Received: 29 June 2023 | Revised: 13 October 2023 | Accepted: 2 January 2024

DOI: 10.1002/net.22210

RESEARCH ARTICLE

WILEY

Enhanced methods for the weight constrained shortest path problem

Saman Ahmadi[1] | **Guido Tack**[2] | **Daniel Harabor**[2] | **Philip Kilby**[3] | **Mahdi Jalili**[1]

[1]School of Engineering, RMIT University, Melbourne, Victoria, Australia
[2]Department of Data Science and Artificial Intelligence, Monash University, Melbourne, Victoria, Australia
[3]CSIRO Data61, Canberra, Australian Capital Territory, Australia

Correspondence
Saman Ahmadi, School of Engineering, RMIT University, Australia.
Email: saman.ahmadi@rmit.edu.au

Funding information
Australian Research Council, Grant/Award Numbers: DP190100013, DP200100025; Victorian Government.

Abstract

The classic problem of *constrained pathfinding* is a well-studied, yet challenging, network optimization problem with a broad range of applications in various areas such as communication and transportation. The weight constrained shortest path problem (WCSPP), the base form of constrained pathfinding with only one side constraint, aims to plan a cost-optimum path with limited weight/resource usage. Given the bi-criteria nature of the problem (i.e., dealing with the cost and weight of paths), methods addressing the WCSPP have some common properties with bi-objective search. This article leverages the recent state-of-the-art techniques in both constrained pathfinding and bi-objective search and presents two new solution approaches to the WCSPP on the basis of A* search, both capable of solving hard WCSPP instances on very large graphs. We empirically evaluate the performance of our algorithms on a set of large and realistic problem instances and show their advantages over the state-of-the-art algorithms in both time and space metrics. This article also investigates the importance of priority queues in constrained search with A*. We show with extensive experiments on both realistic and randomized graphs how bucket-based queues without tie-breaking can effectively improve the algorithmic performance of exhaustive A*-based bi-criteria searches.

KEYWORDS

bi-objective shortest path, constrained pathfinding, heuristic search, weight constrained shortest path

1 | INTRODUCTION

The weight constrained shortest path problem (WCSPP) is well known as a technically challenging variant of the classical shortest path problem. The objective of the point-to-point WCSPP is to find a minimum-cost (shortest) path between two points in a graph such that the total weight (or resource consumption) of the path is limited. Formally, given a directed graph $G = (S, E)$ with a finite set of states S and a set of edges E, each edge labeled with a pair of attributes $(cost, weight)$, the task is to find a path π from $start \in S$ to $goal \in S$ such that $cost(\pi)$ is minimum among all paths between $start$ and $goal$ subject to $weight(\pi) \leq W$ where W is the given weight limit. The problem has been shown to be NP-complete [17].

The WCSPP, as a core network optimization problem or a subroutine in larger problems, can be seen in various real-world applications in diverse areas such as telecommunication networks, transportation, planning and scheduling, robotics and game development. A typical example of the WCSPP as a core problem is finding the least-cost connection between two nodes in a network with relays such that the weight of any path between two consecutive relays does not exceed a given upper bound

[20]. As a subroutine, examples can be solving the WCSPP as a sub-problem in the context of column generation (a solving technique in linear programming) [35], or obtaining valid (constrained) paths needed for trip planning such as in vehicle routing problems. A valid path in this context can be defined as the quickest path between two points such that its energy requirement is less than the (limited) available energy/petrol [31] or an energy-efficient path that is at most a constant factor longer than the shortest time/distance path for every transport request [1].

Looking at the performance of the state-of-the-art search techniques for constrained pathfinding over the last decade, we can see fast algorithms that are able to optimally solve small-size instances in less than a second. However, given the significance and difficulty of the problem, especially in large-size graphs, the WCSPP still remains a challenging problem that needs more efficient solutions in terms of time and space (memory usage). The main motivations behind this research are therefore as follows.

1. Hardware/resource requirement: we are mainly concerned by the space requirement of recent solutions to the WCSPP, as this metric may drastically limit their application in large graphs. Although space usage is highly correlated with runtime and (faster) methods are likely to consume less memory, there are still some procedures in the state of the art that are space-demanding. As an example, the *partial path matching* procedure in the recent bidirectional approaches needs to store partial paths of each direction to be able to build a complete *start-goal* path later during the search. In terms of memory, this procedure is quite demanding, since the number of partial paths grows exponentially over the course of the search.
2. Applicability of algorithms: Although recent algorithms have shown that the problem can be solved faster bidirectionally, there are several problems were searching backwards is not possible or at least is not straightforward. As an example, in planning feasible paths for electric vehicles, the forward search can propagate estimated battery levels based on the current battery level at the origin location, but a backward search would require the final battery level at the destination to be known without actually completing the search.

Among other variants of the shortest path problem, the bi-objective shortest path problem (BOSPP) is considered conceptually closest to the WCSPP in terms of the number of criteria involved. Compared to the WCSPP where we look for a single optimal solution, methods to the BOSPP find a set of Pareto-optimal solution paths, a set in which every individual solution offers a path that minimizes the bi-criteria problem in both *cost* and *weight*. The BOSPP is a difficult problem, technically more difficult than the WCSPP. Therefore, if there exists a time and space-efficient solution to the BOSPP, it is worth investigating whether it can be adapted for the WCSPP as it may help us to address our serious concerns about the applicability of recent approaches.

Contributions: For our first contribution, we design two new solution approaches to the WCSPP, called WC-EBBA*$_{\text{par}}$ and WC-A*, by extending the recent fast algorithms available for both WCSPP and BOSPP. We conduct an extensive analysis of our A*-based methods through a set of challenging WCSPP instances, and compare their algorithmic performance with *six* recent WCSPP approaches to understand the strengths and weaknesses of the algorithms in both time and space metrics. The results show that the proposed approaches are very effective in reducing the search time and also the total memory requirement of the constrained search in almost all levels of tightness, outperforming some recent solutions by orders of magnitude faster runtime.

As our second contribution, we empirically study variations in a fundamental component of bi-criteria search with A*: priority queue. We briefly explain our motivations for this extension. The bi-criteria search with A* works based on enumeration, that is, all promising paths are evaluated until there is no feasible solution path better than the discovered optimum path. In such search schemes, it is highly expected that the search generates dominated paths, that is, paths for which we can find at least one other path with better *cost* and *weight*. Recent studies have shown that the lazy dominance test in bi-objective A* contributes to faster query times. Nonetheless, it inevitably increases the number of suspended paths in the queue, mainly because the queue accepts dominated paths, making queue operations lengthier. To this end, we extend our experiments and investigate the impacts of three types of priority queues on the performance of constrained search with A*, namely bucket, hybrid and binary heap queues. We define our hybrid queue to be a two-level bucket-heap structure. Among these types, we are interested in finding a type that appropriately addresses the inefficiencies incurred by the lazy dominance test in the exhaustive search of A*. The results of our extensive experiments show that bucket-based queues (bucket queue and hybrid queue) perform far better than the traditional binary heap queues in managing the significant number of paths produced in difficult WCSPP instances.

For our third contribution, we investigate the impact of tie-breaking on the algorithmic performance of constrained search with A*, that is, the total computational cost of handling cases where two paths in the priority queue have the same estimated *cost*. For this purpose, we evaluate two variants of our priority queues (with and without tie-breaking) on realistic and also randomized graphs, and show how recent A*-based algorithms can expedite their search by simply ordering unexplored paths solely based on their primary cost, without a tie-breaking mechanism. Our detailed analysis reveals that tie-breaking has a very limited ability to reduce the number of path expansions. Instead, it incurs significant computational overheads to break the ties

among paths in the queue. In other words, in difficult bi-criteria problems, the priority queue's effort in breaking the ties between paths is not paid off by the search, and we are better off simply ordering paths in the queue based on their primary costs.

The article is structured as follows. In the next section, we survey related work in the WCSPP literature and explain, at a high level, the main features of our selected algorithms. We formally define the problem in Section 3. The main features of our constrained A* searches are explained in this section. In Section 4, we introduce our unidirectional search scheme WC-A*, followed by our parallel algorithm WC-EBBA*$_{\text{par}}$ presented in Section 5. Section 6 describes our practical considerations for the empirical study in Section 7. Our bucket and hybrid priority queues are introduced in this section. Section 7 contains the experimental results and also our detailed study on the impacts of tie-breaking and priority queues on the computation time. Finally, Section 8 presents conclusions and directions for future work.

2 | BACKGROUND AND SELECTED ALGORITHMS

This section reviews classic and recent methods proposed to address the WCSPP. In our literature review, we also provide cases where BOSPP approaches have been extended to address the WCSPP. We conclude this section by nominating two promising extensions from both domains, along with six recent algorithms in the literature for the WCSPP as our baseline for experiments.

The WCSPP and its extended version with more than one weight constraint, which is known as the resource constrained shortest path problem (RCSPP), are well-studied topics in the mathematical optimization literature. Since the WCSPP is just a special case of the RCSPP, all algorithms designed to solve the RCSPP are naturally capable of addressing the WCSPP. This applicability can be seen in almost all works on the RCSPP in the literature, where solution approaches are also computationally evaluated on benchmark instances of the WCSPP. Given the close relationship between the two problems, there exist various types of solution approaches in the literature designed to tackle constrained pathfinding in different settings. Pugliese and Guerriero [25] presented a summary of traditional exact solution approaches to the RCSPP (methods that solve the problem to optimality). In their survey paper, strategies for the gap-closing step (the main search for the optimal solution) are discussed in three categories: path ranking, dynamic programming and branch-and-bound (B&B) approach. In the first category (path ranking), k-paths are obtained, usually by solving the k-shortest path problem, and then sorted (ranked) based on their cost in ascending order. For the search phase, the shortest paths are successively evaluated until a feasible path (with a valid resource consumption) is obtained [28]. As the computational effort in the first phase depends on the value of k, and since the value of k exponentially grows with respect to the size of the network, selecting an efficient path ranking method is crucial in this category of solutions. Solution approaches in the second category generally establish a dynamic programming framework to extend partial paths from origin to destination, normally via a labeling method. Labels in this context can be defined as tuples that contain some essential information about partial paths such as cost, resource consumption and parent pointers. The search in labeling methods can be improved by using pruning rules, or sometimes by integrating other methods to reduce the search space, normally after a preprocessing phase [13]. The solution in this case is a feasible least-cost label (with a valid resource consumption) at the destination. The third category of algorithms (B&B) tries to find feasible solutions by applying an enumeration procedure (e.g., via a depth-first search) along branches of the search tree. This step usually involves adding a node to the end of the partial path while using lower-bounding information to prune infeasible paths and remove nodes from the end of the partial path (backtracking). It is evident that the performance of B&B approaches greatly depends on the quality of the bounds and also the pruning strategy [23].

Ferone et al. [14] presented a short survey on the recent state-of-the-art exact algorithms designed for constrained pathfinding. According to their report, we can still see effective methods from each category of solution approaches. The B&B-like solution method presented by Lozano and Medaglia [21] conducts a systematic search via the depth-first search scheme for the RCSPP. Their so-called *Pulse* algorithm was equipped with *infeasible* and *dominance* pruning strategies and was computationally compared with the path ranking method of Santos et al. [28] on their benchmark instances for the WCSPP. Lozano and Medaglia concluded that, regardless of the tightness of the constraint, the Pulse algorithm performs roughly 40 times faster than the path ranking method of Santos et al. [28]. Horváth and Kis [19] extended the integer programming techniques presented by Garcia [15] for the RCSPP and designed a B&B solution with improved cutting and heuristic methods. Their computational results on the WCSPP instances of Santos et al. [28] show that their proposed branch-and-cut procedure delivers a comparable performance to the Pulse algorithm, both showing larger computational times with increasing input size. However, Sedeño-Noda and Alonso-Rodríguez [29] later developed an enhanced path ranking approach called *CSP*, which was able to exploit pruning strategies of Lozano and Medaglia [21]. They computationally tested their CSP algorithm and compared its performance against Pulse on a new set of realistic large-size instances for the WCSPP. Sedeño-Noda and Alonso-Rodríguez reported that although Pulse might perform better than their CSP algorithm mainly on small instances, it generally performs poorly on difficult large instances, leaving 28% of the instances unsolved even after a two-day timeout. Their results also show that the enhanced path ranking algorithm CSP runs generally faster across a range of constraint tightness and solves more instances than Pulse, yet still

leaving 4% of the instances unsolved. Later in [7], Bolívar et al. presented several acceleration strategies for Pulse to solve the WCSPP with replenishment, mainly by introducing a queuing strategy that limits the depth of the Pulse search, and also a path completion strategy that allows the search to possibly update the primal upper bound early. However, Bolívar et al. reported that the accelerated Pulse method is just about 1%–5% faster than the standard Pulse algorithm on the WCSPP instances.

Among recent solutions, the dynamic programming approach of Thomas et al. [32] solves the RCSPP with the help of heuristic search. Inspired by the idea of a *half-way point* utilized in the bidirectional dynamic programming approach of Righini and Salani [27], Thomas et al. developed a bidirectional A* search informed with forward and backward lower bounding information required for both pruning and the A*'s best-first search. In their *RC-BDA** algorithm, the resource budget is divided equally and then allocated to the forward and backward searches. In other words, the algorithm stops expanding partial paths with a resource consumption larger than half of the total resource budget, allowing both searches to meet at 50% resource half-way points. A complete path in this method can be obtained by joining forward and backward partial paths. Thomas et al. evaluated RC-BDA* on some instances of Sedeño-Noda and Alonso-Rodríguez [29] for the WCSPP. Their empirical results illustrate the effectiveness of their bidirectional search on large instances, being able to solve 99% of the total instances within five hours. However, it can be seen that RC-BDA* is dominated by both Pulse of Lozano and Medaglia [21] and CSP of Sedeño-Noda and Alonso-Rodríguez [29] on many small-size instances. Thomas et al. also tested a unidirectional version of RC-BDA* by running a single forward constrained A* search. Compared to the bidirectional search RC-BDA*, they reported weaker performance (on all levels of tightness) in both runtime and memory requirement for the unidirectional variant on the instances of [28]. Given the success of the bidirectional search in large instances, Cabrera et al. [8] developed a parallel framework to execute Pulse bidirectionally. To prevent the search from falling into unpromising deep branches, bidirectional Pulse (*BiPulse*) limits the depth of the Pulse search and employs an adapted form of the queuing strategy proposed by Bolívar et al. [7] to store and later expand halted partial paths in the breadth-first search manner. Similar to RC-BDA*, BiPulse handles collision between search frontiers by joining forward and backward partial paths. This method in BiPulse is further extended by joining partial paths with their complementary cost/resource shortest path to obtain and update the incumbent solution early. Cabrera et al. evaluated BiPulse on a subset of large instances of Sedeño-Noda and Alonso-Rodríguez [29] for the WCSPP and compared BiPulse with the unidirectional Pulse and RC-BDA* of Thomas et al. [32]. Their results show that BiPulse delivers better performance and solves more instances than both Pulse and RC-BDA* on medium-size instances, while leaving 3% of the instances unsolved after 4 h of runtime. Both RC-BDA* and BiPulse recently won the Glover-Klingman prize, awarded by networks [16].

The growing significance of the WCSPP can also be seen in its increasing prominence in the recent AI literature. Motivated by the applications of the WCSPP, we modified and then improved RC-BDA* of Thomas et al. [32] for the WCSPP [5]. We optimized various components of the RC-BDA* search and proposed an enhanced dynamic programming framework that was able to solve, for the first time, 100% of the benchmark instances of Sedeño-Noda and Alonso-Rodríguez [29], each within just 9 min of runtime. We showed that our enhanced biased bidirectional A* algorithm for the WCSPP, which we call *WC-EBBA**, outperforms the state-of-the-art algorithms on almost all instances by several orders of magnitude. WC-EBBA* follows the two-phase search of the conventional approaches. In the first (initialization) phase, the search establishes both lower and upper bounds needed for the main search while reducing the search space by removing states that are out-of-bounds. Further, compared to RC-BDA* where the bidirectional searches are expected to meet at the 50% half-way point, and also other approaches with dynamic half-way point [33], the initialization phase of WC-EBBA* decides on the budgets of each search direction, making the main bidirectional searches biased by enabling their search frontiers to meet at any fraction of the resource budget (not just at the 50% half-way point). In the second phase, or the main search, WC-EBBA* employs several strategies to expedite the searches in each direction. For the dominance checking procedure, WC-EBBA* borrows a fast strategy from the bi-objective search context to detect and prune dominated partial path in a constant time. WC-EBBA* also takes advantage of node ordering in A* to perform more efficient partial path matching, a procedure that plays an important role in both computation time and space usage of the algorithm.

Given the success of the recent bidirectional search algorithms in solving difficult WCSPP instances, we still need to investigate algorithms that are more efficient in terms of space, and even sometimes algorithms that are simpler in terms of search structure. For instance, some of the recent algorithms (BiPulse, RC-BDA*, and WC-EBBA*) employ a path-matching procedure that handles frontier collision. To fulfil this task, the search needs to store all the explored partial paths in both directions. However, this requirement can significantly increase the memory usage of the algorithms, since the number of partial paths can grow exponentially over the course of the search. Quite recently in Ahmadi et al. [4], we leveraged our bi-objective bidirectional A*-based search algorithm BOBA* [2, 3] and introduced WC-BA* as a bidirectional WCSPP method that does not need to handle frontier collision. WC-BA* executes its forward and backward searches in parallel and on different attribute orderings (e.g., forward A* search working on *cost* and backward A* search on *weight*). As each individual search in WC-BA* is complete, partial paths are no longer stored. Besides the early solution update (ESU) method we proposed in Ahmadi et al. [5], WC-BA* has a unique method called *heuristic tuning*, which allows the search to improve its initial lower bounds during the search. We evaluated WC-BA* on very large graphs and compared its performance against the recent algorithms in the literature, including

TABLE 1 An overview of the main features of the studied algorithms.

Feature/Alg.	WC-A*	WC-BA*	WC-EBBA*$_{par}$	WC-EBBA*	BiPulse	Pulse	RC-BDA*	CSP
Bidirectional	No	Yes	Yes	Yes	Yes	No	Yes	No
Frontier collision	-	No	Yes	Yes	Yes	-	Yes	-
Parallel framework	No	Yes	Yes	No	Yes	No	No	No
Early solution update	Yes	Yes	Yes	Yes	Yes	No	No	No
Heuristic tuning	No	Yes	No	No	No	No	No	No

WC-EBBA*. The results of our experiments over 2000 new instances showed that WC-BA* outperforms WC-EBBA* by up to 60% (35%) in terms of computation time (space) in various constrained problem instances. This observation motivated us to further explore other efficient BOSPP methods for their constrained variant. We briefly describe the main features of our studied algorithms as follows.

Selected algorithms: From the existing WCSPP solutions, we consider the recent WC-BA* and WC-EBBA* algorithms and also the award-winning BiPulse algorithm as our baseline. For our extended empirical study, we also evaluate the other award-winning algorithm RC-BDA*, the ranking method CSP and the B&B method Pulse.

Proposed algorithms: We target the recent fast A*-based BOSPP and WCSPP methods and develop two new algorithms for the WCSPP, namely:

- WC-A*: The adapted version of the bi-objective A* search algorithm (*BOA**) originally presented by Ulloa et al. [34] and enhanced by us in Ahmadi et al. [2]. BOA* is a simple unidirectional search method, so its WCSPP variant will help us to investigate a simple solution to problems that cannot be solved bidirectionally. Our adapted algorithm WC-A* leverages improvements proposed for BOA*.
- WC-EBBA*$_{par}$: WC-EB The extended version of our recent WC-EBBA* algorithm for the WCSPP [5], improved with parallelism. In contrast to the standard WC-EBBA* algorithm where the search only explores one direction at a time, the new variant provides the algorithm with the opportunity of executing its forward and backward searches concurrently. This new feature allows us to accelerate the WC-EBBA*'s bidirectional search and solve more instances in a limited time.

The suffix *par* in WC-EBBA*$_{par}$ denotes that the algorithm is run on a *parallel* framework, ideally with two CPU cores. Table 1 summarizes the main features of all eight algorithms studied in this article, including their proposed speed-up techniques.

3 | NOTATION AND SEARCH STRATEGY

Consider a directed graph $G = (S, E)$ with a finite set of states S and a set of edges $E \subseteq S \times S$. Every edge $e \in E$ has two non-negative attributes that can be accessed via the cost function $\mathbf{cost} : E \to \mathbb{R}^+ \times \mathbb{R}^+$. For the sake of simplicity, in our algorithmic description, we replace the conventional $(cost, weight)$ attribute representation with $(cost_1, cost_2)$. Further, in our notation, every boldface function returns a tuple, so for the edge cost function, we have $\mathbf{cost} = (cost_1, cost_2)$. Expanding a typical state u *generates* a set of successor states, denoted $Succ(u)$. A path is a sequence of states $u_i \in S$ with $i \in \{1, \dots, n\}$ and $(u_i, u_{i+1} \in E)$ for every $i \in \{1, \dots, n-1\}$. The **cost** of path $\pi = \{u_1, u_2, u_3, \dots, u_n\}$ is then the sum of corresponding attributes on all the edges constituting the path, namely $\mathbf{cost}(\pi) = \sum_{i=1}^{n-1} \mathbf{cost}(u_i, u_{i+1})$. The WCSPP aims to find a **cost**-optimal *start-goal* solution path such that the secondary cost of the optimum path is within the upper bound W (conventionally *weight* limit). We formally define **cost**-optimal solution paths below.

> **Definition.** π^* is a **cost**-optimal *start-goal* solution path for the WCSPP if it has the lexicographically smallest $(cost_1, cost_2)$ among all paths from $start \in S$ to $goal \in S$ such that $cost_2(\pi^*) \leq W$.

We abstract from the two possible attribute orderings $(1, 2)$ and $(2, 1)$ in our notation by using a pair (p, s) (for *primary* and *secondary*) with $p, s \in \{1, 2\}$ and $p \neq s$. Given an attribute ordering (p, s), we now define lexicographical order on $(cost_p, cost_s)$, followed by the definition of search *directions*.

> **Definition.** Path π is lexicographically smaller than path π' in the $(cost_p, cost_s)$ order if $cost_p(\pi) < cost_p(\pi')$, or $cost_p(\pi) = cost_p(\pi')$ and $cost_s(\pi) < cost_s(\pi')$. Path π is equal to path π' if $cost_p(\pi) = cost_p(\pi')$ and $cost_s(\pi) = cost_s(\pi')$.

> **Definition.** The search is called *forward* if it explores the graph from the *start* state to the *goal* state. Otherwise, searching from *goal* towards *start* is called *backward*.

In our notation, we generalize both possible search directions by searching in direction $d \in \{forward, backward\}$ from an *initial* state to a *target* state. Therefore, the (*initial*, *target*) pair would be (*start*, *goal*) in the forward search and (*goal*, *start*) in the backward search. In addition, we define d' to always be the opposite direction of d. To keep our notation consistent in the bidirectional setting, we always use the reversed graph or *Reversed*(G) if we search backwards. Compared to the original graph G, *Reversed*(G) has the same set of states but with all the directed edges reversed.

We follow the standard notation in the heuristic search literature and define our search objects to be *nodes* (equivalent to partial paths). A node x is a tuple that contains the main information of the partial path to state $s(x) \in S$. The node x traditionally stores a value pair $\mathbf{g}(x)$ which measures the **cost** of a concrete path from the initial state to state $s(x)$. In addition, x is associated with a value pair $\mathbf{f}(x)$ which is an estimate of the **cost** of a complete path from the initial state to the target state via $s(x)$; and also a reference *parent*(x) which indicates the parent node of x.

We consider all operations of the boldface costs to be done element-wise. For example, we define $\mathbf{g}(x) + \mathbf{g}(y)$ as $(g_1(x) + g_1(y), g_2(x) + g_2(y))$. We use $(\prec, \succ)$ or $(\preccurlyeq, \succcurlyeq)$ symbols in direct comparisons of boldface values, for example, $\mathbf{g}(x) \preccurlyeq \mathbf{g}(y)$ denotes $g_1(x) \leq g_1(y)$ and $g_2(x) \leq g_2(y)$. Analogously, if one (or both) of the relations cannot be satisfied, we use $(\not\prec, \not\succ)$ or $(\not\preccurlyeq, \not\succcurlyeq)$ symbols. For instance, $\mathbf{g}(x) \not\preccurlyeq \mathbf{g}(y)$ denotes $g_1(x) > g_1(y)$ or $g_2(x) > g_2(y)$. Unless otherwise stated, if the search is bidirectional, we assume that nodes are only compared within the same direction of the search. This means that we do not compare a forward search node with a backward search node, even if they are associated with the same state. We now define *dominance* over nodes generated in the same search direction.

> **Definition.** For every pair of nodes (x, y) associated with the same state $s(x) = s(y)$, we say node y is dominated by x if we have $g_1(x) < g_1(y)$ and $g_2(x) \leq g_2(y)$ or if we have $g_1(x) = g_1(y)$ and $g_2(x) < g_2(y)$. Node x weakly dominates y if $\mathbf{g}(x) \preccurlyeq \mathbf{g}(y)$.

With the search dominance criteria defined, we now describe state lower and upper bounds.

> **Definition.** For every state $u \in S$, $\mathbf{h}^d(u)$ and $\mathbf{ub}^d(u)$ denote the lower and upper bounds on the **cost** of paths, respectively, from state u to the target state in the search direction d (*goal* in forward and *start* in backward search).

Note that $\mathbf{h}^d$ and $\mathbf{ub}^d$ can be established by conducting two simple unidirectional single-objective searches from the target state in the reverse direction d', one on $cost_1$ and the other one on $cost_2$. We define the *validity* condition and terminal states as follows.

> **Definition.** A path/node/state x is valid if its estimated cost $\mathbf{f}(x)$ is within the search global upper bounds defined as $\bar{f} = (\bar{f_1}, \bar{f_2})$, that is, x is valid if $\mathbf{f}(x) \preccurlyeq \bar{\mathbf{f}}$. Otherwise, x is invalid if $\mathbf{f}(x) \not\preccurlyeq \bar{\mathbf{f}}$, that is, if we have $f_1(x) > \bar{f_1}$ or $f_2(x) > \bar{f_2}$.

> **Definition.** State u is a terminal state in direction d if $\mathbf{h}^d(u) = \mathbf{ub}^d(u)$, meaning that there exists a path from u to the target state optimum for both costs.

As we will show later in the article, a valid node associated with such a terminal state yields a tentative solution [2, 4].

3.1 | Constrained pathfinding with A*

The main search in A* is guided by the *start-goal* cost estimates or **f**-values. These **f**-values are traditionally established based on a consistent and admissible heuristic function $\mathbf{h} : S \to \mathbb{R}^+ \times \mathbb{R}^+$ [18]. In other words, for every search node x, we have $\mathbf{f}(x) = \mathbf{g}(x) + \mathbf{h}(s(x))$ where $\mathbf{h}(s(x))$ estimates lower bounds on the **cost** of paths from state $s(x)$ to the target state.

> **Definition.** The heuristic function h_p is admissible iff $h_p(u) \leq cost_p(\pi)$ for every $u \in S$ where π is the optimal path on $cost_p$ from state u to the target state. It is also consistent if we have $h_p(u) \leq cost_p(u, v) + h_p(v)$ for every edge $(u, v) \in E$ [18].

We distinguish forward and backward heuristic functions by incorporating the search direction d, that is, $\mathbf{h}^d$. In A*, we perform a systematic search by *expanding* nodes in best-first order. That is, the search is led by a partial path that shows the lowest cost estimate. To this end, we expand one (lexicographically) least-cost node in each iteration and store all the descendant (new) nodes in an *Open* list. More accurately, $Open^d$ is a priority queue for the A* search of direction d that contains generated (but not expanded) nodes. To commence the search, we initialize the $Open^d$ list with a node associated with the initial state and $\mathbf{g} = (0, 0)$. For the purpose of further expansion, the $Open^d$ list reorders its nodes according to their f-values such that the least-cost node is at the front of the list. In the constrained setting, since **f** represents a pair of costs, there are two possible lexicographic orderings of the nodes in the $Open^d$ list. Therefore, depending on the search requirement, the $Open^d$ list can order nodes based on (f_1, f_2) or (f_2, f_1) lexicographically. For example, if a search method needs unexplored nodes to be lexicographically ordered on (f_1, f_2), the $Open^d$ list first orders nodes based on their f_1-value, and then based on their f_2-value if two (or more) of them have the same f_1-value. The latter operation (ordering based on the second element) is called *tie-breaking*. Nodes associated with the target

Procedure 1: Setup(d)

1 $Open^d \leftarrow \emptyset$
2 $g^d_{min}(u) \leftarrow \infty$ for each $u \in S$
3* $\chi^d(u) \leftarrow \emptyset$ for each $u \in S$
4 **if** $d = forward$ **then** $u_i = start$
5 **else** $u_i = goal$
6 $x \leftarrow$ new node with $s(x) = u_i$
7 $\mathbf{g}(x) \leftarrow (0, 0)$
8 $\mathbf{f}(x) \leftarrow (h^d_1(u_i), h^d_2(u_i))$
9 $parent(x) \leftarrow \varnothing$
10 Add x to $Open^d$

* This list is used in WC-EBBA* search only.

state represent solution paths, thus A* does not need to expand such solution nodes. Finally, A* terminates if the **cost**-optimal solution path is found, or if there is no node in the $Open^d$ list to expand.

Section SM. 1 in Supplementary Material (SM) revisits the principles of constrained search with A* and discusses in detail the correctness of each component of the search, including (lazy) dominance rules, the termination criteria and optimality of the solution. In the next parts, we present the search setup and the pruning strategies as part of our common methods.

3.2 | Search setup

The first common method in our A* searches is Setup(d) in Procedure 1. This procedure shows the essential data structures we initialize for our A*-based constrained search in direction $d \in \{forward, backward\}$. Following the literature, the procedure first initializes $Open^d$ as the priority queue of search. It then initializes for every $u \in S$ the scalar $g^d_{\min}(u)$, an important parameter that will keep track of the secondary cost of the last node successfully expanded for state u during the search in direction d. A* uses this parameter to prune some dominated nodes. Depending on the search direction, the procedure sets the initial state u_i. If the search direction is *forward*, u_i is chosen to be *start*, otherwise, *goal* is chosen as the initial state. To commence the search, the procedure generates a new node x with the initial state u_i and inserts it into $Open^d$. Node x, as the initial node, has zero actual costs (i.e., g-values) and a null pointer as its parent node, but it can use the heuristic functions $\mathbf{h}^d$ to establish its cost estimate $f(x)$, that is, f-values.

In addition, if we want to run a constrained search via WC-EBBA* or WC-EBBA*$_{\text{par}}$, the procedure initializes $\chi^d(u)$ as an empty list for every $u \in S$. As we described in Ahmadi et al. [5], this list will be populated with expanded nodes (partial paths) of state u during the search in direction d. WC-EBBA*$_{\text{par}}$ (and similarly WC-EBBA*) will use this list to offer complementary paths to nodes that are under exploration for state u in the opposite direction d' (as part of partial paths matching).

3.3 | Path expansion with pruning

Expanding partial paths is a key component in A*. Procedure 2 shows the main steps involved in Expanding and Pruning (ExP) a typical node x in the traditional (f_1, f_2) order and in direction d, as established in Ahmadi et al. [4]. The ExP(x, d) procedure *generates* a set of new descendant nodes via $s(x)$'s successors, that is, $Succ(s(x))$, and then checks new nodes against the pruning criteria. Each successor state is denoted by v. Lines 2–5 of Procedure 2 show the essential operations involved in the node expansion. Given the partial path information carried by the current node x, each new node y is initialized with the successor state v, actual costs $\mathbf{g}(y)$ and cost estimates $\mathbf{f}(y)$ of the extended path, capturing x as the node y's parent. Procedure 2 also applies three types of pruning strategies before inserting new nodes into $Open^d$. We briefly explain their type and functionality below.

- Line 6: Prune by dominance; ignore the new node y if it is dominated by the last non-dominated node expanded for $s(y)$.
- Line 7: Prune by dominance; ignore node y if it is dominated by one of the preliminary shortest paths to $s(y)$.
- Line 8: Prune by invalidity; ignore node y if it shows cost estimates beyond the search global upper bounds in $\bar{\mathbf{f}}$.

Note that the pruning strategy employed in Line 7 will only be used in bidirectional searches, where search in direction d has access to states' upper bound function $\mathbf{ub}^{d'}$. Finally, the ExP(x, d) procedure inserts every new node y into $Open^d$ if y is not pruned by dominance or validity tests.

Procedure 2: ExP(x, d) in (f_1, f_2) ord

for *all* $v \in Succ(s(x))$ **do**
  $y \leftarrow$ new node with $s(y) = v$
  $\mathbf{g}(y) \leftarrow \mathbf{g}(x) + \mathbf{cost}(s(x), v)$
  $\mathbf{f}(y) \leftarrow \mathbf{g}(y) + \mathbf{h}^d(v)$
  $parent(y) \leftarrow x$
  if $g_2(y) \geq g^d_{min}(v)$ **then continue**
* **if** $\mathbf{g}(y) \npreceq \mathbf{ub}^{d'}(v)$ **then continue**
  if $\mathbf{f}(y) \npreceq \bar{\mathbf{f}}$ **then continue**
  Add y to $Open^d$

* This pruning is used in bidirectional searches only.

Procedure 3: ESU(x, d) in (f_1, f_2) ord

$f'_1 \leftarrow g_1(x) + ub^d_1(s(x))$
$f'_2 \leftarrow g_2(x) + ub^d_2(s(x))$
if $f'_2 \leq \overline{f_2}$ **then**
  if $f_1(x) < \overline{f_1}$ **or** $f'_2 < f^{sol}_2$ **then**
    $\overline{f_1} \leftarrow f_1(x)$
    $f^{sol}_2 \leftarrow f'_2$
    $Sol \leftarrow (x, \emptyset)$
else if $f'_1 < \overline{f_1}$ **then**
  $\overline{f_1} \leftarrow f'_1$
  $f^{sol}_2 \leftarrow \infty$

3.4 | Early solution update

The ESU strategy allows our algorithms to update the global upper bounds and possibly establish a feasible solution *Sol* before reaching the target state. Here, we choose the ESU procedure we developed for bi-criteria search in Ahmadi et al. [2] and adapt it to the WCSPP. The idea of this strategy is quite straightforward and is shown in Procedure 3 for the traditional objective ordering (f_1, f_2) in the search direction d. We briefly describe the procedure as follows.

Consider node x as a partial path, the procedure ESU(x, d) tries to establish a complete *start-goal* path by joining x to its two complementary shortest paths from $s(x)$ to the target state. In our notation, we use $(\overline{f_1}, f^{\text{sol}}_2)$ to keep track of the actual costs of the best-known solution path during the search. The procedure tries two cases, as explained below.

1. It first joins node x with its complementary shortest path for the primary attribute, that is, the path with costs $(h^d_1(s(x)), ub^d_2(s(x)))$ in the (f_1, f_2) order. It then retrieves the actual costs of the joined path via $f_1(x) = g_1(x) + h^d_1(s(x))$ and $f'_2 = g_2(x) + ub^d_2(s(x))$. If the resulting joined path is valid, the procedure treats x as a tentative solution node. In this case, if the tentative solution is lexicographically smaller than the best-known solution, the procedure updates the global upper bound $\overline{f_1}$ with $f_1(x)$. This is followed by storing f'_2 in f^{sol}_2 as the secondary cost of the new solution and also capturing x as a new solution node via *Sol*.
2. If the joined path in the first part is found invalid, the procedure still has a chance to improve its primal upper bound using the shortest path optimum for the second criterion, with costs $(ub^d_1(s(x)), h^d_2(s(x)))$. In this case, the costs of the joined path can be retrieved as $f_2(x) = g_2(x) + h^d_2(s(x))$ and $f'_1 = g_1(x) + ub^d_1(s(x))$. If the resulting joined path is valid, the procedure updates the primary upper bound $\overline{f_1}$ with f'_1 and then resets f^{sol}_2 for the upcoming solution (with cost f'_1 or better).

Note that the procedure does not need to explicitly check $f_1(x)$ and $f_2(x)$. This is mainly because our constrained A* searches prune invalid nodes before attempting the ESU strategy. In addition, if x enters Procedure 3 with the target state, it will definitely pass case 1 above and will be captured as a valid solution node. This is because we always have $\mathbf{h}^d(s(x)) = \mathbf{ub}^d(s(x))$ for such nodes, which immediately yields $f'_s = f_s(x)$ and a valid path consequently. There is also one important difference between the two cases: valid joined paths compute a potential solution only in case 1 (see Lemma SM. 1.10 for the formal proof).

Algorithm 4: WC-A* high-level

Input: A problem instance (G, **cost**, *start*, *goal*) with the weight limit W
Output: A node pair corresponding with the cost-optimal feasible solution *Sol*
1 $\mathbf{h}, \mathbf{ub}, \bar{\mathbf{f}}, S' \leftarrow$ Initialize WC-A* (G, **cost**, *start*, *goal*, W) ▷ Algorithm 5
2 $Sol \leftarrow (\emptyset, \emptyset), f_2^{\text{sol}} \leftarrow \infty$
3 $Sol \leftarrow$ Run a WC-A* search on (G, **cost**, *start*, *goal*) in the forward direction and (f_1, f_2) order with global upper bounds $\bar{\mathbf{f}}$, heuristic functions ($\mathbf{h}, \mathbf{ub}$) and initial solution *Sol* with secondary cost f_2^{sol}. ▷ Algorithm 6
4 **return** *Sol*

Solution path construction: If the search terminates with x as an optimal solution node with a non-target state, we just need to join x with its shortest path on the primary cost ($cost_p$) for solution path construction. Therefore, solution path recovery with the ESU strategy is a two-stage procedure: first, we follow back-pointers from *Sol* to the initial state and build the partial path. Second, we recover the concrete path from the *Sol*'s state to the target state (via the pre-established initial shortest path on $cost_p$) and then concatenate this complementary path to the partial path obtained in the first stage.

4 | CONSTRAINED PATHFINDING WITH WC-A*

The BOSPP and WCSPP are two inextricably linked problems, with often shared procedures and terminologies. Compared to the WCSPP where we look for a single optimal path, BOSPP aims to find a representative set of Pareto-optimal solution paths, that is, a set in which every individual solution offers a path that minimizes the bi-criteria problem in both *cost* and *weight*. More accurately, given $(cost_1, cost_2)$ as our edge attributes, Pareto optimality is a situation where we cannot improve $cost_2$ of point-to-point paths without worsening their $cost_1$ and vice versa. It is not difficult to show that any method that generates all Pareto-optimal solutions to the BOSPP is also able to deliver an optimal solution to the WCSPP. From the WCSPP's point of view, if a **cost**-optimal solution path exists, it can always be found in a Pareto-optimal set of BOSPP. With this introduction, we now explain how our new A*-based weight constrained search algorithm WC-A* can be derived from the recent Bi-objective A* algorithm (BOA*) [34], a unidirectional search algorithm to find a set of cost-unique Pareto-optimal paths.

WC-A* is a simple solution approach specialized to solve the WCSPP and follows the search strategy of BOA* and its improvements in Ahmadi et al. [2]. WC-A* works on the basis of A* and explores the graph in the forward direction (from *start* to *goal*) in the traditional (f_1, f_2) order. Algorithm 4 shows the high-level design of WC-A* and its two essential phases. Similar to other A*-based algorithms, WC-A* needs to establish its heuristic functions via an initialization phase. However, since the main constrained search is unidirectional, WC-A* needs to compute lower and upper bound functions in only one direction. Therefore, the initialization phase consists of two single-objective backward searches. When the preliminary heuristic searches are complete, WC-A* initializes a node pair *Sol* with the secondary cost f_2^{sol} to keep track of the lexicographically smallest solution node during the main search. In the next phase, WC-A* executes a forward constrained A* search in the (f_1, f_2) order. When the second phase is also complete, the algorithm returns the node pair *Sol*, which corresponds to a **cost**-optimal solution. Note that since WC-A* is a unidirectional search algorithm, there is no backward counterpart. Clearly, there is no feasible solution if the returned node pair is empty. We illustrate operations in each phase of WC-A* as follows.

4.1 | Initialization

WC-A* is a unidirectional search algorithm and only requires heuristics in one direction. Algorithm 5 shows the main steps involved in WC-A*'s initialization phase. It starts with initializing the search global upper bounds, namely $\bar{f}$-values. The secondary global upper bound $\overline{f_2}$ is set to be the weight limit W, while the primary upper bound $\overline{f_1}$ is initially unknown and will be determined by one of the heuristic searches. The main constrained search will be run in the forward direction, so we obtain the required heuristics h_1^f and h_2^f by running two cost-bounded backward A* searches. The first unidirectional A* search finds lower bounds on $cost_2$ and stops before expanding a state with an estimated cost larger than the global upper bound $\overline{f_2}$. The first search on $cost_2$ is also capable of initializing the global upper bound $\overline{f_1}$ using the upper bound obtained for the *start* state, that is, $ub_1^f(start)$. When the first cost-bounded A* terminates, all not-yet-expanded states in this preliminary search are guaranteed not to be part of any solution paths, reducing the graph size to obtain better-quality heuristics. This method is known as *resource-based network reduction* [6]. To this end, our cost-bounded A* search on $cost_1$ will only consider the states expanded in the first search and stops before expanding a state with a cost estimate larger than $\overline{f_1}$.

Algorithm 5: Initialization phase of WC-A*

Input: The problem instance (G, **cost**, *start*, *goal*) and the weight limit W
Output: WCSPP's uni-directional heuristic functions **h** and **ub**, also global upper bounds $\bar{\mathbf{f}}$

1 Set global upper bounds: $\overline{f_1} \leftarrow \infty$ and $\overline{f_2} \leftarrow W$
2 $h_2^f, ub_1^f \leftarrow$ Run f_2-bounded backward A* (from *goal* using an admissible heuristic) on $cost_2$, use $cost_1$ as a tie-breaker, update $\overline{f_1}$ with $ub_1^f(start)$ when *start* is going to get expandedand stop before expanding a state with $f_2 > \overline{f_2}$.
3 $h_1^f, ub_2^f \leftarrow$ Run f_1-bounded backward A* (using an admissible heuristic) on $cost_1$, use $cost_2$ as a tie-breaker, ignore unexplored states in the search of line 2, and stop before expanding a state with $f_1 > \overline{f_1}$.
4 $S' \leftarrow$ Non-dominated states explored in the last bounded A* search of line 3
5 **return** $\boldsymbol{h}^f, \boldsymbol{ub}^f, \bar{\mathbf{f}}, S'$

Algorithm 6: Constrained search of WC-A*/BA*

Input: Problem (G, **cost**, *start*, *goal*), search direction d, objective ordering (f_p, f_s), global upper bounds $\bar{\mathbf{f}}$, search heuristics $(\mathbf{h}, \mathbf{ub})$, and an initial solution *Sol* with the secondary cost $f_2^{\min}$
Output: A node pair corresponding with an optimal solution

1 Setup(d) ▷ Procedure 1
2 $d' \leftarrow$ opposite direction of d
3 **while** $Open^d \neq \emptyset$ **do**
4 Remove from $Open^d$ node x with the lexicographically smallest (f_p, f_s) values
5 **if** $f_p(x) > \overline{f_p}$ **then break**
6+ **if** $f_s(x) > \overline{f_s}$ **then continue**
7 **if** $g_s(x) \geq g^d_{\min}(s(x))$ **then continue**
8* **if** $g^d_{\min}(s(x)) = \infty$ **then**
9* $h_p^{d'}(s(x)) \leftarrow g_p(x)$
10* $ub_s^{d'}(s(x)) \leftarrow g_s(x)$
11 $g^d_{\min}(s(x)) \leftarrow g_s(x)$
12 ESU(x, d) in (f_p, f_s) order ▷ Procedure 3 or Algorithm 7
13 **if** $h_p^d(s(x)) = ub_p^d(s(x))$ **then continue**
14 ExP(x, d) in (f_p, f_s) order ▷ Procedure 2 or 8
15 **return** *Sol*

+ This pruning method is used in the (f_2, f_1) order only.
* This heuristic is used in WC-BA* only.

The fact that WC-A* only needs two cost-bounded searches to establish its required heuristic functions can be seen as WC-A*'s strength; easy problems will especially benefit from a speedy search setup. However, we can also see this simple initialization phase as a weakness, mainly because WC-A* will have a limited capability to reduce the graph size. Compared to the bidirectional algorithms where both attributes effectively reduce the graph size in two rounds of searches (forward and backward), as in WC-EBBA* and WC-BA*, WC-A* benefits from this effective feature only in one round of bounded searches. In addition, there may be cases where starting the bounded searches with $cost_2$ results in limited graph reduction, mainly due to a lack of informed heuristics for $cost_2$. We will investigate the impact of this possible search ordering on WC-A*'s performance in our extended empirical study.

4.2 | Constrained search

The constrained search in the second phase of WC-A* is relatively straightforward and involves in general two types of strategies: expand and prune. Algorithm 6 shows a pseudocode of WC-A* in the generic search direction d and in objective ordering (f_p, f_s). In this notation, f_p and f_s denote *primary* and *secondary* cost of the search, respectively, generalizing the two possible objective orderings (f_1, f_2) or (f_2, f_1). Although this article studies WC-A* in the standard setting of BOA*, we intentionally present WC-A* in a generic form to emphasize its capability in solving the WCSPP from both directions and also in both objective orderings. One such application of WC-A* is in the parallel searches of WC-BA* (see SM. 3 for more details). For now, let WC-A* be a

Procedure 7: ESU(x, d) in (f_2,f_1) order

```
1  f'_1 ← g_1(x) + ub^d_1(s(x))
2  f'_2 ← g_2(x) + ub^d_2(s(x))
3  if f'_1 ≤ f̄_1 then
4      if f'_1 < f̄_1 or f_2(x) < f^sol_2 then
5          f̄_1 ← f'_1
6          f^sol_2 ← f_2(x)
7          Sol ← (∅, x)
8  else if f_1(x) < f̄_1 and f'_2 ≤ f̄_2 then
9      f̄_1 ← f_1(x)
10     f^sol_2 ← ∞
```

Procedure 8: ExP(x, d) in (f_2,f_1) order

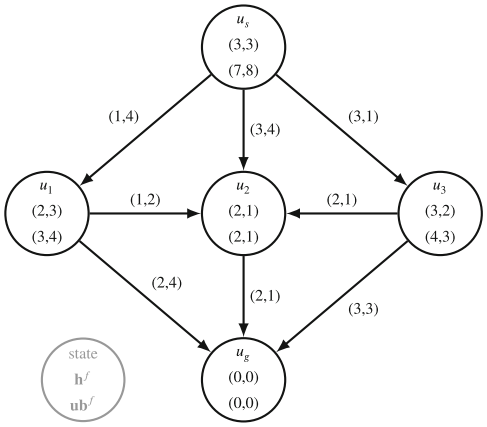
```
1  for all v ∈ Succ(s(x)) do
2      y ← new node with s(y) = v
3      g(y) ← g(x) + cost(s(x), v)
4      f(y) ← g(y) + h^d(v)
5      parent(y) ← x
6      if g_1(y) ≥ g^d_min(v) then continue
7*     if g(y) ⋠ ub^{d'}(v) then continue
8      if f(y) ⋠ f̄ then continue
9      Add y to Open^d
```

* This pruning is used in bidirectional searches only.

forward constrained A* search in the (f_1,f_2) order, that is, we have $d = forward$ and (p, s)=(1,2). In addition, for the operations given in Algorithm 6, we focus on those applicable to WC-A* (lines in black without $^+$ or * symbols).

Algorithm description: WC-A* in Algorithm 6 employs Setup(d) (Procedure 1) to initialize data structures required by the search in the forward direction. This initialization involves inserting a node with *start* state in the priority queue to commence the constrained search. Let $Open^f$ be a (non-empty) priority queue of the forward search in any arbitrary iteration of the algorithm. WC-A* extracts a node x from the priority queue with the lexicographically smallest (f_1,f_2) among all nodes in $Open^f$. Note that A* essentially needs x to be the least-$cost_p$ node, however, lexicographical ordering of nodes in $Open^f$ will help WC-A* to prune more dominated nodes. If $f_1(x)$ is out-of-bounds (line 5), A* guarantees that all not-yet-expanded nodes would be out-of-bounds and the search can terminate (see Lemma SM. 1.9). Otherwise, the algorithm considers x as a valid node and checks it for dominance at line 7. If x is not weakly dominated by the last node expanded for $s(x)$, WC-A* will then explore x and update $g^f_{\min}(s(x))$ with the secondary (actual) cost $g_2(x)$ at line 11 of Algorithm 6. In the next step (line 12), node x is matched with the complementary shortest paths from $s(x)$ to the *goal* state to possibly obtain a tentative solution or update the primary upper bound $\overline{f_1}$ before reaching *goal*. This matching is done via our new ESU procedure (Procedure 3 in the (f_1,f_2) order or Procedure 7 in the (f_2,f_1) order). At this point, the ESU strategy guarantees that x has been captured as a tentative solution if the joined *start-goal* path via x is lexicographically smaller in the ($cost_1, cost_2$) order than the current solution in *Sol* with costs ($\overline{f_1}$,f_2^{sol}). However, WC-A* can still skip expanding node x if it carries a terminal state via line 12, that is, if $h^f_1 = ub^f_1$. This is because nodes associated with a terminal state are tentative solution nodes (already captured by the ESU strategy) and thus their expansion is not necessary (see Lemma SM. 1.11 for the formal proof). Finally, x will be expanded via ExP(x, d) in the forward direction (Procedure 2 in the (f_1,f_2) order or Procedure 8 in the (f_2,f_1) order) if none of the above cases holds.

Example: We explain the constrained search of WC-A* by solving a WCSPP for the graph depicted in Figure 1. On this graph, we want to find the **cost**-optimal shortest path between states u_s and u_g with the weight limit $W = 6$. Let us assume the initialization phase has already been completed, and we have both lower and upper bounds from all states to u_g (our target state) via two single objective backward searches. For the states in Figure 1, we have shown $\mathbf{h}^f$ and $\mathbf{ub}^f$ as part of the states' information, and have updated the primary global upper bound with $ub^f_1(u_s)$ by setting $\overline{f_1} = 7$ (via our initial solution). Nonetheless, we can see that the graph size has not changed as all the lower bounds (**h**-values) are within the search global



It.	*Open* list [$\mathbf{f}(x), \mathbf{g}(x), s(x)$]	Nodes pruned	$\overline{f_1}$	f_2^{sol}
1	↑[(3,3), (0,0), u_s]	[(3,7), (1,4), u_1]	7	∞
2	↑[(5,5), (3,4), u_2] [(6,3), (3,1), u_3]		5	5
3	↑[(6,3), (3,1), u_3]		5	5

parent arrays	u_s	u_1	u_2	u_3	u_g
`parent_state`	[∅]		[u_s]		
`parent_path_id`	[0]		[1]		

FIGURE 1 Left: An example graph with **cost** on the edges, and with (state identifier, **h**, **ub**) inside the states. Right: Status of WC-A* in every iteration (It.): nodes in the *Open* list at the beginning of each iteration, nodes pruned by the algorithm in each iteration, the (updated) value of the global upper bound $\overline{f_1}$ and also the secondary cost of the best-known solution f_2^{sol} at the end of each iteration. The search is conducted in the forward direction and in the (f_1, f_2) order. The symbol ↑ beside nodes denotes the expanded min-cost node in the iteration. The second table shows the status of the parent arrays of the states when the search terminates (see Section 6 for details).

upper bounds. Figure 1 also shows a summary of changes in each iteration of WC-A*. In particular: the status of the *Open* list at the beginning of each iteration, nodes pruned in the iteration and also the latest value of solution costs $(\overline{f_1}, f_2^{\text{sol}})$ at the end of each iteration. We explain in three iterations below how WC-A* finds a **cost**-optimal solution path with just one node expansion.

- **Iteration 1:** At the beginning of this iteration, the search sees only one node in *Open* associated with the start state u_s. We extract this node as $x \leftarrow [(3,3),(0,0),u_s]$ and interpret the node's information as $[(f_1,f_2),(g_1,g_2),state]$. The algorithm then checks x against the termination criterion and also the dominance test. x is a valid and non-dominated node, so WC-A* updates $g^d_{\min}(u_s) \leftarrow 0$ and proceeds with the ESU strategy. In this procedure, we can see that the shortest path from u_s to u_g on $cost_1$ is invalid. In addition, the shortest path on $cost_2$ does not offer a valid path better than the initial solution. So the procedure is unable to improve the upper bound in this iteration. $s(x)$ is not a terminal state either, so WC-A* expands x and generates new nodes when it discovers adjacent states u_1, u_2, and u_3. Before inserting new nodes into the priority queue, the pruning criteria indicate that the new node associated with u_1 is invalid and should be pruned, essentially because its secondary cost estimate is out-of-bounds ($f_2 > \overline{f_2}$ or $7 > 6$). However, the two other nodes generated for states u_2 and u_3 will be added to the *Open* list.
- **Iteration 2:** There are two nodes in the priority queue. WC-A* extracts the node associated with u_2 as it is lexicographically smaller than the other node in the queue (in the (f_1,f_2) order). Let $x \leftarrow [(5,5),(3,4),u_2]$ be the extracted node. This node does not meet the termination criterion and is not a dominated node, mainly because this is the first time WC-A* visits state u_2, so it updates $g^d_{\min}(u_2) \leftarrow 4$. The algorithm then checks x against the ESU strategy. In the ESU(x,d) procedure, we realize that joining x with its complementary shortest path on $cost_1$ yields a valid path. For this complete joined path, that is, path $\{u_s,u_2,u_g\}$, we have $\mathbf{cost} = (5,5)$. So we can see that the joined path is valid because its primary cost is smaller than the best-known cost ($f_1(x) < \overline{f_1}$ or $5 < 7$). Therefore, the ESU strategy captures x as a tentative solution node in *Sol* and updates $\overline{f_1} \leftarrow 5$ and $f_2^{\text{sol}} \leftarrow 5$. x also carries a terminal state, since we have $h^f_1(u_2) = ub^f_2(u_2) = 2$. Hence, WC-A* does not need to expand x considering the fact that the tentative solution via u_2 is already stored in *Sol*.
- **Iteration 3:** WC-A* did not insert any new node in *Open* in the previous iteration, so there exists only one unexplored node in the priority queue. The algorithm extracts this node as $x \leftarrow [(6,3),(3,1),u_3]$. The extracted node x is then checked against the termination criterion, similar to other extracted nodes. Now, we can see that x is an invalid node since its f_1-value is no longer within the bound. In other words, we have $f_1(x) > \overline{f_1}$ or $6 > 5$ equivalently. Therefore, WC-A* successfully terminates with a **cost**-optimal solution node in *Sol* with costs $(5,5)$.

We now prove the correctness of constrained pathfinding with WC-A* (see SM. 3.2 for the correctness without tie-breaking).

Theorem 1. *WC-A* returns a node corresponding to a* **cost***-optimal solution path for the WCSPP.*

Proof. WC-A* enumerates all valid partial paths from the initial state towards the target state in search of an optimal solution. Our proposed pruning strategies ensure that WC-A* never removes a solution node from the search space (Lemmas SM. 1.5 and SM. 1.6). In addition, the ESU strategy correctly keeps track of the lexicographically smallest

Algorithm 9: WC-EBBA*$_{\text{par}}$ High-level

Input: A problem instance (G, **cost**, *start*, *goal*) with the weight limit W
Output: A node pair corresponding with the cost-optimal feasible solution *Sol*
1 $\mathbf{h}, \mathbf{ub}, \bar{\mathbf{f}}, S' \leftarrow$ Initialize WC-EBBA*$_{\text{par}}$ (G, **cost**, *start*, *goal*, W) ▷ Algorithm 10
2 $\beta^f, \beta^b \leftarrow$ Obtain forward and backward budget factors using h_1-values of non-dominated states in S'. ▷ Equation (3)
3 $Sol \leftarrow (\emptyset, \emptyset), f_2^{\text{sol}} \leftarrow \infty$
4 **do in parallel**
5 Run a biased WC-EBBA*$_{\text{par}}$ search for (G, **cost**, *start*, *goal*) in the *forward* direction with global upper bounds $\bar{\mathbf{f}}$, heuristic functions ($\mathbf{h}, \mathbf{ub}$), budget factors (β^f, β^b) and initial solution *Sol* with secondary cost f_2^{sol}. ▷ Algorithm 11
6 Run a biased WC-EBBA*$_{\text{par}}$ search for (G, **cost**, *start*, *goal*) in the *backward* direction with global upper bounds $\bar{\mathbf{f}}$, heuristic functions ($\mathbf{h}, \mathbf{ub}$), budget factors (β^f, β^b) and initial solution *Sol* with secondary cost f_2^{sol}. ▷ Algorithm 11
7 **return** *Sol*

tentative solutions discovered during the search (Lemma SM. 1.10), while avoiding unnecessary expansion of nodes with terminal state (Lemma SM. 1.11). Therefore, we conclude that WC-A* terminates with a **cost**-optimal solution, even with zero-weight cycles (Lemma SM. 1.9). ■

5 | CONSTRAINED PATHFINDING WITH PARALLEL WC-EBBA*

The enhanced biased bidirectional A* search algorithm for the WCSPP, or WC-EBBA* [5], is constructed based on the RC-BDA* algorithm of Thomas et al. [32], both following the standard bidirectional search first presented by Pohl [24]. In contrast to RC-BDA* where each search direction is allocated 50% of the *weight* budget, the search in direction d of WC-EBBA* works with a budget factor $\beta^d \in [0, 1]$. We say the (bidirectional) search is *biased* towards direction d if $\beta^d > 0.5$, which means allocating more budget to the search of direction d than that of the opposite direction d'. WC-EBBA* uses separate priority queues for its forward and backward searches, namely $Open^f$ and $Open^b$, and only explores the graph in one direction at a time. Depending on the location of the least cost node in $Open^f \cup Open^b$, we have either $d = forward$ or $d = backward$. For example, if the least-cost node in $Open^f$ is smaller than the least-cost node in $Open^b$, WC-EBBA* attempts a forward expansion. In this interleaved search scheme, there might be cases where the algorithm mostly performs long runs of uni-directional expansions, a likely case when the search frontier of one direction reaches a dense part of the graph. Although WC-EBBA* tries to balance bidirectional search effort by changing the weight budget of each direction, there is always a chance for slower directions to become dominant with this type of queuing strategy. In addition, WC-EBBA* still suffers from a lengthy initialization phase, even with its more efficient bounded searches. Faster than the traditional one-to-all approaches, the initialization phase of WC-EBBA* obtains its necessary heuristics via four bounded A* searches in series. However, there might be cases where the algorithm spends more time in its initialization phase than the actual constrained search, especially in easy instances with loose weight constraints. Section SM. 2 revisits the standard WC-EBBA* algorithm in the context of heuristic search, providing a detailed overview of all key steps enriched with some of the procedures considered in this article, such as the ESU strategy. Further, it introduces refined procedures that can handle a variant of WC-EBBA* with no tie-breaking in the priority queue (see SM. 2.3).

In this section, we present a new variant of the WC-EBBA* algorithm that leverages parallelism to overcome the aforementioned shortcomings. Algorithm 9 shows, at a high level, three main steps for our new parallel version WC-EBBA*$_{\text{par}}$. Similar to its standard variant, WC-EBBA*$_{\text{par}}$ obtains its required heuristics and also global upper bounds in the first place via two rounds of parallel searches. It then uses a set of non-dominated states obtained from the initialization phase to determine the forward and backward budget factors β^f and β^b, *biasing* the main search towards the direction showing a larger budget factor (see SM. 2.2 for details). In the next step, the search initializes a (shared) solution node pair *Sol* with an unknown secondary cost f_2^{sol} to keep track of the best solution obtained in both directions. In the last step, the algorithm runs two biased WC-EBBA*$_{\text{par}}$ searches concurrently, each capable of updating shared parameters, such as the global upper bound $\bar{f}_1$ and the solution node pair *Sol*. Comparing the structure of both algorithms at this high level, we can see that WC-EBBA*$_{\text{par}}$ differs from WC-EBBA* in two aspects: initialization and search structure. We will explain each of these in detail below.

5.1 | Initialization

Our bidirectional WC-EBBA*$_{\text{par}}$ requires both forward and backward lower/upper bound functions. To speed up the preliminary searches, following WC-BA*, we compute the necessary functions in two rounds of parallel searches as shown in Algorithm 10.

Algorithm 10: Initialization phase of WC-EBBA*$_{par}$ and WC-BA*

Input: The problem instance (G, **cost**, *start*, *goal*) with the weight limit W
Output: WCSPP's bidirectional heuristic functions **h** and **ub**, global upper bounds $\bar{\mathbf{f}}$, also non-dominated states S'

1 Set global upper bounds: $\overline{f_1} \leftarrow \infty$ and $\overline{f_2} \leftarrow W$
2 **do in parallel**
3 $h_2^f, ub_1^f \leftarrow$ Run f_2-bounded backward A* (from *goal* using an admissible heuristic) on $cost_2$, use $cost_1$ as a tie-breaker, update $\overline{f_1}$ with $ub_1^f(start)$ when *start* is going to get expandedand stop before expanding a state with $f_2 > \overline{f_2}$.
4 $h_1^b, ub_2^b \leftarrow$ Run f_1-bounded forward A* (from *start* using an admissible heuristic) on $cost_1$, use $cost_2$ as a tie-breaker, and stop before expanding a state with $f_1 > \overline{f_1}$.
5 **do in parallel**
6 $h_2^b, ub_1^b \leftarrow$ Run f_2-bounded forward A* (using h_2^f as an admissible heuristic) on $cost_2$, use $cost_1$ as a tie-breaker, ignore unexplored states in the previous round lines 3 and 4,update $\overline{f_1}$ via paths matching if feasible and stop before expanding a state with $f_2 > \overline{f_2}$.
7 $h_1^f, ub_2^f \leftarrow$ Run f_1-bounded backward A* (using h_1^b an admissible heuristic) on $cost_1$, use $cost_2$ as a tie-breaker, ignore unexplored states in the previous round lines 3 and 4,update $\overline{f_1}$ via paths matching if feasible and stop before expanding a state with $f_1 > \overline{f_1}$.
8 $S' \leftarrow$ non-dominated states explored in both searches of the last round (lines 6 and 7)
9 **return** $(\boldsymbol{h}^f, \boldsymbol{h}^b)$, $(\boldsymbol{ub}^f, \boldsymbol{ub}^b)$, $\bar{\mathbf{f}}$, S'

1. *Round one*: The algorithm performs a bounded forward search on $cost_1$ and, in parallel, a bounded backward search on $cost_2$. As the initial global upper bound $\overline{f_1}$ is not known beforehand, the bounded search on $cost_2$ updates the global upper bound $\overline{f_1}$ with $ub_1^f(start)$ as soon as it computes $h_2^f(start)$. Since both searches explore the graph concurrently and have direct access to shared parameters, the forward A* search on $cost_1$ will turn into a bounded search as soon as $\overline{f_1}$ gets updated by the concurrent search in the opposite direction. When the parallel searches of the first round terminate, the initialization phase has computed two heuristic functions, namely h_2^f and h_1^b. There are two cases where we can terminate the parallel searches of the first round early without needing to go into the second round of concurrent searches: (1) the problem has no optimal solution if we have found $h_2^f(start) > W$; (2) if the shortest path on $cost_1$ has been found feasible and $ub_2^b(goal) \leq W$.
2. *Round two*: The algorithm runs a complementary backward A* search on $cost_1$ and, at the same time, a bounded forward A* search on $cost_2$. Benefiting from the results of the first round, the searches of the second round use two functions h_2^f and h_1^b as informed heuristics to guide A*. Hence, we can expect faster searches in round two. The second round will also perform two tasks during the search. First, it only explores the expanded states of round one, that is, none of the searches in the second round will explore states identified as out-of-bounds in round one. Second, it tries to update the global upper bound $\overline{f_1}$ with partial path matching. Since each state expanded in the searches of the second round has access to at least one complementary shortest path obtained via the first round, we can update the global upper bound $\overline{f_1}$ if joining partial paths with their complementary optimum paths yields a valid *start-goal* path.

When the initialization phase is complete, WC-EBBA*$_{par}$ uses lower bound heuristics h_1^f and h_1^b to determine the search budget factors β^f and β^b as in WC-EBBA* (via Equation (3) in SM. 2). It then initializes a (shared) node pair *Sol* with an unknown secondary cost f_2^{sol} to be updated during the main search. Note that the order of searches in the initialization phase of WC-EBBA*$_{par}$ is slightly different from that of WC-EBBA*. The sequential preliminary searches in WC-EBBA* allows us to first reduce the graph size via two bounded searches on $cost_2$, followed by two complementary searches on $cost_1$. However, this ordering might not help reduce the graph size in the parallel scheme, mainly because we would not be able to use heuristics obtained in one search to inform the search in the opposite direction. For example, if we run concurrent forward and backward searches on $cost_2$ in the first round, there would be no lower bound heuristics for either of the searches, and thus they would turn into less informed bounded A*. The other possible configuration could be running parallel searches of each round in one direction, for example, searching backwards on $cost_1$ and $cost_2$ in the first round. The main problem with this configuration is that heuristics obtained in the direction of the second round would become much more informed than heuristics of the first round in the opposite direction. Hence, the search in one direction would always be weaker on both $cost_1$ and $cost_2$ heuristics.

In summary, our proposed parallel search ordering ensures that: (1) the second round can benefit from the heuristics obtained in the first round to reduce the graph size, delivering S' to the next phase as a subset of valid states explored in both rounds;

Algorithm 11: WC-EBBA*$_{\text{par}}$ search

Input: Problem (G, **cost**, *start*, *goal*), search direction d, global upper bounds $\bar{\mathbf{f}}$, search heuristics (**h**, **ub**), budget factors β^f and β^b, and an initial solution *Sol* with the secondary cost $f_2^{\min}$
Output: A node pair corresponding with an optimal solution

```
1 Setup(d)                                                        ▷ Procedure 1
2 d' ← opposite direction of d
3 while Open^d ≠ ∅ do
4     x ← A node with the lexicographically smallest (f1,f2) costs in Open^d
5     Remove node x from Open^d
6     if f1(x)>f̄1 then break
7     Explore(x, d, d')                                           ▷ Procedure 12
8 return Sol
```

Procedure 12: Explore(x, d, d')

```
1 if g2(x) ≥ g^d_min(s(x)) then continue
2 g^d_min(s(x)) ← g2(x)
3 ESU(x, d) in (f1,f2) order                                      ▷ Procedure 3
4 if h^d_1(s(x)) = ub^d_1(s(x)) then continue
5 if g2(x) ≤ β^d ×f̄2 then
6     ExP(x, d) in (f1,f2) order                                  ▷ Procedure 2
7 if h^d_2(s(x)) ≤ β^d' ×f̄2 or |χ^d'(s(x))| > 0 then
8     Match(x, d')                                               ▷ Procedure SM. 4
9     Store(x, d)                                                ▷ Procedure SM. 5
```

(2) each constrained search in the next phase can benefit from a set of informed heuristics for either pruning or guiding the search in the bidirectional setting; (3) the initialization time is improved as there are only two rounds of bounded A*.

5.2 | Constrained search

In this phase, WC-EBBA*$_{\text{par}}$ executes two constrained bidirectional searches in parallel. The parallel framework allows both constrained searches of WC-EBBA* to explore the graph concurrently, so the search is no longer led by one direction at a time. Algorithm 11 shows the main procedures of the constrained search in the generic direction d.

Algorithm description: Given $\bar{\mathbf{f}}$ and *Sol* as the shared data structures of the algorithm, each (constrained) search starts with initializing the search in direction d via the Setup(d) procedure and also with identifying d' as the opposite direction. At this point, the search has one node in $Open^d$ associated with the initial state in direction d. In contrast to WC-EBBA*, each search in WC-EBBA*$_{\text{par}}$ only works with one priority queue. As can be seen in line 4 of Algorithm 11, the constrained search in direction d is continued by exploring nodes in $Open^d$ in the (f_1, f_2) order. Given x as the least cost node in the priority queue, the algorithm removes x from $Open^d$ and checks it against the termination criterion via line 6. If the algorithm finds x (as the least cost node) out-of-bounds, the search in direction d can safely terminate (see Lemma SM. 1.9). Otherwise, if the node's estimated cost is within the bounds, the search will follow the standard procedures of WC-EBBA* to explore x as a valid node. The procedure includes matching x with candidate nodes of the opposite direction stored in $\chi^{d'}(s(x))$ and then storing x in $\chi^{d}(s(x))$ for future expansions with $s(x)$ in direction d', only if x is in the coupling area (see SM. 2 for more details). For the sake of clarity, we present the main steps involved in node exploration via Explore (x, d, d') in Procedure 12, with its refined Match(x, d') and Store(x, d) methods described in Procedures SM. 4 and SM. 5, respectively.

Unlike WC-EBBA* where the search is led by the least-cost node of one direction, WC-EBBA*$_{\text{par}}$ runs two parallel searches with independent priority queues. This means that there are no dependencies between the f_1-value of nodes in the forward and backward priority queues, and it is very likely for one of the directions to show faster progress on f_1-values. In other words, the faster search is no longer halted by the slower search, and WC-EBBA*$_{\text{par}}$ can potentially find the optimal solution faster than WC-EBBA*. Note that when WC-EBBA* extracts a node x from $Open^d$, it ensures that all nodes in $Open^{d'}$ have a primary cost

no smaller than $f_1(x)$. Given this important feature in WC-EBBA*$_{\text{par}}$, we now prove the correctness of the algorithm (see SM. 2.3 for the correctness of WC-EBBA* without tie-breaking).

Theorem 2. WC-EBBA*$_{\text{par}}$ *returns a node pair corresponding with a* **cost***-optimal solution path for the WCSPP.*

Proof. As WC-EBBA*$_{\text{par}}$ employs WC-EBBA*'s procedures for its node expansion, the correctness of the involved strategies are directly derived from WC-EBBA*'s correctness (provided in SM. 2). Thus, we just discuss the correctness of the stopping criterion. We already know that when the search in direction d terminates, there is no promising node in the corresponding priority queue with a cost estimate smaller than the best-known solution cost $\overline{f_1}$. However, since both searches use the same stopping criterion, WC-EBBA*$_{\text{par}}$ will not terminate until both (constrained) searches terminate. Generally, if one direction has already completed its search by surpassing the global upper bound $\overline{f_1}$, the algorithm waits for the other (active) direction to confirm the optimality of the solution stored in *Sol*. In this situation, if the active direction finds a better solution, it can still update $\overline{f_1}$ and *Sol* accordingly, otherwise, *Sol* remains optimal. In either case, the active search will eventually confirm the optimality of the existing solution by surpassing the established upper bound. Furthermore, reducing the global upper bound $\overline{f_1}$ in the active search does not affect the correctness of the other (terminated) search, essentially because the algorithm never builds a solution path with invalid nodes. Therefore, WC-EBBA*$_{\text{par}}$ guarantees the optimality of *Sol* when it terminates. ∎

Memory: Both WC-EBBA* and WC-EBBA*$_{\text{par}}$ use the same data structures to initialize their bidirectional searches, some of them shared between the searches in WC-EBBA*$_{\text{par}}$. Therefore, there is no difference in the minimum space requirement of the algorithms. However, there is a minor search overhead associated with parallel search in WC-EBBA*$_{\text{par}}$. Recall the queuing strategy in WC-EBBA*. The sequential search guarantees that the entire algorithm never explores nodes with an f_1-value larger than the optimal cost $\overline{f_1}$. However, this is not always the case in WC-EBBA*$_{\text{par}}$. As an example, the algorithm may terminate its constrained search in direction d with a tentative solution of cost $\overline{f_1}$, but later discovers an optimal solution pair (x, y) with the primary cost $f_1' < \overline{f_1}$ in the opposite direction d'. In this case, it is not difficult to see that the expansion of nodes with f_1-values larger than the optimal cost f_1' in direction d was unnecessary. Therefore, we expect WC-EBBA*$_{\text{par}}$ to use more space than WC-EBBA* due to this resulting search overhead.

6 | PRACTICAL CONSIDERATIONS

As A*-based algorithms enumerate all valid paths, the size of the *Open* lists can grow exponentially during the constrained search. This case is more serious in A* with lazy dominance tests, essentially because priority queues also contain dominated nodes. Furthermore, we can similarly see that the number of generated nodes will show exponential growth and the search may need significant memory to store all nodes for solution path construction. For instance, A*-algorithms can easily generate and expand billions of search nodes in hard problems. Following the practical considerations presented in Ahmadi et al. [2] for the bi-objective search, we now describe two techniques to handle search nodes generated in the constrained search more efficiently.

6.1 | More efficient priority queues

The performance of constrained A* search with lazy dominance test can suffer when the queue size grows to very large numbers of nodes. Contrary to eager dominance or other conservative approaches where the search rigorously removes dominated nodes in the queue, as in Pulido et al. [26], none of our A* searches tries to remove the dominated nodes from the queue unless they are extracted. There are some Dijkstra-like approaches [12] in the literature that only keep one best candidate node of each state in the priority queue, as in Sedeño-Noda and Colebrook [30]. However, our A*-based algorithms do not work with such substitution of nodes during the search. Therefore, we need to invest in designing efficient priority queues to effectively order significant numbers of nodes in constrained search with A*. We describe our node queuing strategies as follows.

In all of our A*-based algorithms, the lower and upper bounds on the f_p-values of the search nodes are known prior to the constrained searches, namely via the heuristic function h_p^d and the global upper bound $\overline{f_p}$. Let $[f_{\min}, f_{\max}]$ be the range of all possible f_p-values generated by A* in the (f_p, f_s) order. To achieve faster operations in our *Open* lists, we use fixed-size bucket-based queues. Although there may be cases where the number of nodes in the priority queue is bounded and the bucket list is sparsely populated, for the majority of cases where the number of nodes grows exponentially in our A* searches, we expect to see almost all the buckets filled. To this end, we investigate two types of bucket-based priority queues based on the multi-level bucket data structures in the literature [9–11]. Consider a bucket list with $\Delta f \in \mathbb{N}^+$ (as a fixed parameter) identifying

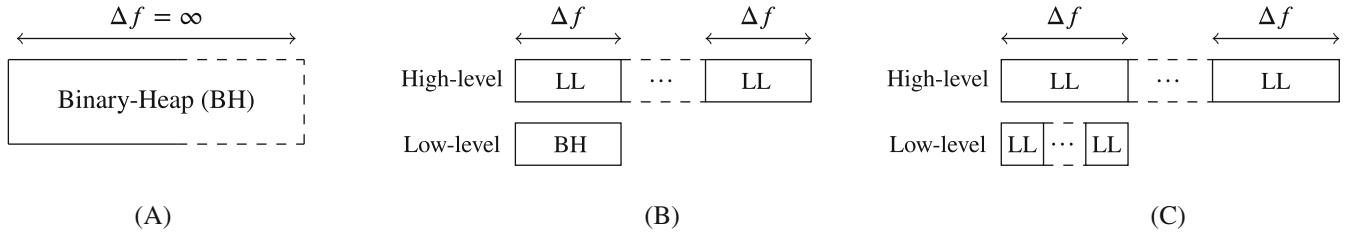


FIGURE 2 Schematic of priority queues studied: (A) Conventional binary heap (BH) queue with no limit on f_p-values; (B) hybrid queue with buckets in the higher level and binary heap in the lower level. Δf denotes bucket width; (C) two-level bucket queue with linked list (LL) in buckets of both levels. The width of the low-level buckets is one.

the bucket width. We limit the size of the bucket list such that a node with $f_{\min}$ (resp. $f_{\max}$) is always placed in the first (resp. last) bucket, that is, we have

$$\text{Bucket size } (BS) = \left\lfloor \frac{f_{\max} - f_{\min}}{\Delta f} \right\rfloor + 1. \tag{1}$$

Given BS and Δf denoting the bucket size and width, respectively, we now explain each queue type as follows.

1. Bucket queue: a two-level queue with buckets in both levels [9, 10]. A high-level bucket $i \in \mathbb{N}^0_{<BS} = \{i \in \mathbb{N}^0 | i < BS\}$ contains all nodes whose f_p-value falls in the $[f_{\min} + i \times \Delta f, f_{\min} + (i+1) \times \Delta f - 1]$ range except for the nonempty high-level bucket with the smallest index k. A node x with $f_p(x)$ in the $[f_{\min} + k \times \Delta f, f_{\min} + (k+1) \times \Delta f - 1]$ range is maintained in the low-level bucket. The size of the low-level bucket is Δf, so we have one bucket for every distinct f_p-value in the range. In other words, a low-level bucket $j \in \mathbb{N}^0_{<\Delta f} = \{j \in \mathbb{N}^0 | j < \Delta f\}$ contains all nodes whose f_p-value is equal to $f_{\min} + k \times \Delta f + j$. To handle nodes in each bucket, we can use a linked list structure with two node extraction strategies: first-in, first-out (FIFO) or last-in, first-out (LIFO). This queue is only able to handle integer costs.
2. Hybrid queue: a two-level priority queue with buckets in the higher level and a binary heap in the lower level [10]. Similar to the bucket queue, a high-level bucket $i \in \mathbb{N}^0_{<BS} = \{i \in \mathbb{N}^0 | i < BS\}$ contains all nodes whose f_p-value falls in the $[f_{\min} + i \times \Delta f, f_{\min} + (i+1) \times \Delta f - 1]$ range except for the nonempty high-level bucket with the smallest index k. A node x with $f_p(x)$ in the range $[f_{\min} + k \times \Delta f, f_{\min} + (k+1) \times \Delta f - 1]$ is maintained in the low-level binary heap structure. This queue type can handle both integer and non-integer costs.

Figure 2 illustrates our priority queues and compares them against the traditional binary heap queue. In this figure, Δf denotes the bucket width. We can see that the hybrid queue can be converted into a form of binary heap if we pick a large enough bucket width. We can also convert our two-level bucket queue into a one-level bucket queue [11] by simply setting $\Delta f = 1$. In this case, nodes can be directly added (removed) to (from) the high-level buckets and the use of the low-level bucket is no longer necessary.

Queue operations: To add new nodes to the queue, we simply insert them into high-level buckets corresponding with their f_p-value. To find and extract the least-cost node, since f_p-values in A* are monotonically non-decreasing, we can simply scan the high-level buckets from left ($f_{\min}$) to right ($f_{\max}$). The first bucket is always a non-empty bucket when our constrained searches are initialized. Let k be the smallest index of a non-empty high-level bucket. We transfer all nodes in the nonempty high-level bucket k to the low-level structure for upcoming extractions. In the meantime, if the queue receives a new node with an f_p-value corresponding with index k, we directly add the node to the low-level data structure. Nodes in the low-level structure are explored during the search in the order of their f_p-values. Before each node extraction, if all nodes in the low-level data structure are already extracted, we increase k to the index of the next non-empty (high-level) bucket and then transfer all nodes of the non-empty bucket to the low-level data structure. Nodes are extracted from the low-level structure only.

Tie-breaking: The bucket queue with linked lists is obviously unable to handle tie-breaking, but it instead offers fast queue operations. In the hybrid queue with binary heaps, however, we can break ties between f_p-values by simply comparing nodes on their f_s-values in the low-level binary heap.

6.2 | Memory efficient backtracking

Node generation is a necessary part of all of our WCSPP algorithms with A*. Each node occupies a constant amount of memory and represents a partial path. A typical node representation would contain information such as **f**-values, the node's corresponding state, its position in the priority queue, and its parent, required for solution path construction. Considering the huge number of generated nodes in difficult problems, we use our memory-efficient approach in Ahmadi et al. [2] as part of solution path construction for all of our studied A* algorithms. Since constrained search in A* only expands each node at most once (the

search prunes cycles), every time a node is expanded, we store its backtracking information in compact data structures outside the node and then recycle the memory of the processed node for future node expansions. In other words, since the majority of a node's information will no longer be required for solution path construction (parameters such as position of the node in the queue and **f**-values), we only extract the minimal backtracking information and return the node to the memory manager. Clearly, the memory occupied by nodes pruned during the search can similarly be recycled (see the example given in SM. 4 for further illustration). The final size of the compact structure varies from instance to instance, but it is shown that, compared to the conventional back-pointer approach, the compact approach is on average five times more efficient in terms of memory [2].

7 | EXPERIMENTAL ANALYSIS

We compare our new algorithms with the state-of-the-art solution approaches available for the WCSPP and RCSPP and evaluate them on a set of 2000 WCSPP instances [4] for 12 maps in the 9th DIMACS implementation challenge,[1] with the largest map containing around 24 M nodes and 57 M edges (see SM. 5.1 for the benchmark details). Following the literature, we define the weight limit W based on the tightness of the constraint δ as:

$$\delta = \frac{W - h_2}{ub_2 - h_2} \quad \text{for} \quad \delta \in \{10\%, 20\%, \dots, 70\%, 80\%\}, \tag{2}$$

where h_2 and ub_2 are, respectively, lower and upper bounds on $cost_2$ of *start-goal* paths. In this setup, high (resp. low) values of δ mean that the weight limit W is loose (resp. tight).

Studied algorithms: we consider the award-winning BiPulse algorithm of Cabrera et al. [8], WC-EBBA* [5], and WC-BA* [4] algorithms. For the other constrained search methods RC-BDA* [32], Pulse [21] and CSP [29], although shown to be slower than WC-EBBA* in Ahmadi et al. [5], we evaluate their performance against the benchmark instances in SM. 5.3.

Implementation: We implemented all the A*-based algorithms (WC-A*, WC-BA*, WC-EBBA*, and WC-EBBA*$_{\text{par}}$) in C++ and used the Java implementation of the BiPulse algorithm kindly provided to us by its authors. For WC-BA*, we implemented its standard variant (with the HTF method). Our graph implementation removes duplicate edges from the DIMCAS graphs, that is, if there are two (or more) edges between a pair of states in the graph, we only keep the lightest edge. In addition, for the initialization phase of the A*-based algorithms, we use spherical distance as an admissible heuristic function for both (distance and time) objectives. All the A*-based algorithms use the same type of priority queue and solution construction approach. In particular, since all costs in the benchmark instances are integer, we use bucket queues with linked lists (using the LIFO strategy with $\Delta f = 1$) for the priority queue of constrained A* searches, along with the compact approach discussed in Section 6 for solution path construction. There is no procedure for the solution path construction in the Pulse and BiPulse implementations, so the code only returns optimal costs. All C++ code was compiled with O3 optimization settings using the GCC7.5 compiler. The Java code was compiled with OpenJDK version 1.8.0_292. We ran all experiments on an AMD EPYC 7543 processor running at 2.8 GHz and with 128 GB of RAM, under the SUSE Linux Server 15.2 environment and with a 1-h timeout. In addition, we allocated two CPU cores to all parallel algorithms. There are five algorithms and 2000 instances, resulting in 10 000 runs. However, we realized that BiPulse is unable to handle our two largest maps CTR and USA (400 instances altogether) due to some inefficiencies in its graph implementation. We can similarly see that BiPulse has not been tested on these two graphs in its original paper [8] either. In order to achieve more consistent computation time in the parallel setting, especially in easy instances with small runtime, we perform five consecutive runs of each algorithm and store the results of the run showing the median runtime (algorithms attempt each instance five times). Thus, we perform $5 \times 9600 = 48\,000$ runs. All runtimes we report in this article include initialization time. Our codes, benchmark instances and detailed results are publicly available.[2]

7.1 | Algorithmic performance

We now analyse the performance of the selected algorithms over the benchmark instances as provided in Table 2. We have 160 point-to-point WCSPP instances in each map, except in the USA map, for which we have 240 instances. As stated before, BiPulse was unable to handle instances of two maps. We report for each algorithm the number of solved cases $|S|$, the runtime in seconds and memory usage in MB. Note that because of the difficulties in reporting the memory usage, we allow 1 MB tolerance in our experiments. For the runtime of unsolved cases, we generously report a runtime of 1 h (the timeout) by assuming that the algorithm would have found the optimal solution right after the timeout. We discuss the results in three aspects as follows.

[1]DIMACS - Shortest Paths; 2005. http://www.diag.uniroma1.it/challenge9/.
[2]https://bitbucket.org/s-ahmadi/biobj.

TABLE 2 Number of solved cases $|S|$ (out of 240 for USA and 160 for the other maps), runtime and memory use of the algorithms.

			Runtime(s)			Memory(MB)	
Map	**Algorithm**	**$\|S\|$**	**Min**	**Avg.**	**Max**	**Avg.**	**Max**
NY	WC-A*	160	**0.01**	**0.06**	**0.15**	**2**	**7**
	WC-BA*	160	**0.01**	**0.06**	0.19	3	8
	WC-EBBA*	160	**0.01**	0.08	0.16	**2**	13
	WC-EBBA*$_{par}$	160	**0.01**	**0.06**	**0.15**	3	14
	BiPulse	160	0.46	0.85	2.35	13	126
BAY	WC-A*	160	**0.01**	0.10	0.40	**3**	20
	WC-BA*	160	**0.01**	**0.09**	**0.39**	**3**	**16**
	WC-EBBA*	160	**0.01**	0.13	0.46	4	35
	WC-EBBA*$_{par}$	160	**0.01**	0.11	0.49	5	20
	BiPulse	160	0.54	1.08	4.05	41	596
COL	WC-A*	160	**0.02**	0.22	1.77	**7**	67
	WC-BA*	160	0.03	**0.20**	1.30	8	**60**
	WC-EBBA*	160	0.04	0.25	**1.01**	10	138
	WC-EBBA*$_{par}$	160	0.04	0.22	1.25	11	84
	BiPulse	160	0.73	2.85	27.46	160	1035
FLA	WC-A*	160	0.19	1.45	15.33	**38**	337
	WC-BA*	160	**0.16**	**1.11**	4.92	40	**183**
	WC-EBBA*	160	0.22	1.29	4.99	**38**	199
	WC-EBBA*$_{par}$	160	0.17	1.15	**4.57**	50	212
	BiPulse	160	1.47	35.29	396.44	287	1271
NW	WC-A*	160	0.12	1.83	12.90	**57**	**314**
	WC-BA*	160	**0.10**	2.20	18.50	86	585
	WC-EBBA*	160	0.13	2.11	16.20	113	1039
	WC-EBBA*$_{par}$	160	0.11	**1.81**	**12.54**	138	1084
	BiPulse	160	1.58	163.50	1440.52	849	6586
NE	WC-A*	160	0.17	2.07	31.51	**53**	**557**
	WC-BA*	160	**0.15**	2.19	31.50	80	1003
	WC-EBBA*	160	0.21	2.67	32.00	82	699
	WC-EBBA*$_{par}$	160	0.20	**1.87**	**27.11**	86	735
	BiPulse	156	2.11	237.58	3600.00	496	6350
CAL	WC-A*	160	0.12	4.10	57.78	96	1295
	WC-BA*	160	0.10	1.87	21.08	**57**	**711**
	WC-EBBA*	160	0.12	1.93	11.71	60	460
	WC-EBBA*$_{par}$	160	**0.09**	**1.57**	**10.31**	78	834
	BiPulse	160	2.26	154.10	2640.61	518	5068
LKS	WC-A*	160	0.09	25.45	298.96	**507**	**4246**
	WC-BA*	160	0.08	28.18	334.46	830	8122
	WC-EBBA*	160	**0.07**	29.14	349.12	872	5839
	WC-EBBA*$_{par}$	160	**0.07**	**20.13**	**244.63**	1022	8294
	BiPulse	109	2.76	1439.84	3600.00	980	5134
E	WC-A*	160	**0.07**	36.16	349.42	**673**	**5180**
	WC-BA*	160	0.09	35.72	366.17	984	7496
	WC-EBBA*	160	0.10	43.15	418.95	1388	10 581
	WC-EBBA*$_{par}$	160	0.10	**33.45**	**310.04**	1522	13 373
	BiPulse	99	3.66	1617.31	3600.00	573	4000
W	WC-A*	160	**0.32**	**31.25**	**349.29**	**560**	**4990**
	WC-BA*	160	0.43	38.57	412.05	920	6794
	WC-EBBA*	160	0.36	40.62	367.12	985	6796
	WC-EBBA*$_{par}$	160	0.39	35.66	390.92	1294	13 354
	BiPulse	101	7.76	1620.46	3600.00	547	5421

TABLE 2 (Continued)

Map	Algorithm	$\|S\|$	Runtime(s)			Memory(MB)	
			Min	Avg.	Max	Avg.	Max
CTR	WC-A*	160	0.27	74.62	663.09	**1063**	**7548**
	WC-BA*	160	**0.26**	80.27	631.31	1738	11 073
	WC-EBBA*	160	0.29	99.39	758.78	2223	17 463
	WC-EBBA*$_{par}$	160	**0.26**	**67.13**	**536.69**	2257	17 508
	BiPulse	-	-	-	-	-	-
USA	WC-A*	236	0.20	471.61	3600.00	5272	35 783
	WC-BA*	240	**0.17**	394.11	3444.40	8577	68 279
	WC-EBBA*	240	0.26	379.19	2120.83	7709	63 995
	WC-EBBA*$_{par}$	240	0.21	**298.72**	**1651.07**	8791	66 932
	BiPulse	-	-	-	-	-	-

Note: Runtime of unsolved instances is assumed to be 3600 s. Memory is reported only for the solved cases and 1 MB=10^3 KB.

Solved cases: All A*-based algorithms have been able to fully solve all instances of 11 maps. For the BiPulse algorithm, however, we can see it has been struggling with some instances from the NE, LKS, E and W maps. Hence, we could expect even more unsolved cases in the larger maps CTR and USA if we were able to evaluate BiPulse on that set of instances. Comparing the number of solved instances of A*-based algorithms in the USA map, we can see that WC-EBBA* (both sequential and parallel versions) and WC-BA* are the best-performing algorithms with all instances solved. For WC-BA*, we can see that it solves its most difficult instance just 2.5 min before the timeout. For WC-A*, however, we see it is the only algorithm that shows unsolved cases within the 1-h timeout when compared with its A*-based competitors.

Runtime: We report the minimum, average, and maximum runtime of the algorithms in each map. Boldface values denote the smallest runtime among all algorithms. We can see that the runtime of all algorithms increases with the graph size, and we have obtained larger values in larger graphs. From the results, it becomes clear that the improved bidirectional search scheme in BiPulse does not contribute to runtimes competitive with A* and the depth-first search nature of BiPulse is still a potential reason for its poor performance on large graphs. Comparing the average runtime of BiPulse against the A*-based algorithms, we see it can be up to two orders of magnitude slower than its competitors. Among the A*-based methods, we can nominate our WC-EBBA*$_{par}$ as the fastest algorithm due to showing smaller runtimes overall, specifically in larger maps CTR and USA by showing the maximum runtime of 28 min. For more detailed runtime analyses, please see SM. 5.2.

Memory: Table 2 also presents the average and maximum memory usage of the algorithms over the solved instances. The results show that, although BiPulse's implementation does not handle solution path construction, its memory requirement is considerably higher than its competitors. Among the A*-based algorithms, we can see that WC-A* and WC-BA* show smaller values than WC-EBBA* and its parallel variant WC-EBBA*$_{par}$. In the E map, for example, the results show that the maximum memory requirement of WC-EBBA* and WC-EBBA*$_{par}$ in difficult instances can roughly be as big as 10.6 and 13.4 GB, respectively, but both WC-A* and WC-BA* manage such instances with about 5.2–7.5 GB of memory in nearly the same amount of time. Although time and space are highly correlated in search strategies and slower algorithms are likely to generate more nodes, we can see how frontier collision in WC-EBBA* contributes to faster runtime but in turn higher memory usage due to the partial path matching procedure. In the NW, E and LKS maps for example, the results show that WC-A* outperforms WC-EBBA* in terms of runtime while using two times less memory on average. WC-A* also shows comparable performance to WC-EBBA*$_{par}$ in the NW map (only 20 ms slower on average) but consumes about 2.4 times less memory on average. Comparing the average memory usage of WC-A* and WC-BA*, we realize that WC-BA* might consume up to 65% more space than WC-A* on average, and in almost all of our large graphs (except the USA map), they both perform better than WC-EBBA* and WC-EBBA*$_{par}$ in memory use. We have provided a more detailed memory analysis of the algorithms in SM. 5.2.

7.2 | Performance impact of constraint tightness

We now analyse the performance of each algorithm with varying levels of tightness. In the experiment setup, we used eight values for the tightness of the WCSPP constraint, namely values ranging from 10% to 80%. We use box plots for all of our constraint-based analyses to show the distribution of values over the instances. For the sake of clarity, we show in Figure 3 what type of statistical information each box plot presents.

To better study the strengths and weaknesses of the algorithms across various levels of tightness, we need to undertake a form of one-to-one comparison. To this end, we define our baseline to be a virtual oracle that cannot be beaten by any of the

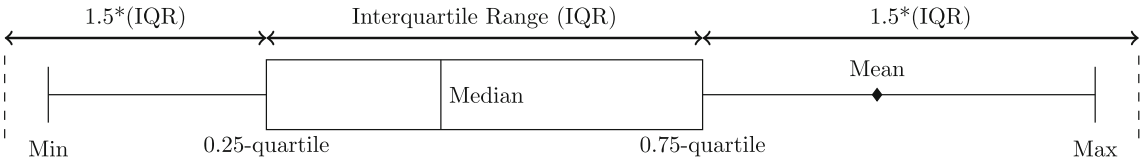


FIGURE 3 A schematic of box-plot visualizing the distribution of data: The plot shows the minimum, the first quartile (25% of data), the median, the third quartile (75% of data), the mean and the maximum (minimum and maximum within the 1.5× IQR).

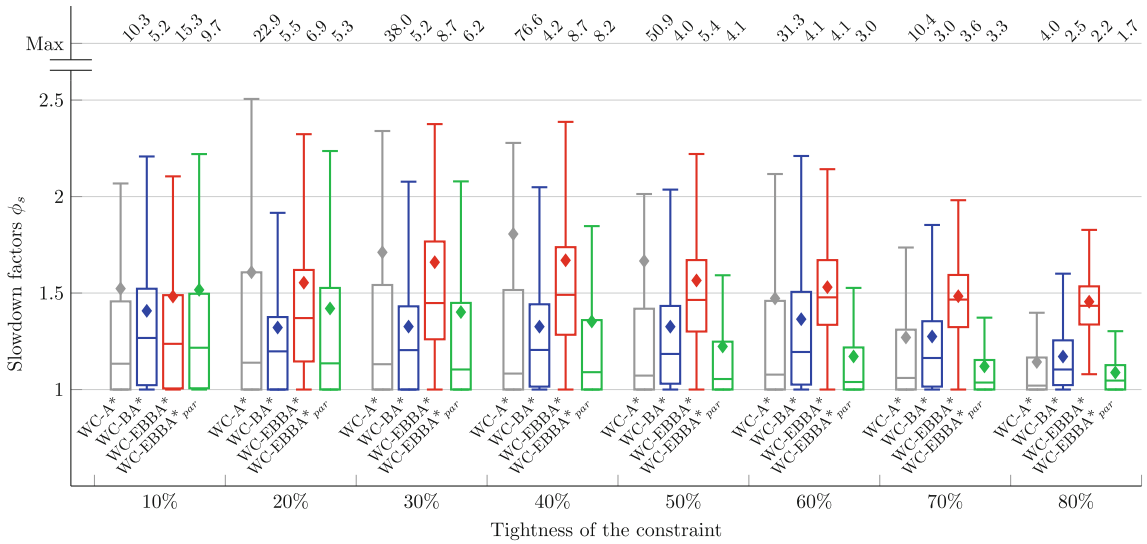


FIGURE 4 The distribution of slowdown factors for all the A*-based algorithms against the virtual best oracle in each level of tightness. Values above the plots show the maximum slowdown factor observed in the experiments (outliers are not shown).

algorithms in terms of runtime. In other words, for every instance, the virtual oracle is given the best (i.e., smallest) runtime of all algorithms. Given the virtual best oracle as the baseline, we then calculate for every runtime (across all algorithms) a slowdown factor ϕ_s with the virtual oracle's runtime as the baseline, that is, we have $\phi_s \geq 1$ and at least one algorithm with $\phi_s = 1$ for every instance. This means that algorithms with ϕ_s-values close to one are as good as the virtual best oracle. For the sake of better readability, this analysis does not include BiPulse as it shows significantly larger slowdown factors compared to our A*-based algorithms, and there was no instance for which BiPulse achieves $\phi_s = 1$ (it never beats the virtual oracle). Figure 4 shows the range of slowdown factors obtained for each algorithm across all levels of tightness.

We describe patterns for each algorithm.

- WC-A*: This algorithm works consistently well across all levels of tightness in terms of the median slowdown factor. However, WC-A*'s performance gets closer to that of the virtual best oracle when the weight constraint becomes loose. The main reason for this fast response in loose constraints is in WC-A*'s simple (unidirectional) initialization phase. Problems with loose constraints normally appear to be easy, mainly because the optimal solution is not far from the initial solution. Therefore, there might be easy cases where the initialization time is the dominant term in the total execution time, making algorithms with a faster initialization phase the best performer. Having compared the maximum slowdown factors of WC-A* with other algorithms, we can observe that WC-A* is the only algorithm that shows the largest maximum slowdown factors in almost all levels of tightness (except in 10%). This means that it will be more likely to get the worst performance with WC-A* than with other A*-based algorithms.
- WC-BA*: Similar to WC-A*, WC-BA* performs well in loose constraints, but is relatively slow in tightly constrained instances. However, WC-BA* shows the best average factor in the 10%–40% tightness range. More importantly, the maximum slowdown factors of WC-BA* are considerably better than that of WC-A*, and even smaller than WC-EBBA* in almost all levels of tightness. We can nominate one potential reason for this behavior. WC-BA* is a bidirectional algorithm, so its worst-case performance is far less severe than WC-A*. More accurately, if the search in one direction is slow in some difficult instances, the concurrent search can help WC-BA* in reducing the search space. In addition, WC-BA* uses bidirectional lower bounds and is also able to tune its heuristics during the search, so it effectively does more pruning in tight constraints and will consequently perform better in the runtime comparison. In loose constraints, WC-A* and WC-BA* perform quite similarly as WC-BA*'s initialization phase is done in parallel, but there may be cases (in the 80% tightness) where WC-A* performs better mainly because the search space is very small and heuristic tuning and parallel search in WC-BA* only adds unnecessary overhead.

- WC-EBBA*: This algorithm presents comparable runtimes only in very tightly constrained instances (10%) and is the weakest solution method in mid-range and loose constraints if we compare algorithms based on their median slowdown factors. However, the average performance of WC-EBBA* is still better than WC-A* in the 10%–60% range. In terms of the worst-case performance (maximum slowdown), WC-EBBA* lies between WC-A* and WC-BA*, but still performs far better than WC-A* mainly due to its bidirectional framework. However, compared to the other bidirectional algorithm WC-BA*, we can see that WC-EBBA* gradually becomes weaker when we move from tight (left) to loose constraints (right) in both median and mean slowdown factors. There is one potential reason for this pattern. WC-EBBA*'s preliminary heuristics are more informed than those of WC-BA* due to sequential heuristic searches. Therefore, WC-EBBA* prunes more nodes and performs faster in very tight constraints. In loose constraints, however, the sequential searches (in both initialization and the constrained search) become the lengthy part of WC-EBBA* and lead to increasing the runtime.
- WC-EBBA*$_{par}$: This algorithm can be seen as the best performer across the 40%–80% range. It also shows the best average factor in the 50%–80% tightness range. Compared to its standard version WC-EBBA*, the parallel variant significantly improves the runtime and worst-case performance (maximum factors) on almost all levels of tightness, especially in loose constraints. This is mainly because of faster initialization via parallelism. However, WC-EBBA*$_{par}$ is slightly weaker than its sequential version in the 10% constraint, mainly because its heuristic functions are not as informed as those of WC-EBBA* with sequential searches. Having compared WC-EBBA*$_{par}$ with the other parallel search algorithm WC-BA*, we can see that WC-EBBA*$_{par}$ outperforms WC-BA* in loosely constrained instances (the 40%–80% range), whereas WC-BA* performs better on average in the 10%–30% range. A potential reason for this behavior is that, unlike WC-BA* where its backward search has limited impact on reducing the search space in loose constraints, bidirectional searches of WC-EBBA*$_{par}$ work in the same objective ordering, and thus they actively contribute to reducing the (already small) search space.

Summary: WC-EBBA* is a good candidate for very tight constraints, as it benefits from more informed heuristics via sequential preliminary searches. WC-BA* can be seen as a great candidate for tightly constrained problem instances. Its backward search is very effective in reducing the search space in such cases. WC-EBBA*$_{par}$ works very well on loosely constrained problems, mainly because its bidirectional searches are both capable of reducing the search space. Finally, WC-A* can be seen as a good candidate for very loose constraints where the search space is already small and does not need better informed heuristics.

A different observation: We focus on the performance of WC-A* and WC-EBBA*. Thomas et al. reported that the unidirectional forward A* search is less effective than the bidirectional search of RC-BDA* on the WCSPP instances of Santos et al. [28]. In their paper, the forward A* search is reportedly 20 times slower than RC-BDA* on average. It is also outperformed by RC-BDA* in all tested levels of tightness. They concluded that the bidirectional nature of RC-BDA* plays a critical role in RC-BDA*'s success. Given WC-EBBA* as the enhanced weight constrained version of RC-BDA*, if we assume our WC-A* is a unidirectional variant of WC-EBBA*, the experimental results in Figure 4 show that the bidirectional search does not always yield smaller runtimes. As we discussed above, there are cases in which unidirectional WC-A* delivers outstanding performance (in both median and mean runtimes) compared to the bidirectional WC-EBBA* (in the 60%–80% range). However, we can confirm that the unidirectional variant is still dominated by the bidirectional variant in tight constraints.

7.3 | On the importance of initialization

In order to investigate the significance of preliminary searches in the initialization phase of our non-parallel algorithms WC-A* and WC-EBBA*, we implemented a version of these two algorithms where the order of preliminary searches is changed to:

(1) Run standard A* on $cost_2$ to obtain an initial upper bound on $cost_1$ and initialize $\overline{f_1}$.
(2) Run bounded A* on $cost_1$ using the upper bound obtained in the previous step.
(3) Run bounded A* on $cost_2$ using the weight limit as the upper bound, and only using the states expanded in the previous step.

In this ordering, the bounded search on $cost_2$ is done in the last step. Note that WC-EBBA* requires steps (2) and (3) above to be done in both directions. Moreover, the bounded search of step (3), in one direction, can be seen as a continuation of step (1). With this extension, we aim to analyse the performance of both WC-A* and WC-EBBA* algorithms with heuristic functions of different qualities. For this analysis, we evaluate both variants of the algorithms on all of our benchmark instances. We perform three runs of each variant and store the run showing the median runtime, and then calculate the speedup factors achieved by using the new variant. The speedup factor is larger than one if the new variant runs faster (with smaller runtime) than the standard version. We have 250 instances in each level of tightness, and unsolved cases are considered to have a runtime of one hour. Figure 5 shows the distribution of speedup factors achieved across all levels of tightness with respect to the standard variants.

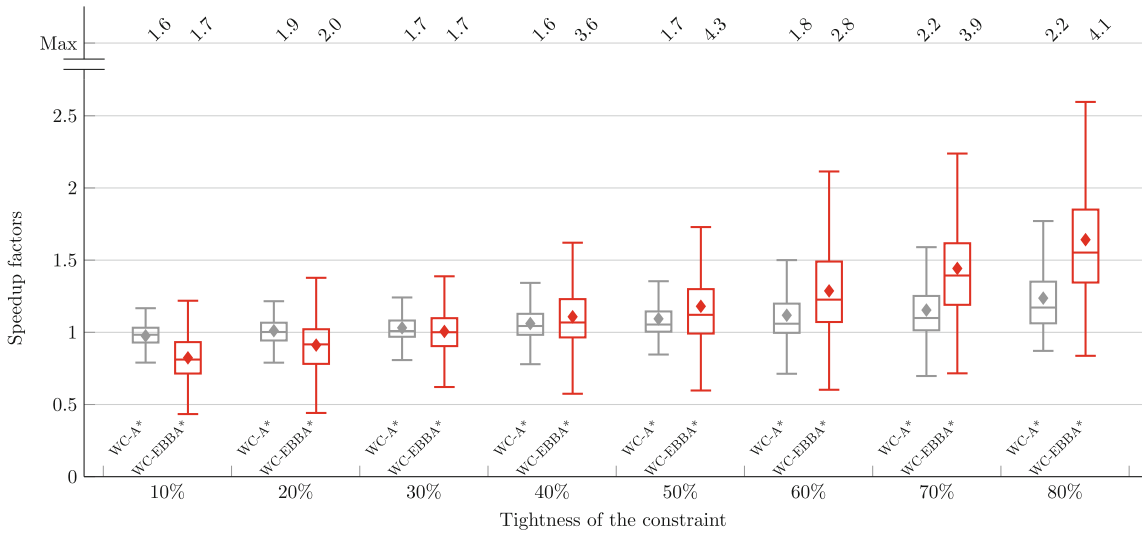


FIGURE 5 The distribution of speedup factors achieved by changing the order of preliminary searches in the initialization phase of WC-A* and WC-EBBA* (outliers are not shown).

The results in Figure 5 highlight that WC-EBBA* is much more affected by the changes in the initialization phase than WC-A*. Looking at the range of speedups achieved in tight constraints (the 10%–30% range), we notice that both algorithms become weaker when we perform bounded searches on $cost_1$ earlier than $cost_2$. In particular, the detailed results illustrate that the new strategy can make WC-EBBA* up to six times slower in the 10% tightness level. However, based on the pattern, the new search order (starting bounded searches with $cost_1$) becomes more effective when we move from tight (left) to loose constraints (right). More precisely, in the 40%–80% range, we see an improvement in the performance of the algorithms (via both median and mean speedup factors) with the new search order. In the 80% tightness level, for example, we see WC-EBBA* performing 60% better than its standard version on average. There are two potential reasons for having such a pattern. First, the upper bound on $cost_2$ (the weight limit) is smaller in tight constraints, so starting with the bounded search on $cost_2$ would be more effective in reducing the graph size in such constraints. Second, our heuristic function (spherical distance) is more informed in the bounded search on $cost_1$ (distance) than $cost_2$ (time). As a result, in loose constraints with larger upper bounds on $cost_2$, we can expect that bounded search on $cost_1$ performs better than $cost_2$ in graph reduction. In terms of the best and worst-case performances in very loose constraints (70% and 80%), we see maximum speedup factors around two and four for WC-A* and WC-EBBA*, respectively, but maximum slowdown factors around 1.15–1.45 for both algorithms. This means that the worst-case performance of the new variants is far less severe in loose constraints.

We can conclude that, depending on the quality of heuristics, both WC-A* and WC-EBBA* (with an admissible heuristic on $cost_1$) can benefit from the proposed (reversed) search order for their initialization phase in loosely constrained instances to perform faster, but such a setting is less effective in tightly constrained problems. Based on this observation, we recommend using procedures that can change the order of bounded searches in the initialization phase based on the tightness of the constraint.

7.4 | Extended experiments: Tie-breaking and priority queues

We now study the significance of priority queues in constrained search with A* and show how disabling tie-breaking and using bucket-based queues can together contribute to substantial improvement in algorithmic performance. Earlier in Section 6, we presented two types of bucket-based queues for the WCSPP: bucket queues with linked lists, and hybrid queues with binary heaps. For the analysis of this section, we also consider binary heaps as the conventional queuing method. Obviously, both binary heaps and hybrid queues (with binary heaps) can order nodes with and without tie-breaking. For both bucket and hybrid queues, we set the bucket width to be $\Delta f = 1$. Bucket queues (with linked lists), however, are not able to break the tie between nodes in cases two nodes have the same primary cost. For this category of queues, we study LIFO and FIFO node extraction strategies instead.

For this extended set of experiments, we choose the unidirectional search algorithm WC-A* as it offers a more stable performance than our bidirectional algorithms in terms of runtime and the number of node expansions. For our benchmark instances, since the objectives (distance and time) in the DIMACS road networks are highly correlated, we design a set of randomized graphs by changing the $cost_2$ of the edges in the DIMCAS maps (the time attribute) with random (integer) values in the $[1, 10\,000]$ range. We do not change instance specifications, that is, (origin, destination) pairs and the tightness values. The new set of (randomized) graphs will help us to investigate the queuing strategies in cases where the search generates more

TABLE 3 Cumulative runtime of the WC-A* algorithm (in *min*) with different types of priority queues per map (**normal weights**).

Queue/map	NY	BAY	COL	FLA	NW	NE	CAL	LKS	E	W	CTR	USA
Bucket-LIFO	0.16	0.24	0.57	3.54	4.38	4.36	10.85	68.29	107.25	89.50	208.19	2025.00
Bucket-FIFO	0.16	0.25	0.59	4.01	4.82	4.81	14.55	72.82	113.27	109.36	222.44	2205.61
Hybrid w/o tie	0.16	0.26	0.64	4.24	5.28	5.22	13.87	80.88	130.08	103.65	248.62	2331.18
Hybrid w tie	0.17	0.28	0.71	5.04	6.49	6.48	16.64	114.37	174.41	146.05	313.83	2875.61
Bin-Heap w/o tie	0.21	0.37	1.06	8.43	11.25	11.24	31.87	225.36	348.82	278.86	602.25	4189.57
Bin-Heap w tie	0.21	0.38	1.07	9.09	12.00	12.63	35.62	257.55	395.29	303.87	681.50	4600.02

Note: We report total time needed to solve all cases of the instance and consider the timeout (1-h) as the runtime of unsolved cases.

TABLE 4 Cumulative runtime of the WC-A* algorithm (in *min*) with different types of priority queues per map (**random weights**).

Queue/map	NY	BAY	COL	FLA	NW	NE	CAL	LKS	E	W	CTR	USA
Bucket-LIFO	0.46	0.39	1.51	44.61	13.13	41.75	46.98	763.38	324.08	683.44	2800.15	6476.14
Bucket-FIFO	0.49	0.42	1.54	42.45	13.86	44.63	46.05	757.41	328.41	693.51	2807.20	6557.63
Hybrid w/o tie	0.52	0.43	1.73	49.18	16.46	47.63	54.44	873.85	377.23	759.35	2953.02	6685.09
Hybrid w tie	0.63	0.48	2.09	65.48	20.41	68.16	69.86	1174.14	529.29	1067.34	3194.13	7138.32
Bin-Heap w/o tie	0.94	0.66	3.40	127.65	37.91	137.14	130.24	1724.87	929.88	1761.51	3666.08	7951.32
Bin-Heap w tie	1.03	0.69	3.51	132.23	37.36	150.44	133.88	1870.27	1050.45	1894.05	3837.63	8217.28

Note: We report total time needed to solve all cases of the instance and consider the timeout (1-h) as the runtime of unsolved cases.

non-dominated nodes with random costs. We ran all experiments on the machine described earlier in the section and with a one-hour timeout (the runtime of unsolved cases). The runtimes we report for this extended experiment are the median of three runs for each instance. We present the cumulative runtime of WC-A* with all types of studied priority queues over the instances of each map in Table 3 for the original DIMACS networks and in Table 4 for the DIMACS graphs with random weights.

Tie-breaking impact: Comparing the performance of WC-A* in Tables 3 and 4 in both settings, that is, with and without tie-breaking, we can see that the algorithm performs better if we simply do not break ties in the hybrid and binary heap priority queues. For the one-level bucket queue with linked lists, the results show that WC-A* with the LIFO strategy outperforms WC-A* with the FIFO strategy in almost all maps. There is one potential reason for this observation. In the LIFO strategy, recent insertions (in each bucket) appear earlier in the queue. Since recent insertions are normally more informed than earlier insertions, the search potentially prunes more dominated nodes in the LIFO strategy. Interestingly, the detailed results show that WC-A* with worst-case tie-breaking (in bucket-queue with the FIFO strategy) leads to up to 100% extra expansions but still runs faster than the variant with tie-breaking (hybrid queue and binary heap with tie-breaking). Further, hybrid queues without tie-breaking are not as effective as bucket queues with linked lists, mainly due to overheads in the two-level bucket-heap data structure. We have provided more detailed analyses on the impacts of tie-breaking in SM. 5.4.

Queue type impact: Comparing the performance of WC-A* based on the priority queues in Tables 3 and 4, we see that bucket queues are consistently faster than the other queue types across all maps, whereas the hybrid queues are ranked second in the head-to-head comparison between the three priority queue types in both tables. Nonetheless, WC-A* with the hybrid queue is still significantly faster than WC-A* with the conventional binary heap queue. In particular, even the hybrid queue with tie-breaking shows better cumulative runtimes than the binary heap without tie-breaking. This observation highlights the effectiveness of using bucket-based queues in the exhaustive search of WC-A*.

Memory: We did not observe any significant difference in the memory usage of WC-A* with the three queue types, as they normally contain nearly the same number of nodes over the course of the search. Nonetheless, hybrid queues can be seen as slightly more efficient than binary heap queues in terms of memory use, as the low-level binary heap in the hybrid queue handles a portion of the total nodes and thus is less likely to grow into a big list that contains all nodes (as in conventional binary heaps).

Bucket width impacts: For our last experiment in this article, we study the impact of the bucket width on the search performance. In particular, we are interested in cases where buckets are required to order nodes based on their primary cost, as in problem instances with non-integer costs via hybrid queues. To this end, we evaluated the performance of WC-A* on both realistic and randomized graphs using bucket widths $\Delta f \in \{10, 100, 1000\}$ in the hybrid queue, with and without tie-breaking. Figure 6 shows the results for the given (increased) bucket widths, along with the two extreme cases $\Delta f = 1$ and $\Delta f = \infty$ (i.e., binary heap). To better see the difference between the plots, we do not show the first 500 instances.

We can see from the results in Figure 6 that increasing the bucket width adversely affects the algorithmic performance of WC-A* on both graph types. In both variants (with and without tie-breaking), WC-A* solves fewer instances if we increase the

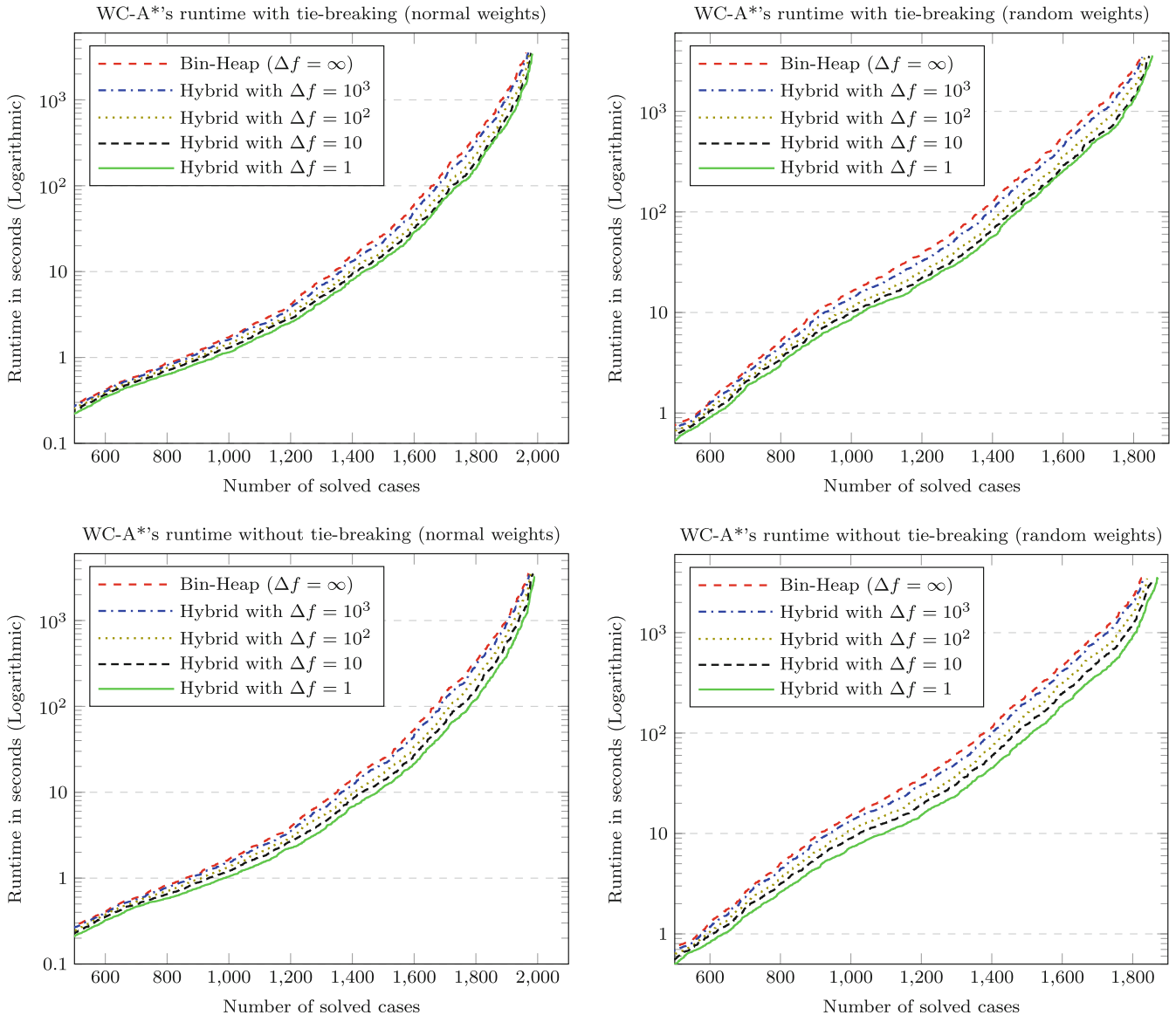


FIGURE 6 Cactus plots of WC-A*'s performance with hybrid queues of different widths with and without tie-breaking on the original DIMACS graphs (left) and randomized graphs (right). The first 500 easy instances are not shown.

bucket width to 10, 100, or 1000. The pattern also shows that WC-A*'s performance with the hybrid queue becomes closer to that of the binary heap queue if we further increase the bucket width Δf. A potential reason for this behavior is related to the number of nodes in each bucket of the hybrid queue. Increasing the bucket width means covering a larger range of f_p-values, and consequently, dealing with more nodes in each bucket. As a result, high-level buckets become more populated and the low-level binary heap operations would take longer.

8 | CONCLUSION AND FUTURE WORK

This article presented two new solution approaches to the hard WCSPP: WC-A* and WC-EBBA*$_{\text{par}}$. WC-A* is a unidirectional algorithm derived from the techniques used in recent bi-objective search algorithms BOA* and BOBA*. Our second algorithm, WC-EBBA*$_{\text{par}}$, is the extended version of the state-of-the-art bidirectional constrained search algorithm WC-EBBA*, enhanced with parallelism. We evaluated our WCSPP algorithms WC-A* and WC-EBBA*$_{\text{par}}$ on very large graphs through a set of 2000 realistic instances and compared their performance against *six* recent WCSPP algorithms in the literature, namely CSP [29], the award-winning algorithms BiPulse [8] and RC-BDA* [32], and our recent algorithms WC-EBBA* [5] and WC-BA* [4]. The results show that the new A*-based algorithms of this article are effective in improving the runtime over the state-of-the-art algorithms. In particular, they both outperform BiPulse in almost all instances by showing up to two orders of magnitude faster runtime on average. We summarize the strengths and weaknesses of each algorithm as follows.

- WC-A*: Very fast on loose constraints and excellent in memory efficiency. This algorithm uses a unidirectional search strategy and can be considered an effective solution approach for problems that cannot be solved bidirectionally. WC-A* outperforms the state-of-the-art WC-EBBA* algorithm on problems with relatively small search space. In addition, it is very space efficient and consumes up to three times less memory than the other studied algorithms. However, compared to bidirectional algorithms, it performs poorly on tightly constrained instances and shows more critical worst cases.
- WC-EBBA*$_{par}$: Very fast on almost all levels of tightness. This algorithm can be a great choice for applications that need fast solution approaches. WC-EBBA*$_{par}$ considerably improves the standard version WC-EBBA* in both runtime and worst-case performance, especially in instances with non-tight constraints, and solves more instances in a limited time than its fast competitors. However, it shows the worst performance in terms of memory usage among the other A*-based algorithms due to its space-demanding path-matching procedure.

We also investigated the importance of the initialization phase in the performance of constrained search with A*. By changing the order of preliminary heuristic searches, we realized that prioritizing the attribute with more informed heuristic can contribute to performing several factors faster in a specific range of constraints, but likely slower in the remaining tightness levels. Therefore, a better initialization procedure can be the one that decides search ordering based on the tightness of the constraint.

Furthermore, we studied two main components of the constrained search with A*: priority queue and lexicographical ordering (tie-breaking). We showed via extensive experiments on realistic and randomized graphs that bi-criteria A* algorithms with lazy dominance procedures can take advantage of bucket-based queues to expedite the queue operations. In addition, we empirically proved that although lexicographical ordering of search objects in the priority queue is generally supposed to be a standard method for node expansion in the literature, we can achieve far better runtimes if we just order search objects based on their primary cost (without tie-breaking). The results of our experiments on large graphs show that the priority queues' effort in breaking ties among search objects in the queue is not paid off by delivering smaller runtimes, as the tie-breaking overhead is normally far higher than the number of pruned objects via tie-breaking. We also studied the impact of bucket width on search performance and observed that, regardless of tie-breaking, hybrid queues with sparsely distributed nodes (via smaller bucket widths) can be more effective than hybrid queues with fewer but (densely) populated buckets.

Future work: We already observed that improving the quality of search heuristics can potentially lead to faster runtime for WC-A* and WC-EBBA* in loose constraints. An interesting direction for future work can be applying the graph reduction technique of branch-and-bound methods to the initialization phase of our A*-based algorithms. For example, we can run several rounds of forward-backward bounded search to remove more nodes from the graph and obtain better quality heuristics for the main search. We currently limit the number of preliminary heuristic searches to at most two rounds. As a further optimization, we can improve the bidirectional heuristics of WC-EBBA* and WC-EBBA*$_{par}$ during the search, like heuristic tuning in WC-BA*. However, as both searches of these algorithms are conducted on the same objective ordering, we can improve primary lower bounds instead. In such a setting, we need to make sure that the consistency requirement of the A* heuristic function is still satisfied.

FUNDING INFORMATION

Research at Monash University is supported by the Australian Research Council (ARC) under Grant numbers DP190100013 and DP200100025 as well as a gift from Amazon. The project was also partly supported by Victorian Government through Victorian Higher Education State Investment Fund scheme.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Bitbucket at https://bitbucket.org/s-ahmadi/biobj/.

ACKNOWLEDGMENT

Open access publishing facilitated by Monash University, as part of the Wiley - Monash University agreement via the Council of Australian University Librarians.

ORCID

Saman Ahmadi https://orcid.org/0000-0002-7326-3384

REFERENCES

[1] S. Ahmadi, G. Tack, D. Harabor, and P. Kilby, "*Vehicle dynamics in pickup-and-delivery problems using electric vehicles*," *Proc. 27th Int. Conf. Princ. Pract. Constraint Program.*, Vol **210**, L. D. Michel (ed.), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Wadern, Germany, 2021, pp. 11:1–11:17. https://doi.org/10.4230/LIPIcs.CP.2021.11.

[2] S. Ahmadi, G. Tack, D. Harabor, and P. Kilby, "*Bi-objective search with bi-directional a**," *Proc. 29th Annu. Eur. Symp. Algorithms*, Vol **204**, P. Mutzel, R. Pagh, and G. Herman (eds.), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Wadern, Germany, 2021, pp. 3:1–3:15. https://doi.org/10.4230/LIPIcs.ESA.2021.3.

[3] S. Ahmadi, G. Tack, D. Harabor, and P. Kilby, "*Bi-objective search with bi-directional a* (extended abstract)*," *Proc. 14th Int. Symp. Comb. Search*, H. Ma and I. Serina (eds.), AAAI Press, Washington, DC, 2021, pp. 142–144 https://ojs.aaai.org/index.php/SOCS/article/view/18563.

[4] S. Ahmadi, G. Tack, D. Harabor, and P. Kilby, "*Weight constrained path finding with bidirectional a**," *Proc. 15th Int. Symp. Comb. Search*, L. Chrpa and A. Saetti (eds.), AAAI Press, Washington, DC, 2022, pp. 2–10 https://ojs.aaai.org/index.php/SOCS/article/view/21746.

[5] S. Ahmadi, G. Tack, D. D. Harabor, and P. Kilby, "*A fast exact algorithm for the resource constrained shortest path problem*," *Proc. 35th AAAI Conf. Artif. Intell.*, AAAI Press, Washington, DC, 2021, pp. 12217–12224 https://ojs.aaai.org/index.php/AAAI/article/view/17450.

[6] Y. P. Aneja, V. Aggarwal, and K. P. K. Nair, *Shortest chain subject to side constraints*, Networks **13** (1983), no. 2, 295–302. https://doi.org/10.1002/net.3230130212.

[7] M. A. Bolívar, L. Lozano, and A. L. Medaglia, *Acceleration strategies for the weight constrained shortest path problem with replenishment*, Optim. Lett. **8** (2014), no. 8, 2155–2172. https://doi.org/10.1007/s11590-014-0742-x.

[8] N. Cabrera, A. L. Medaglia, L. Lozano, and D. Duque, *An exact bidirectional pulse algorithm for the constrained shortest path*, Networks **76** (2020), no. 2, 128–146. https://doi.org/10.1002/net.21960.

[9] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, *Shortest paths algorithms: Theory and experimental evaluation*, Math. Program. **73** (1996), 129–174. https://doi.org/10.1007/BF02592101.

[10] E. V. Denardo and B. L. Fox, *Shortest-route methods: 1. Reaching, pruning, and buckets*, Oper. Res. **27** (1979), no. 1, 161–186. https://doi.org/10.1287/opre.27.1.161.

[11] R. B. Dial, *Algorithm 360: Shortest-path forest with topological ordering [H]*, Commun. ACM **12** (1969), no. 11, 632–633. https://doi.org/10.1145/363269.363610.

[12] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numer. Math. **1** (1959), 269–271. https://doi.org/10.1007/BF01386390.

[13] I. Dumitrescu and N. Boland, *Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem*, Networks **42** (2003), no. 3, 135–153. https://doi.org/10.1002/net.10090.

[14] D. Ferone, P. Festa, S. Fugaro, and T. Pastore, "*On the shortest path problems with edge constraints*," *Proc. 22nd Int. Conf. Transparent Opt. Netw.*, IEEE, New York, 2020, pp. 1–4. https://doi.org/10.1109/ICTON51198.2020.9203378.

[15] R. Garcia. *Resource constrained shortest paths and extensions*, Ph.D. thesis, Georgia Institute of Technology, 2009.

[16] B. L. Golden and D. R. Shier, *2019-2020 Glover-Klingman prize winners*, Networks **79** (2021), 431. https://doi.org/10.1002/net.22072.

[17] G. Y. Handler and I. Zang, *A dual algorithm for the constrained shortest path problem*, Networks **10** (1980), no. 4, 293–309. https://doi.org/10.1002/net.3230100403.

[18] P. E. Hart, N. J. Nilsson, and B. Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Trans. Syst. Sci. Cybern. **4** (1968), no. 2, 100–107. https://doi.org/10.1109/TSSC.1968.300136.

[19] M. Horváth and T. Kis, *Solving resource constrained shortest path problems with LP-based methods*, Comput. Oper. Res. **73** (2016), 150–164. https://doi.org/10.1016/j.cor.2016.04.013.

[20] G. Laporte and M. M. B. Pascoal, *Minimum cost path problems with relays*, Comput. Oper. Res. **38** (2011), no. 1, 165–173. https://doi.org/10.1016/j.cor.2010.04.010.

[21] L. Lozano and A. L. Medaglia, *On an exact method for the constrained shortest path problem*, Comput. Oper. Res. **40** (2013), no. 1, 378–384. https://doi.org/10.1016/j.cor.2012.07.008.

[22] K. Miettinen, *Nonlinear multiobjective optimization*, International Series in Operations Research and Management Science, Vol **12**, Kluwer, Alphen aan den Rijn, The Netherlands, 1998.

[23] R. Muhandiramge and N. Boland, *Simultaneous solution of Lagrangean dual problems interleaved with preprocessing for the weight constrained shortest path problem*, Networks **53** (2009), no. 4, 358–381. https://doi.org/10.1002/net.20292.

[24] I. Pohl, *Bi-directional search*, Mach. Intell. **6** (1971), 127–140.

[25] L. D. P. Pugliese and F. Guerriero, *A survey of resource constrained shortest path problems: Exact solution approaches*, Networks **62** (2013), no. 3, 183–200. https://doi.org/10.1002/net.21511.

[26] F. J. Pulido, L. Mandow, and J. Pérez-de-la-Cruz, *Dimensionality reduction in multiobjective shortest path search*, Comput. Oper. Res. **64** (2015), 60–70. https://doi.org/10.1016/j.cor.2015.05.007.

[27] G. Righini and M. Salani, *Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints*, Discret. Optim. **3** (2006), no. 3, 255–273. https://doi.org/10.1016/j.disopt.2006.05.007.

[28] L. Santos, J. Coutinho-Rodrigues, and J. R. Current, *An improved solution algorithm for the constrained shortest path problem*, Transp. Res. B Methodol. **41** (2007), no. 7, 756–771 http://www.sciencedirect.com/science/article/pii/S0191261507000124.

[29] A. Sedeño-Noda and S. Alonso-Rodríguez, *An enhanced K-SP algorithm with pruning strategies to solve the constrained shortest path problem*, Appl. Math. Comput. **265** (2015), 602–618. https://doi.org/10.1016/j.amc.2015.05.109.

[30] A. Sedeño-Noda and M. Colebrook, *A biobjective Dijkstra algorithm*, Eur. J. Oper. Res. **276** (2019), no. 1, 106–118. https://doi.org/10.1016/j.ejor.2019.01.007.

[31] S. Storandt, "*Quick and energy-efficient routes: Computing constrained shortest paths for electric vehicles*," *Proc. 5th ACM SIGSPATIAL Int. Workshop Comput. Transp. Sci.*, S. Winter and M. Müller-Hannemann (eds.), ACM, Redondo Beach, CA, 2012, pp. 20–25. https://doi.org/10.1145/2442942.2442947.

[32] B. W. Thomas, T. Calogiuri, and M. Hewitt, *An exact bidirectional a⋆ approach for solving resource-constrained shortest path problems*, Networks **73** (2019), no. 2, 187–205. https://doi.org/10.1002/net.21856.

[33] C. Tilk, A. Rothenbächer, T. Gschwind, and S. Irnich, *Asymmetry matters: Dynamic half-way points in bidirectional labeling for solving shortest path problems with resource constraints faster*, Eur. J. Oper. Res. **261** (2017), no. 2, 530–539. https://doi.org/10.1016/j.ejor.2017.03.017.

[34] C. H. Ulloa, W. Yeoh, J. A. Baier, H. Zhang, L. Suazo, and S. Koenig, "*A simple and fast bi-objective search algorithm*," *Proc. 13th Int. Conf. Autom. Plan. Sched.*, J. C. Beck, O. Buffet, J. Hoffmann, E. Karpas, and S. Sohrabi (eds.), AAAI Press, Washington, DC, 2020, pp. 143–151 https://aaai.org/ojs/index.php/ICAPS/article/view/6655.

[35] X. Zhu and W. E. Wilhelm, *A three-stage approach for the resource-constrained shortest path as a sub-problem in column generation*, Comput. Oper. Res. **39** (2012), no. 2, 164–178. https://doi.org/10.1016/j.cor.2011.03.008.

SUPPORTING INFORMATION

Additional supporting information can be found online in the Supporting Information section at the end of this article.

How to cite this article: S. Ahmadi, G. Tack, D. Harabor, P. Kilby, and M. Jalili, *Enhanced methods for the weight constrained shortest path problem*, Networks. **84** (2024), 3–30. https://doi.org/10.1002/net.22210