Bi-Objective Search for the Traveling Salesman Problem with Time Windows and Vacant Penalties

Shizhe Zhao[1], **Yancheng Wu**[1], **Zhongqiang Ren**[1,2*]

[1]UM-SJTU Joint Institute, Shanghai Jiao Tong University, China
[2]Department of Automation, Shanghai Jiao Tong University, China
{shizhe.zhao,wuyancheng,zhongqiang.ren}@sjtu.edu.cn

Abstract

This paper investigates a Traveling Salesman Problem with Time Windows and Vacant Penalties (TSP-TW-VP), which plans a path to service a set of machines at different locations within their respective time windows while minimizing two objective functions: the finish time and penalty for machine vacancy. There is often no single solution that optimizes both objectives simultaneously, and the problem thus seeks the Pareto-optimal solutions. TSP-TW-VP generalizes TSP-TW and is therefore NP-hard. To solve the problem, this paper develops an algorithm called Search with Look-Ahead Pruning (S-LAP) that is guaranteed to find all Pareto-optimal solutions for TSP-TW-VP. S-LAP gains computational efficiency by introducing a novel look-ahead pruning rule, and a fast dominance checking method based on both the objective functions and path history. Experimental results show that the proposed look-ahead pruning and fast dominance can speed up the search for 2-8 times over 4 different datasets.

Introduction

Given a complete graph, the Traveling Salesman Problem with Time Windows (TSP-TW) seeks a tour for an agent to service each machine located at a vertex of the graph within a pre-specified time window and return to the initial vertex (i.e., the depot), while minimizing the time to finish the tour, i.e., makespan. This paper investigates TSP-TW with Vacant Penalties (TSP-TW-VP), a variant of the TSP-TW, where both makespan and penalties are considered as objectives. Consider a mobile robot in a factory transporting parts to load the machines, each with a time window constraint. In addition to minimizing the makespan for the robot to service all the machines, an additional goal is to maintain high utilization rates of key machines, i.e., minimizing the total vacant time of these machines. These two objectives are often competitive to each other, as keeping high utilization rates of certain key machines may prolong the overall tour to service all the machines. Fig. 1 shows an example. In the presence of bi-objective, there is no single solution that optimizes both the objectives at the same time, and the problem instead seeks a set of Pareto-optimal solutions. A solution is Pareto-optimal if one cannot improve one objective without worsening another objective.

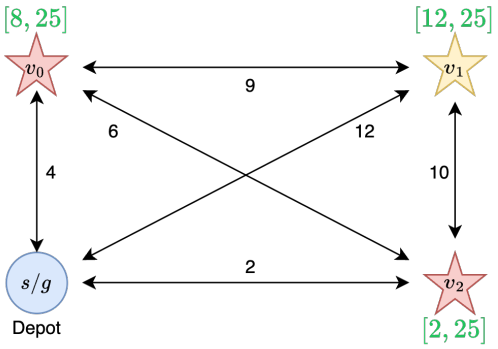


Figure 1: A toy example of TSP-TW-VP. The arrows with numbers indicate the minimum travel time between vertices. s (or g) indicates the depot where the robot must start and end at. v_0, v_1, and v_2 are vertices representing locations of machines that the robot needs to visit and serve. Each machine can only be served within a time window shown in green. For simplicity, assume the service time of each machine is 0. v_0 and v_2 are two key vertices where a penalty is incurred based on the starting time of the service. This instance has two Pareto-optimal solutions: $\langle s, v_2, v_0, v_1, g\rangle$ with makespan 29 and penalty 10; and $\langle s, v_2, v_1, v_0, g\rangle$ with makespan 25 and penalty 23, showing the trade-off between makespan and penalty.

When the set of key machines is empty, TSP-TW-VP becomes TSP-TW. TSP-TW is NP-complete (Savelsbergh 1985) and so is TSP-TW-VP. TSP-TW-VP is computationally challenging: On the one hand, TSP-TW-VP inherits the difficulty of TSP-TW about determining the servicing order of machine subject to the time window constraints. On the other hand, TSP-TW-VP suffers from large numbers of Pareto-optimal paths as in many bi-objective problems. In particular, with bi-objective, there exist multiple Pareto-optimal paths from the starting vertex to any other vertices and an algorithm has to properly maintain and compare these paths during the computation.

We are not aware of any existing work on TSP-TW-VP,

*Corresponding Author.
Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

so we start with a dynamic programming method for TSP-TW (Dumas et al. 1995). In particular, our algorithm S-LAP leverages our prior multi-objective search framework (Ren et al. 2025) and develop new techniques to expedite the search while guaranteeing find all Pareto-optimal solutions for TSP-TW-VP. S-LAP iteratively expands paths from the starting vertex to all other vertices in the graph until these paths service all machines and end at the goal location. During the search, the planner leverages the time window constraints and the path costs for early pruning of infeasible and unpromising paths using the notion of dominance (Cao et al. 2024; Zhao et al. 2025). Additionally, to address challenges in TSP-TW-VP, two new ideas in S-LAP are proposed. First, S-LAP introduces a new pruning rule that looks ahead to prune a path if there is a future path that serves more vertices without worsening the makespan, while ending with the same successor vertex during the expansion. This new look-ahead pruning rule is applicable to both TSP-TW and TSP-TW-VP when minimizing the makespan. Second, inspired by the recent progress in multi-objective path planning (Hernández et al. 2023; Ahmadi et al. 2021; Ren et al. 2022), S-LAP introduces a new fast dominance checking method based on both the cost vector and the path history (i.e., the set of vertices that are serviced along that path).

We test S-LAP on 4 different datasets with ablation study. The experimental results show that, the proposed look-ahead pruning and fast dominance can individually speed up the search for 2-4 times in different scenarios depending on the number of vertices and the time window constraints, and the two combined can speed up the search for 2-8 times.

Related Works

TSP is one of the most well-known NP-hard problems and has been extensively studied (Desrosiers et al. 1995). TSP-TW generalizes TSP by introducing the time window constraints at each vertex, within which that vertex must be serviced by the agent. Methods to solve TSP and TSP-TW can be roughly divided into exact (Fischetti, Salazar González, and Toth 1997), approximation (Garg, Konjevod, and Ravi 2000) and unbounded sub-optimal (Johnson and McGeoch 2003), and this work focuses on exact algorithms that are guaranteed to solve the problem to optimality. To solve TSP-TW to optimality, various approaches were developed such as integer programming (IP) (Langevin et al. 1993; Kara and Derya 2015) and dynamic programming (DP) (Dumas et al. 1995; Mingozzi, Bianco, and Ricciardelli 1997; Kuroiwa and Beck 2024), where DP is shown to often outperform IP (Langevin et al. 1993) since the existence of time windows allows DP to efficiently prune branches that will lead to infeasible tours (Dumas et al. 1995). Most TSP-TW algorithms minimize the tour length without incurring costs when the agent waits at vertices to satisfy the time window constraints (Dumas et al. 1995; Ascheuer, Fischetti, and Grötschel 2001; Pesant et al. 1998; Focacci, Lodi, and Milano 2002), and only a few algorithms minimize the makespan where the waiting time at vertices are also included into the tour cost (Christofides, Mingozzi, and Toth 1981; Baker 1983; López-Ibáñez et al. 2013). This work seeks to minimize the makespan as one of the objectives.

In the presence of multiple objectives, there are multiple non-dominated paths from the starting vertex to any other vertices in the graph, and efficiently maintaining and comparing these non-dominated paths (namely fast dominance checking) during the search is a decisive factor for search algorithms (Pulido, Mandow, and Pérez-de-la Cruz 2015). Fast dominance checking techniques received attention recently (Hernández et al. 2023; Ren et al. 2022; Mandow and Pérez de la Cruz 2023), but mainly for the multi-objective path planning (MOPP) problem that seeks a path from given start to goal. For TSPs, when comparing two paths that reach the same vertices, one has to compare not only the cost vector of the path, but also the path history (Cao et al. 2024), the set of vertices that are serviced along the path. This work leverages the fast dominance methods in MOPP (Pulido, Mandow, and Pérez-de-la Cruz 2015; Hernández et al. 2023) and further extends them to handle bi-objective TSPs.

Problem Formulation

Graph Let $G = (V, C)$ denote a complete graph where the vertex set V represents possible locations and cost matrix $C \in \mathbb{R}_{\geq 0}^{|V|^2}$ represents *minimum travel cost* between vertices. Each vertex $v \in V$ has a machine to be serviced by the robot, and let $s(v) \in \mathbb{R}^+$ denote the service time required by the robot at $v \in V$.

Constraints Machine at each vertex can only be served once. Serving a machine at v doesn't affect the graph G, i.e., the vertex v can *still be visited afterward*. Let $(t^a(v), t^b(v)) \in \mathbb{R}^+ \times \mathbb{R}^+$ denote the time window for a machine at vertex $v \in V$, within which the robot can service the machine. In other words, the robot must start and finish the service within the time window, which is referred to as the time window constraint. Time window constraints apply only to machine servicing, while vertices can be visited at any time.

Vacant Penalty Let $K \subseteq V$ denote the key machine set. Servicing any machine $v_i \in K$ at time t yields a penalty [1]$p(v_i) = t$. Intuitively, $p(v_i)$ indicates the vacancy time incurred at key machine v_i.

Path Let $\pi = \pi(v_1, v_l) = \{v_1, v_2, \cdots, v_l\}$ denote a path from vertex v_1 to v_l. Let $t^r(v)$ denote the makespan of v, i.e., the service finishing time of v. The service starting time at each $v_{i+1} \in \pi(v_1, v_l)$ is either the starting time of the time window at v_{i+1} or the arrival time, so the makespan of v_{i+1} is:

$$t^r(v_{i+1}) = \max\{t^r(v_i) + C(v_i, v_{i+1}), t^a(v_{i+1})\} + s(v_{i+1}). \tag{1}$$

Let $t^r(\pi)$ denote the makespan of the path, which is the same as the makespan of the last vertex in π, i.e., $t^r(\pi) = t^r(v_l)$. Let $p(\pi) = \sum_{v_i \in K \cap \pi} p(v_i)$ denote the penalty of the path π, and let $\vec{g}(\pi) := (t^r(\pi), p(\pi))$ denote the cost vector of π. A path is feasible if no time window constraint is violated.

[1] One can also define the penalty as the difference between the service starting time and the starting time of time window, i.e., $t - t^a(v_i)$. This doesn't change the solution as $t^a(v)$ is a constant for all key vertices $v \in K$.

Definition 1. (**Path Dominance**) Given any two non-identical paths $\pi_1(v_1, v_l)$ and $\pi_2(v_1, v_l)$, their corresponding cost vectors are $\vec{g_1} = (t^r(\pi_1), p(\pi_1))$ and $\vec{g_2} = (t^r(\pi_2), p(\pi_2))$ respectively. π_1 *strongly dominates* π_2 if $t^r(\pi_1) \leq t^r(\pi_2), p_1 \leq p_2$ and $\vec{g_1} \neq \vec{g_2}$, denoted as $\vec{g_1} \prec \vec{g_2}$ (or $\pi_1 \prec \pi_2$). Also, we call π_1 *weakly dominates* π_2 if $\vec{g_1} \prec \vec{g_2}$ or $\vec{g_1} = \vec{g_2}$, denoted as $\vec{g_1} \preceq \vec{g_2}$ (or $\pi_1 \preceq \pi_2$).

TSP-TW-VP Let v_o be the starting vertex and v_d be the ending vertex. Among all feasible paths from v_o to v_d, the non-dominated subset of paths is called the Pareto-optimal set. The goal of TSP-TW-VP is to find a maximal cost-unique Pareto-optimal set Π_*, where (i) each path $\pi(v_o, v_d) \in \Pi_*$ is Pareto-optimal, (ii) along each path $\pi(v_o, v_d) \in \Pi_*$, all vertices are serviced exactly once, and (iii) any two paths in Π_* have different cost vectors.

Method

Concepts and Notations

Definition 2. (**Label**) During the search, a label, defined by a tuple $l = (v, \vec{g}, B)$, represents a path $\pi_l = \pi(v_o, v)$, which can be reconstructed by iteratively following the parent labels. Here, $v \in V$ is the current vertex, $B \subseteq V$ is the set of serviced vertex along the path, including both key and non-key vertices that are serviced, which is also referred to as the *path history*. The vector $\vec{g} = (t^r(\pi_l), p(\pi_l))$ in a label is the cost vector the current path π_l. For convenience, in the remaining sections, we use $v(l), \vec{g}(l)$ and $B(l)$ to represent the $v, \vec{g}$ and B component of l respectively. A label l is a *target label* if $v(l) = v_d$ and $B(l) = V$. Labels l_1, l_2 are *identical* if $v(l_1) = v(l_2), \vec{g}(l_1) = \vec{g}(l_2)$, and $B(l_1) = B(l_2)$.

Definition 3. (**Lexical Order**) For any pair of two-dimensional cost vectors $\vec{g_1} = (t_1^r, p_1)$ and $\vec{g_2} = (t_2^r, p_2)$, $\vec{g_1}$ is lexicographically smaller than $\vec{g_2}$ if one of the following two conditions holds: (i) $t_1^r < t_2^r$ or (ii) $t_1^r = t_2^r \wedge p_1 < p_2$. In the case that $t_1^r = t_2^r$ and $p_1 = p_2$ (i.e., $\vec{g_1} = \vec{g_2}$), we break tie arbitrarily.

For a vector, we use bracket to indicate the component in the vector. E.g. $\vec{g}[1]$ means the first component in vector $\vec{g}$.

Definition 4. **(Transition)** Given a label $l = (u, \vec{g}, B)$, and a vertex v, a transition generates a new label $l' = (v, \vec{g'}, B \cup \{v\})$ from l with:

$$\vec{g'}[1] = t^r(v) = \max\{t^r(u) + C(u, v), t^a(v)\} + s(v)$$

$$\vec{g'}[2] = \begin{cases} \vec{g}[2], & \text{if } v \notin K \\ \vec{g}[2] + (t^r(v) - s(v)), & \text{if } v \in K \end{cases}$$

During the search, there can be multiple labels at the same vertex v, representing multiple path from v_o to v. To compare and prune these paths, we use the notion of dominance. The existing dominance pruning techniques in MOPP are not applied here since their dominance checking only considers the cost vector, while our work requires further considering the path history.

Definition 5. **(Label dominance)** Let $l_1 = (v, \vec{g_1}, B_1)$ and $l_2 = (v, \vec{g_2}, B_2)$ be two non-identical labels at the same vertex v, l_1 dominates l_2 if:

- $\vec{g_1} \prec \vec{g_2}$ and $B_2 \subseteq B_1$, or;
- $\vec{g_1} = \vec{g_2}$ and $B_2 \subset B_1$;

denoted as $l_1 \prec l_2$. Similarly, we call $l_1 \preceq$ than l_2 if $l_1 \prec l_2$ or $l_1 = l_2$.

Algorithm 1: Pseudocode for S-LAP

Require: v_o, v_d
1: $l_o \leftarrow (v_o, (0, 0), \emptyset)$
2: add l_o into OPEN
3: $\mathcal{F}_{trunc}(v) \leftarrow \emptyset, \forall v \in V$
4: $\mathcal{F}^* \leftarrow \emptyset$
5: **while** OPEN not empty **do**
6: $l \leftarrow$ pop from OPEN
7: **if** $IsDominated(l)$ **then**
8: **continue**
9: $\mathcal{F}_{trunc}(v(l)) \leftarrow UpdateFrontier(l)$
10: **if** $v(l) = v_d$ and $B(l) = V$ **then** ▷ l is a target label
11: $\mathcal{F}^* \leftarrow \mathcal{F}^* \cup \{l\}$
12: **continue**
13: **for** $i \in V \setminus B(l)$ **do**
14: $l' \leftarrow Transition(l, i)$ ▷ Def. 4
15: **if** $t^r(i) \geq t^b(i)$ **then**
16: **continue**
17: **if** $LookAhead(l')$ **then**
18: **continue** ▷ Def. 6
19: **if** $\exists j \in V/B(l')$ s.t. $t^r(i) + C(i, j) > t^b(j)$ **then**
20: **continue** ▷ PostCheck (Dumas et al. 1995)
21: parent(l') $\leftarrow l$
22: add l' into OPEN
23: **return** $BuildPath(\mathcal{F}^*)$ ▷ Build Π_* based on parent

Search with Look-Ahead Pruning (S-LAP)

We develop a best-first search approach as shown in Algorithm 1. The search employs an OPEN list that prioritizes labels by their $\vec{g}$ in non-decreasing lexical order. The search starts with an initial label at v_o (line 2), then repeatedly pop out labels to generate new labels by transition, until the OPEN is empty.

Multi-objective search often truncates the cost vectors for fast dominance comparison, which is explained later. Let $\mathcal{F}_{trunc}(v)$ and $\mathcal{F}^*$ denote the non-dominated truncated label set at vertex v and the Pareto-optimal solutions respectively, which are initialized to empty sets (line 3 and 4). In each iteration, *IsDominated* is applied to a popped out label l (line 7). This procedure determines whether l is dominated by existing frontiers at $v(l)$. Then *UpdateFrontier* adds the label l to $\mathcal{F}(v(l))$. This procedure filters existing labels in $\mathcal{F}(v(l))$ that are dominated by l, before adding l to $\mathcal{F}(v(l))$. More details about *IsDominated* and *UpdateFrontier* will be discussed in the next section.

For all generated labels, we ensure the time window constraints are satisfied (line 15). Then we perform pruning strategies *PostCheck* (line 19) and *LookAhead* (line 17). *Post Check* is an existing pruning technique (Dumas et al. 1995) to ensure that, for the newly generated label l', the future transition from l' to any unserviced vertices $u \notin B(l')$ satisfies the time window constraint at u. This simple checking procedure is quick to conduct and was shown to be able to speed up the search (Dumas et al. 1995), which is leveraged

Algorithm 2: *IsDominated* and *UpdateFrontier*

```
1: function IsDominated(l)
2:     for all i ∈ F_trunc(v(l)) do
3:         if g⃗[2](i) ≤ g⃗[2](l) and B(l) ⊂ B(i) then
4:             return True
5:     return False
6: procedure UpdateFrontier(l)
7:     filtered ← ∅
8:     for all i ∈ F_trunc(v(l)) do
9:         if g⃗[2](l) ≤ g⃗[2](i) and B(i) ⊂ B(l) then
10:            filtered ← filtered ∪ {i}
11:    F_trunc(v(l)) ← (F_trunc(v(l)) ∪ {l} \ filtered)
12:    return F_trunc(v(l))
```

in S-LAP. *LookAhead* is a new pruning technique proposed by this work, which will be discussed shortly.

Fast Dominance Checking

The size of $\mathcal{F}(v)$ for any $v \in V$ may grow dramatically during the search, causing costly dominance checking. To address this problem, we adapted two techniques from the existing works to our problem setting:

- Lazy dominance check (Hernández et al. 2023): instead of eagerly running *IsDominated* after a new label is generated (Alg. 1, line 22), we run dominance checking after a label is popped out (Alg. 1, line 6).
- Dimensionality reduction (DR) (Pulido, Mandow, and Pérez-de-la Cruz 2015): given the fact that labels are popped out in the lexical ordering, the first dimension of cost vectors $\vec{g}(l)$ of labels l that are expanded at the same vertex $v(l)$ must be *monotonically non-decreasing*. Thus this dimension can be ignored in dominance checking. Without the first component (i.e., makespan $t^r(\pi)$), the resulting truncated cost vector $\vec{g}_{trunc}$ becomes a scalar (i.e., $p(\pi)$). Consequently, for dominance checking, instead of iterating all vectors in $\mathcal{F}(v)$, only a subset $\mathcal{F}_{trunc}$ (which consists of non-dominated truncated vectors) of $\mathcal{F}$ (which contains the non-dominated original vectors) needs to be stored. $\mathcal{F}_{trunc}$ is often much smaller than $\mathcal{F}$, which thus expedites the dominance checking.

Alg. 2 shows how we maintain $\mathcal{F}_{trunc}$ with the help of DR. In addition, the path history of two labels must be properly compared using the aforementioned label dominance. The for-loops in *IsDominated* and *UpdateFrontier* (line 2 and 8) gets benefit from $\mathcal{F}_{trunc}$ due to its smaller size. The label l in *UpdateFrontier* is guaranteed to be non-dominated (Alg. 1, line 9), and we use it to further filter existing labels that are dominated by l before adding it to $\mathcal{F}_{trunc}$ (line 11).

Now we illustrate how the DR speeds up the dominance checking in our problem. Consider an instance as shown in Fig. 2a. When $K = \emptyset$, and the search has generated set of labels at v_1:

- $l_1 = (v_1, \vec{g_1} = (t^a(v_1), 0), B_1 = \{v_o, v_1\})$;
- $l_2 = (v_1, \vec{g_2} = (t^a(v_2), 0), B_2 = \{v_o, v_2, v_1\})$;
- $\cdots$;
- $l_n = (v_1, \vec{g_n} = (t^n(v_n), 0), B_n = \{v_o, v_n, \cdots, v_1\})$

The labels are popped out in the order of $\vec{g_i}$, i.e., $l_1, l_2, \cdots, l_n$. Note that even $\vec{g_i} \prec \vec{g_j}$ when $i < j$, $l_i \not\prec l_j$ as $B_i \subset B_j$. Consequently, without DR, we have to compare the popped out label against all of them in $\mathcal{F}(v_1)$. With the DR, $UpdateFrontier(l_1)$ gives $\mathcal{F}_{trunc}(v_1) = \{l_1\}$. Then $UpdateFrontier(l_2)$ removes l_1 and adds l_2, etc. After the procedure $UpdateFrontier(l_n)$, $\mathcal{F}_{trunc}(v_1)$ stores only the label l_n. In all procedures, only one label is compared.

Remark 1. At Alg. 2 line 2 and 8, we can use data structure like AVL-Tree to further speed up the checking and filtering (Ren et al. 2022). Due to the overhead of the data structure, a linear search is efficient enough to process $\mathcal{F}_{trunc}$ in several existing datasets for TSP-TW-VP.

Remark 2. The DR technique poses requirements on the f-vector of labels, which makes the heuristic design challenging, and we therefore do not use any heuristic in this work. The following example illustrates the challenge. Given a set of vertices $V = \{S, 0, 1, \cdots, N\}$, let $v_o = v_d = S$. Assume travel cost $C(S, 0) = C(0, 1) = 1$, and there is no time window constraint. Consider two labels generated at the vertex 0 during the search (assuming no penalty is applied at 0):

- $l_1 = (v = 0, \vec{g} = [1, 0], B = \{0\})$, representing a tour from S to 0;
- $l_2 = (v = 0, \vec{g} = [2, 0], B = \{0, 1\})$, representing a tour from S to 1 then 0;

Since l_1 and l_2 do not dominate each other (Def. 5), both should be expanded. When the DR is applied, l_1 must be expanded before l_2. Otherwise, since $Trunc(l_2) = (\vec{g} = [0], B = \{0, 1\}))$ dominates $Trunc(l_1) = (\vec{g} = [0], B = \{0\})$, l_1 will be pruned. Assume an admissible heuristic is applied, and let $h_1 = [3, 0]$, and $h_2 = [1, 0]$ denote the underestimated heuristic values of l_1 and l_2 respectively. Then we have $f(l_1) = [1 + 3, 0], f(l_2) = [2 + 1, 0]$. When labels are prioritized by the lexical order of their f-vectors, l_1 will be expanded after l_2, meaning that l_1 will be pruned. To ensure l_1 is expanded earlier, the heuristic function needs additional property besides the admissibility, possibly by considering the path history B, which is non-trivial.

Look Ahead Pruning (LAP)

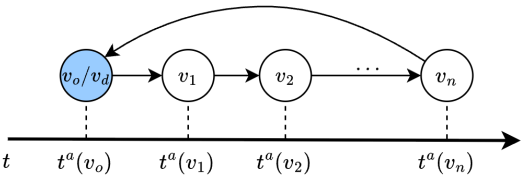
When generating a label l' from l, LAP checks whether the newly generated label l' is dominated (Def. 5) by another label l'' that may not be generated. If so, l' will be not be generated. More specifically:

Definition 6. (Look Ahead) Given a label $l_u = (u, \vec{g_u}, B_u)$, and its successor $l_v = (v, \vec{g_v}, B_v)$, where $v \notin B_u$ and $B_v = B_u \cup \{v\}$. Then l_v can be pruned if $\exists m \in V \setminus B_u$, s.t.,

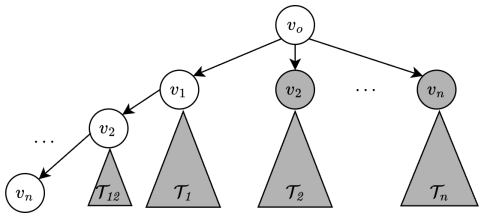
$$\begin{aligned} &\max\{t^r(u) + C(u, m), t^a(m)\} \\ &+s(m) + C(m, v) \leq t^a(v) \end{aligned} \tag{2}$$

Intuitively, it prunes a path where the agent waits at certain vertex while there is enough time to make a detour to service more vertices. We will prove the correctness of the pruning at the end of this section.

Now, we illustrate how LAP effectively reduces the search space. Consider a graph as shown in Fig. 2a. The graph has a symmetric cost matrix for the minimum travel cost. The



(a) The graph has a symmetric cost matrix for the minimum travel cost. For each vertex, the starting time of its time window is shown in t-axis, and the end time is infinite. Arrows indicate the optimal path of this instance when $K = \emptyset$.



(b) The search tree of instance in Fig. 2a. Triangles represent subtree of a search node (i.e., label). Gray objects are pruned by LAP.

Figure 2: An example showcasing LAP.

costs of the edges are small enough so that one can always start service at v with starting time $t^a(v)$, and $t^a(v_o) < t^a(v_1) < \cdots < t^a(v_n) < t^a(v_d)$.

LAP for TSP-TW Assuming there is no key nodes, i.e., $K = \emptyset$, and the service time at each vertex is simply zero. Here, the optimal tour is $\pi = (v_o, v_1, \ldots, v_n, v_d)$. Fig. 2b shows the search tree of this instance, where all gray nodes $(v_2, \cdots, v_n)$ and subtrees $(\mathcal{T}_{12}, \mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_n)$ can be pruned by LAP. For example, $v_o \to v_2$ can be pruned as $\max\{t^r(v_o) + C(s, v_1), t^a(v_1)\} + C(v_1, v_2) \leq t^a(v_2)$, so as its subtree $\mathcal{T}_2$. With the help of LAP, we only need to generate n labels while the entire search space is $O(n!)$.

LAP for TSP-TW-VP In the example above, the search space can also be significantly reduced by dominance checking, given that the penalty dimension is always 0 (i.e., $K = \emptyset$). However, dominance checking becomes less effective when key nodes are present. For example, let $K = \{v_1\}$, and label l_{s12}, l_{s2} represent the partial path $\{v_o, v_1, v_2\}, \{v_o, v_2\}$ respectively. For l_{s12}, the penalty at v_1 is $t^a(v_1)$, so

$$l_{s12} = (v_2, (t^a(v_2), t^a(v_1)), \{v_o, v_1, v_2\}).$$

Then $l_{s2} = (v_2, (t^a(v_2), 0), \{v_o, v_2\})$ would be generated if LAP is not applied, because neither can dominate the other according to Def. 5. This demonstrates that LAP can be more effective in certain multi-objective settings. The result section further discusses the scenarios where LAP or fast dominance checking expedites the search.

Remark 3. Or-interchanges (Savelsbergh 1985) in local search is similar to LAP. The difference is that Or-interchanges is applied on an initial feasible solution to improve it; while LAP is applied on a partial solution to prune successors using dominance checking.

Theoretical Analysis

We first introduce *redundant transition* for the convenience of analysis.

Definition 7. (**Redundant Transition**) For a label $l = (v, \vec{g} = (t^r(v), p), B)$, a transition from l to $i \in B$ is redundant if it generates a label at a previously serviced vertex:

$$l_i = (i, \vec{g'} = (t^r(v) + C(v, i) + w, p), B)$$

where $w \geq 0$ indicates the waiting time before starting the transition.

Clearly, any labels generated by a redundant transition can not lead to a Pareto-optimal solution. We will prove a label is unpromising by showing it is dominated by another label that is generated by a redundant transition.

Lemma 1. Pruning a dominated label would never eliminate a Pareto-optimal solution.

Proof. Let $l_1 = (v, \vec{g}_1, B_1)$, $l_2 = (v, \vec{g}_2, B_2)$ and $l_1 \prec l_2$. Consider any transition from l_2 to $i \in V \setminus B_2$ that generates

$$l_2^i = (i, \vec{g}_2^i = (t_2^r(i), p_2^i), B_2 \cup \{i\}).$$

We can apply the same transition on l_1 and an additional redundant transition to generate a label with the same makespan as l_2^i:

$$l_1^i = (i, \vec{g}_1^i = (t_2^r(i), p_1^i), B_1 \cup \{i\})$$

We show that $l_1^i \preceq l_2^i$ always hold for all possible cases: (i) $i \in B_1$; (ii) $i \notin B_1$ and $i \notin K$; (iii) $i \notin B_1$ and $i \in K$.

In case (i), i has been serviced, so the penalty p_1^i remains unchanged. Then we have $p_1^i = p(l_1) \leq p_2^i$ and $B(l_2^i) \subseteq B(l_1^i) = B(l_1)$, meaning that $l_1^i \preceq l_2^i$;

In case (ii), i is an unserviced vertex but not a key node, so the penalty is unchanged. Then we have, $p_1^i = p(l_1) \leq p_2^i$ and $B(l_2^i) \subseteq B(l_1^i) = B(l_1) \cup \{i\}$, meaning that $l_1^i \preceq l_2^i$;

In case (iii), i is an unserviced key node, same penalty would be applied on both l_1^i and l_2^i. So we have $p(l_1) + \triangle p \leq p(l_2) + \triangle p$ and $B(l_2^i) \subseteq B(l_1^i) = B(l_1) \cup \{i\}$, meaning that $l_1^i \preceq l_2^i$.

Therefore, for any successor l' of l_2, we can always generate a label from l_1 that is no worse than l'. So pruning l_2 would never eliminate any Pareto-optimal solution. □

Lemma 2. Look Ahead Pruning never eliminates any Pareto-optimal solution

Proof. Let $Trans(l, v)$ denote the transition from label l to vertex v. Given a label $l_u = (u, \vec{g_u}, B_u)$, let l_v denote the label generated by $Trans(l_u, v)$, and l_{mv} denote the label generated by $Trans(Trans(l_u, m), v)$, where l_u, l_v, m satisfy Eq. (2). There are two cases: (i) $m \notin K$, and; (ii) $m \in K$. In the first case, $\vec{g}(l_v) = \vec{g}(l_{mv})$ and $B(l_v) \subset B(l_{mv})$, so $l_{mv} \prec l_v$. In the second case, for arbitrary sequence of transitions from l_v that reaches m, denoted as l_v^m, we can apply the exactly same transition sequence on l_{mv}, generating l_{mv}^m. Recall that m has been serviced from l_{mv}, so that the last transition of l_{mv}^m is a redundant transition that does not increase the penalty. Then, we have $t^r(l_v^m) = t^r(l_{mv}^m)$ (Eq. (2)), $B(l_v^m) = B(l_{mv}^m)$, and $p(l_v^m) > p(l_{mv}^m)$, so

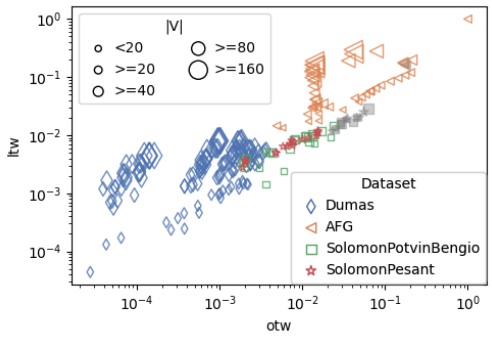


Figure 3: All instances from the four datasets. The shape and color represents the dataset, the size of markers indicates the number of nodes in an instance, and gray markers are instances that S-LAP are unable to solve within 300s when $k = 0$.

$l^m_{mv} \prec l^m_v$. Therefore, any successor of l_v that reaches m is dominated. With Lemma 1, LAP never prunes a Pareto-optimal solution. □

Theorem 1. S-LAP can find all Pareto-optimal solutions.

Proof. Without DR and LAP, Alg. 1 is a regular multi-objective search algorithm that systematically enumerates all non-dominated paths from v_o to v_d while servicing all vertices, and is complete and optimal (Hernández et al. 2023; Ren et al. 2022). During the search, S-LAP only prunes dominated labels which cannot lead to Pareto-optimal solutions (Lemma 1), and LAP never eliminates Pareto-optimal solutions (Lemma 2), so S-LAP can find all Pareto-optimal solutions. □

Remark 4. Our method is not restricted to a complete graph. For any finite graph, since servicing machines do not affect the traversability of vertices, we can always precompute the all-pair shortest distance. This preprocessing allows our search to focus on scheduling (i.e., "which machine to serve next") while ignoring path planning (i.e., "how to move from u to v"). Additionally, since travel cost and service time (C and s) always appear together, many existing works, e.g., (López-Ibáñez and Blum 2010), incorporate the service time $s(v)$ into the cost matrix by shortening $t^b(v)$ by $s(v)$ and increasing the cost of outgoing edge by $s(v)$. In this work, we denote the travel cost and service time separately, as revisiting a serviced vertex is allowed (Def. 7).

Numerical Results

We tested S-LAP on four datasets: *AFG* (Ascheuer 1996), *Dumas* (Dumas et al. 1995), *SolomonPesant* (Pesant et al. 1998) and *SolomonPotvinBengio* (Potvin and Bengio 1996). *Dumas* has 135 symmetric instances (i.e., $C(i,j) = C(j,i)$) and is widely used in TSP-TW. *AFG* has 50 asymmetric instances (i.e., $C(i,j)$ may not be the same as $C(j,i)$). *SolomonPotvinBengio* and *SolomonPesant* have 57 asymmetric instances in total, and are very diverse in transition cost, time window tightness, and positioning of the time windows (Solomon 1987).

Variant	#Sol.	Time (min)	#Gen. (10^8)
Base	1258	122.7	24.77
FD	1266	52.0 (↓ 57%)	25.34 (↑ 2.3%)
LAP	1267	93.9 (↓ 23%)	4.57 (↓ 81.5%)
FD+LAP	1278	26.8 (↓ 78%)	4.83 (↓ 80.5%)

Table 1: Summary of all instances for all ks, there are 1446 instances in total (241 per k). The table shows the number of solved instances (*#Sol*), total runtime (*Time*), and total label generation (*#Gen.*). The differences to *Base* are shown in parenthesis.

To generate bi-objective instances, for a given k, we randomly select $k\%$ of nodes from the input as key nodes. In all experiments, we vary $k \in [0, 20, \cdots, 100]$ under a fixed random seed, where $k = 0$ means solving the classic TSP-TW that only minimizes makespan.

Two properties of time windows may affect the difficulty of a TSP-TW instance for a search algorithm: (i) the length of time windows (*ltw*) and (ii) the overlapped length of time windows (*otw*). The higher the *ltw*, the looser the time window constraints are. Looser time windows make it hard for a search algorithm to prune based on the time window constraints, and often increase the search space. For *otw*, consider an extreme instance that has no overlapped time window, LAP would ensure all non-key vertices are serviced in the order of their starting times, significantly reducing the search space. So we anticipate that instances with lower *otw* are easier. To estimate the difficulty of an instance, we measure the *ltw* and *otw* as follows:

- *ltw*: the mean of time window length divided by the mean of edge length.
- *otw*: the mean of overlapped time window length divided by the mean of edge length.

Fig. 3 shows all instances based on their *ltw* and *otw*. For visualization purposes, we apply the min-max normalization to map *ltw* and *otw* between 0 and 1.

We implemented S-LAP in C++ compiled with `-O3` flag. All experiments were run on a desktop with a 16-core i7-13700 CPU and 32GB RAM on Ubuntu 22.04. The runtime budget is 300 seconds per instance.

Ablation Study

In this experiment, we vary k, and examine the effectiveness of Fast dominance and Look ahead pruning in S-LAP. The following variants are derived from S-LAP:

- *FD*: apply the Fast dominance only;
- *LAP*: apply the Look ahead pruning only;
- *Base*: without neither Fast dominance nor Look ahead pruning;

For clarity, we denote S-LAP as *FD+LAP*.

Table 1 shows FD+LAP obviously reduces the total runtime and the total label generation over all instances. FD has

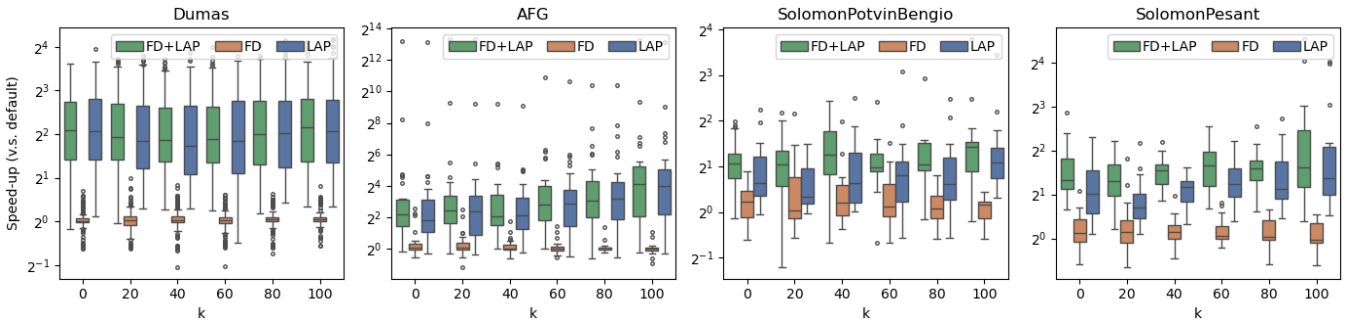
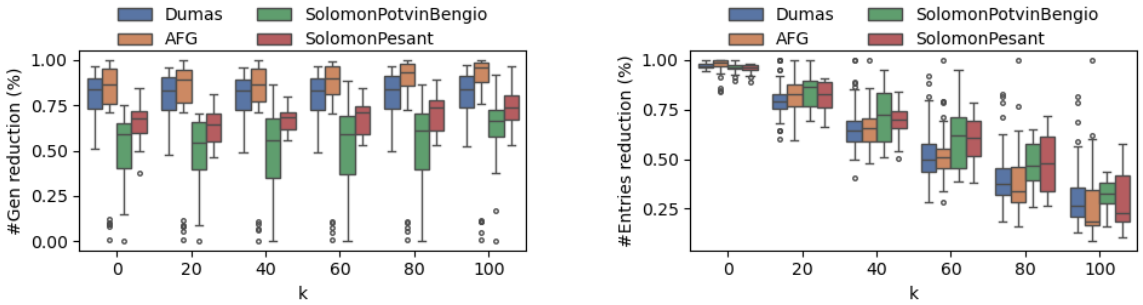


Figure 4: Speed-up factors compared to *Base* on fast instances (runtime < 1s).



(a) (b)

Figure 5: (a) Reduction of *FD+LAP* on generated labels compared to *Base* on fast instances (runtime < 1s); (b) Among fast instances of *FD+LAP*, #entries reduction is $1.0 - (|\mathcal{F}_{trunc}|/|\mathcal{F}|)$, indicating the reduced comparisons in dominance checking.

generated slightly more labels than Base, even though FD doesn't impact label generation. The reason is that FD has solved more instances, which generates more labels. For the same reason, FD+LAP has more labels than LAP.

Now we demonstrate how each components impacts the performance on different k values. We define the speed-up factor of a variant (i.e., *FD,LAP,FD+LAP*) as the runtime of *Base* divided by the runtime of the respective variant. The speed-up distribution among *fast instances* (runtime $<$ 1s) and *slow instances* (runtime $\geq$ 1s) differs, hence we discuss the results in separate.

Fig. 4 shows the distribution of speed-up for *fast instances* compared to *Base*, and Fig. 5 shows the reduction of *FD+LAP* on label generation and frontier size. LAP is the main reason for the speed-up, and the overall trend of k follows the label generation reduction as shown in Fig. 5a. This indicates the main benefit of these instances stem from the reduction in labels. The reason is $\mathcal{F}$ is often small and the overhead on dominance checking has minor influence.

Fig. 6 shows the distribution of speed-up for *slow instances* compared to *Base*, and Fig. 7 shows the reduction of *FD+LAP* on label generation and frontier size. We can see that FD becomes the primary contributor, excluding Dumas. The influence of FD diminishes as k increases, while the impact of LAP intensifies. This trend is consistent with #entries reduction of $\mathcal{F}$ as shown in Fig. 7b. This implies that the significant component of runtime is dominance checking. This outcome is due to the generation of more labels in these instances, which increases the size of $\mathcal{F}$. Additionally, as k increases, labels are less likely be filtered (Alg 2, line 10) because the penalty becomes more complicated as the number of key vertices increases.

Compare with TSP-TW Solvers

In this experiment, we fix $k = 0$, where TSP-TW-VP degenerates to TSP-TW and the goal is to minimize the makespan of the tour. We compare our S-LAP against two recent state-of-the-art solvers that can minimize makespan (as opposed to minimizing path length) for TSP-TW (denoted as TSP-TW-makespan problem):

- *Peel-and-Bound (PnB)* (Rudich, Cappart, and Rousseau 2023). PnB is based on the decision diagram. It can generate stronger bound compared to the *branch-and-bound* methods, and outperforms them in terms of running speed.
- *ACS+* (Fontaine, Dibangoye, and Solnon 2023). ACS+ is a variant of A* and combines a variety of techniques, including constraint propagation and local search.

We sort the instances by their runtime from the minimum to the maximum, and then show the cumulative runtime of the instances in Fig. 8. Initially, S-LAP is orders of magnitude faster than its competitors in solving the same number of instances, which demonstrates the runtime benefit of employing LAP and FD in the search.

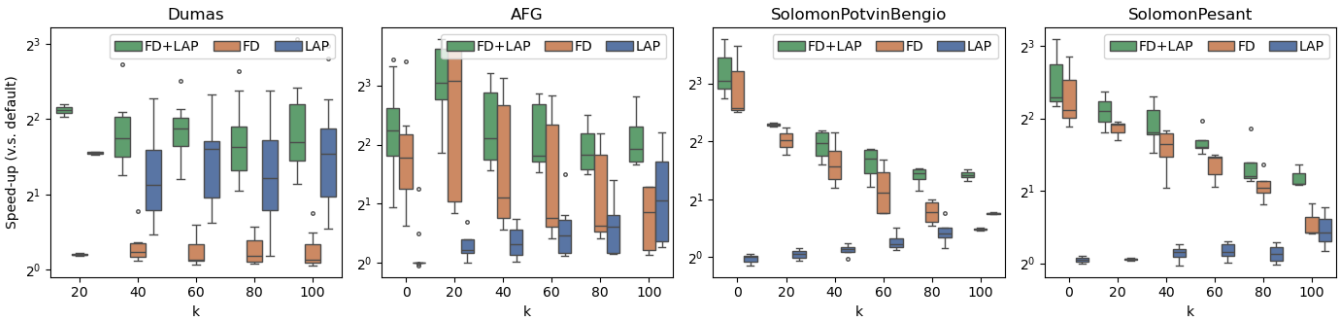
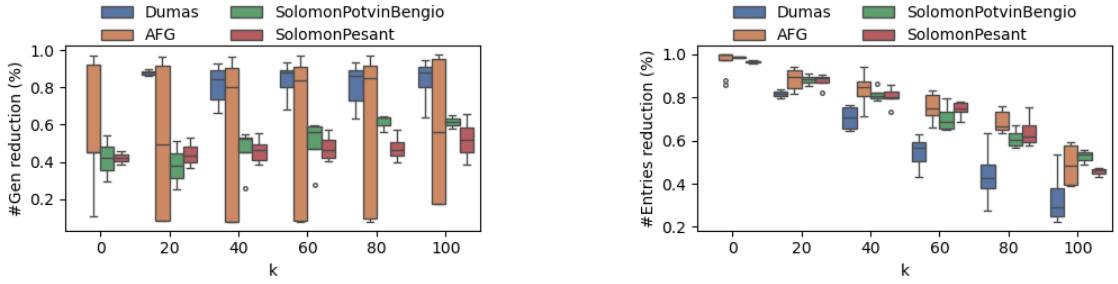


Figure 6: Speed-up factors compared to *Base* on slow instances (runtime $\geq$ 1s)



(a) (b)

Figure 7: (a) Reduction of *FD+LAP* on generated labels and entries compared to *Base* on fast instances (runtime $\geq$ 1s); (b) Among slow instances of *FD+LAP*, #entries reduction is $1.0 - |\mathcal{F}_{trunc}|/|\mathcal{F}|$, indicating the reduced comparisons in dominance checking.

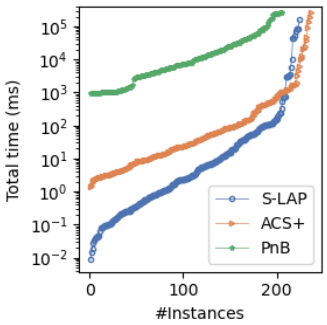


Figure 8: Compare with SOTA solvers on the TSP-TW-makespan. Given 300s runtime budget, S-LAP, PnB and ACS+ solved 223, 206 and 236 instances out of 241, respectively.

However, as the number of instances increases, the advantage of S-LAP diminishes and it is eventually outperformed by *ACS+*. Furthermore, S-LAP solved less number of instances than *ACS+* (223 v.s. 236). When the time budget was extended to 1 hour, it was observed that both competitors could solve all 241 instances, while S-LAP still failed on the same ones. This may due to the lack of heuristic guidance, and the lack of additional techniques such as constraint propagation and local search as in ACS+, which leads the search in S-LAP explore the more search space. This observation also points out the possible future work directions.

Conclusion

This work studies a new problem TSP-TW-VP. The proposed method, S-LAP, adapts the dimensionality reduction technique from MOPP to reduce the overhead on dominance checking, and employs a new pruning strategy called look-ahead pruning to avoid generating unpromising labels during the search. Experimental results show that the ideas of fast dominance checking and look ahead pruning in S-LAP can reduce runtime up to 78% and memory consumption (measured by the number of generated labels) up to 80% for instances in several datasets. In the TSP-TW problem (i.e., when there is no key vertices in TSP-TW-VP), compared to the recent state-of-the-art solvers, S-LAP has smaller runtime on solved instances, but still needs further investigation to handle hard instances.

Acknowledgements

This work was supported by the Natural Science Foundation of Shanghai under Grant 24ZR1435900, and the Natural Science Foundation of China under Grant 62403313.

References

Ahmadi, S.; Tack, G.; Harabor, D. D.; and Kilby, P. 2021. Bi-objective Search with Bi-directional A*. In *Proceedings*

of the International Symposium on Combinatorial Search, volume 12, 142–144.

Ascheuer, N. 1996. *Hamiltonian path problems in the online optimization of flexible manufacturing systems*. Ph.D. thesis.

Ascheuer, N.; Fischetti, M.; and Grötschel, M. 2001. Solving the asymmetric travelling salesman problem with time windows by branch-and-cut. *Mathematical programming*, 90: 475–506.

Baker, E. K. 1983. An exact algorithm for the time-constrained traveling salesman problem. *Operations Research*, 31(5): 938–945.

Cao, C.; Xu, J.; Zhang, J.; Choset, H.; and Ren, Z. 2024. Heuristic Search for the Orienteering Problem with Time-Varying Reward. In *Proceedings of the International Symposium on Combinatorial Search*, volume 17, 11–19.

Christofides, N.; Mingozzi, A.; and Toth, P. 1981. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2): 145–164.

Desrosiers, J.; Dumas, Y.; Solomon, M. M.; and Soumis, F. 1995. Chapter 2 Time constrained routing and scheduling. In *Network Routing*, volume 8 of *Handbooks in Operations Research and Management Science*, 35–139. Elsevier.

Dumas, Y.; Desrosiers, J.; Gelinas, E.; and Solomon, M. M. 1995. An optimal algorithm for the traveling salesman problem with time windows. *Operations research*, 43(2): 367–371.

Fischetti, M.; Salazar González, J. J.; and Toth, P. 1997. A branch-and-cut algorithm for the symmetric generalized traveling salesman problem. *Operations Research*, 45(3): 378–394.

Focacci, F.; Lodi, A.; and Milano, M. 2002. A hybrid exact algorithm for the TSPTW. *INFORMS journal on Computing*, 14(4): 403–417.

Fontaine, R.; Dibangoye, J.; and Solnon, C. 2023. Exact and anytime approach for solving the time dependent traveling salesman problem with time windows. *Eur. J. Oper. Res.*, 311(3): 833–844.

Garg, N.; Konjevod, G.; and Ravi, R. 2000. A polylogarithmic approximation algorithm for the group Steiner tree problem. *Journal of Algorithms*, 37(1): 66–84.

Hernández, C.; Yeoh, W.; Baier, J. A.; Zhang, H.; Suazo, L.; Koenig, S.; and Salzman, O. 2023. Simple and efficient bi-objective search algorithms via fast dominance checks. *Artificial intelligence*, 314: 103807.

Johnson, D. S.; and McGeoch, L. A. 2003. *8. The traveling salesman problem: a case study*, 215–310. Princeton: Princeton University Press. ISBN 9780691187563.

Kara, I.; and Derya, T. 2015. Formulations for minimizing tour duration of the traveling salesman problem with time windows. *Procedia Economics and Finance*, 26: 1026–1034.

Kuroiwa, R.; and Beck, J. C. 2024. Parallel Beam Search Algorithms for Domain-Independent Dynamic Programming. 20743–20750. AAAI Press.

Langevin, A.; Desrochers, M.; Desrosiers, J.; Gélinas, S.; and Soumis, F. 1993. A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows. *Networks*, 23(7): 631–640.

López-Ibáñez, M.; and Blum, C. 2010. Beam-ACO for the travelling salesman problem with time windows. *Comput. Oper. Res.*, 37(9): 1570–1583.

López-Ibáñez, M.; Blum, C.; Ohlmann, J. W.; and Thomas, B. W. 2013. The travelling salesman problem with time windows: Adapting algorithms from travel-time to makespan optimization. *Appl. Soft Comput.*, 13(9): 3806–3815.

Mandow, L.; and Pérez de la Cruz, J.-L. 2023. Improving Bi-Objective Shortest Path Search with Early Pruning. *ECAI 2023*, 1680–1687.

Mingozzi, A.; Bianco, L.; and Ricciardelli, S. 1997. Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. *Operations research*, 45(3): 365–377.

Pesant, G.; Gendreau, M.; Potvin, J.-Y.; and Rousseau, J.-M. 1998. An exact constraint logic programming algorithm for the traveling salesman problem with time windows. *Transportation Science*, 32(1): 12–29.

Potvin, J.-Y.; and Bengio, S. 1996. The vehicle routing problem with time windows part II: genetic search. *INFORMS journal on Computing*, 8(2): 165–172.

Pulido, F.-J.; Mandow, L.; and Pérez-de-la Cruz, J.-L. 2015. Dimensionality reduction in multiobjective shortest path search. *Computers & Operations Research*, 64: 60–70.

Ren, Z.; Hernández, C.; Likhachev, M.; Felner, A.; Koenig, S.; Salzman, O.; Rathinam, S.; and Choset, H. 2025. EMOA*: A framework for search-based multi-objective path planning. *Artificial Intelligence*, 339: 104260.

Ren, Z.; Zhan, R.; Rathinam, S.; Likhachev, M.; and Choset, H. 2022. Enhanced multi-objective A* using balanced binary search trees. In *Proceedings of the International Symposium on Combinatorial Search*, volume 15, 162–170.

Rudich, I.; Cappart, Q.; and Rousseau, L. 2023. Improved Peel-and-Bound: Methods for Generating Dual Bounds with Multivalued Decision Diagrams. *J. Artif. Intell. Res.*, 77: 1489–1538.

Savelsbergh, M. W. 1985. Local search in routing problems with time windows. *Annals of Operations research*, 4: 285–305.

Solomon, M. M. 1987. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2): 254–265.

Zhao, S.; Nandy, A.; Choset, H.; Rathinam, S.; and Ren, Z. 2025. Heuristic Search for Path Finding With Refuelling. *IEEE Robotics and Automation Letters*, 10(4): 3230–3237.