L_u})\}$
      - **break**
    - **if** $v \notin V^{\text{map}}[l]$ **then**
      - /* Either initiates or reuses $\mathtt{LB}^-(\overline{L_v})$ */
      - $\mathtt{LB}^-(v) \leftarrow$ INITLB$^-(v)$
      - $V^{\text{map}}[l_v]$.INSERT(v)
    - $\widehat{\mathtt{LB}}^-(u) \leftarrow \min\{\widehat{\mathtt{LB}}^-(u), \mathtt{d}(u,v) + \mathtt{LB}^-(v)\}$
    - $\mathtt{UB}^-(u) \leftarrow \min\{\mathtt{UB}^-(u), \mathtt{d}(u,v) + \mathtt{UB}^-(v)\}$
    - $\mathtt{UB}^-(0^+) \leftarrow \min\{\mathtt{UB}^-(0^+), \mathtt{c}(P) + \mathtt{UB}^-(u)\}$
  - **if** all successors of u checked **then**
    - $\mathtt{LB}^-(\overline{L_u}) \leftarrow \max\{\mathtt{LB}^-(\overline{L_u}), \min_{v \in \sigma_u} \mathtt{LB}^-(v)\}$
  - $\mathtt{LB}^-(u) \leftarrow \widehat{\mathtt{LB}}^-(u)$
  - $u \leftarrow$ some node $v^* \in \sigma_u^*$
  - P.PUSH(u)

6.6.2. Backward-Oriented Search

To demonstrate the behavior of the modifications in backward-oriented search, we will once more define the adapted routine for the backward-oriented search in Algorithm 6.

As opposed to the forward-oriented search, the search path is in the form $(0^-, l_1, .., l_v)$.

Algorithm 6: Improved inner search routine in backward orientation

Function GENERATESEARCHPATH⟨Backward⟩**:**
  $P \leftarrow []$
  $u \leftarrow 0^-$
  P.PUSH(u)
  while $\neg$CTD$^-$ **do**
    $\widehat{\texttt{LB}}^+(u) \leftarrow \infty$
    foreach $l \in \mathcal{L}(l_u)$ **do**
      if l not a valid predecessor location for u **then**
        continue
      $v \leftarrow (l, L_u \cup \{l\})$
      if MSB$^-(u, v)$ **then**
        break
      if CTB$^-(u, v)$ **then**
        $\widehat{\texttt{LB}}^+(u) \leftarrow \min\{\widehat{\texttt{LB}}^+(u), \texttt{d}(u, v) + \texttt{LB}^+(L_v)\}$
        break
      if $v \notin V^{\text{map}}[l_v]$ **then**
        /* Either initiates or reuses LB$^+(L_v)$ */
        LB$^+(v)$, LB$^+(L_v) \leftarrow$ INITLB$^+(v)$
        UB$^+(v) \leftarrow$ LB$^+(v)$ for $v \in V^d, d \leq 4$
        $V^{\text{map}}[l_v]$.INSERT(v)
      $\widehat{\texttt{LB}}^+(u) \leftarrow \min\{\widehat{\texttt{LB}}^-(u), \texttt{d}(u, v) + \texttt{LB}^+(v)\}$
      UB$^+(u) \leftarrow \min\{$UB$^-(u), \texttt{d}(u, v) + \texttt{UB}^+(v)\}$
      UB$^+(0^-) \leftarrow \min\{UB^+(0^-), \texttt{c}(P) + \texttt{UB}^+(u)\}$
    if all successors of u checked **then**
      LB$^+(L_v) \leftarrow \max\{\texttt{LB}^+(L_v), \min_{u \in \pi_v} \texttt{LB}^+(u)\}$
    LB$^+(u) \leftarrow \widehat{\texttt{LB}}^+(u)$
    $u \leftarrow$ some node $v^* \in \pi_u^*$
    P.PUSH(u)

6.7. Impact of Predicates on Termination of the Search Routine

We now focus on another suspicion that might arise: Will using all the suggested breaking-predicates still guarantee termination of our algorithm as well as finding the optimal solution? In general, termination could be questioned if an infinite loop cannot be ruled out. In the outer search routine Algorithm 1, an infinite loop could (hypothetically) occur, when the inner search routine always travels the same search path and finishes at the same sate node $u^\bullet$ where an breaking-predicate is applied ruling out $\sigma_u^* \in u^\bullet$.

Given coherent lower and upper bounds for all nodes, finding the optimal solution would be in doubt, if at the time of termination of the outer search routine at least one of the following applies:

1. not all all state nodes $v \in t^{\mathrm{OPT}}$ have been explored – exception: state nodes $w \in V^d, d \geq |\mathcal{L}| - 3$ that do not have to be explored because of the fixation of upper bounds of predecessors see Section 5.11
2. the values of $\texttt{UB}(v_i, v_j)$ for *all* nodes $v_i, v_j \in t^{\mathrm{OPT}}, j > i$ would be strictly larger than $\texttt{LB}(v_i, v_j)$

To answer this question, we will consider both of the suggested breaking-predicates:

6.7.1. Breaking Based on $\texttt{LB}^-(u)$ (MSB)

Lemma 6.2. SolveTSPPD *with minimizing-successor-breaking (MSB) terminates having found the optimal solution* t^{OPT}.

Proof. This rule applies according to its very definition iff

$$\texttt{d}(u, v^\bullet) + \texttt{LB}^-(\overline{L_u}) \geq \widehat{\texttt{LB}}^-(u) \tag{35}$$

for some $v^\bullet \in \sigma_u$. Starting from the most basic relationship between lower and upper bounds of nodes, we have for all $v^* \in \sigma_u^*$:

$$\begin{aligned} \texttt{d}(u, v^\bullet) + \texttt{UB}^-(v^\bullet) &\geq \texttt{d}(u, v^\bullet) + \texttt{LB}^-(v^\bullet) && (36)\\ &\geq \texttt{d}(u, v^\bullet) + \texttt{LB}^-(\overline{L_u}) && (37)\\ &\geq \widehat{\texttt{LB}}^-(u) && (38) \end{aligned}$$

Now for the function GenerateSearchPath⟨Forward⟩ to terminate here (due to the while-loop condition not being true anymore), it must additionally hold:

$$\texttt{LB}^-(v^*) = \texttt{UB}^-(v^*) \tag{39}$$

Hence, the upper bound $\texttt{UB}^-(u)$ of getting from u over $v^* \in \sigma_u^*$ to the depot is also at most as large as $\texttt{d}(u, v) + \texttt{UB}^-(v)$ for any other neighbor $v \in \sigma_u, v \notin \sigma_u^*$.

This implies:

1. $\texttt{LB}^-(u)$ would be set to $\texttt{UB}^-(u)$ after the iteration and u would not even be qualifying for being a successor node to some node $t \in \pi_u$. Hence, this exact search path resulting in $\texttt{MSB}(u, v^\bullet)$ being true cannot occur twice.
2. A state node v' that could succeed u in an optimal solution t^{OPT} (supposed that u is part of the optimal solution) cannot be filtered out here. v' could only be included in the optimal solution if it is preceeded by some node $u' \neq u$.

□

6.7.2. Breaking and Termination Based on $\texttt{UB}^-(0^+)$ (CTB and CTD)

Lemma 6.3. SolveTSPPD *using complete-tour-upper-bound-breaking (CTB) and complete-tour-discontinuation (CTD) rule terminates having found the optimal solution* t^{OPT}.

The proof will be similar to Theorem 6.1. However, here we are handling the interaction between CTB and CTD here.We are going to show that using CTB and CTD no search path is generated that has an existing search path as a prefix and no search path is generated more than *twice*. Furthermore, the search path that will result in finding t^{OPT} is always generated.

Proof. Let $P = (0^+, v_1, \ldots, v_k)$ be a search path and $v^\bullet$ be the first successor of v_k scanned by the foreach-loop such that $\texttt{CTB}(v_k, v^\bullet)$ is true. That is the case if:

$$\texttt{c}(P) + \texttt{d}(v_k, v^\bullet) + \texttt{LB}^-(\overline{L_{v_k}}) \geq \texttt{UB}^-(0^+)$$

According to Corollary 6.3, at least one of the following must be true then:

1. $\texttt{d}(v_k, v^\bullet) + \texttt{LB}^-(\overline{L_{v_k}}) > \widehat{\texttt{LB}}^+(v_k)$: In this case, we know that $v^\bullet$ would not minimize the progression rule anyway. Here, we can refer to the proof for Lemma 6.2 that showed that ignoring such successors $v^\bullet$ based on the current lower bound values will not prevent the outer search from finding the optimal solution and terminating.
2. $\texttt{c}(P) + \texttt{LB}^-(\widehat{\sigma^*_{v_k}}) > \texttt{UB}^-(0^+)$: In this case $\texttt{CTD}(v_k, v^*)$ would be true for all $v^* \in \sigma^*_{v_k}$ and the search path would not be extended beyond u.

If exclusively Item 2 applies and not Item 1 (the only case where termination remains to be shown), we know that $\texttt{d}(v_k, v^\bullet) + \texttt{LB}^-(\overline{L_{v_k}}) \leq \widehat{\texttt{LB}}^+(v_k)$. Hence, we can specify the intermediate assigment performed by Line 7 after $\texttt{CTB}(v_k, v^\bullet)$ resolving to true:

$$\widehat{\texttt{LB}}^-(v_k) \leftarrow \min\{\widehat{\texttt{LB}}^+(v_k), \texttt{d}(v_k, v^\bullet) + \texttt{LB}^-(\overline{L_{v_k}})\} \tag{40}$$

to the final assignment of

$$\widehat{\texttt{LB}}^-(v_k) \leftarrow \texttt{d}(v_k, v^\bullet) + \texttt{LB}^-(\overline{L_{v_k}}) \tag{41}$$

Immediately afterwards, the assignment $\texttt{LB}^-(v_k) \leftarrow \widehat{\texttt{LB}}^-(v_k)$ is performed.

Let us suppose Line 7 would generate the same search path again up to v_{k_1}, i.e. some path $P' = (0^+, v_1, \ldots, v_{k-1})$. Hence, $\texttt{c}(P) = \texttt{c}(P') + \texttt{d}(v_{k-1}, v_k)$.

In the next inner search iteration, $\sigma^*_{v_{k-1}}$ will be determined. Primarily, there two possibilites:

1. v_k is still in $\sigma^*_{v_{k-1}}$ and is added to the search path exactly *once* more. This again has two crucial consequences
   a) $\texttt{LB}^-(v_k)$ is limiting $\texttt{LB}^-(v_{k-1})$, which results in an effective increase of the latter (intermediate incumbent skipped for simplicity):

   $$\texttt{LB}^-(v_{k-1}) \leftarrow \texttt{d}(v_{k-1}, v_k) + \texttt{LB}^-(v_k) = \texttt{d}(v_{k-1}, v_k) + \texttt{d}(v_k, v_k^\bullet) + \texttt{LB}^-(\overline{L_{v_k}})$$

   This will ensure that in a future inner search iteration, after v_{k-1} has been added to the search path, $\texttt{CTD}(v_{k-1})$ must be true.

b) $\texttt{CTD}(v_k)$ will apply in the following inner search iteration before even evaluating successors of v_k. This is by assumption, because of $\texttt{CTB}(v_k, v^\bullet)$ being true and because of the subsequent increase of $\texttt{LB}^-(v_k)$ to the value $\texttt{d}(v_k, v^\bullet) + \texttt{LB}^-(\overline{L_{v_k}})$.

$$\begin{aligned}
\texttt{c}(P') + \texttt{LB}^-(v_{k-1}) &= \texttt{c}(P') + \texttt{d}(v_{k-1}, v_k) + \texttt{d}(v_k, v_k^\bullet) + \texttt{LB}^-(\overline{L_{v_k}}) && (42)\\
&= \texttt{c}(P) + \texttt{d}(v_k, v_k^\bullet) + \texttt{LB}^-(\overline{L_{v_k}}) && (43)\\
&\overset{\text{by assumption}}{\geq} \texttt{UB}^-(0^+) && (44)\\
& && (45)
\end{aligned}$$

2. There now exists another limiting successor $u_k \neq v_k, u_k \in \sigma^*_{v_{k-1}}$. Then either

   a) $\texttt{d}(v_{k-1}, u_k) + \texttt{LB}^-(u_k)$ is sufficiently small, such that

$$\begin{aligned}
& \texttt{c}(P') + \texttt{d}(v_{k-1}, u_k) + \texttt{LB}^-(\overline{L_{u_{k-1}}}) && (46)\\
\overset{\neg\texttt{CTB}(v_{k-1},u_k)}{\leq}\ & \texttt{c}(P') + \texttt{d}(v_{k-1}, u_k) + \texttt{LB}^-(u_k) && (47)\\
\overset{\neg\texttt{CTD}(u_k)}{<}\ & \texttt{UB}^-(0^+) && (48)
\end{aligned}$$

   In this case, $\texttt{LB}^-(v_{k-1})$ will be increased accordingly and u_k will be added to the search path.

   b) $\texttt{d}(v_{k-1}, u_k) + \texttt{LB}^-(u_k)$ is not sufficiently small. In this case, we can restart the proof, with u_k now taking the role v_k did play here.

That leaves us with the proof of optimality. For this, we only have to prove that, as long as no t^{OPT} has been found, CTB will not prevent pushing some u_{k+1} to some search path $P = (0^+, u_1, \ldots, u_k)$ that is a prefix to some $t^{\text{OPT}} = (0^+, u_1, \ldots, u_k, u_{k+1}, \ldots, 0^-)$. This is very easy to show, as $\texttt{CTB}(u_k, u_{k+1})$ compares the following term against $\texttt{UB}^-(0^+)$

$$\begin{aligned}
\texttt{c}(P) + \texttt{d}(u_k, u_{k+1}) + \texttt{LB}^-(\overline{L_{u_k}}) &= \texttt{c}(0^+, u_{k+1}) + \texttt{LB}^-(\overline{L_{u_k}}) && (49)\\
&\leq \texttt{c}(0^+, u_{k+1}) + \texttt{c}(u_{k+1}, 0^-) && (50)\\
&= \texttt{c}(t^{\text{OPT}}) && (51)
\end{aligned}$$

However, $\texttt{UB}^-(0^+)$ is strictly larger than $\texttt{c}(t^{\text{OPT}})$ as long as no t^{OPT} has been found. In turn, $\texttt{CTB}(u_k, u_{k+1})$ can not be true in such a case. □

7. Parallel Aspects

State-space-search offers some potential for parallelization. The technique, we are going to use, is of pragmatic nature: Every thread is extending and initiating their own search paths concurrently. Whenever a thread A comes upon a situation where another thread B currently executes some costly operation on a state node or preliminary lower bound A would like to access as well, thread A makes a greedy choice.

7.1. Introduction of Occasional Greedy Choices

In the case of lower bound initializations of nodes, we assume $\texttt{LB}^-(v)$ to be 0 and simply choose v the next location in the search path of the current thread.

For the preliminary lower bound $\texttt{LB}^-(\overline{L_u})$, we however assume the value to be ∞ to force the thread to choose the nearest neighbor to the current node u in the search path. This is considered more reasonable than waiting another thread to finish the computation of $\texttt{LB}^-(\overline{L_u})$ or doing duplicate work by calculating the value once more. As opposed to the initialization of $\texttt{LB}^-(v)$, we do not want the preliminary lower bound to be assumed of a value of 0, because that would (in most cases) lever out the breaking-predicates.

7.2. Modifications to Storage of State Nodes

The most important change is affecting the data structure V^{map}. It is changed from a normal hash map to a *sharded* hash map $V_{\text{sh}}^{\text{map}}$. These are data structures consisting of an array of *shards*, where each shard typically contains exactly one hash map. This requires another hashing layer in front of the hash maps themselves. Namely, a hash function

$$h\colon (l, 2^{\mathcal{L}}) \to \mathbb{N} \tag{52}$$

maps locations along with a set of visited locations to some natural number.

This allows several threads to access the whole data structure concurrently, as long as they do not read and write to the same shard at once. On the other hand, individual shards require a mutual exclusion (mutex) property to prevent more than one thread reading and writing to it at the same time. In this way, it is still possible that threads face idle time, because of waiting for a write access of another thread to a specific shard to be completed. This waiting time will be miniscule compared to the idle time of waiting for another thread to execute some costly operation. The reason is, that whenever a costly operation needs to be executed by some thread – most prominently computing $\texttt{LB}^-(v)$ for some state node v, the responsible thread inserts v into $V_{\text{sh}}^{\text{map}}[h(l_v, L_v)]$ and then immediately frees the corresponding shard going ahead with computing $\texttt{LB}^-(v)$. Only when this costly operation is finished, access to the shard is requested again.

Fortunately, only insertions and removal (which our algorithm does not do) are considered as write modifications *to the map*. Changing the transient properties of a state node $v \in V^{\text{map}}$, e.g. $\texttt{LB}^-(v)$ is not considered as a write operation to the map. The cost of employing this consideration is that the individual state nodes $v \in V_{\text{sh}}^{\text{map}}$ still require an additional *atomic lock protection*, because concurrent read and writes on v could result in undefined behaviour.

Algorithm 7: Search routine as adapted for parallelization

Function GENERATESEARCHPATHPARALLEL$\langle Forward \rangle$ $(u \in V)$**:**
  $u \leftarrow 0^+$
  while $\neg\texttt{CTD}(u)$ **do**
    $\widehat{\texttt{LB}}^-(u) \leftarrow \infty$ **if** $\texttt{LB}^-(\overline{L_u})$ is currently computed by another thread **then**
      $\texttt{LB}^-(\overline{L_u}) \leftarrow \infty$
    else
      $\texttt{LB}^-(\overline{L_u}) \leftarrow c^{\text{MST}}(\overline{L_u} \setminus \{0^-\}) + \min_{w \in \pi(0^-)} \texttt{d}(l_w, 0^-)$
    foreach $l \in \overline{\mathcal{L}}(l_u)$ **do**
      if l not a valid successor location for u **then**
        continue
      $v \leftarrow (l, L_u \cup \{l\})$
      if $\texttt{MSB}(u, v)$ **then**
        break
      if $\texttt{CTB}(u, v)$ **then**
        $\widehat{\texttt{LB}}^-(u) \leftarrow \min\{\widehat{\texttt{LB}}^-(u), \texttt{d}(u, v) + \texttt{LB}^-(\overline{L_u})\}$
        break
      if $v \in V_{\text{sh}}^{\text{map}}[h(l_v, L_v)]$ & $\texttt{LB}^-(v)$ is currently computed **then**
        $u \leftarrow v$
        break
      if $v \notin V_{\text{sh}}^{\text{map}}[h(l_v, L_v)]$ **then**
        /* Either initiates or reuses $\texttt{LB}^-(\overline{L_v})$ */
        $\texttt{LB}^-(v) \leftarrow$ INITLB$^-(v)$
        $V^{\text{map}}[l_v]$.INSERT(v)
      $\widehat{\texttt{LB}}^-(u) \leftarrow \min\{\widehat{\texttt{LB}}^-(u), \texttt{d}(u, v) + \texttt{LB}^-(v)\}$
      $\texttt{UB}^-(u) \leftarrow \min\{\texttt{UB}^-(u), \texttt{d}(u, v) + \texttt{UB}^-(v)\}$
      $\texttt{UB}^-(0^+) \leftarrow \min\{\texttt{UB}^-(0^+), \texttt{c}(P) + \texttt{UB}^-(u)\}$
    if all successors of u checked **then**
      $\texttt{LB}^-(\overline{L_u}) \leftarrow \max\{\texttt{LB}^-(\overline{L_u}), \min_{v \in \sigma_u} \texttt{LB}^-(v)\}$
    $\texttt{LB}^-(u) \leftarrow \widehat{\texttt{LB}}^-(u)$
    $u \leftarrow$ some node $v^* \in \sigma_u^*$
    P.PUSH(u)

8. Experiments

The primarily algorithm compared against, is a simple ILP formulation for the TSPPD with Miller-Tucker-Zemlin (MTZ) subtour elimination constraints [MTZ60]given in Section 8.1. MTZ constraints are limited in their number, as opposed to the exponential amount of traditional, subset-based subtour elimination constraints. However, they do

not allow for same performance as the most sophisticated subtour elimination techniques in which subset-based constraints are introduced on demand during the solving process, most notably Dumitrescu et al. [DRCL] for the TSSPD. The latter publication will also serve as an approximate comparison, as the corresponding code was not published and the experiments were run on an AMD Opteron 250 CPU released in 2004. Furthermore, we use the benchmark instances created by Dumitrescu et al, which are described in section 7.1 of [DRCL]:

> The first set of instances was created by generating $2n + 1$ points randomly in the square $[0, 1000] \times [0, 1000]$. The first point is the depot while the remaining points are paired to form requests (point i is paired with point $n + i$). The travel costs (c_{ij}) were set equal to the Euclidean distances. Instances with $n = 5, 10, 15, 20, 25, 30, 35$ were created, five for each size. The instances are named probnX, where X is one of the letters a, b, c, d and e used to distinguish between instances with the same size.

The original instance files themselves were plain text files containing an enumerations of the (x, y)-coordinates of all locations. To ease readability, they were converted into a suitable comma-separated-value-file (csv) leaving the possibility for additional location attributes irrelevant for the TSPPD. Importantly, it has to be stated that after the computation of the Euclidean distance, all respective travel costs (c_{ij}) (which refer to the distances $\mathtt{d}(i, j)$ between two locations $i \in \mathcal{L}, j \in \mathcal{L}$ in this report) were rounded to the nearest integer (because Dumitrescu et al. decided to do so as well).

8.1. Reference Integer-Linear-Programming-Approach

8.1.1. Decision Variables

The most prominent decision variables are the arc-indexed variables x_{ij} for all edges (i, j) in the location graph LG.

$$x_{ij} \in \{0, 1\}, \quad \forall i, j \in \mathcal{L}, i \neq j \tag{53}$$

$x_{ij} = 1$ indicates that the vehicle is supposed to visit location j directly after location i.

The location-indexed variables u_i are used to prevent subtours (solution existing of several cycles) and to enforce the precedence constraints of the TSPPD:

$$u_i \in \mathbb{R}, \quad \forall i \in \mathcal{L} \tag{54}$$

$$\tag{55}$$

8.1.2. Objective Function

$$\min \sum_{i \in \mathcal{L}} \sum_{j \neq i \in \mathcal{L}} c_{ij} \cdot x_{ij} \tag{56}$$

8.1.3. Constraints

Depot departure constraints: The depot departure has an outgoing edge to exactly one pickup location, no outgoing edges to delivery locations and no incoming edges:

$$\sum_{i^+ \in \mathcal{L}} x_{0^+ i^+} = 1, \sum_{i^- \in \mathcal{L}} x_{0^+ i^-} = 0, \sum_{i \in \mathcal{L}} x_{i 0^+} = 0 \tag{57}$$

Depot arrival constraints: The depot arrival has an incoming edge from exactly one delivery location, no incoming edges from pickup locations and no outgoing edges:

$$\sum_{i^- \in \mathcal{L}} x_{i 0^-} = 1, \sum_{i^+ \in \mathcal{L}} x_{i^+ 0^-} = 0, \sum_{i \in \mathcal{L}} x_{0^- i} = 0, \tag{58}$$

Out-degree constraints: Each non-depot location has exactly one outgoing edge:

$$\sum_{j \neq i \in \mathcal{L}} x_{ij} = 1, \quad \forall i \in \mathcal{L} \setminus \{0^+, 0^-\} \tag{59}$$

In-degree constraints: Each non-depot location has exactly one incoming edge:

$$\sum_{j \neq i \in \mathcal{L}} x_{ij} = 1, \quad \forall j \in \mathcal{L} \setminus \{0^+, 0^-\} \tag{60}$$

MTZ subtour elimination constraints: Prevents subtours among non-depot locations:

$$u_{0^+} = 1, \qquad u_{0^-} = |\mathcal{L}| \tag{61}$$

$$2 \leq u_i \leq |\mathcal{L}| - 1, \qquad \forall i \in \mathcal{L} \setminus \{0^+, 0^-\} \tag{62}$$

$$u_i - u_j + |\mathcal{L}| \cdot x_{ij} \leq |\mathcal{L}| - 1, \qquad \forall i, j \in \mathcal{L} \setminus \{0^+, 0^-\}, i \neq j \tag{63}$$

Pickup-delivery precedence: Each pickup location must be visited before its delivery location:

$$u_{i^+} + 1 \leq u_{i^-}, \quad \forall i \in \mathcal{R} \tag{64}$$

8.1.4. Variable Domains

$$x_{ij} \in \{0, 1\}, \qquad \forall i, j \in \mathcal{L}, i \neq j \tag{65}$$

$$x_{ii} = 0, \qquad \forall i \in \mathcal{L} \tag{66}$$

$$u_i \in \mathbb{R}, \qquad \forall i \in \mathcal{L} \tag{67}$$

8.2. Methods

Our solving approach was tested with the following configurations using a time limit of one hour:

1. Pure – without `MSB` or `CTB`, using `TD` as discontinuation-predicate
2. `MSB` enabled, using `TD` as discontinuation-predicate
3. `CTB` enabled, using `CTD` as discontinuation-predicate
4. both `MSB` and `CTD` using `CTD` as discontinuation-predicate
5. parallel outer search with both `MSB` and `CTB` using `CTD`

For all configurations, the lower bound initiation described in Section 5.11 was used. The cost of minimum spanning trees c^{MST} used for preliminary lower bounds (Section 6.1) and state node lower bound initialization (Section 5.11) were stored in a hashmap mapping a subset of the locations to the cost value for repeated use.

Apart from these configurations, for comparison an integer-linear-program (ILP) for the TSPPD given in Section 8.1 is run as well.

Thereby, we are interested in different performance indicators. These performance indicators can be separated in primary and secondary indicators. Primary indicators are figures routinely used in the analysis of optimization algorithms, while secondary indicators are specific to this state-space-based approach (and do not apply for the ILP).

1. Primary: (Average) time to prove optimality - if optimality has been proven within the time limit of one hour
2. Primary: (Average) time to find the (provenly) optimal solution - if optimal solution has been found within the time limit of one hour
3. Primary: Gap between Lower Bound and Upper Bound for $c(t^{\mathrm{OPT}})$ at termination
4. Secondary (only for state-space-search): number of outer search iterations; deterministic per instance, as long as no parallelization is used
5. Secondary (only for state-space-search): $|V^{\mathrm{map}}|/|V|$ – Ratio of state nodes added and total number of state nodes for instance size; deterministic per instance, as long as no parallelization is used

The benchmarks where executed with a python-script that was created with the help of ChatGPT 5.4. Notably, the script decides the number of runs per configuration and instance dynamically. The goal was to execute enough runs to yield a representative estimate for the average *time to prove optimality*. We decided for this figure, because it is the most common indicator used in benchmarking exact optimization algorithms. Under the assumption that the mean is approximately normally distributed with variance

unknown, we employ the 95%-confidence interval of a t-distribution. When the halfwidth of this interval has a relative size of less than 5% of the current mean estimator for the running time, the value is considered to be narrowed down sufficiently enough and we procede with the next instance.

Apart from that:

1. A maximum number of runs of 100 was set
2. Specifically for the longer runs, additional exclusion criteria for initiating more runs are enforced:
   a) when the solving time has been more than 20 minutes and we are testing a multi-threaded solver or configuration (which includes the ILP)
   b) when the solving time has been more than 10 minutes in all the other configurations. In deterministic algorithms, relative deviations are miniscule above this range.

All benchmarks were executed on a machine with an Intel Core i9 13900F (24 CPU cores, 32 threads) and 32GB RAM. For the multicore-benchmarks, pilot runs strongly indicated that enabling more than 16 threads did not lead to additionally benefit, so this was set as the maximum number of threads used.

Evaluation of the benchmarks was done with a python-script that was created with the help of Claude Sonnet 4.6 (Thinking). Because the large number of indicators evaluated here and the different output formats of the state space search and the ILP, the script is comparably complex and required many adjustments until it was verifiably producing the correct results.

8.3. Results and Discussion

The complete results are captured in Section A of the appendix. Besides, Section B of the appendix contains plots of the lower and upper bound evolutions for selected instances.

8.3.1. Comparison between configurations

Overall, the pure and most simplistic approach has the least favorable indicators. The only seemingly favorable indicator for this configuration is the that the number of outer search iteration is lowest here.

Average ratio of mapped state nodes to total state nodes In Table A5, as expected and on average, the most sophisticated configuration (`CTB`, `MSB` + `CTD`) has the most favorable values (i.e. the lowest). The most surprising insight is, that the parallel configuration was able to get along with inserting less nodes in V^{map} for some instances. In general, the shortcomings of the greedy decision the parallel mode has to make did not seem to create significant increases in state node insertion into V^{map}. It is also

visible that complete-tour-breaking and -discontinuation (CTD, CTB) in isolation is more effective than only incorporating minimizing-successor-breaking MSB, TD in isolation.

Importantly, it has to be noted that for instances (all instances with 25 requests) for which optimality could not be proven, we have minor deviations between configurations and instances of the same size. The reason for this is that, if we have could not prove optimality, the solver had always run for a full hour. In this time, the number of state nodes to be added V^{map} is quite uniform. The differences observed here (all smaller than 0.0011%) are most likely random noise, with more variation for the non-deterministic parallel configuration.

Average number of outer search iterations In Table A4, the picture seems inverse: The more sophisticated of a configuration we use, the more outer search iterations have to be executed. This might seem unfavorable on first sight, but it is a mere consequence of discontinuing outer search iterations early. The pure configuration will have longer outer search iteration yielding more information. In particular, the pure configuration scans more successor nodes $v \in \sigma_u$, even when the minimizing successor v^* has already been identified and it continues extending the search path P, even when $\mathtt{c}(P)+\mathtt{LB}^-(u)$ is larger than $\mathtt{UB}^-(0^+)$. This will lead to often indecisive extra information, but occasionally, these advancements can be useful in future iterations. For example, when $\mathtt{c}(P)$ is at a later iteration sufficiently lower when considering u, having progressed on u might occasionally prove useful. However, this does not explain the small differences between the pure and the MSB, TD configuration. We traced the root cause for this via diagnostic testing: the update of the preliminary lower bound at the end of an inner search routine (Line 25/26 in Line 7). When no breaking-predicate is active, this update still occurs. In fact, it will occur in every single inner search iteration, because the inner search will not exit the foreach-loop early. Hence the preliminary lower bound has been initiated as $\mathtt{LB}^-(\overline{L_v})$ for a state node v, we can expect it to tighten whenever v is added into the search path. This tightened preliminary lower bound can be reused for the lower bound initialization for any state node v' with $\overline{L_v} = \overline{L'_v}$ that is inserted into V^{map} later and occasionally increase the initial lower bounds for state nodes. It does not occur very often, but enough to produce the observed effect.

Average time to prove optimality Table A2 is the indicator with the largest discrepancy between configurations. Moreover, the discrepancies seem to grow with growing instance size. For instances with 10 or less requests, the CTB, MSB + CTD has always more than a third of the runtime of the pure configuration. ALso, the parallel configuration does not introduce a significant benefit here. The latter fact comes unsurprising, as the overhead of multi-threaded computing is a more limiting factor for algorithms that finish in less than 100ms.

On the other hand, the difference between pure and CTB, MSB + CTD are often be more than tenfold for the instances with 20 requests. For prob20e, pure did even hit the time limit while the most other configurations did not come close to it.

Average time to find the optimal solution (Table A1) While on average, the inter-configuration ranking appears the same (more sophisticated correlates with lower time to find the optimal solution), there are some notable exceptions here: For prob20c, `MSB`, `TD` was able to find the optimal solution faster than all other configuration, except the parallel one. And for prob20a, it was at least faster than `CTD`, `CTB` in this respect. In general, the differences between the configurations `MSB`, `TD` and `CTD`, `CTB` seem to be a lot lower than for the previously studied indicators.

Average gap at termination (Table A3) When optimality could not be proven, we can still observe (as expected) that the more sophisticated configurations behave in a preferable way. However, the difference between `MSB`, `TD` and `CTD`, `CTB` seem to be significantly lower than the difference these show in time to prove optimality, even though both of these indicators are closely related. The speculated reason for this may be hardware constraints: Unfortunately, system memory usage was not logged, but upilot runs showed that the it will run full in around 45 minutes, on average (the speed of inserting state nodes is assumed to be approximately constant). This memory overflow marks an important milestone, as from now on, a swap partition on the hard drive has to be utilised which leads to tremendously larger access times on state nodes. After this milestone, we have lower, but likely more evened-out speed of process progress among the configuration.

Interestingly, for instances where no configuration was able to prove optimality, the parallel configurations ended with, on average, a slightly higher gap. We can only give an unproven assumption for this effect: While the parallel configuration does not insert significantly more state nodes into V^{map} than its single core counterpart `CTB`, `MSB` + `CTD`, it does so at a faster rate. This leads to earlier termination for smaller instances, but for larger instances it will lead to the system memory running full earlier. Pilot runs showed that the number of state nodes that are inserted into V^{map} by the parallel configuration will lead the system memory to be exhausted by around 20 minutes. Utilisation of the swap partition is probably even more impactful for the parallel mode, as the shared access of V^{map} forces threads to wait often for other threads to finish writing to a shard or reading/writing to a specific state node $v \in V^{\text{map}}$.

8.3.2. Comparison Between State Space Search and ILP

Regardless of instance size, the ILP took overwhelmingly longer than any configuration of the state-space search in both finding the optimal solution and proving optimality. It has to be noted that the time for finding the optimal was estimated based on the output: The CPLEX log contains checkpoints and does not log the timestamp for every single event. So, the mean value between the time of the last checkpoint (or 0, of there were none before) and the time of the next checkpoint serves as an estimator here.

8.3.3. Comparison Between State Space Search and Branch-and-Cut algorithm by Dumistrescu et al.

Figure Section A shows the results by Dumistrescu et al. [DRCL]. The results have to be taken with a large grain of salts due to the already mentioned outdated hardware. Taking this into account, it is visible that instances with 10 or less requests may be solved faster by the state space search than the sophisticated Branch-and-Cut algorithm on modern hardware. However, with 15 or more requests, it is certain that the Branch-and-Cut algorithm is superior in terms of proving optimality.

8.3.4. Notes on the bound evolution plots

The plots are considered complementary material and will not be discussed in detail here. Since we are talking about complete tours, $\texttt{LB}(0^+, 0^-)$ is colloquially called the lower bound here, while $\texttt{UB}(0^+, 0^-)$ is referred to as th upper bound. It has to be noted that the lower bound is subject to much more frequent changes than the upper bound (as you may be able to surmise from the envelopes). Hence, to limit the size of the log file and reduce access to the shared data structure (for the parallel configuration), a scanning rate of $1/s$ was imposed to the lower bound. If you try to look past this limitation and in particular at the figures in logarithmic scale, the lower bound evolution seems to behave like a logarithmic curve as the changes seem very constant in this representation.

References

[Bel62] Richard Bellman: Dynamic Programming Treatment of the Travelling Salesman Problem. *J. ACM*, 9(1):61–63, 1962, 10.1145/321105.321111.

[DRCL] Irina Dumitrescu, Stefan Ropke, Jean François Cordeau, and Gilbert Laporte: The traveling salesman problem with pickup and delivery: polyhedral results and a branch-and-cut algorithm. page 269–305.

[GKS21] Daniela Gaul, Kathrin Klamroth, and Michael Stiglmayr: Solving the Dynamic Dial-a-Ride Problem Using a Rolling-Horizon Event-Based Graph. 96:8:1–8:16, 2021, 10.4230/OASIcs.ATMOS.2021.8.

[HK62] Michael Held and Richard M. Karp: A Dynamic Programming Approach to Sequencing Problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962, 10.1137/0110015.

[HK70] Michael Held and Richard M. Karp: The Traveling-Salesman Problem and Minimum Spanning Trees. *Operations Research*, 18(6):1138–1162, 1970, 10.1287/opre.18.6.1138.

[MTZ60] C. E. Miller, A. W. Tucker, and R. A. Zemlin: Integer Programming Formulation of Traveling Salesman Problems. *J. ACM*, 7(4):326–329, October 1960, 10.1145/321043.321046.

[RN16] Stuart Jonathan Russell and Peter Norvig: Artificial Intelligence: A Modern Approach. 2016, 10.5555/3112658.3112903.

A. Tabular results

Tab. A1: Average time (ms) to find the optimal solution. — indicates the run did not solve to optimality.

Instance	Pure	`MSB` + `TD`	`CTB` + `CTD`	`CTB`, `MSB` + `CTD`	Parallel	ILP [3]
prob5a	0	0	0	0	0.00	18
prob5b	0	0	0	0	1	15
prob5c	0	0	0	0	1	14
prob5d	0	0	0	0	0	15
prob5e	0	0	0	0	1	20
prob10a	72	44	27	20	13	23454
prob10b	31	18	13	11	5	7140
prob10c	11	7	12	9	6	1142
prob10d	36	24	25	19	16	1995
prob10e	9	4	8	4	4	1605
prob15a	1574	476	387	227	31	—
prob15b	6849	2298	810	499	110	—
prob15c	21	5	5	2	2	17063
prob15d	1018	345	443	202	169	592342
prob15e	227	64	38	30	16	7562
prob20a	96472	25180	31798	24744	10642	—
prob20b	364360	103499	52206	41744	8163	—
prob20c	63392	4664	11067	11826	2763	—
prob20d	170288	48528	29041	21544	12358	—
prob20e	—	137062	226356	248156	172582	—
prob25a	—	—	—	—	—	—
prob25b	—	—	—	—	—	—
prob25c	—	—	—	—	—	—
prob25d	—	—	—	—	—	—
prob25e	—	—	—	—	—	—

Tab. A2: Average time (ms) to prove optimality ($\texttt{LB}(0^+, 0^-) = \texttt{UB}(0^+, 0^-)$ for state space search). — indicates the run did not solve to optimality.

Instance	Pure	MSB + TD	CTB + CTD	CTB, MSB + CTD	Parallel	ILP
prob5a	1	1	1	1	3	37
prob5b	1	1	1	1	2	29
prob5c	1	1	0	0	2	29
prob5d	1	1	1	1	2	30
prob5e	1	1	1	1	2	52
prob10a	86	54	35	30	19	26354
prob10b	55	33	18	18	16	10675
prob10c	27	19	14	11	9	2093
prob10d	66	44	32	26	23	6430
prob10e	29	19	12	9	8	3028
prob15a	15023	6110	1802	1708	732	—
prob15b	27088	11816	3606	3346	1566	—
prob15c	137	40	11	12	10	22020
prob15d	8623	3578	1176	989	393	1098680
prob15e	651	192	48	41	30	379000
prob20a	431002	127763	34593	27706	14721	—
prob20b	1112278	335894	80658	73026	44782	—
prob20c	323664	102944	27226	25615	15124	—
prob20d	517658	163812	44130	38064	22760	—
prob20e	—	3267925	475297	430472	277839	—
prob25a	—	—	—	—	—	—
prob25b	—	—	—	—	—	—
prob25c	—	—	—	—	—	—
prob25d	—	—	—	—	—	—
prob25e	—	—	—	—	—	—

Tab. A3: Average gap at termination (%) computed as $(\texttt{UB}(0^+, 0^-) - \texttt{LB}(0^+, 0^-))/\texttt{UB}(0^+, 0^-)$ for state space search.

Instance	Pure	MSB + TD	CTB + CTD	CTB, MSB + CTD	Parallel	ILP
prob5a	0.00	0.00	0.00	0.00	0.00	0.00
prob5b	0.00	0.00	0.00	0.00	0.00	0.00
prob5c	0.00	0.00	0.00	0.00	0.00	0.00
prob5d	0.00	0.00	0.00	0.00	0.00	0.00
prob5e	0.00	0.00	0.00	0.00	0.00	0.00
prob10a	0.00	0.00	0.00	0.00	0.00	0.00
prob10b	0.00	0.00	0.00	0.00	0.00	0.00
prob10c	0.00	0.00	0.00	0.00	0.00	0.00
prob10d	0.00	0.00	0.00	0.00	0.00	0.00
prob10e	0.00	0.00	0.00	0.00	0.00	0.00
prob15a	0.00	0.00	0.00	0.00	0.00	4.77
prob15b	0.00	0.00	0.00	0.00	0.00	8.59
prob15c	0.00	0.00	0.00	0.00	0.00	0.00
prob15d	0.00	0.00	0.00	0.00	0.00	0.00
prob15e	0.00	0.00	0.00	0.00	0.00	0.00
prob20a	0.00	0.00	0.00	0.00	0.00	7.33
prob20b	0.00	0.00	0.00	0.00	0.00	20.39
prob20c	0.00	0.00	0.00	0.00	0.00	4.46
prob20d	0.00	0.00	0.00	0.00	0.00	9.23
prob20e	2.89	0.00	0.00	0.00	0.00	9.66
prob25a	12.64	9.07	10.15	7.97	8.16	20.22
prob25b	12.56	8.43	6.92	4.82	5.78	12.52
prob25c	9.28	4.87	1.23	0.76	1.04	22.10
prob25d	11.56	8.62	8.05	5.17	5.52	19.32
prob25e	11.23	7.96	5.15	2.67	2.86	12.21

Tab. A4: Average number of outer search iterations by instance and configuration.

Instance	Pure	`MSB` + `TD`	`CTB` + `CTD`	`CTB`, `MSB` + `CTD`	Parallel
prob5a	87	87	101	101	317
prob5b	26	28	27	28	271
prob5c	36	36	42	42	267
prob5d	56	57	59	60	218
prob5e	70	70	74	74	304
prob10a	8826	8892	10486	10486	20529
prob10b	3601	3616	4248	4250	9798
prob10c	1536	1539	1864	1859	4537
prob10d	6575	6598	7552	7554	15400
prob10e	1083	1109	1286	1305	3334
prob15a	364812	365302	482642	482642	828550
prob15b	701938	708224	889437	889590	1653740
prob15c	1452	1452	1644	1644	4734
prob15d	212831	213548	277666	277624	457005
prob15e	8937	8943	10441	10441	22067
prob20a	3316007	3318823	4050794	4024174	9198511
prob20b	8057682	8070587	10181710	10181681	25305700
prob20c	2954717	2960595	3915173	3912827	8257102
prob20d	4366906	4382815	5258394	5258926	12239508
prob20e	10027727	45054427	56930945	56264966	128495954
prob25a	2632137	22413831	15489737	77873383	120178248
prob25b	2771026	22807521	44152423	144387733	224672603
prob25c	2546405	21317459	208255838	292529714	542471118
prob25d	2811714	22675156	24482252	105414681	148954641
prob25e	3276530	22640373	60162558	175277856	349621384

Tab. A5: Average ratio of mapped state nodes to total state nodes ($|V^{\text{map}}|/|V|$) in %.

Instance	Pure	MSB + TD	CTB + CTD	CTB, MSB + CTD	Parallel
prob5a	55.6790	39.2593	20.6173	19.1358	17.5309
prob5b	27.7778	15.0617	10.0000	7.1605	7.0282
prob5c	38.0247	23.9506	10.0000	9.5062	9.6156
prob5d	46.1728	28.0247	13.5802	12.4691	12.3456
prob5e	57.1605	35.3086	21.2346	17.6543	16.7828
prob10a	38.2139	19.3865	5.4575	4.6558	4.6652
prob10b	21.7713	9.2176	1.9502	1.7261	1.6891
prob10c	8.0681	3.2967	0.6963	0.6285	0.6293
prob10d	26.6888	14.2458	5.6965	4.2194	4.1845
prob10e	9.1208	3.6628	1.2544	0.7585	0.8255
prob15a	9.7013	3.5240	0.5045	0.4900	0.4888
prob15b	18.5554	6.4201	0.9485	0.9206	0.9247
prob15c	0.0927	0.0211	0.0025	0.0018	0.0019
prob15d	5.7834	2.1585	0.3869	0.3085	0.3177
prob15e	0.5045	0.1189	0.0133	0.0117	0.0123
prob20a	0.6626	0.1606	0.0220	0.0166	0.0165
prob20b	1.5705	0.3952	0.0382	0.0355	0.0349
prob20c	0.4985	0.1274	0.0124	0.0123	0.0123
prob20d	0.7672	0.2052	0.0253	0.0210	0.0211
prob20e	1.8729	1.7800	0.2150	0.1987	0.2003
prob25a	0.0053	0.0056	0.0056	0.0054	0.0057
prob25b	0.0052	0.0056	0.0056	0.0055	0.0052
prob25c	0.0052	0.0055	0.0052	0.0049	0.0046
prob25d	0.0052	0.0055	0.0056	0.0054	0.0053
prob25e	0.0054	0.0056	0.0056	0.0054	0.0050

Table 6 Branch-and-cut results for data set 1: randomly generated instances

Name	n	Best IP	Seconds	Opt	Root LB	Best LB	RLB/UB	BLB/UB	BC Nodes	#cuts
prob5a	5	3585	0	✓	3585.00	3585.00	100.0	100.0	1	19
prob5b	5	2565	0	✓	2565.00	2565.00	100.0	100.0	1	6
prob5c	5	3787	0	✓	3787.00	3787.00	100.0	100.0	1	6
prob5d	5	3128	0	✓	3128.00	3128.00	100.0	100.0	1	6
prob5e	5	3123	0	✓	3123.00	3123.00	100.0	100.0	1	45
prob10a	10	4896	3	✓	4851.08	4896.00	99.1	100.0	4	134
prob10b	10	4490	2	✓	4490.00	4490.00	100.0	100.0	1	135
prob10c	10	4070	0	✓	4070.00	4070.00	100.0	100.0	1	93
prob10d	10	4551	1	✓	4551.00	4551.00	100.0	100.0	1	93
prob10e	10	4874	4	✓	4874.00	4874.00	100.0	100.0	1	97
prob15a	15	5150	8	✓	5030.48	5150.00	97.7	100.0	6	268
prob15b	15	5391	21	✓	5085.48	5391.00	94.3	100.0	45	580
prob15c	15	5008	0	✓	5008.00	5008.00	100.0	100.0	1	165
prob15d	15	5566	14	✓	5417.93	5566.00	97.3	100.0	15	390
prob15e	15	5229	0	✓	5229.00	5229.00	100.0	100.0	1	140
prob20a	20	5698	12	✓	5647.92	5698.00	99.1	100.0	9	438
prob20b	20	6213	20	✓	6125.61	6213.00	98.6	100.0	20	473
prob20c	20	6200	19	✓	6042.74	6200.00	97.5	100.0	28	444
prob20d	20	6106	17	✓	5985.96	6106.00	98.0	100.0	28	369
prob20e	20	6465	58	✓	6181.51	6465.00	95.6	100.0	115	962
prob25a	25	7332	14400		6632.90	7168.14	90.5	97.8	14377	3244
prob25b	25	6665	3138	✓	6259.86	6665.00	93.9	100.0	7952	2266
prob25c	25	7095	291	✓	6767.27	7095.00	95.4	100.0	878	1255
prob25d	25	7069	14323	✓	6515.85	7069.00	92.2	100.0	21627	3873
prob25e	25	6754	72	✓	6529.99	6754.00	96.7	100.0	125	704
prob30a	30	7309	14400		6745.14	7196.27	92.3	98.5	8296	3238
prob30b	30	6857	2843	✓	6461.94	6857.00	94.2	100.0	4528	2262
prob30c	30	7723	1891	✓	7258.73	7723.00	94.0	100.0	4269	2019
prob30d	30	7310	573	✓	7028.66	7310.00	96.2	100.0	1783	1372
prob30e	30	7213	14400		6683.29	7166.34	92.7	99.4	10523	3412
prob35a	35	7746	2104	✓	7338.12	7746.00	94.7	100.0	3541	1649
prob35b	35	7904	14400		7084.73	7496.03	89.6	94.8	6167	2764
prob35c	35	7949	14400		7509.94	7858.39	94.5	98.9	8427	3194
prob35d	35	7905	14400		7306.69	7686.77	92.4	97.2	9025	2944
prob35e	35	8530	14400		7690.94	8069.74	90.2	94.6	9791	3562
				28			96.5	99.5		

B. Bound evolution plots for selected instances

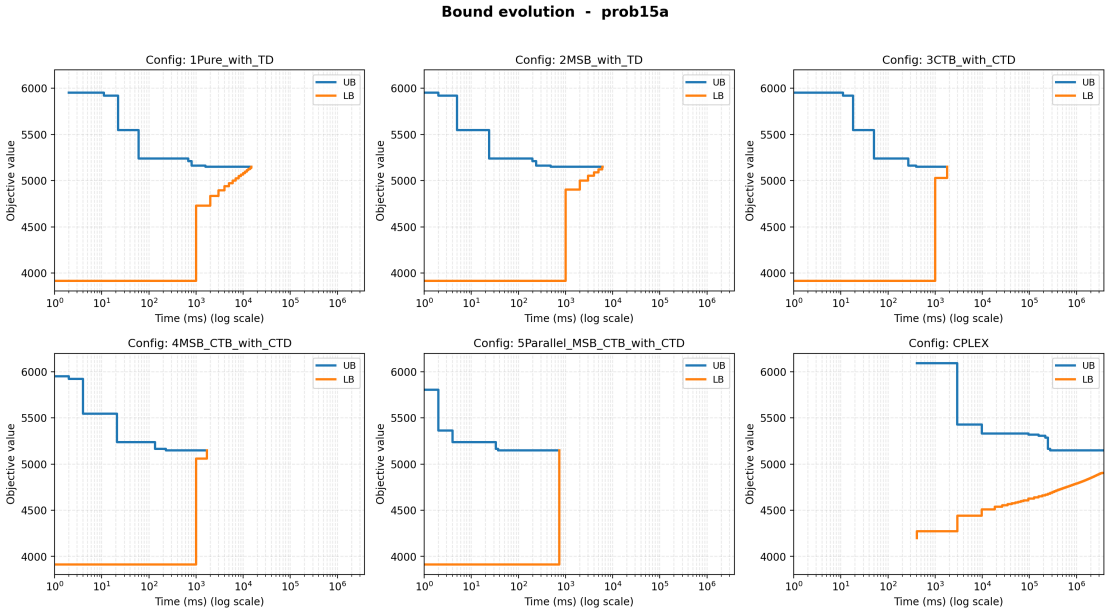


Fig. 2: Evolution of lower (orange) and upper (blue) bound in instance prob15a (normed logarithmic scale)

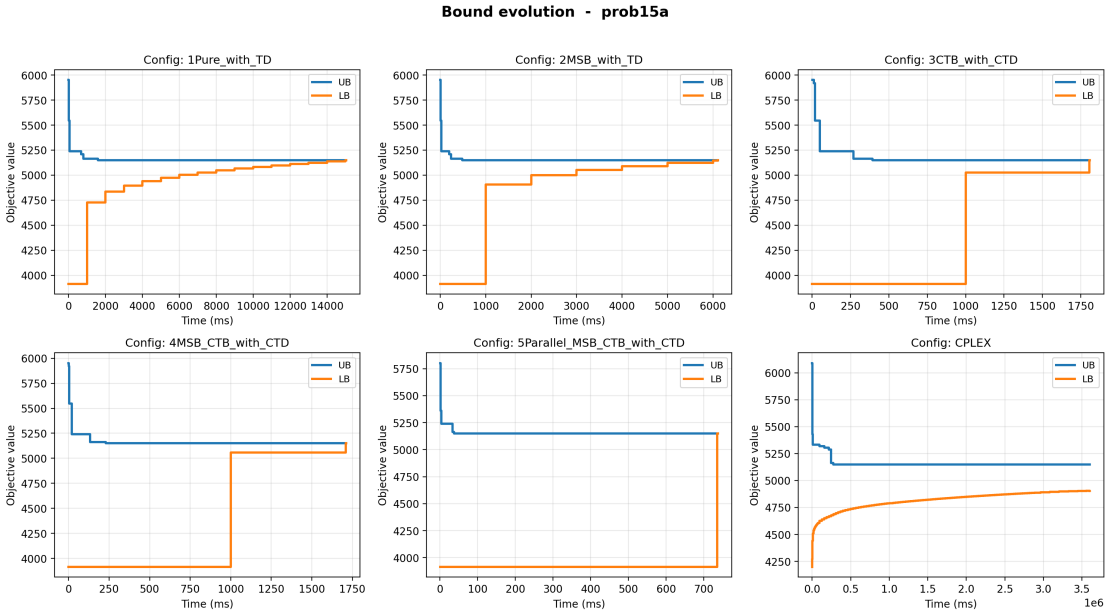


Fig. 3: Evolution of lower (orange) and upper (blue) bound in instance prob15a

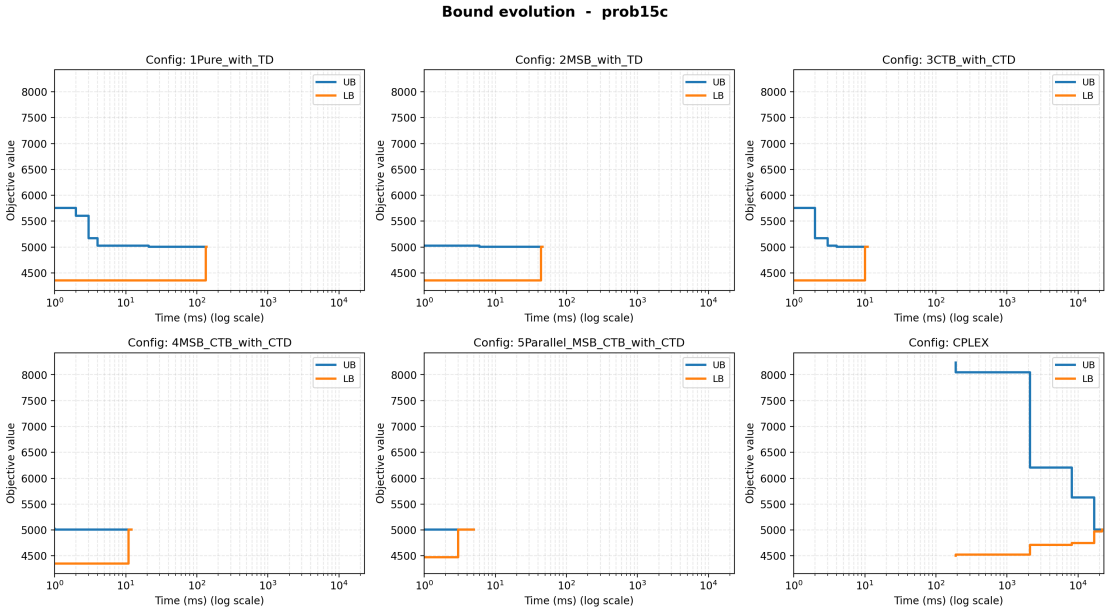


Fig. 4: Evolution of lower (orange) and upper (blue) bound in instance prob15c (normed logarithmic scale)

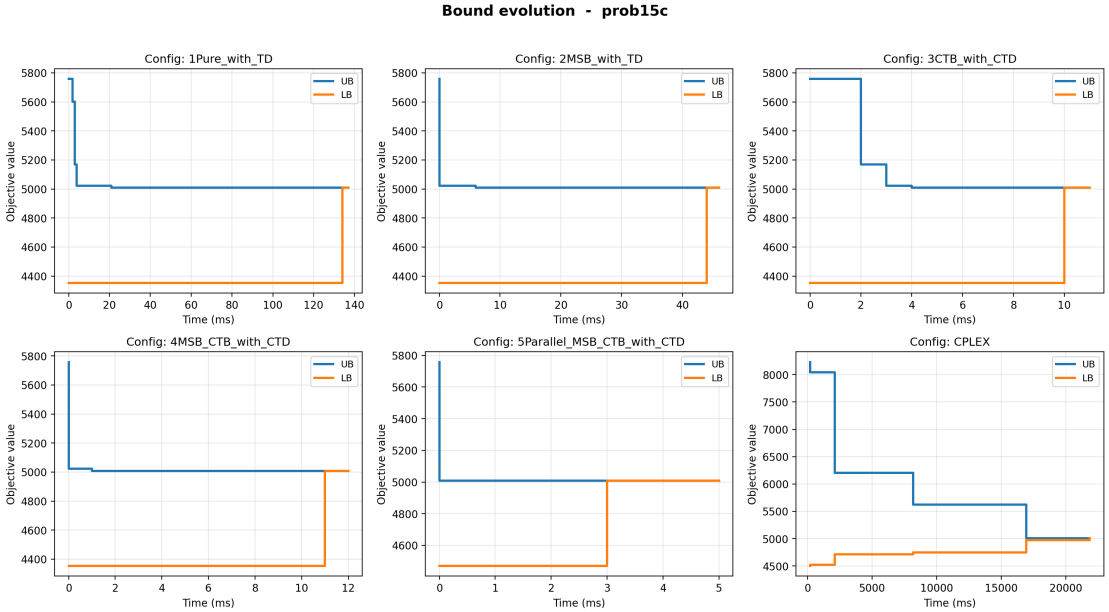


Fig. 5: Evolution of lower (orange) and upper (blue) bound in instance prob15c

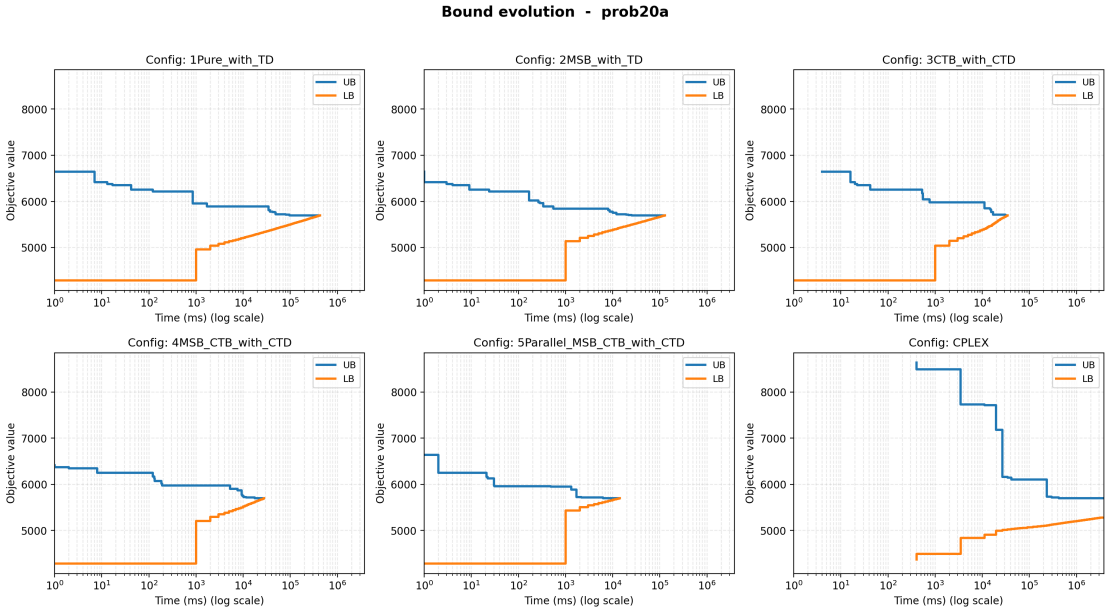


Fig. 6: Evolution of lower (orange) and upper (blue) bound in instance prob20a (normed logarithmic scale)

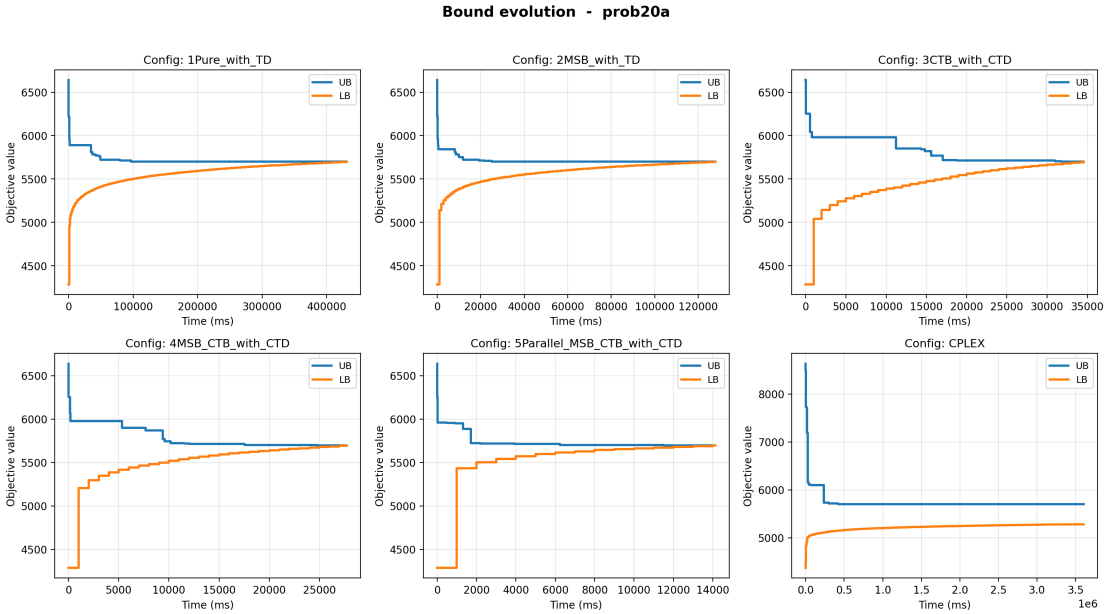


Fig. 7: Evolution of lower (orange) and upper (blue) bound in instance prob20a

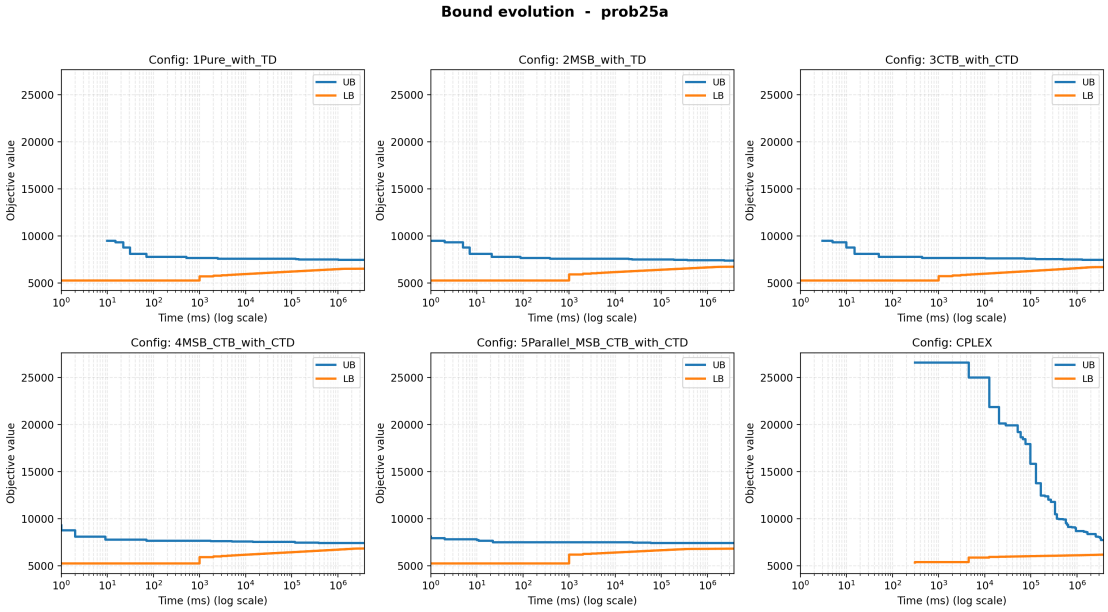


Fig. 8: Evolution of lower (orange) and upper (blue) bound in instance prob25a (normed logarithmic scale)

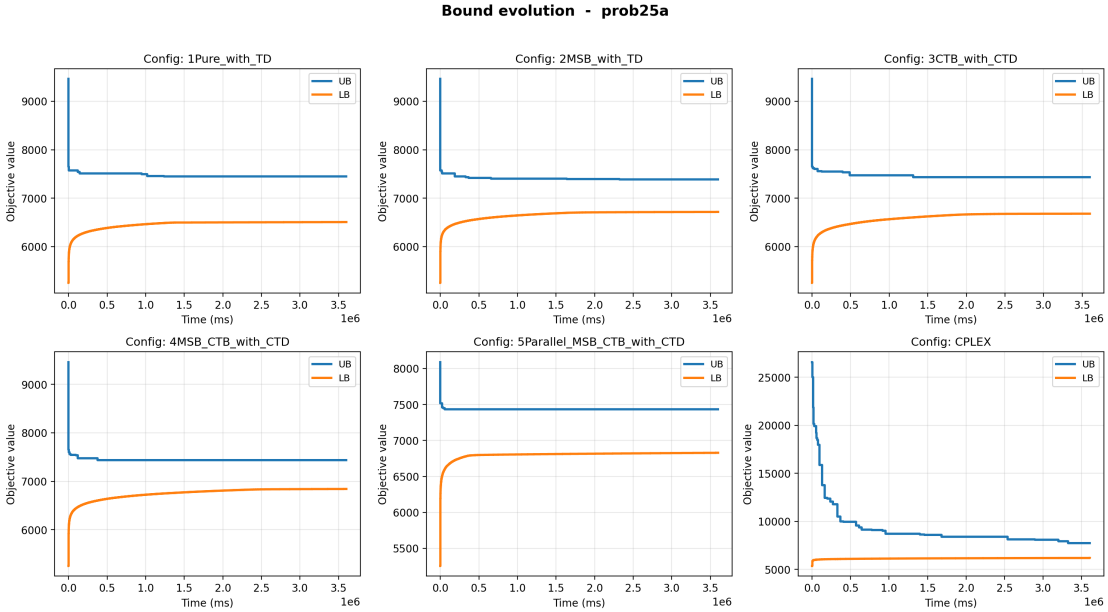


Fig. 9: Evolution of lower (orange) and upper (blue) bound in instance prob25a

C. Complete symbol reference

$\mathcal{R}$	set of all requests
n	number of requests
$\mathcal{L}$	set of all $\lvert\mathcal{L}\rvert$ locations
l_v	location corresponding to $v \in V$
L_v	subset of locations $\mathcal{L}$ — indicates (already) visited locations at state node v
$\overline{L_v}$	subset of locations $\mathcal{L}$ — indicates (yet) unvisited locations at state node v
$\overline{\mathcal{L}}(l)$	list of locations other than $l \in \mathcal{L}$ ascendingly sorted by distance to l
V	set of nodes in the state space-graph $G(V)$
V^{map}	data structure storing all nodes that are encountered during search
V^d	subset of nodes in the state space-graph $G(V)$ at depth d, such that $\forall v \in V^d : \lvert L_v\rvert = d$
π_u	set of predecessor nodes (incoming arcs) of $u \in V$ in state space-graph; It holds $\forall v \in \pi_u : \lvert L_v\rvert = \lvert L_u\rvert - 1$
σ_u	set of successor nodes (outgoing arcs) of $u \in V$ in state space-graph; It holds $\forall v \in \sigma_u : \lvert L_v\rvert = \lvert L_u\rvert + 1$
π_u^*	short for $\arg\min_{v\in\pi_u} \texttt{LB}^+(u) + \texttt{d}(v,u)$ with u being the last node in the search Path P
σ_u^*	short for $\arg\min_{v\in\sigma_u}(\texttt{d}(u,v) + \texttt{LB}^-(v))$, with u being the last node in the search Path P
$\texttt{TD}(u)$	Tightness-discontinuation predicate for some node u, true iff $\texttt{LB}^-(v) = \texttt{UB}^-(v)$
$\texttt{CTD}(u)$	complete-tour-discontinuation predicate, true iff $\texttt{UB}^+(u) + \texttt{LB}^-(u) \geq \texttt{UB}^-(0^+)$
$\texttt{MSB}(u,v)$	Minimizing-successor-breaking-predicate, true iff $\texttt{d}(u,v) + \texttt{LB}^-(\overline{L_u}) \geq \widehat{\texttt{LB}}^-(u)$
$\texttt{CTB}(u,v)$	complete-tour-breaking-predicate, true iff $\texttt{UB}^+(u) + \texttt{d}(u,v) + \texttt{LB}^-(\overline{L_u}) \geq \texttt{UB}^-(0^+)$

Tab. A6: Relevant symbols

$\mathtt{d}(l_1, l_2)$	cost of edge between l_1 and l_2 in the location graph L^2
$\mathtt{d}(u, v)$	cost of arc $(u, v) \in A$, matches $\mathtt{d}(l_u, l_v)$
$\mathtt{c}(v, w)$	Actual minimum cost required of getting from from state node v to state node w
$t^{\mathrm{OPT}}(v, w)$	Optimal, (partial) tour(s) from v to w with total cost of exactly $\mathtt{c}(v, w)$
t^{OPT}	Optimal complete tour
$\mathtt{LB}(v, w)$	Lower bound for the costs $\mathtt{c}(v, w)$ of getting from state node v to w
$\mathtt{UB}(v, w)$	Upper bound for the costs $\mathtt{c}(v, w)$ of getting from state node v to w
$\mathtt{LB}^-(\overline{L_u})$	Lower bound for the costs of getting from any $v \in \sigma_u, u$ to 0^- (preliminary lower bound)
$\mathtt{LB}^+(L_u)$	Lower bound for the costs of getting from 0^+ to $u \in \sigma_v, v$ to (preliminary lower bound)
$c^{\mathrm{MST}}(L)$	Total cost of a minimum spanning tree over all locations in L
$\mathtt{LB}^-(v)$	Short for $\mathtt{LB}(v, 0^-)$
$\widehat{\mathtt{LB}}^-(v)$	incumbent value for and equivalent to $\mathtt{LB}^-(v)$ at the end of an inner search iteration
$\mathtt{UB}^-(v)$	Short for $\mathtt{UB}(v, 0^-)$
$\mathtt{LB}^+(v)$	Short for $\mathtt{LB}(0^+, v)$
$\widehat{\mathtt{LB}}^+(v)$	incumbent value for and equivalent to $\mathtt{LB}^+(v)$ at the end of an inner search iteration
$\mathtt{UB}^+(v)$	Short for $\mathtt{UB}(0^+, v)$
$(\mathtt{d}(u, v) + \mathtt{LB}^-(v))$	Short for $\mathtt{d}(u, v) + \mathtt{LB}^-(v)$ with $v \in \sigma_u$
u^{INC}	snapshot of state node u when $\mathtt{UB}(0^+, 0^-)$ was decreased the last time
P_u^{INC}	snapshot of search path P when $\mathtt{UB}(0^+, 0^-)$ was decreased the last time

Tab. A7: Relevant cost properties

D. Usage of AI

The use of large language models (LLMs) for this report covers Python-scripts and isolated components of the code itself that are unspecific to the solving algorithm presented in this report. In total, LLMs were used in the creation of exactly the following files:

1. runbenchmark.py – script for executing the benchmarks deciding on the number of runs per instance to reach a precise estimate for runningtime
2. analyze.py – script for evaluating the runs benchmark instances afterwards, producing plots and LaTeX-tables as output
3. inc/Cost.h – a memory- and cost-efficient type for represent costs of arcs and search paths that is able to detect overflows and supports operator overloading
4. src/Instance.pp – reading of the input instances
5. misc/ShardedMap.h – implementation of both a static and a dynamic sharded map used for the parallel mode; The former is used for caching the costs of preliminary lower bounds, the latter for storing state nodes

To avoid interactions with the rest of the codebase, only web interfaces were used (lmarena.ai) and no AI-agents like Github Copilot or Claude Code were involved.