Noname manuscript No.
(will be inserted by the editor)

Heuristic Search for One-to-Many Shortest Path Queries

**Roni Stern · Meir Goldenberg ·
Abdallah Saffidine · Ariel Felner**

Received: date / Accepted: date

Abstract In this paper we study the One-to-Many Shortest Path Problem (OMSPP), which is the problem of solving k shortest path problems that share the same start node. This problem has been studied in the context of routing in road networks. While past work on routing relied on pre-processing the network, which is assumed to be provided explicitly. We explore how OMSPP can be solved with heuristic search techniques, allowing the searched graph to be given either explicitly or implicitly. Two fundamental heuristic search approaches are analyzed: searching for the k goals one at a time, or searching for all k goals as one compound goal. The former approach, denoted k×A*, is simpler to implement, but can be computationally inefficient, as it may expand a node multiple times, by the different searches. The latter approach, denoted kA*, can resolve this potential inefficiency, but implementing it raises fundamental questions such as how to combine k heuristic estimates, one per goal, and what to do after the shortest path to one of the goals has been found. We propose several ways to implement kA*, and characterize the required and sufficient conditions on the available heuristics and how they are aggregated such that the solution is admissible. Then, we analytically compare the runtime and memory requirements of k×A* and kA*, identifying when each

R. Stern
Ben Gurion University of the Negev, Be'er Sheva, Israel,
Palo Alto Research Center, CA, USA
E-mail: sternron@post.bgu.ac.il,rstern@parc.com

A. Felner
Ben Gurion University of the Negev, Be'er Sheva, Israel
E-mail: felner@post.bgu.ac.il

M. Goldenberg
The Jerusalem College of Technology, Jerusalem, Israel
E-mail: mgoldenbe@gmail.com

A. Saffidine
The University of New South Wales, Sydney, Australia
E-mail: abdallah.saffidine@gmail.com

approach should be used. Finally, we compare these approaches experimentally on two representative domains, providing empirical support for our theoretical analysis. These results shed light on when each approach is beneficial.

Keywords Heuristic Search · Path Finding

1 Introduction

The Shortest Path Problem (SPP) in a graph is a fundamental problem in Computer Science, with many applications in Artificial Intelligence and Operations Research. The input to an SPP instance is a graph and two nodes — a *start* node and a *goal* node — and the task is to find the lowest-cost path in the graph from start to goal. Classical algorithms such as Dijkstra's algorithm [9] and A* [22] have been proposed for solving SPP.

This paper addresses a generalization of SPP, where the task is to solve k SPP problems, such that all the problems share the same start node. That is, we have a single start node and k goal nodes, and the task is to find k shortest paths $\pi_1, \ldots, \pi_k$, where every π_i is a shortest path from the start to the i^{th} goal. This problem is known as the One-to-Many Shortest Path Problem (OMSPP) [8].

Efficient solutions to OMSPP can be helpful in robotics. For example, a path planning engine for multiple drones flying from a central dispatcher location to k target locations will need to solve a OMSPP instance. Another robotic application is when using the Incremental Roadmap Spanner technique [34], which is a motion planning technique that requires searching for a shortest path to a limited number of nearby locations to speed up solving future pathfinding problems. In fact, preliminary work on using heuristic search to solve OMSPP was done exactly for this problem [10].

OMSPP also arises when developing an intelligent agent with a planning component that needs to choose between achieving one of k alternative goals [35, 5]. The preference between these goals may be hard to fully quantify, e.g., requiring human feedback. However, the cost of achieving these goals is an important input to the agent's decision making process. Computing the cost of achieving each of these k goals is exactly an instance of OMSPP. Finally, OMSPP has been studied in the context of path finding in road networks [8], where performing batch one-to-many queries can be useful to optimize the utilization of a GPS navigation server. This prior work focused on pre-processing the network, which was explicitly given. In this work, we explore how heuristic search can be applied to solve OMSPP, and consider also OMSPP in implicitly given, combinatorially large graphs.

A trivial algorithm for solving OMSPP is to run an SPP solver k times, one for each goal. We refer to this algorithm as k×A*. k×A* potentially explores parts of the search space multiple times, introducing redundant overlaps and missing opportunities for sharing information between the k searches. An alternative is to search for the shortest paths to all k goals together, in a single best-first pass of the search space. We call such an algorithm kA*. Like any best-first search, a key question when implementing kA* is how to choose which node to expand next. In A*, this is done by considering for every candidate node its distance to the starting point and a heuristic estimate of its distance to the goal — the g and h values. In kA*, there are k goals and thus k heuristic values. A fundamental question is therefore: how to aggregate these k heuristic values in an effective and admissible way? The **first contribution** of this work is a complete theory of the necessary and sufficient conditions for a heuristic aggregation function that can be used by kA* and guarantee an optimal path to each of the k goals will be found. As we show, the conditions required for this heuristic aggregation function to be admissible depend on the properties of the underlying k heuristics. We also provide sufficient conditions for when the heuristic aggregation function can be used in kA* such that it expands every state at most once.

A fundamental question that arises when implementing kA* is what to do after the shortest path to one of the goals has been found. An **eager** implementation of kA* would then re-compute the heuristic aggregation function for all previously expanded nodes. We propose an alternative implementation of kA* that re-computes the heuristic aggregation function in a **lazy** manner, and characterize when this lazy re-computation preserves the admissibility of kA*. This is the **second contribution** of this work.

Then, we analyze theoretically the runtime and memory requirements of kA* and compare it analytically with the runtime and memory requirements of k$\times$A*. This analysis provides useful guidelines for when to use each algorithm, and constitutes the **third contribution** of this work. Finally, we support this analysis by an empirical study on two standard search benchmarks: Grid path finding and the n-Pancake problem. The results show that in some cases k$\times$A* is better while in other cases kA* is more efficient. In particular, we show while kA* can be advantageous in gird pathfinding, it is almost never useful for the n-Pancake problem, suggesting that for implicitly-given graphs kA* might not be the preferred approach. Moreover, even in explicitly given graphs, it is better in some cases to simply perform a simple variant of Dijkstra's algorithm instead of kA*.

We are not the first to address the OMSPP problem, and some of the algorithmic ideas we present here have been proposed before in various contexts [45, 1, 7, 8]. However, the theoretical contributions in this work go well beyond all prior work we are familiar with.

This paper is structured as follows. Section 2 provides background and basic definitions. Section 3 introduces the kA* algorithm, and Section 4 develops our theory on how to aggregate heuristic values for kA*. Section 5 examines several ways to update the heuristic values of nodes after the shortest path to a goal has been found. Section 6 compares k$\times$A* and kA* theoretically in terms of memory consumption and runtime. Section 7 describes our experimental results, where we compared k$\times$A* and kA* empirically on two domains. Section 8 discusses related work. Section 9 concludes the paper and discusses some directions for future work.

w	A function that maps an edge to a non-negative cost.
G and s	The searched graph and the source, respectively.
$d(x, y)$	The cost of a lowest-cost path from x to y.
t_i	The i^{th} goal in a OMSPP problem.
Π_i	The SPP problem from s to t_i.
A^*_i	An A* for finding a lowest-cost path to goal t_i.
$f(n)$	The evaluation function of A*: $f(n) = g(n) + h(n)$.
kA^*_Φ	kA* using Φ to aggregate heuristic values, e.g. $kA^*_{\min}$.
$F(n)$	The evaluation function used by kA*.
$\text{Gen}(X)$	The nodes generated by algorithm X.
$\text{Gen}_i(kA^*)$	The nodes generated by kA* until t_i is expanded.
$\text{Open}_i(X)$	The nodes in Open when algorithm X expands goal t_i.
$\text{Time}(X)$	The runtime required to run algorithm X.

Table 1 This table summarizes some of the notations used in this paper.

2 Definitions and Background

Let $G = (V, E, w)$ be a finite weighted directed graph, where V is the set of nodes, E is the set of edges, and $w : E \to \mathbb{R}_{\geq 0}$ is a function that assigns non-negative weights to edges. For any edge $(n_1, n_2) \in E$, we denote its weight by $w(n_1, n_2)$. The cost of a path in a graph is the sum its edge weights.

Definition 1 (Shortest Path Problem (SPP)) A shortest path problem (SPP) is defined by a tuple $\langle G, s, t\rangle$, where G is a weighted directed graph, s and t are nodes in G, and we refer to s as the *start* and t as the *goal* (or target). The objective is to find a lowest-cost path in G from s to t.

Note that in some SPP instances, the graph is given explicitly, e.g., representing roadmaps, grids and game maps [40]. In other cases, the graph is defined implicitly by an initial node and a set of transition functions, e.g., representing a space of possible robot configurations, puzzle permutations, or STRIPS-like states in a domain-independent planning problem.

In the rest of the paper, for any pair of nodes x and y we use $d(x, y)$ to denote the minimal cost of a path from x to y. When there are no paths from x to y, we set $d(x, y) = +\infty$. Table 1 lists some of the key notations introduced throughout the paper.

2.1 The A* Algorithm

SPP has been well-studied in the Computer Science literature. A* [22] is a popular heuristic search algorithm for solving SPP. For completeness, we provide a brief description of A* and provide its pseudo code in Algorithm 1. A* maintains two lists of nodes: Open and Closed. Initially, Closed is empty and Open contains only the start s. Every node that is added to Open is associated with a g value and an f value. The g value of a node n, denoted $g(n)$, is the cost of a lowest-cost path found so far from the start s to n. Hence, $g(s)$ is set to zero. The f value of n, denoted $f(n)$, is the sum of $g(n)$ and a heuristic estimate of the cost of a lowest-cost path from n to the goal t. This heuristic estimate is denoted by $h(n)$.

In every iteration of the main loop of A*, the node with the lowest f value in Open is moved from Open to Closed. If that node is the goal, the search halts returning the path found to it.[1] Otherwise, the node is *expanded.* Expanding a node n means generating every successor node. The g value of every generated node c is computed based on the g value of n and the weight of the edge connecting them. Finally, the generated nodes are added to Open with their f values, to be considered for expansion in future iterations. If the searched graph is not a tree, multiple paths to the same node may be explored. To address this, A* keeps track of the nodes it has generated and maintains for every node only the best (lowest cost) path to it found so far (lines 9–12).

Algorithm 1: A*

```
   Input: start $s$, goal $t$)
1  $g(s) \leftarrow 0$; Open $\leftarrow \emptyset$; Closed $\leftarrow \emptyset$
2  Add $s$ to Open with key $f(s) = g(s) + h(s)$
3  while Open $\neq \emptyset$ do
4      $best \leftarrow$ a node from Open with the smallest key
5      Move $best$ from Open to Closed
6      if $best = t$ then return the lowest-cost path found to $best$
7      for every outgoing edge $(best, c)$ do
8          $g_{new} \leftarrow g(best) + w(best, c)$
9          if $c \in$ Open $\cup$ Closed then
10             if $g_{new} \leq g(c)$ then
11                 $g(c) \leftarrow g_{new}$
12                 Update the key of $c$ in Open to $f(c) = g(c) + h(c)$
13         else
14             $g(c) \leftarrow g_{new}$
15             Add $c$ to Open with key $f(c) = g(c) + h(c)$
16 return No solution exists
```

Definition 2 A heuristic h is *admissible* if for every node x, $h(x) \leq d(x,t)$.

In other words, a heuristic is admissible if it never over-estimates the cost of a lowest-cost path to the goal.

Definition 3 A heuristic h is *consistent* if $h(t) = 0$ and for any pair of nodes x and y we have $h(x) \leq d(x,y) + h(y)$.

A* has several important properties. First, if the heuristic used is *admissible*, then A* is guaranteed to solve SPP, i.e., to return a lowest-cost path from

[1]We omit in this pseudocode how back-pointers are maintained to allow reconstructing the best path to each node.

s to t.[2] Second, if the heuristic is *consistent* then A* is guaranteed to never expand a node more than once [22]. Third, under some conditions, it can be shown that up to tie-breaking between nodes with equal f values, A* will only expand the smallest set of nodes required to find an optimal solution [6]. A recent variant of A* provides a similar guarantee with respect to the number of nodes generated as well [20]. This type of guarantees, often referred to as the *optimally effective* property of A*, is important because in many cases the runtime of an algorithm is correlated with the number of nodes expanded/generated. Thus, ensuring that A* expands/generates the smallest set of nodes provides some sort of optimality guarantee for A*'s runtime efficiency compared to other equally informed algorithms.

2.2 The One-to-Many Shortest Path Problem

In this paper, we focus on OMSPP, which is a generalization of SPP, defined as follows.

Definition 4 (One-to-Many Shortest Path Problem (OMSPP)) An *OMSPP instance* is a tuple $\langle G, s, \mathbf{t}\rangle$ where G is a weighted graph, s is the start node, and $\mathbf{t} = \langle t_1, t_2, \ldots, t_k\rangle$ is a vector of k *goal nodes.* A *solution* to an OMSPP instance is a vector of k paths $\mathbf{p} = \langle p_1, \ldots p_k\rangle$ such that for every $i \in [1, k]$ it holds that p_i is a lowest-cost path from s to t_i.

We say that an OMSPP algorithm is *admissible* if for any OMSPP instance it returns an optimal path to every reachable goal. Algorithm 2 outlines a straightforward admissible OMSPP algorithm: run A* k times, one for each of the k goals, and return the k resulting paths. This simple algorithm has been proposed in the literature under different names. For example, Zhao et al. [45] called this algorithm Brute-force Polyanya in the context of solving the k nearest neighbor problem. In this work, we refer to Algorithm 2 as k×A*, and for every goal t_i we denote by A^*_i the A* search used by k×A* to find an optimal path to t_i.

Algorithm 2: k×A*: OMSPP with k A* s

```
Input: start $s$, goals $t_1, \ldots t_k$
1 SOLUTION ← $\emptyset$
2 for $i = 1$ to $k$ do
3     $p_i \leftarrow$ A* $(s, t_i)$
4     Add $p_i$ to SOLUTION
5 return SOLUTION
```

[2]In fact, it is sufficient that the heuristic is admissible only for the nodes that are on a single lowest-cost path for A* to guarantee optimality [27, 6].

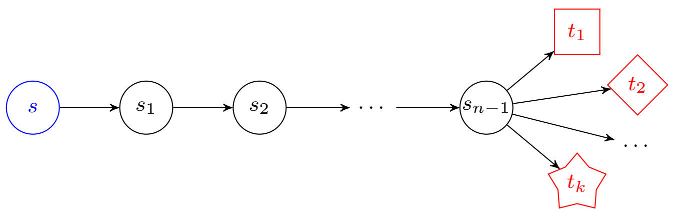


Fig. 1 An example where the k search approach is inefficient.

3 The kA* Algorithm

k×A* overlooks the fact that the sets of nodes generated by the k individual A*s may intersect. As such, some nodes may be expanded in more than one of the k searches. Since expanding a node incurs a computational cost, expanding the same node multiple times should be avoided as much as possible. An extreme example of this inefficiency is depicted in Figure 1. The searched graph is a simple line ending with a star. In this case, k×A* would expand n nodes per sub-search, so it would expand $n+n+\dots+n = n\cdot k$ nodes in total, while it is easy to see that expanding $n+k$ nodes is sufficient to solve this OMSPP instance. The ratio of node generations done by k×A* to the number of node generations needed to solve this instance is $\frac{n\cdot k}{n+k}$ and can be made arbitrarily close to k.

More generally, this potential inefficiency of k×A* stems from the fact that the searches do not share any information. To this end, we explore in the next section an OMSPP algorithm that performs a single search towards all k goals. Compared to k×A*, this single search maintains one Open and one Closed, thereby maximizing information reuse. We call this algorithm kA*. Before presenting a complete pseudo code for kA*, we highlight several aspects that differentiate it from k×A* and from A*.

Maintaining the Set of Active Goals. In kA*, the search does not halt until all of the k goals have been expanded. To this end, kA* tracks the set of goals that have not been expanded yet. We refer to this set of goals as the *active goals* and denote it by $\mathcal{A}$.

Guiding the Search with Multiple Heuristics. In k×A* we implicitly assumed that for each goal t_i there is a corresponding heuristic function h_{t_i}, so that $h_{t_i}(n)$ is the heuristic estimate of the cost to get from node n to goal t_i. In kA*, there is a single search, but we still have access to these k heuristic functions. Formally, every generated node n is associated with a k-ary vector $\mathbf{h}(n) = \langle h_{t_1}(n), \dots h_{t_k}(n)\rangle$. kA* chooses which node to expand in every iteration by considering (1) the g and $\mathbf{h}$ values of every node in Open, (2) the set of active goals $\mathcal{A}$, and (3) a *heuristic aggregation function.* A heuristic aggregation function is a function that accepts as input the set of active goals $\mathcal{A}$ and the heuristics vector $\mathbf{h}$, and outputs a single real value. For example, a possible heuristic aggregation function is one that takes the minimum over the heuristics of the active goals. Formally, we define heuristic aggregation functions as follows.

Definition 5 (Heuristic Aggregation Function) Let k be a fixed positive integer. A *heuristic aggregation function* is a function $\Phi : 2^{\{t_1,\ldots,t_k\}} \times \mathbb{R}^k_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$ such that $\Phi(\mathcal{A}, \mathbf{0}) = 0$ for every $\mathcal{A} \in 2^{\{t_1,\ldots,t_k\}}$, where $\mathbf{0}$ is the k-dimensional zero vector.

The set of active goals $\mathcal{A}$ is a parameter to Φ to allow an implementation of Φ in which the heuristic values for non-active goals (i.e., goals not in $\mathcal{A}$) are not aggregated. This avoids unnecessary computations when a node is generated. kA* computes for every generated node the following value: $F_\Phi(n) = g(n) + \Phi(\mathcal{A}, \mathbf{h}(n))$, where Φ is a given aggregation function. In every iteration, kA* expands a node with the smallest F_Φ value in Open. We denote by kA$^*_\Phi$ the instantiation of kA* that uses a given aggregation function Φ. We discuss in Section 4 various design choices for the heuristic aggregation function Φ, and how they impact kA*'s behavior with respect to the properties of the given heuristic functions. For now, assume the following natural heuristic aggregation function:

$$\Phi\left(\mathcal{A}, \left\langle h_{t_1}(n), \ldots h_{t_k}(n)\right\rangle\right) = \min_{t_i \in \mathcal{A}} h_{t_i}(n) \tag{1}$$

As a notational convenience, when $\mathcal{A}$ is obvious from context, we omit it and denote Φ as a single parameter function, accepting only the vector of heuristics.

Algorithm 3 shows the pseudocode for kA*. Initially, all goals are inserted into the set of active goals (line 2), and s is inserted to Open with its F_Φ value. In every iteration, a node with the smallest F_Φ value, denoted *best*, is selected and removed from Open (line 4). If *best* is an active goal, we store the path to it (line 8) and remove that goal from the active goal set $\mathcal{A}$ (line 9). When this happens, kA* update the F_Φ value of all the nodes in Open, reordering the nodes according to their updated F_Φ value (line 11). When the $\mathcal{A}$ list is empty, we halt the search, having found a path to each goal (line 12). When expanding non-goal nodes, kA* performs a standard node expansion process similar to A*. Note that a node may be generated more than once if a new path to it has been found. This may occur even to nodes that were already expanded moved to Closed (line 19). In such a case, the node will be added again to Open, with the updated F_ϕ value. Such *node re-expansions* may occur often in some cases, which negatively affects runtime. However, in Section 4.1, we elicit conditions over the heuristics and heuristic aggregation function that guarantee such node re-expansions never occur.

kA* halts and returns a solution after all k goals have been expanded (line 12). When a node is expanded, it means a path to it has been found. Thus, kA* is *sound*, in the sense that if it returns a solution then that solution contains a path from s to each of the k goals. A key question is whether each of these k paths is indeed an optimal path to its corresponding goal. We answer this question in the Section 4.

Algorithm 3: kA*

```
Input: start $s$, goals $t_1, \dots, t_k$
1  $g(s) \leftarrow 0$; OPEN $\leftarrow \emptyset$; CLOSED $\leftarrow \emptyset$
2  $\mathcal{A} \leftarrow \{t_1, \dots, t_k\}$
3  Add $s$ to OPEN with key $F_\Phi(s) = g(s) + \Phi(\mathcal{A}, \mathbf{h}(s))$
4  while OPEN $\neq \emptyset$ do
5      $best \leftarrow$ a node in OPEN with the smallest $F_\Phi$
6      Move $best$ from OPEN to CLOSED
7      if $best \in \mathcal{A}$ then
8          Add the path to $best$ to SOLUTION
9          Remove $best$ from $\mathcal{A}$
10         for every node $m$ in OPEN do
11             Update the key of $m$ in OPEN to $F_\Phi(m) = g(m) + \Phi(\mathcal{A}, \mathbf{h}(m))$
12         if $\mathcal{A} = \emptyset$ then return SOLUTION
13     foreach outgoing edge $(best, c)$ do
14         $g_{new} \leftarrow g(best) + w(best, c)$
15         if $c \in$ OPEN then
16             if $g_{new} < g(c)$ then
17                 $g(c) \leftarrow g_{new}$
18                 Update the key of $c$ in OPEN to $F_\Phi(c) = g(c) + \Phi(\mathcal{A}, \mathbf{h}(c))$
19         else if $c \in$ CLOSED then
20             if $g_{new} < g(c)$ then
21                 $g(c) \leftarrow g_{new}$
22                 Remove $c$ from CLOSED
23                 Add $c$ to OPEN with key $F_\Phi(c) = g(c) + \Phi(\mathcal{A}, \mathbf{h}(c))$
24         else
25             $g(c) \leftarrow g_{new}$
26             Add $c$ to OPEN with key $F_\Phi(c) = g(c) + \Phi(\mathcal{A}, \mathbf{h}(c))$
27 return No solution exists
```

Since the underlying graph G is finite, kA* is guaranteed to terminate. Indeed, in every iteration of the main loop kA* either adds a new node to open or closed, or decreases the g value of an existing node. In addition, kA* returns *no solution exists* only if OPEN is empty (line 27). A node is removed from OPEN only after its children have been added to OPEN. So, if kA* reaches an iteration when OPEN is empty it means that some of the k goals are not reachable. Therefore, the following statement holds for any OMSPP problem over a finite graph.

Proposition 1 (Completeness) *If a goal is reachable, then it will eventually be expanded by kA*. If a goal is unreachable, kA* will return* no solution exists.

4 Aggregating Heuristic Values

In this section, we examine conditions under which kA$^*_\Phi$ is admissible. To do so, we examine the traditional assumptions on heuristics—admissibility and consistency—in combination with newly introduced assumptions on aggregation functions. As we shall see, the is a tradeoff between the assumptions made on the k available heuristics $h_1, \ldots, h_k$ and the assumptions made on the heuristic aggregation function Φ: stronger assumptions on the heuristics allow a larger class of heuristic aggregation functions that guarantee that kA^*_Φ is admissible. Moreover, we show that these assumptions over Φ are also *necessary*, in the sense that there are cases where kA^*_Φ is inadmissible without making these assumptions.

As a preliminary, we note the following.

Lemma 1 *In every iteration of* kA^*_Φ*, for every active goal* t_i*, there exists a state* n_i *in* OPEN *that is on an optimal path to* t_i*, i.e.,* $g(n_i) + d(n_i, t_i) = d(s, t_i)$*.*

Lemma 1 can be proven by induction over the iterations of kA^*_Φ. Namely, it trivially holds in the first iteration and continues to hold in subsequent iterations because when a node with $g(\cdot) + d(s, \cdot) = d(s, t_i)$ is expanded then one of its children must also have $g(\cdot) + d(s, \cdot) = d(s, t_i)$. An equivalent to Lemma 1 has been proven for many other best-first search algorithms.

4.1 Consistent Heuristics

Consider first the case where all the k heuristic functions are *consistent* (Definition 3). We provide below conditions over the heuristic aggregation function that are necessary and sufficient to guarantee that kA^* is admissible.

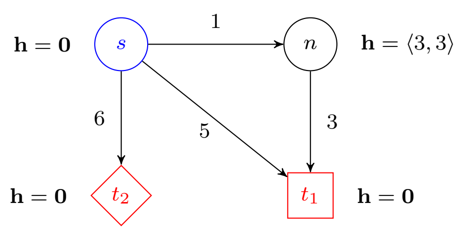
Definition 6 (Consistent heuristic aggregation function) A heuristic aggregation function Φ is *consistent* if for every pair of vectors $\mathbf{v}$ and $\mathbf{u}$, we have that if there exists i such that $u_i = 0$ then $\Phi(\mathbf{v}) - \Phi(\mathbf{u}) \leq \max(\mathbf{v} - \mathbf{u})$, where u_i is the i^{th} element in $\mathbf{u}$.

Theorem 1 (Consistency is a necessary and sufficient condition) *Let* Φ *be a heuristic aggregation function. If* Φ *is consistent, then for any* OMSPP *instance and tuple of consistent heuristics* $\mathbf{h}$*,* kA^*_Φ *is admissible. If* Φ *is not consistent, then there exists an* OMSPP *instance and a tuple of consistent heuristics such that* kA^*_Φ *is not admissible.*

To simplify readability, the proofs for all the theoretical statements in this section are listed in Appendix A. A broad class of heuristic aggregation functions are consistent, including the following natural heuristic aggregation functions.

Corollary 1 Minimum*,* maximum*,* mean*,* median*, and* projection of a single element *in a vector are all consistent heuristic aggregation functions.*

Nevertheless, some heuristic aggregation functions are not consistent, and thus, as stated in Theorem 1, they cannot be safely used in kA^*. A prime


$\mathbf{h} = \mathbf{0}$
s
1
n
$\mathbf{h} = \langle 3, 3\rangle$
6
5
3
$\mathbf{h} = \mathbf{0}$
t_2
t_1
$\mathbf{h} = \mathbf{0}$

Node	g	h_1	h_2	$\sum \mathbf{h}$	$F_{\sum}$
t_1	5	0	0	0	5
t_2	6	0	0	0	6
n	1	3	3	6	7

Fig. 2 Counter-example for $\text{kA}^*_{\sum}$. Running $\text{kA}^*_{\sum}$ returns a suboptimal path to t_1. The table shows the g, $\sum \mathbf{h}$, and $F_{\sum}$ values for n, t_1, and t_2 after expanding s.

example is the *sum* function: $\Phi(\mathbf{v}) = \sum \mathbf{v} = \sum_i v_i$. For example, consider the OMSPP instance defined by the graph depicted in Figure 2, with start vertex s, and goals t_1 and t_2. The heuristic h_1 to goal t_1 is consistent, and so is h_2 for goal t_2. Yet, running $\text{kA}^*_{\sum}$ will find a suboptimal path to t_1. Indeed, the optimal path to t_1 is through n and costs 4, but $\text{kA}^*_{\sum}$ will expand t_1 before n, returning a path of length 5 to t_1.

In the Appendix, we characterize two broad classes of heuristic aggregation functions: one that consists of only heuristic aggregation functions that are consistent (Proposition 3 in Appendix), and includes all the functions given in Corollary 1, and one that consists of only heuristic aggregation functions that are not consistent (Proposition 4 in Appendix) and includes the sum function.

4.2 Node Re-Expansion in kA*

The main benefit of using a consistent heuristic in A* is that it avoids the need to re-expand nodes. This means the complexity of A* with a consistent heuristic is linear in the number of states in the search space. Unfortunately, using a consistent heuristic aggregation function in kA* does not have the same property. For example, the heuristic aggregation function $\lfloor \min(\mathbf{u}) \rfloor$ is consistent, but there are cases where kA* would have to re-expand a node twice to find an optimal solution.

To guarantee that kA* will be admissible and avoid re-expanding nodes, we require a similar but different form of condition for heuristic aggregation functions and the domains in which they are used.

Definition 7 (Re-expansion-avoiding heuristic aggregation function) A heuristic aggregation function Φ is *re-expansion-avoiding* in a given domain if for every pair of nodes n and n' in that domain we have that

$$\Phi(\mathbf{h}(n)) - \Phi(\mathbf{h}(n')) \leq \max(\mathbf{h}(n) - \mathbf{h}(n')) \tag{2}$$

Theorem 2 (Sufficient condition for avoiding re-expansions) *Let Φ be a heuristic aggregation function. If Φ is re-expansion-avoiding, then for any*

OMSPP instance and tuple of consistent heuristics $\mathbf{h}$*, when* kA^*_Φ *expands a node* n *then* $g(n) = d(s, n)$*.*

The direct implication of the above Theorem is that kA* with a re-expansion-avoiding heuristic aggregation function is guaranteed to expand every node at most once. Interestingly, some heuristic aggregation functions are consistent but are not re-expansion-avoiding in some domains. The $\lfloor \min(\mathbf{u}) \rfloor$ function mentioned above is an example of such a function. But, all the heuristic aggregation functions mentioned in Corollary 1 are consistent and also re-expansion-avoiding in all domains.

4.3 Admissible but Possibly Inconsistent Heuristics

So far we assumed that the k heuristics are consistent. This is a strong requirement in some cases, and there are highly effective heuristics that are admissible but not consistent [15]. Relaxing the consistency requirement for the heuristics requires stricter restrictions on the heuristic aggregation function, as described below.

Definition 8 A heuristic aggregation function Φ is *admissible* if for every vector $\mathbf{v}$, we have $\Phi(\mathbf{v}) \leq \min \mathbf{v}$.

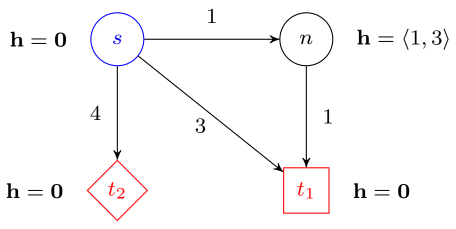
Our next result shows that with admissible heuristics, the aggregation function being admissible is a sufficient and necessary condition for proving that kA^*_Φ is admissible.

Theorem 3 (Admissibility is a necessary and sufficient condition) *Let* Φ *be a heuristic aggregation function. (1) If* Φ *is admissible then for any OMSPP instance and for any tuple of admissible heuristics* $\mathbf{h}$*,* kA^*_Φ *is admissible. (2) If* Φ *is not admissible, then there exists an OMSPP instance and a tuple of admissible heuristics such that* kA^*_Φ *is not admissible.*

Some heuristic aggregation functions, however, are not admissible, although they are consistent. For example, the max heuristic aggregation function is not admissible, and thus, due to Theorem 3, it cannot be safely used in kA* with admissible heuristics. For example, consider the OMSPP instance defined by the graph depicted in Figure 3, with start vertex s, and goals t_1 and t_2. The heuristic h_1 to goal t_1 is admissible, and so is h_2 for goal t_2. Yet, running $\mathrm{kA}^*_{\max}$ will find a suboptimal path to t_1. Indeed, the optimal path to t_1 is through n and costs 2, but $\mathrm{kA}^*_{\max}$ will expand t_1 before n, returning a path of length 3 to t_1.

Finally, Theorem 4 states that if the available heuristics are not admissible, then one cannot safely use them in kA*.

Theorem 4 *If there exists a heuristic* h_i *that is not admissible, and the heuristic aggregation function* Φ *is not the constant 0, then there exists a OMSPP instance and a tuple of arbitrary heuristics such that* kA^*_Φ *is not admissible.*


$\mathbf{h} = \mathbf{0}$
s
1
n
$\mathbf{h} = \langle 1, 3 \rangle$
4
3
1
$\mathbf{h} = \mathbf{0}$
t_2
t_1
$\mathbf{h} = \mathbf{0}$

Node	g	h_1	h_2	$\max \mathbf{h}$	$F_{\max}$
t_1	3	0	0	0	3
t_2	4	0	0	0	4
n	1	1	4	4	5

Fig. 3 Counter-example for $kA^*_{\max}$. Running $kA^*_{\max}$ returns a suboptimal path to t_1. The table shows the g, $\max \mathbf{h}$, and $F_{\max}$ values for n, t_1, and t_2 after expanding s.

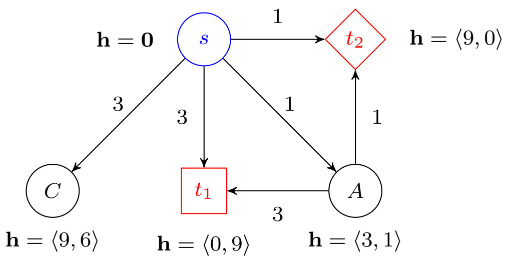
In such cases, one can run a variant of Dijkstra's algorithm [9] that halts when all goal nodes have been expanded. This algorithm, which we call k-Dijkstra, can be viewed as kA* with a heuristic aggregation function that always returns zero.

Unless stated otherwise, we assume for the rest of this paper that we use a combination of heuristics and heuristic aggregation function such that kA^*_{Φ} is admissible. In other words, either, the heuristics are consistent and Φ is consistent, or the heuristics are admissible and Φ is admissible. Hereinafter we omit the subscript $_{\Phi}$ from kA^*_{Φ} and F_{Φ} when Φ is clear from the context or is not relevant for the discussion.

5 Maintaining Open in kA*

According to Theorems 1 and 3, when a goal t_i is expanded by kA* we are guaranteed that the optimal path to it has been found. Therefore, we would like to guide the search towards finding optimal paths to the other goals. To do so, kA* removes t_i from the set of active goals (see line 9 in Algorithm 3). Removing a goal from the set of active goals affects the how the F value will be computed for nodes generated in subsequent kA* iterations. But what about the nodes already in OPEN? they were inserted into OPEN with F values that were computed w.r.t. a set of active goals that is different from the current set of active goals. Thus, some nodes may have two different F values – the F values computed w.r.t. to the set of active goals before and after t_i was expanded.

For example, consider the OMSPP instance depicted in Figure 4 and assume we use the min heuristic aggregation function. The shortest paths to t_1 and t_2 do not go through any other intermediate node and cost 3 and 1 respectively. Initially, the set of active goals contain t_1 and t_2 and the first node expanded is the initial state s. After s is expanded, the nodes in OPEN are t_2, A, t_1,



$\mathbf{h} = \mathbf{0}$
s
1
t_2
$\mathbf{h} = \langle 9, 0\rangle$
3
3
1
1
C
t_1
A
3
$\mathbf{h} = \langle 9, 6\rangle$
$\mathbf{h} = \langle 0, 9\rangle$
$\mathbf{h} = \langle 3, 1\rangle$

		Stale		Up-to-date	
Path	g	$\Phi_{t_1,t_2}(\mathbf{h})$	$F_{\Phi_{t_1,t_2}}$	$\Phi_{t_1}(\mathbf{h})$	$F_{\Phi_{t_1}}$
St_2	1	0	1		
SA	1	1	2	3	4
St_1	3	0	3	0	3
SAt_1	4	0	4	0	4
SAt_2	2	0	2		
SC	3	6	9	9	12

Fig. 4 Example illustrating the pitfalls of not performing an eager update step after a goal is expanded (i.e., omitting line 11 from Algorithm 3). Running kA* without this step in this example results in expanding a surplus state (A). Running Lazy kA* recomputes the value of A after expanding t_2, but avoids expanding A; Lazy kA* does not recompute the value of C.

and C, with F values 1, 2, 3, and 9, respectively. Thus, t_2 is expanded next, and is subsequently removed from the set of active goals. At this stage, Open contains nodes A, t_1, and C. Node A was inserted to Open with an F value of 2. However, if we compute $F(A)$ w.r.t. the current set of active goals, which contains only t_2, then A will have an F value of 4. To distinguish between these possible F values, we refer to the F value as computed by the current set of active goals as the *up-to-date* F value. We say that a node has a *stale* F value if it is stored in Open with an F value that is different from the up-to-date F value. In the example above, the F value of node A becomes stale after expanding t_2.

Considering stale F values when expanding nodes can introduce inefficiencies to the search. In the example above, considering stale F values will result in expanding A after expanding t_2, as its stale F value is 2 while $F(t_1) = 3$. Note that A is a surplus nodes (Definition 11) with respect to both t_1 and t_2. By contrast, considering the up-to-date F value of A will result in expanding t_1 after expanding t_2, halting the search without expanding A at all. The kA* pseudo code listed in Algorithm 3 describes an eager approach to avoid considering stale F value: re-compute the F value of all nodes in Open whenever a goal is removed from the set of active goal (line 11 in Algorithm 3). We refer to this implementation as Eager kA*.

Eager kA* is very easy to implement. However, its updating the F values of all the nodes Open every time a goal is expanded can be very time consuming, as it requires reordering Open $k - 1$ times.[3] In this section, we explore an

[3]We analyze the runtime overhead induced by these reorderings in Section 6.

alternative implementation of kA* in which F values are updated in a lazy manner.

5.1 Lazy kA*

Eager kA* computes the F value of all the nodes that have a stale F value. This can be wasteful, as some of these nodes will never be expanded by the search. To address this potential inefficiency, we propose an alternative kA* implementation in which the F values of states are re-computed in a lazy manner. We call this algorithm Lazy kA*, since it is directly inspired by the Lazy A* algorithm [3, 44], where multiple heuristics are used towards the same goal, and are evaluated lazily in a similar manner.

In more detail, Lazy kA* works as follows. When a goal is expanded, it is removed from the set of active goals, but all nodes in Open retain their current F value. Then, when a node is selected for expansion (line 4 in Algorithm 3), Lazy kA* checks if that node's F value is up-to-date or stale. If its F value is up-to-date, then that node is expanded. Otherwise, we re-compute its F value. If its up-to-date F value is smaller than the F value of all other nodes in Open, we expand it. Otherwise, the node is re-inserted to Open with its new F value.

Example. Consider running Lazy kA* on our running example from Figure 4. Lazy kA* behaves exactly as Eager kA* until the first goal is expanded, which in this case is t_2. At this stage, Open contains t_1, A, and C with F values 3, 2, and 9, respectively. The F values of A and C are stale, and therefore Eager kA* will re-compute their F values and re-insert them into Open. Subsequently, t_1 will be expanded and the search will halt. By contrast, Lazy kA* continues the search with the stale F values of A and C, and select A from Open after expanding t_2. Lazy kA* observes that $F(A)$ is stale and re-computes it. The updated F value of A is 4, which is greater than the $F(t_1) = 3$. So, A is re-inserted into Open and the next state selected from Open is t_1. After expanding t_1 the search halts, having an optimal path to both goals. As we can see, Lazy kA* did not re-compute the F value of C although it was stale. Nonetheless, Lazy kA* did not expand more node than Eager kA*.

5.1.1 Tracking Responsible Goals

Checking if a node has a stale value involves computing its F value, which may be costly. However, in some cases it is possible to identify that an F value is up-to-date even without computing it. To this end, we define the notion of *responsible goals.*

Definition 9 A set of goals $\mathcal{R}$ is responsible for a node n w.r.t a set of active goals if removing all the goals in $\mathcal{R}$ will make its F value stale, but removing any proper subset of $\mathcal{R}$ from the set of active goals will not.

In the example in Figure 4, the set of responsible goals for A w.r.t $\{t_1, t_2\}$ is $\{t_2\}$. By definition, an F value of a node n can only become stale if all its responsible goals are removed from the set of active goals.

We exploit this understanding to improve our implementation of Lazy kA* as follows. For every node in Open we arbitrarily choose one of the goals in its set of responsible goals. Then, when a node is chosen for expansion by Lazy kA*, we only compute its F value to check if it is stale if the chosen responsible goal for n has been removed from the set of active goals. Otherwise, we can safely infer its F value is up-to-date.

5.2 Lazy kA* for General Aggregation Functions

In the above example, Lazy kA* always expanded the node with the smallest up-to-date F value. We say that a kA* implementation that preserves this property is follows a best-first order w.r.t the up-to-date F values. This is a desirable property for a kA* implementation, since it means the search considers only the active goals when it chooses which node to expand next. Next, we explore under which restrictions over the heuristic aggregation function can we guarantee that Lazy kA* follows such a best-first order.

We slightly abuse the previous notation by defining Φ as an aggregation function that can accept vectors of different sizes, specifically any size between 1 and k. Common aggregation functions such as max, min, and average are all such functions, as we can apply them to vectors of different sizes.

Definition 10 We say that a heuristic aggregation function Φ is *isotone* if for any two sets $\mathcal{A}' \subseteq \mathcal{A}$ and for any vector $\mathbf{v}$, it holds that $\Phi(\mathcal{A}', \mathbf{v}) \leq \Phi(\mathcal{A}, \mathbf{v})$. Conversely, Φ is called *antitone* if $\Phi(\mathcal{A}', \mathbf{v}) \geq \Phi(\mathcal{A}, \mathbf{v})$ for any sets $\mathcal{A}' \subseteq \mathcal{A}$ and vector $\mathbf{v}$.

Informally, a heuristic aggregation function Φ is isotonoe if removing an active goal never increases its value, and it is called antitone if removing an active goal never decreases it. The min and max operators are trivial examples of antitone and isotone aggregation functions, respectively.

Theorem 5 *If a heuristic aggregation function Φ is antitone then Lazy* kA$^*_\Phi$ *follows a best-first order w.r.t the up-to-date F values.*

Proof When Lazy kA* chooses to expand a node n then its F value is up-to-date and it is smaller than or equal to all the (stale or up-to-date) F values in Open. Since the heuristic aggregation function is antitone, the F value of a node can only increase when a goal is removed from the set of active goals. Therefore, $F(n)$ must also be smaller than or equal to the up-to-date F value of all nodes in Open. Formally, let Φ be an antitone heuristic aggregation function, n be a node chosen for expansion by Lazy kA*, and n' be some other node in Open. Now, assume that the F value of n' was computed w.r.t. a set of active goals $\mathcal{A}$ and that the current set of active goals is $\mathcal{A}' \subset \mathcal{A}$. When using Lazy kA*, at a given point in time a node may have a stale F value (according to which it is positioned in OPEN), and an up-to-date F value. To differentiate between them, we denote the former by F and the latter by F^u. Since n is chosen for expansion, we have that (1) its F value is up-to-date, i.e., $F(n) = F^u(n) = g(n) + \Phi(\mathcal{A}', \mathbf{h}(n))$, and (2) $F(n)$ is the smallest F value in OPEN, i.e., $F(n) \leq F(n') = g(n') + \Phi(\mathcal{A}, \mathbf{h}(n'))$. Since Φ is antitone, $F(n') \leq F^u(n')$, as required. □

The implication of Theorem 5 is that for $\Phi = \min$, both Lazy kA* and Eager kA* follow a best-first order w.r.t. the up-to-date F values, except that Lazy kA* does so more efficiently, without the need to re-compute the F value of some stale nodes in OPEN. However, the exact sets of nodes they expand may be different due to tie breaking. For example, consider a tie-breaking rule that chooses the node that has the smallest sum of f values (among the nodes with the same F value). Now, assume that we have two nodes in OPEN, n_1 and n_2, with $\mathbf{f}$ values $\langle 5, 6, 10\rangle$ and $\langle 5, 7, 7\rangle$, respectively. They have the same F value (5), but according to this tie-breaking rule kA* will choose n_2 before n_1. Now, assume that t_3 was expanded. Using Eager kA* would result in n_1 now being expanded before n_2, while Lazy kA* would not do so. Since tie-breaking rules can make a significant difference in performance [2], this means that Eager kA* may still outperform Lazy kA* or vice versa. In our experimental results, however, this did not occur and Lazy kA* was in general better.

Observation 6 *If a heuristic aggregation function Φ is isotone then Lazy* kA^*_Φ *never chooses to re-insert a node back into* OPEN.

Proof Since the heuristic aggregation function is isotone, the F value of a node can only decrease when a goal is removed from the set of active goals. Therefore, if n has the smallest F value in OPEN, then its up-to-date value $F^u(n)$ will also be smaller than or equal to the F value of all nodes in OPEN. □

As an example of an isotone heuristic aggregation function, consider max. Consider using max for kA* over the example in Figure 4. As stated in Observation 6, Lazy $\text{kA}^*_{\max}$ will not re-insert any node back into OPEN. In this example, this may result in expanding node C before t_1 and consequently having Lazy $\text{kA}^*_{\max}$ expand in non-best-first order.

6 Resource Analysis

In this section, we compare analytically the behavior of kA* and k×A* in terms of the set of nodes they expand and in terms of the computational resources — memory requirements and CPU runtime — they require.

6.1 Expanded Nodes

Under certain conditions A* expands only the necessary set of nodes needed to find an optimal path to that goal [6]. This raises the question of either k×A* or kA* have this property as well. To answer this question, we use and then extend the notions of *surely expanded* nodes and *surplus* nodes [6, 21].

Definition 11 (Surely expanded and surplus nodes) Let $\langle G, s, t\rangle$ be an SPP instance and let h be an admissible heuristic. A node n is *surely expanded* w.r.t h, if there exists a path from s to n where all nodes n' on that path (except s and including n) have $d(s,n') + h(n') < d(s,t)$. A node n is *surplus* if in every path from s to n there exists a node $n' \neq s$ such that $d(s,n') > d(s,t)$.

Dechter and Pearl [6] and others have noted that if h is consistent (Definition 3), then a node n is *surely expanded* w.r.t h, iff $d(s,n) + h(n) < d(s,t)$, and a node is *surplus* w.r.t. h iff $d(s,n) + h(n) > d(s,t)$.

Without any additional knowledge of the graph G, a forward search algorithm needs to expand every surely expanded node in order to optimally solve a given SPP, and a forward search algorithm can optimally solve a given SPP without expanding any surplus node [6, 21].[4] A* does exactly that: expands all surely expanded nodes but never expands any surplus node [6].

The notion of surplus and surely expanded nodes can be extended to k×A*.

Definition 12 (Surplus and surely expanded for k×A*) A node is *surely expanded by k×A** with respect to an OMSPP instance if it is surely expanded for **at least one** of the SPPs $\Pi_1, \ldots, \Pi_k$. A node is *surplus for k×A** with respect to an OMSPP instance if it is a surplus node for **all** SPPs $\Pi_1, \ldots, \Pi_k$.

The following observation is straightforward.

Proposition 2 *For any OMSPP instance, k×A* expands all nodes that are surely expanded by k×A* and never expands a node that is surplus for k×A*.*

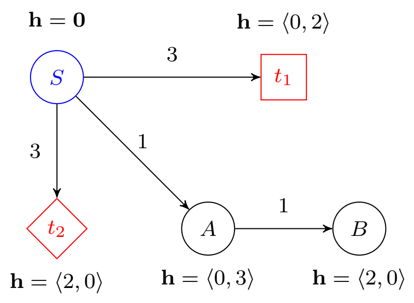
The relation between the set of nodes expanded by kA* and k×A* depend on the properties of the heuristic and heuristic aggregation functions being used.

Theorem 7 *Let Φ be a heuristic aggregation function. (1) If $\Phi = \min$, for every OMSPP instance and tuple of consistent heuristics, kA^*_Φ never expands any node that is a surplus node for k×A*. (2) If Φ is admissible but it is not $\min$, then there exists an OMSPP instance, a tuple of consistent heuristics, and a node A that is expanded by kA^*_Φ and is a surplus node for k×A*. (3) If $\Phi = \max$, then there exists an OMSPP instance, a tuple of consistent heuristics, and a node A that is a expanded by kA^*_Φ and is a surplus node for k×A*.*

Note that our assumption that the heuristics in **h** are consistent is necessary for the proof of Theorem 7. With an admissible but inconsistent heuristic, $\text{kA}^*_{\min}$ may, in fact, expand nodes that are surplus for k×A*. Figure 5 shows a OMSPP instance with $k = 2$ where this occurs. Running $\text{kA}^*_{\min}$ on this OMSPP instance will expand all nodes in the figure, while B is surplus for w.r.t both goals and is therefore surplus for k×A*.

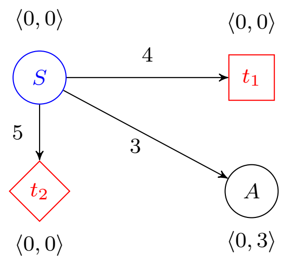
From the above, it may seem that $\text{kA}^*_{\min}$ is the preferred heuristic aggregation function. This is not necessarily so, as demonstrated by the following.

[4]A forward search here means any search that progresses by expanding nodes and starts from s, as oppose to bi-directional search.


$\mathbf{h} = \mathbf{0}$
$\mathbf{h} = \langle 0, 2\rangle$
S
3
t_1
3
1
t_2
A
1
B
$\mathbf{h} = \langle 2, 0\rangle$
$\mathbf{h} = \langle 0, 3\rangle$
$\mathbf{h} = \langle 2, 0\rangle$

Path		$A^*_{t_1}$		$A^*_{t_2}$		$kA^*_{\min}$		
	g	f	S	f	S	$\Phi(\mathbf{h})$	F	S
S	0	0	✗	0	✓	0	0	✗
St_1	3	3	✗	5	✓	0	3	✗
St_2	3	5	✓	3	✗	0	3	✗
SA	1	1	✗	4	✓	0	1	✗
SAB	2	4	✓	2	✓	0	2	✓

Fig. 5 An example where $kA^*_{\min}$ expands a surplus state, B. The table on the right indicates which nodes are surplus (S) for $A^*_{t_1}$, $A^*_{t_2}$, and $kA^*_{\min}$. This example relies on the inconsistency of the second heuristic: $h_{t_2}(A) > d(A, B) + h_{t_2}(B)$.



Path		$A^*_{t_1}$		$A^*_{t_2}$		$kA^*_{\max}$		S
	g	f	S	f	S	max($\mathbf{h}$)	F	
S	0	0	✓	0	✓	0	0	✓
St_1	4	4	✓	4	✓	0	4	✓
St_2	5	5	✗	5	✓	0	5	✓
SA	3	3	✓	6	✗	3	6	✓

Fig. 6 An example where $kA^*_{\max}$ with consistent heuristics does not expand a node (A) that is surely expanded by k×A*. The vector of heuristic values is displayed next to each node. The table on the right indicates which nodes are surely expanded (S) by $A^*_{t_1}$, $A^*_{t_2}$, and $kA^*_{\max}$.

Theorem 8 *Let Φ be a heuristic aggregation function. (1) If Φ is admissible, then for any OMSPP instance and for any tuple of admissible heuristics* $\mathbf{h}$, kA^*_{Φ} *expands all the nodes that are surely expanded by $k{\times}A^*$. (2) If Φ is not admissible, then there exists an OMSPP instance and a tuple of consistent heuristics such that* kA^*_{Φ} *does not expand some nodes that are surely expanded by $k{\times}A^*$.*

To illustrate the second part of Theorem 8, we look at an example with consistent heuristics and the max aggregation function which is not admissible, and show that kA* does not expand a node that is surely expanded by k×A*. Consider the OMSPP instance in Figure 6. Both heuristics are consistent and every node is surely expanded by k×A*. Consider node A, which is surely expanded by k×A* due to t_1. $kA^*_{\max}$ will not expand A because A's F value is larger than the F value of both goals.

By cloning node A in Figure 6 one can easily let kA^*_{Φ} skip an arbitrarily large number of nodes that are surely expanded by k×A*. One implication of Theorem 8 is that in some situations, $kA^*_{\max}$ can be better than $kA^*_{\min}$, since min is admissible, max is not admissible, but $kA^*_{\max}$ is admissible when used with consistent heuristics. $kA^*_{\max}$ may expand fewer nodes than $kA^*_{\min}$ even in domains restricted to a bounded number of successors per node and unit cost, as shown in Fig 7. In this example, kA^*_{max} will only expand the nodes on the right until it guarantees an optimal solution has been found, while kA^*_{min} is required to expand the entire subtree to the right. Thus, in this OMSPP instance kA^*_{max} may expand exponentially fewer nodes than kA^*_{min}.

While the results above demonstrate that there are cases where kA^*_{max} will expand fewer nodes than kA^*_{min}, in all our experiment this was not the case. To provide an intuitive explanation for why this occurs, consider using kA^*_{max} to solve an OMSPP in a grid pathfinding domain. As nodes closer to one goal are being expanded, the heuristic for that goal will decrease. However, the heuristic to the other goals may actually increase. As a result, kA^*_{max} ends up behaving quite similarly to a breadth-first search. This is illustrated in Figure 8, which shows the nodes expanded (in brown) and the nodes in the open list (in yellow) when using kA^*_{min} (left) and kA^*_{max} (right). As can be seen, kA^*_{min} is more "goal-driven", while the behavior of kA^*_{max} is similar to a breadth-first search.

6.2 Memory Requirements

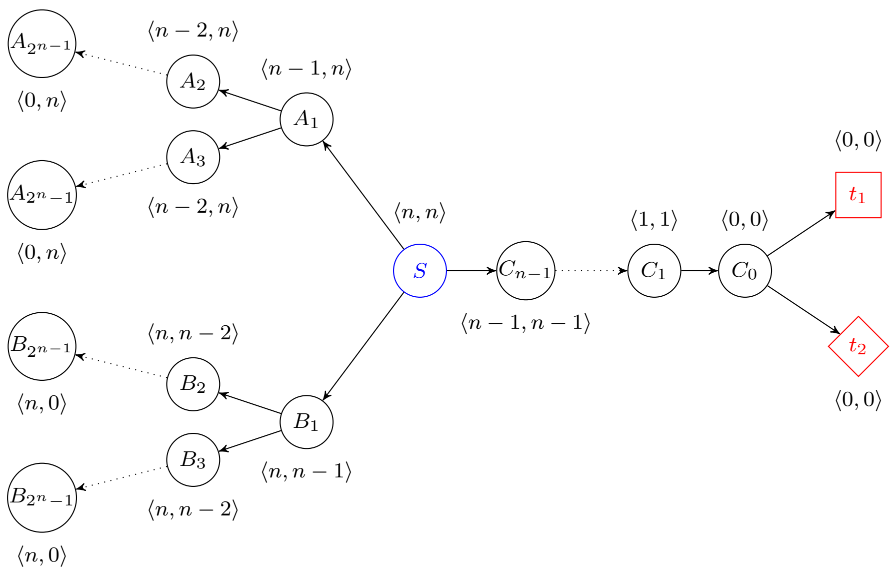
The common implementation of A* includes storing in memory all generated nodes for the entire duration of the search.[5] Hence, the memory required for the algorithms discussed in this paper is proportional to the memory required to store a single node times the number of distinct nodes generated. In our analysis we make the simplifying assumption that each node requires one memory unit to store for all algorithms. This assumption is not exactly correct in practice, since in kA* we store for a node a vector of k values ($\mathbf{f}(n) = \langle f_1, \ldots, f_k \rangle$) while in A* we only store a single value. Nonetheless, this assumption is reasonable for cases where the memory required to store the node details is significantly larger than these k values. The number of distinct nodes generated is strongly related with the number of distinct nodes expanded, for which we can establish the following corollary.[6]

Corollary 2 *For any OMSPP and for any tuple of consistent heuristics* $\mathbf{h}$*,* $k{\times}A^*$ *and* kA^*_{min} *expands the same set of nodes, up to tie breaking.*

Corollary 2 follows directly from Theorems 8, Theorem 7, and Proposition 2. Let Mem(X) and Gen(X) denotes the memory required and the set of nodes generated when running algorithm X, respectively. Then, up to tie-breaking

[5]We note that other implementations of A* are also possible, e.g., storing only the nodes in Open and their predecessors [47, 32], or storing some nodes in external memory [46, 13, 12].

[6]If b is the branching factor then there are at most b times more generated nodes than expanded nodes.



Node		$A^*_{t_1}$		$A^*_{t_2}$		$kA^*_{\min}$		$kA^*_{\max}$		
	g	f	S	f	S	$\min(\mathbf{h})$	F	$\max(\mathbf{h})$	F	S
S	0	n	✓	n	✓	n	n	n	n	✓
C_{n-1}	1	n	✓	n	✓	$n-1$	n	$n-1$	n	✓
⋮										
C_1	n	n	✓	n	✓	1	n	1	n	✓
C_0	n	n	✓	n	✓	1	n	1	n	✓
t_1	$n+1$	$n+1$	✓	$n+1$	✗	0	$n+1$	0	$n+1$	✓
t_2	$n+1$	$n+1$	✗	$n+1$	✓	0	$n+1$	0	$n+1$	✓
A_1	1	n	✓	$n+1$	✗	$n-1$	n	n	$n+1$	✓
A_2	2	n	✓	$n+2$	✗	$n-2$	n	n	$n+2$	✓
A_3	2	n	✓	$n+2$	✗	$n-2$	n	n	$n+2$	✓
⋮										
$A_{2^{n-1}}$	n	n	✓	$2n$	✗	0	n	n	$2n$	✓
B_1	1	$n+1$	✗	n	✓	$n-1$	n	n	$n+1$	✓
B_2	2	$n+2$	✗	n	✓	$n-2$	n	n	$n+2$	✓
B_3	2	$n+2$	✗	n	✓	$n-2$	n	n	$n+2$	✓
⋮										
B_{2^n-1}	n	$2n$	✗	n	✓	0	n	n	$2n$	✓

Fig. 7 Generic instance with consistent heuristics where $kA^*_{\max}$ does not expand an exponential number of nodes that are surely expanded by k×A*. The table indicates which nodes are surely expanded (S) for $A^*_{t_1}$, $A^*_{t_2}$, and kA^*_{Φ}.

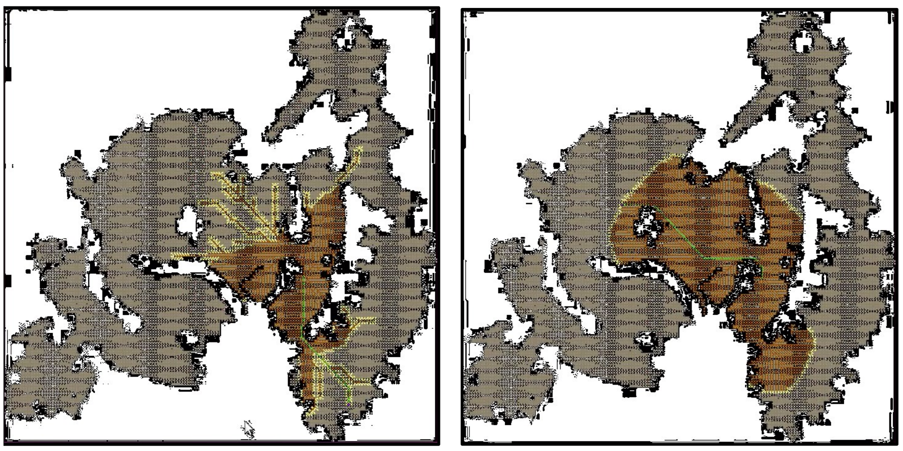


Fig. 8 A comparison of the set of expanded nodes for kA^*_{min} and for kA^*_{max} over a grid pathfinding probelm.

between nodes with the same F value, it holds that:

$$\text{Mem}(\text{k}\times\text{A}^*) = \max_{j\in[1,k]} |\text{Gen}(\text{A}^*_j)| \tag{3}$$

$$\text{Mem}(\text{kA}^*_{\text{min}}) = |\bigcup_{j\in[1,k]} \text{Gen}(\text{A}^*_j)| \tag{4}$$

$$\text{Mem}(\text{k}\times\text{A}^*) \le \text{Mem}(\text{kA}^*_{\text{min}}) \le \sum_{j\in[1,k]} \text{Mem}(\text{A}^*_j) \tag{5}$$

The correctness of Equations (3)–(5) is now established. Since $\text{k}\times\text{A}^*$ runs the k searches independently, there is no need to store the states generated by A^*_i when running A^*_j. Thus, to run $\text{k}\times\text{A}^*$ we require memory sufficient to run each A^*_i individually (Equation (3)). In kA^*_{min} we expand the same set of states as $\text{k}\times\text{A}^*$ and thus generate the same set of states, but we must store them throughout the search. Thus, kA^* stores every state n that is generated by one of the k searches in $\text{k}\times\text{A}^*$, i.e., the union $\bigcup_{j\in[1,k]} \text{Gen}(\text{A}^*_j)$ (Equation (4)). The size of this union cannot be smaller than the cardinality of the set of stated generated by any individual A^*, but cannot be larger than their sum (Equations (5)). Note that these bounds around $\text{Mem}(\text{kA}^*_{\text{min}})$ are tight, in the sense that there are k-goal problems where $\text{Mem}(\text{k}\times\text{A}^*) = \text{Mem}(\text{kA}^*_{\text{min}})$ and other k-goal problems where $\text{Mem}(\text{kA}^*_{\text{min}}) = \sum_{j\in[1,k]} \text{Mem}(\text{A}^*_j)$. For example, in a 2-goal instance, if $\text{Gen}(\text{A}^*_1) \subset \text{Gen}(\text{A}^*_2)$ then $\text{Mem}(\text{k}\times\text{A}^*) = \text{Mem}(\text{kA}^*_{\text{min}})$ while if $\text{Gen}(\text{A}^*_1) \cap \text{Gen}(\text{A}^*_2) = \{s\}$ then $\text{Mem}(\text{kA}^*_{\text{min}}) = \sum_{j\in[1,k]} \text{Mem}(\text{A}^*_j) - 1$, where the minus one is for the start state.

It important to note that the notion of "up to tie breaking" in the discussion above is problematic in our context, since ties in kA^*_{min} refer to nodes with the same F value while ties in $\text{k}\times\text{A}^*$ refer to nodes with the same f_i value for some particular goal t_i. Nonetheless, for ease of presentation, we make the simplifying assumption hereinafter that k×A* and kA$^*_{\text{min}}$ expand the same set of nodes.

6.3 Runtime Analysis

Now we analyze the runtime of k×A* and several implementations of kA*. Observe that while k×A* and kA* generate the same set of nodes, (Corrolary 2), their runtimes differ due to the number of times each node is generated and the cost of these generations. For the following runtime discussion, we assume that the heuristics used by kA* are all consistent, and that the heuristic aggregation function is min.

6.3.1 Runtime of k×A*

To analyze the runtime of k×A*, consider the operations performed by an A* search whenever a node is generated (lines 7–15 in Algorithm 1):

1. **Node Generation** (lines 7–12 and 15). This refers to computing the generated state by applying an action to the expanded node, performing duplicate detection (updating the g value to the generated if needed), and inserting the generated node to Open (if needed). We denote the computational cost of this step as C_{gen}.
2. **Heuristic computation** (lines 14 and 15). This refers to computing the h value of the generated state. We denote the computational cost of this step as C_h.

The exact values of C_{gen} and C_h depends on the domain and various implementation details, such as the data structure used to implement Open and the state representation. In the textbook implementation of A* on standard search benchmarks, computing the node created by applying an action is easy, previously generated nodes are stored in a hashtable, and Open is implemented as a Binary heap.[7] Thus, C_{gen} is linear in the size of the node representation and logarithmic in the size of Open. The cost of the heuristic computation (C_h) can vary widely as well, where some heuristics require constant time to compute while more sophisticated heuristics may involve poly-time computations or even worse.

We perform our runtime analysis with respect to the above cost model, and under the simplifying assumption that C_{gen} and C_h are approximately constant for all nodes. Under this assumption, the runtime of k×A* is straightforward.

$$\text{Time}(\text{k}\times\text{A}^*) = \sum_{i\in[1,k]} |\text{Gen}(\text{A}^*_i)| \cdot (C_{gen} + C_h) \tag{6}$$

where Time(X) denote the runtime of algorithm X.

[7] But faster implementations are also possible in some domains, where adding to Open incurs a constant time [17, 4].

6.3.2 Runtime of kA*

To analyze the runtime of kA*, a deeper analysis is needed. In addition to node generation (C_{gen}) and heuristic computation (C_h), in kA* we also have the cost of reordering Open (either Eagerly or Lazily) after a goal is removed from the set of active goals. Let C_r denote the cost of re-ordering a single node. Next, we consider the number of times each of the costs components — node generation (C_{gen}), heuristic computation (C_h), and node reordering (C_r) — is incurred.

Node generation (C_{gen}). Since we use Φ = min and a set of consistent heuristics, every generated node incurs C_{gen} exactly once. Thus, node generation contributes to the overall runtime of kA$^*_{\text{min}}$:

$$|\text{Gen}(\text{kA}^*)| \cdot C_{gen} = |\bigcup_{i\in\{1,k\}} \text{Gen}(\text{A}^*_i)| \cdot C_{gen}$$

Heuristic computation (C_h). Let Gen$_i$(kA*) denote the set of nodes generated by kA* until goal g_i is expanded. Until the first goal is expanded, kA* computes k heuristics. Then, until the second is expanded, kA* computes $k-1$ heuristics, and so on. So, $|\text{Gen}_1(\text{kA}^*)|$ nodes are generated with k heuristics, $|\text{Gen}_2(\text{kA}^*)| - |\text{Gen}_1(\text{kA}^*)|$ nodes are generated with $k-1$ heuristics, $|\text{Gen}_3(\text{kA}^*)| - |\text{Gen}_2(\text{kA}^*)|$ nodes are generated with $k-2$ heuristics, and so on. Summing all this, we have that heuristic computations add to the overall runtime of kA$^*_{\text{min}}$:

$$\begin{aligned}
&C_h \cdot (k\cdot(|\text{Gen}_1(\text{kA}^*)|\\
&+(k-1)\cdot(|\text{Gen}_2(\text{kA}^*)| - |\text{Gen}_1(\text{kA}^*)|)\\
&+(k-2)\cdot(|\text{Gen}_3(\text{kA}^*)| - |\text{Gen}_2(\text{kA}^*)|)\\
&\quad\cdots\\
&\quad+1\cdot(|\text{Gen}_k(\text{kA}^*)| - |\text{Gen}_{k-1}(\text{kA}^*)|)))\\
&\quad= C_h \cdot \sum_{i\in[1,k]} |\text{Gen}_i(\text{kA}^*)|
\end{aligned}$$

Reordering of Open (C_r). We perform here only a rough analysis of the overhead incurred by reordering Open in Eager kA$^*_{\text{min}}$. This serves as an upper bound for the runtime incurred for reordering nodes Open with Lazy kA* implementations.

Let Open$_i(X)$ denote the states in Open when algorithm X expanded goal t_i, and let C_r be the cost of re-computing the F value for a state and updating its position in Open accordingly. Let t_i be the i^{th} goal that has been found. After t_i was expanded, there are $|\text{Open}_i(\text{kA}^*)|$ states in Open. Thus, the total computational cost incurred by Eager kA* due to recomputing F

Cost	k×A*	$\mathrm{kA}^*_{\min}$
C_{gen}	$\sum_{i=1}^{k} \lvert \mathrm{Gen}(\mathrm{A}^*_i) \rvert$	$\lvert \bigcup_{i=1}^{k} \mathrm{Gen}(\mathrm{A}^*_i) \rvert$
C_h	$\sum_{i=1}^{k} \lvert \mathrm{Gen}(\mathrm{A}^*_i) \rvert$	$\sum_{i=1}^{k} \lvert \mathrm{Gen}_i(\mathrm{kA}^*) \rvert$
C_r	0	$\sum_{i=1}^{k-1} \lvert \textsc{Open}_i(\mathrm{kA}^*) \rvert \cdot C_r$

Table 2 Analysis of the computational costs incurred by k×A* and $\mathrm{kA}^*_{\min}$.

values and reordering Open accordingly after expanding all the goals is

$$C_{eager} = \sum_{i=1}^{k-1} \lvert \textsc{Open}_i(\mathrm{kA}^*) \rvert \cdot C_r \tag{7}$$

6.3.3 Practical Implications

Table 2 provides a summary of our runtime analysis, comparing the runtimes of k×A* and $\mathrm{kA}^*_{\min}$. Each row represents one of the computational cost factors (C_{gen}, C_h, and C_r), showing the contribution of that computational cost factor to the overall runtime for the compared algorithms. To show the usefulness of this analysis, consider the special cases where one of the costs (C_{gen}, C_h, or C_r) dominates the others:

– **Case 1: Node generation cost is dominant.** ($C_{gen} \gg C_h + C_r$) If this case, the preferred algorithm is kA*. To show this, compare the values in Table 2 for C_{gen}: $\sum_{i\in[1,k]} \lvert Gen(\mathrm{A}^*_i) \rvert$ versus $\lvert \bigcup_{i\in[1,k]} Gen(\mathrm{A}^*_i) \rvert$. Clearly, the former is larger than or equal to the latter. The advantage of $\mathrm{kA}^*_{\min}$ grows with the size of the intersection between the sets $Gen(\mathrm{A}^*_i)$ for $i \in [1,k]$, which roughly corresponds to having the goals close to each other.
– **Case 2: Heuristic computation is dominant.** ($C_h \gg C_{gen} + C_r$) If this is the case, the preferred algorithm is k×A*. Again, to show this we look at Table 2 and see that k×A* computes a heuristic function $\sum_{i\in[1,k]} \lvert \mathrm{Gen}(\mathrm{A}^*_i) \rvert$ times, while kA* computes the heuristic $\sum_{i\in[1,k]} \lvert \mathrm{Gen}_i(\mathrm{kA}^*) \rvert$ times. Importantly, for every i it holds that $\lvert \mathrm{Gen}_i(\mathrm{kA}^*) \rvert \geq \lvert \mathrm{Gen}(\mathrm{A}^*_i) \rvert$ because $\mathrm{Gen}_i(\mathrm{kA}^*)$ contains all the states in $\mathrm{Gen}(\mathrm{A}^*_i)$ and states associated with the search for the other goals (those states that A^*_i would not generate but some of other individual A* searches would) that happen to have been added to Open at this stage.
– **Case 3: Distant Goals.** ($\lvert \bigcap\limits_{i\in[1,k]} \mathrm{Gen}(\mathrm{A}^*_i) \rvert \approx 1$) If the goals are far away from each other in the state space, then we expect most nodes to be generated by only one of the k searches. In such a case, $\sum\limits_{i\in[1,k]} \lvert \mathrm{Gen}(\mathrm{A}^*_i) \rvert \approx \lvert \bigcup\limits_{i\in[1,k]} \mathrm{Gen}(\mathrm{A}^*_i) \rvert$ and therefore we expect kA* to be less effective.

k	k-Dijkstra	k×A*	kA*	
			Eager	Lazy
2	53,583	27,389	21,686	**21,682**
4	62,826	52,375	28,424	**28,413**
8	68,505	101,103	36,673	**36,650**
16	72,196	202,715	43,531	**43,489**
32	74,618	407,221	49,732	**49,662**
64	75,998	823,018	54,357	**54,239**
128	76,495	1,613,628	57,932	**57,739**

k	kD	k×A*	kA*	
			Eager	Lazy
2	4.44	3.66	3.07	**3.04**
4	5.09	7.07	4.09	**4.07**
8	5.62	13.55	**5.43**	5.44
16	**5.92**	27.28	7.31	6.98
32	**6.15**	54.45	10.56	9.01
64	**6.37**	111.58	19.45	12.54
128	**6.68**	216.28	51.83	20.49

Table 3 The average number of expanded nodes (left) and runtime in milliseconds (right) for solving OMSPP instances with different number of goals (k) on the large Grid domain. Highlighted in bold are the best results for every k.

7 Experimental Results

In this section, we compare experimentally several implementations of k×A* and kA*. We implemented several heuristic aggregation functions for kA*, including min, max and projection. Using the min heuristic aggregation yielded the best performance among these heuristic aggregation functions, and so unless stated otherwise kA* below means kA^*_{min}. As a baseline, we also compare with k-Dijkstra, which does not use any heuristic function (see Section 4.3).

We evaluate these algorithms on two domains: path finding on Grids and the Pancake puzzle [28]. These two domains represent different types of search problems. In Grid pathfinding, the searched graph is given explicitly as input, and the number of generated nodes grows polynomially with the depth of the search. In contrast, the searched graph for the Pancake puzzle is given implicitly by a start node and a set of operators, and the number of generated nodes grows exponentially with the depth of the search.

7.1 Grid Pathfinding

We experiment with 8-connected grids from the from the Dragon Age video game, available from the Moving AI repository [40]. Specifically, we used the `ost001d` map, which is a 194×194 grid with 10,557 open cells, and the `ost100d` map, which is a 1025×1025 grid with 137,375 open cells. We refer to the former as the *Grid domain* and the latter as the *large Grid domain*. As a heuristic, we used the Octile distance heuristic, which is an admissible and consistent heuristic for 8-connected grids.

7.1.1 The Impact of Varying the Number of Goals

In the first batch of experiments, we investigate the impact of increasing the number of goals on the large Grid domain. We experimented with 2, 4, 8, 16, 32, 64, and 128 goals (k). For each of value of k (number of goals) we randomly generated 100 problem instances where in every instance the start and the k goals were selected randomly. The average number of node expansions and the average runtime in milliseconds are given in the left and right parts of Table 3, respectively. Note that the number of node expansions is not the number of *unique* nodes that were expanded, so if a node is expanded several times, e.g., in the different A* runs in k×A*, this will count as several node expansions.

Consider first the number of node expansions performed by each algorithm. A clear trend is that all kA$^*_{\min}$ implementations expand significantly fewer nodes compared to k×A* and k-Dijkstra. For example, for 32 goals Lazy kA* performed an average of 49,662 node expansions while k×A* and k-Dijkstra expanded 407,221 and 74,618 nodes, respectively. These results are as we expected: k×A* expands more nodes than kA* since it may expand a node multiple times when searching for multiple goals, and k-Dijkstra expands more nodes than kA* because it does not use any heuristic.

Next, consider the average runtime results (right part of Table 3). We observe several interesting trends. First, while Lazy kA* generated fewer nodes than the other algorithms, it is not always the fastest. Up to 8 goals, all kA$^*_{\min}$ variants are the fastest, and the differences beteween them is negligible. However, as the number of goals increases beyond 8, the fastest algorithm is k-Dijkstra. To explain this, consider the most extreme case, where every node is a goal. In this case, kA* will compute at least one heuristic for every node, and for most nodes much more. These heuristic computation times are not spent by k-Dijkstra, and if all nodes will be expanded eventually, then there is no gain from using a heuristic. In other words, the benefit of using a heuristic diminishes as the number of goals increase.

Now, consider the performance of the Eager and Lazy kA* implementations. The differences between Eager kA* and Lazy kA*, in terms of number of nodes expanded, are negligible, and result from having different nodes in Open and consequently ties in F values were broken differently. In terms of runtime, all kA* implementations perform similarly up to 16 goals. However, when solving k Shortest Path Problem (kGP) instances with more goals, the added overhead incurred by Eager kA* re-sorting Open after expanding each goal becomes significant, and grows for kGP instances with more goals. For example, the average runtime of Eager kA* for problem instances with 128 goals is 51.83 milliseconds while it is 20.49 milliseconds for Lazy kA*. Since Lazy kA* provides the best results in most cases, we use it in all subsequent experiments.

7.1.2 The Impact of the Distance between Goals

The results in Table 3 show that while k×A* is always much worse than kA*, the advantage in runtime of kA* over the baseline k-Dijkstra is not large, and appears only for kGP instances with a relatively small number of goals. The reason for this is that the goals in the previous experiment were randomly chosen nodes on the grid. Thus, they tended to be far from each other, and so the overlap of the set of nodes generated by the individual A* searches, was not large. This can be seen by comparing the node expansions of k-Dijkstra and kA* (Table 3, left): while kA* expanded fewer nodes than k-Dijkstra, the

k	R	k-Dijkstra	k×A*	kA*	
				Eager	Lazy
2	1	77,922	47,500	**23,889**	**23,889**
4	1,2	73,946	90,001	22,748	**22,746**
8	1,2	69,373	170,517	21,707	**21,703**
16	1,2	73,875	372,707	23,980	**23,972**
32	1,2,4	67,783	585,962	19,252	**19,237**
64	1,2,4	68,621	967,568	16,222	**16,189**
128	1,2,4,8	67,689	2,364,510	20,414	**20,354**

k	R	k-Dijkstra	k×A*	kA*	
				Eager	Lazy
2	1	55.06	48.68	26.23	**24.98**
4	1,2	51.52	91.42	24.59	**24.29**
8	1,2	48.22	176.71	24.37	**24.16**
16	1,2	52.09	382.70	30.24	**28.69**
32	1,2,4	46.83	606.21	30.09	**25.72**
64	1,2,4	49.71	1002.79	40.78	**28.43**
128	1,2,4,8	49.36	2451.57	91.79	**47.79**

Table 4 The average number of expanded nodes (left) and runtime in milliseconds (right) for solving OMSPP instances with different number of goals (k) on the large Grid domain, when goals are restricted to a given radius bound (column R). Highlighted in bold are the best results for every k.

Radius	k-Dijkstra	k×A*	kA*
1	69,676	365,371	**23,480**
2	71,409	322,065	**20,854**
3	67,341	280,238	**19,005**
4	71,946	306,263	**21,828**
5	70,536	277,826	**22,047**
6	75,533	306,207	**27,265**
7	91,169	378,212	**39,890**

Radius	k-Dijkstra	k×A*	kA*
1	47	379	**28**
2	48	334	**25**
3	42	285	**23**
4	46	315	**27**
5	45	284	**27**
6	48	313	**33**
7	59	386	**48**

Table 5 The average node expansions (left) and runtime in milliseconds (right) for solving OMSPP instances with 16 goals on the large Grid pathfinding domain, for different goal radii.

difference is not large. Following our theoretical analysis (Section 6), we expect that kA* will be more effective when there is a large overlap between the nodes generated by the individual A* searches.

To characterize when kA* will be more effective, we performed a second set of experiments in which we biased the goals to be close to each other. In details, after choosing the first goal's location randomly, we limited the remaining goals to be at most R steps from the first goal, where R is a parameter that we refer to as the *goal radius*. Table 4 shows the results after limiting the goals to a radius bound of 1, 2, 4, and 8. Table 4 follows the same format as Table 3, except that we added a column R to show the radius bounds used for every value of k (number of goals). The difference between these results and the results in which the goals were placed randomly (Table 3) is dramatic: now kA* is always better than all other approaches in both nodes expanded and runtime, even when the number of goals is 128. All other trends remain as before, where Lazy kA* is usually the best performing implementation.

Table 5 provides a deeper analysis of these results, showing the impact of the goal radius parameter on the search. Here we fixed the number of goals to 16. The left part of Table 5 shows the average number of node expansions and the right part shows the average runtime in milliseconds. Here the advantage of kA* over k×A* and k-Dijkstra is very clear, in terms of both runtime and expanded nodes. Also, we observe that while k×A* is completely unaffected by the change in goal radius, while kA* is more effective when the goal radius is smaller.

	2 goals			16 goals			128 goals		
Pivots	kA*	k×A*	k-Dijkstra	kA*	k×A*	k-Dijkstra	kA*	k×A*	k-Dijkstra
1	**1.19**	2.23	3.38	**1.64**	15.61	3.21	7.55	127.78	**4.43**
2	**0.74**	1.39	3.38	**1.15**	9.65	3.21	7.46	80.85	**4.43**
4	**0.48**	0.88	3.38	**0.88**	6.30	3.21	7.60	50.95	**4.43**
8	**0.35**	0.66	3.38	**0.08**	4.47	3.21	8.74	32.28	**4.43**
16	**0.25**	0.45	3.38	**0.77**	3.25	3.21	10.93	25.04	**4.43**

Table 6 Average runtime in milliseconds for the Grid domain experiments. Each row shows the results for a different number of pivots.

7.1.3 The Impact of Using Stronger Heuristics

Next, we evaluated the impact of using a stronger, more accurate heuristic, on the performance of the proposed OMSPP algorithms. To this end, we used differential heuristics (DH) [18, 36, 42] instead of Octile distance. DH is a sophisticated memory-based heuristic for Grid pathfinding that works as follows. A set of nodes (cells in the grid) are chosen referred to as the *pivots*. Then, in a pre-processing step, we compute and store in memory a shortest path between every node and these pivot nodes. When computing the heuristic for a goal that is not one of these pivot nodes, the triangle inequality to obtain an admissible heuristic. For more details, see [18, 36, 42]. Importantly, adding more pivots results in a more accurate heuristic, but one that takes longer time to compute. Thus, DH is especially useful for our analysis, as it can be tuned to be more accurate and more costly to compute.

The average runtime results of kA*, k×A* and k-Dijkstra are given in Table 6 for solving kGP instances with 2, 16, and 128 goals, using 1, 2, 4, 6, 8, and 16 pivots. The goal radius here was set to 2. Highlighted in bold are the best results for every configuration of goals and pivots. The results show several trends. First, kA* is significantly faster for 2 and 16 goals, while k-Dijkstra is better for 128 goals. This follows the trend observed earlier, in which when there are not too many goals, kA* is faster, while k-Dijkstra is better for kGP instances with many goals.

Now consider the impact of adding pivots. When the number of goals is 2, adding pivots only improves the performance of kA*. However, as the number of goals increase, the impact of adding pivots, where adding some pivots helps up to a certain point, after which adding pivots only results in slower runtime. For example, for 16 goals and 8 pivots, kA* required an average of 0.08 milliseconds, which is 40 times faster than both k-Dijkstra and k×A*. But, adding more pivots, results in a slower kA* runtime of 0.77 milliseconds. Moving to 128 goals, we observe that the turning point in terms of runtime is at 2 pivots, afterwhich adding pivots only yielded slower runtime. By contrast, adding pivots only improves the performance of k×A*.

This effect of adding pivots corresponds our theoretical analysis in Section 6. kA* performs fewer node generations compared to k×A*, but it pays by performing more heuristic computations. Adding more pivots means the

	2 goals		16 goals		128 goals	
n	kA*	k×A*	kA*	k×A*	kA*	k×A*
1	**161**	204	**749**	1,575	**4,298**	12,441
2	**181**	204	**944**	1,569	**4,395**	12,902
3	**166**	185	**1,115**	1,523	**6,149**	12,376
4	**188**	205	**1,251**	1,558	**7,813**	12,661
5	**224**	237	**1,395**	1,613	**8,855**	12,315

	2 goals		16 goals		128 goals	
n	kA*	k×A*	kA*	k×A*	kA*	k×A*
1	**1.6**	1.8	14.1	**14.0**	391.8	**111.3**
2	**1.8**	**1.8**	18.5	**14.1**	402.3	**115.9**
3	1.7	**1.6**	22.8	**14.0**	587.0	**112.3**
4	**1.9**	**1.9**	25.5	**14.1**	772.3	**114.8**
5	2.2	**2.1**	28.7	**14.6**	915.6	**111.6**

Table 7 The average node expansions (left) and runtime in milliseconds (right) for solving OMSPP instances in the Pancake domain with 2, 16, and 128 goals, and a goal radius of 1, 2, 4, 8, and 16.

heuristic computation is more costly (increasing C_h), which is incurred more times in kA* than in k×A* (see Table 2).

7.2 Pancake Puzzle

Next, we present the results for the Pancake puzzle. We chose to experiment on a 15 Pancake puzzle, using the GAP heuristic [23], adjusted so that it is suitable for different goal nodes. We experimented with 2, 16, and 128 goals (k). For every k we generated 100 random instances, where goal g_1 was the standard Pancake goal state (where the Pancakes are stacked by size) and the other goals are random permutations that are at most R steps from the first goal (R is the aforementioned radius bound parameter). Since the search space in this puzzle is exponential, k-Dijkstra could not solve even small-sized problems even for $k = 2$. Thus, we only compared k×A* and kA*.

Table 7 shows the average node expansions (left) and runtime in milliseconds (right). As expected, kA* performed fewer node expansions compared to k×A*. The difference, however, is very small in this domain. Moreover, the runtime of kA* is almost always larger than the runtime of k×A*, and the difference between them grows when the number of goals increase. This is because in exponential domains the intersection between the sets of nodes generated by each search is small compared to size of the last f layer in each search. Thus, in exponential domains we recommend k×A* as the algorithm of choice.

7.3 Estimating the Potential Gain of kA*

So far, we observed that in Grid pathfinding, kA* was a better choice than k×A*, while in the Pancake puzzle k×A* was usually better. To better understand these results, we analyze the number of nodes that are expanded by more than one A* search in k×A*.

Table 8 lists the average ratio between the number of nodes expanded by k×A* and the number of unique nodes expanded. If this number is one, it means kA* cannot provide benefit over k×A*, and the potential benefit of kA* grows with this ratio. The table shows this ratio for 2, 4, 8, 16, 32,

	Goals						
Domain	2	4	8	16	32	64	128
Pancakes	1.15	1.23	1.33	1.29	1.38	1.52	1.72
Grid	1.72	3.00	5.10	8.88	16.37	28.49	51.65
Large Grid	1.84	3.35	6.03	10.79	20.58	36.61	69.09

Table 8 The number of nodes that are expanded by multiple A* searching divided by the unique number of expanded nodes.

64, and 128 goals and for goal radii of 1–5 for Pancake and 1–7 for Grids. These results show that the potential benefit of kA* in the Pancake domain is significantly smaller than in the Grid domains. For example, the nodes ratio reported in Table 8 is never larger than 2 for the Pancake domain while for the Grid and Large grid domains it reaches 51.65 and 69.09, respectively. This explains the poor performance of kA* in the Pancake domain compared to the Grid domains.

7.4 Between kA* and k×A*

In kA*, the search for all k goals is done in a single search, while in k×A* these are k different searches. kA* and k×A* can be viewed as two extreme cases of a spectrum of OMSPP algorithms, where algorithms on this spectrum can search for the different goals separately, but share some amount of information between the searches.

In particular, we implemented two such "middle-ground" algorithms. In the first, we run k×A* but shared g values between the searches. That is, when the i^{th} A* search generates a node that was already generated by a previous A* search, then it uses the g value stored for that node if it is smaller than the g value otherwise set for it. We observed that this minor modification to k×A* provides some benefit over k×A*, in terms of runtime and number of nodes expanded. However, since the g values of all nodes expanded by all searches must be stored, the memory consumption is higher than in k×A*.

In the second "middle-ground" algorithm we implemented, we run the full kA* algorithm, but we used *projection* as the heuristic aggregation function. That is, first we use only the heuristic for the first goal. After this goal is expanded, we use the heuristic for the second goal, and so on. As noted in Corollary 1, such a heuristic aggregation function follows the consistent requirement, and thus is admissible if the underlying heuristics are consistent. This version of kA* can be seen as implementing k×A* but sharing OPEN and CLOSED between the different searches. The experimental results of using this projection version of kA* were similar to but usually slightly worse than that of kA* with the min heuristic aggregation function.

To conclude our experimental results, we observed the following trends:

1. In all domains, kA* expands fewer nodes than k×A* and k-Dijkstra.

2. In the Grid pathfinding domain, which is a polynomial domain, many nodes are generated by more than one search, and consequently kA^* is advantageous compared to $\mathrm{k}\times\mathrm{A}^*$ searches.
3. In the Grid pathfinding, when the number of goals becomes very large simply running k-Dijkstra is the best option, as it does not spend time on heuristic computations.
4. In the Pancake domain, which is an exponential domain, only relatively few nodes were generated by more than one search, and so $\mathrm{k}\times\mathrm{A}^*$ is usually faster than running kA^*.

8 Related work

OMSPP has been studied in the context of path finding in road networks. Specifically, one-to-many queries have been mentioned in the context of the PHAST algorithm [7]. PHAST performs a pre-processing of the network, which enables all future one-to-one queries to be fast. Solving one-to-many queries in this way is somewhat similar to using $\mathrm{k}\times\mathrm{A}^*$ that uses a heuristic that requires pre-processing, e.g., PDB. Restricted PHAST (RPHAST) [8] is an extension of PHAST that supports fast one-to-many queries. Both PHAST and RPHAST rely on a pre-processing of the graph that is effective in some networks but less effective for others, and they do not use a heuristic. We explore how one can use heuristics to answer one-to-many queries.

OMSPP is a special case of the k nearest neighbors (kNN) problem.[8] kNN is the problem of finding paths to the k nearest points of interest (POI) from a given initial location. When the total number of POI in the graph is exactly k, then kNN becomes exactly OMSPP. kNN has been studied extensively in many contexts, but recent work has applied heuristic search to solve kNN problems in a Euclidean place with obstacles [45]. They created a graph over this plane and applied heuristic search over it to solve the kNN problem. Many of their techniques are similar to those described in our work. Namely, $\mathrm{k}\times\mathrm{A}^*$ is called there Brute-force Polyanya,[9] the OkNN Polyanya algorithm they propose is a variant of kA^*, both of the heuristics they proposed implement a minimum heuristic aggregation function, and Lazy kA^* is called there *target reassign*. However, our theoretical contribution goes beyond their work, e.g., providing theory for the range of heuristic aggregation functions that can be used, when lazy kA^* can be safely applicable, and analysis of the runtime and memory requirements. In addition, our description is more general, in the sense that it is not related to the specific Polyana state space and the OkNN problem.

OMSPP is similar also to many previously studied problems. OMSPP is also different from a disjunction of k goals, i.e., from the case where there are k possible goals and the task is to find a lowest-cost path to the closest one of them. Unlike this case of a disjunction of k goals, in OMSPP we must find a shortest path to each of the goals, and not just to the closest one.

[8]Not to be confused with the classical kNN supervised learning algorithm.

[9]In other prior work on kNN, it is also referred to as revisited IER [1].

Other related problems are the problem of finding k disjoint paths between a two vertices [43] and the problem of finding the best k paths between two vertices [37, 14]. These are different problems from OMSPP since in OMSPP we have k different goals.

Somewhat related is the work on *multi-objective search*, where we have multiple objectives function that we wish to optimize [33]. For example, in a navigation application one may want to optimize for path length and also for ease of navigation. An optimal solution to a multi-objective optimization problem is usually a solution that is Pareto optimal, and it is often the case that one would like the set of all Pareto-optimal solutions. This is fundamentally different from OMSPP, where we have a single optimization criteria — we must have a lowest-cost path to each of the k goals.

Flerova et al. [16] studied the problem of finding m best solutions to a combinatorial optimization problem. By contrast, OMSPP deals with finding the best solution to each of the k goals. In oversubscription planning, the objective is to find a plan that collect as many goals as possible [11]. In OMSPP, we must find a path to all goals.

The OMSPP problem is related to the problem addressed by incremental search algorithms such as Lifelong Planning A* [30], D*-Lite [29], and Path-Adaptive A* [24]. Incremental search algorithms are designed to solve a sequence of search problems, where the start and goal of these search problems are the same, but the underlying graph has changed. The key idea in incremental search algorithms is to re-use information from previous searches to solve faster the current search problem. The incremental search setting is different from OMSPP: in incremental search we have one goal and the environment is dynamic, while in OMSPP we have k goals but assume the environment does not change. Another related problem is the moving-target search problem [31, 26]. A moving-target search problem is a search problem where the goal changes during the search. This is different from OMSPP where we have k goals, and the goals do not change during the search. Exploring how ideas from incremental search and moving-target search may be imported to help solve OMSPP problems is left for future work, and we outline below some exciting directions.

9 Conclusion and Future Work

In this work, we explored and analyzed two fundamental heuristic search approaches for solving the OMSPP problem. In the first approach, dubbed k×A*, k single-goal searches are performed. In the second approach, dubbed kA*, a single search is performed to find paths for all k goals. To implement kA*, one needs to defined a function that aggregates heuristic values. We analyzed the relation between properties of the available heuristics and the sufficient and required conditions for the corresponding heuristic aggregation function. Then, we proposed several ways in which one can update the aggregated heuristic values after a shortest path to one of the goals have been found. All of the

Heuristics	Aggregation			kA* Properties			
				Admis-sible	Avoids all surplus	Might skip surely	Does not re-expand
Consistent	*Con.*	*Adm.*	$\neq$ min	✓	✗	✗	✗
			min	✓	✓	✗	✓
		$\neg$*Adm.*	$\neq$ max	✓	?	✓	?
			max	✓	✗	✓	✓
	$\neg$*Con.*			✗	?	✓	✗
Admissible	*Adm.*			✓	✗	✗	✗
	$\neg$*Adm.*			✗	✗	✓	✗

Table 9 Summary of the theoretical results in Sections 4 and 6. The columns "Heuristics" and "Aggregation" show properties of the heuristic functions and the aggregation functions. "*Adm.*" and "*Con.*" are shorthand notation for indicating that the heuristic aggregation function (Φ) is admissible and consistent, respectively. "$\neq$ min" and "$\neq$ max" indicate that the heuristic aggregation function is not min and max, respectively. Similarly, "$\neq$ *Adm.*" and "$\neq$ *Con.*" indicate that the heuristic aggregation function is not admissible and consistent, respectively. The columns under the "kA* Properties" column indicate the properties of kA* with the corresponding combination of heuristic. The properties we consider are that kA* (1) is guaranteed to return optimal paths ("Admissible"), (2) avoids all surplus nodes ("Avoids all surplus"), (3) may avoid expanding a surely expanded node ("Might skip surely"), and (4) avoid re-expanding previously expanded nodes ("Does not re-expand").

proposed implementations are sound and complete. We then compared k×A* and kA* theoretically, comparing the number of nodes they expand, the memory the require, and their runtime. This analysis provided guidelines for when to use which algorithm. Table 9 summarizes the related theoretical results.[10]

These guidelines were then tested in practice in two representative standard search benchmarks: Grid pathfinding and the Pancake puzzle. Grid pathfinding represents domains whose size is polynomial in the problem input, while the Pancake represents a state space whose size is exponential in the problem input. Empirically, we showed that kA* is better than k×A* in the polynomial domain, but k×A* is better in the exponential domain. We also observed that in the polynomial domain, when the number of goals grows beyond a certain point, it is better to run a simple variant of Dijkstra's algorithm (k-Dijkstra) instead of kA* since the overhead of computing the heuristics for all k goals is not worthwhile.

To the best of our knowledge, our work is the first study of using heuristic search techniques to solve OMSPP. It opens the way for several exciting directions for future work. One such direction is to explore how to learn valuable information about the underlying graph while searching for one goal, and use this information to improve the search for the other goals. Such information

[10]Regarding the values for the "Does not re-expand" column, observe that min and max are always re-expansion-avoiding. The $\Phi(\mathbf{u}) = \lfloor \min(\mathbf{u}) \rfloor$ heuristic aggregation function is an example of a function that is admissible but not min. As discussed earlier in the paper, this function may cause node re-expansions in some cases. Re-expansions may be necessary when using admissible (but not consistent) heuristics regardless of how they are aggregated, since this is true for a single goal ($k = 1$).

can be in the form of improving the quality of the available heuristics. For example, in undirected graphs the difference between the heuristic for one goal and the heuristic between goals is an admissible heuristic by itself, which may be more accurate than the given heuristic $(h_1, \ldots h_k)$. This idea is inspired by path-finding memory-based heuristics such as Differential heuristics and others [39, 42, 19].

A different approach for future work is to build on the recent resurgence of work on bi-directional search [25, 41, 38] and adapt their approach to OMSPP. Another way to learn information about the search space is to compile OMSPP to an incremental search problem. This can be done by adding an artificial node that is connected to all goals, and running the search from this goal to the start state. Then, every iteration of the incremental search algorithm will change the weight of the edges from the artificial goal to the actual goal, setting the edge to exactly one goal with zero weight and all other edges as unpassable. There are several challenges in implementing this compilation approach, such as how to choose the order of goals that will be solved, so as to optimize the amount of information learned between searches. The benefit of such a compilation approach is that that it may enable using incremental search algorithm such as Path-Adaptive A* [24] to solve the OMSPP problem. Also, we believe that a similar approach for solving OMSPP can be done by compiling it to a moving target search problem.

References

1. Abeywickrama T, Cheema MA, Taniar D (2016) k-nearest neighbors on road networks: A journey in experimentation and in-memory implementation. Very Large Data Bases Conference (VLDB) Endowment 9(6)
2. Asai M, Fukunaga A (2017) Tie-breaking strategies for cost-optimal best first search. Journal of Artificial Intelligence Research (JAIR) 58:67–121
3. Betzalel O, Felner A, Shimony SE (2015) Type system based rational lazy IDA. In: the Symposium on Combinatorial Search (SOCS), pp 151–155
4. Burns EA, Hatem M, Leighton MJ, Ruml W (2012) Implementing fast heuristic search code. In: SoCS
5. Chalupsky H, Gil Y, Knoblock CA, Lerman K, Oh J, Pynadath DV, Russ TA, Tambe M (2001) Electric elves: Applying agent technology to support human organizations. In: IAAI, vol 1, pp 51–58
6. Dechter R, Pearl J (1985) Generalized best-first search strategies and the optimality of A*. Journal of the Association for Computing Machinery 32(3):505–536
7. Delling D, Goldberg AV, Nowatzyk A, Werneck RF (2011) PHAST: Hardware-accelerated shortest path trees. In: IEEE International Parallel Distributed Processing Symposium, pp 921–931
8. Delling D, Goldberg AV, Werneck RF (2011) Faster batched shortest paths in road networks. In: Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems

9. Dijkstra EW (1959) A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271
10. Dobson AJ, Bekris KE (2014) Improved heuristic search for sparse motion planning data structures. In: the Annual Symposium on Combinatorial Search (SOCS)
11. Domshlak C, Mirkis V (2015) Deterministic oversubscription planning as heuristic search: Abstractions and reformulations. J Artif Intell Res 52:97–169
12. Edelkamp S (2005) External symbolic heuristic search with pattern databases. In: ICAPS, pp 51–60
13. Edelkamp S (2016) External-memory state space search. In: Algorithm Engineering, Springer, pp 185–225
14. Eppstein D (1998) Finding the k shortest paths. SIAM Journal on Computing 28(2):652–673
15. Felner A, Zahavi U, Holte R, Schaeffer J, Sturtevant NR, Zhang Z (2011) Inconsistent heuristics in theory and practice. Artificial Intelligence 175(9–10):1570–1603
16. Flerova N, Marinescu R, Dechter R (2016) Searching for the M best solutions in graphical models. J Artif Intell Res 55:889–952
17. Gilon D, Felner A, Stern R (2016) Dynamic potential search - A new bounded suboptimal search. In: Proceedings of the Ninth Annual Symposium on Combinatorial Search (SOCS), Tarrytown, NY, USA, pp 36–44
18. Goldberg AV, Harrelson C (2005) Computing the shortest path: A search meets graph theory. In: Annual ACM-SIAM Symposium on Discrete algorithms (SODA), pp 156–165
19. Goldenberg M, Sturtevant NR, Felner A, Schaeffer J (2011) The compressed differential heuristic. In: AAAI Conference on Artificial Intelligence (AAAI)
20. Goldenberg M, Felner A, Sturtevant N, Holte RC, Schaeffer J (2013) Optimal-generation variants of EPEA*. In: Annual Symposium on Combinatorial Search (SoCS)
21. Goldenberg M, Felner A, Stern R, Sharon G, Sturtevant NR, Holte RC, Schaeffer J (2014) Enhanced partial expansion A. Journal of Artificial Intelligence Research (JAIR) 50:141–187
22. Hart PE, Nilsson NJ, Raphael B (1968) A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics SSC-4(2):100–107
23. Helmert M (2010) Landmark heuristics for the pancake problem. In: Third Annual Symposium on Combinatorial Search (SoCS)
24. Hernández C, Uras T, Koenig S, Baier JA, Sun X, Meseguer P (2015) Reusing cost-minimal paths for goal-directed navigation in partially known terrains. Autonomous Agents and Multi-Agent Systems 29(5):850–895
25. Holte RC, Felner A, Sharon G, Sturtevant NR (2016) Bidirectional search that is guaranteed to meet in the middle. In: Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)

26. Ishida T, Korf RE (1995) Moving-target search: A real-time search for changing goals. IEEE Transactions on Pattern Analysis and Machine Intelligence 17(6):609–619
27. Karpas E, Domshlak C (2012) Optimal search with inadmissible heuristics. In: ICAPS
28. Kleitman D, Kramer E, Conway J, Bell S, Dweighter H (1975) Elementary problems: E2564-e2569. The American Mathematical Monthly 82(10):1009–1010
29. Koenig S, Likhachev M (2005) Fast replanning for navigation in unknown terrain. IEEE Transactions on Robotics 21(3):354–363
30. Koenig S, Likhachev M, Furcy D (2004) Lifelong planning a*. Artificial Intelligence 155(1):93–146
31. Koenig S, Likhachev M, Sun X (2007) Speeding up moving-target search. In: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems (AAMAS), ACM, p 188
32. Korf RE (2004) Best-first frontier search with delayed duplicate detection. In: AAAI, vol 4, pp 650–657
33. Machuca E (2011) An analysis of multiobjective search algorithms and heuristics. In: International Joint Conference on Artificial Intelligence (IJCAI), pp 2822–2823
34. Marble JD, Bekris KE (2013) Asymptotically near-optimal planning with probabilistic roadmap spanners. IEEE Transactions on Robotics 29(2):432–444
35. Myers K, Berry P, Blythe J, Conley K, Gervasio M, McGuinness DL, Morley D, Pfeffer A, Pollack M, Tambe M (2007) An intelligent personal assistant for task and time management. AI Magazine 28(2):47
36. Ng TE, Zhang H (2002) Predicting internet network distance with coordinates-based approaches. In: The Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), IEEE, vol 1, pp 170–179
37. Pollack M (1961) Letter to the editor — the kth best route through a network. Operations Research 9(4):578–580
38. Shaham E, Felner A, Sturtevant NR, Rosenschein JS (2018) Minimizing node expansions in bidirectional search with consistent heuristics. In: the International Symposium on Combinatorial Search (SOCS), pp 81–98
39. Sturtevant NR (2007) Memory-efficient abstractions for pathfinding. AIIDE 684:31–36
40. Sturtevant NR (2012) Benchmarks for grid-based pathfinding. IEEE Transactions on Computational Intelligence and AI in Games 4(2):144–148
41. Sturtevant NR, Felner A (2018) A brief history and recent achievements in bidirectional search. In: AAAI Conference on Artificial Intelligence, pp 8000–8007
42. Sturtevant NR, Felner A, Barer M, Schaeffer J, Burch N (2009) Memory-based heuristics for explicit state spaces. In: IJCAI, pp 609–614
43. Suurballe JW (1974) Disjoint paths in a network. Networks 4:125–145

44. Tolpin D, Beja T, Shimony SE, Felner A, Karpas E (2013) Toward rational deployment of multiple heuristics in A*. In: the International Joint Conference on Artificial Intelligence (IJCAI), pp 674–680
45. Zhao S, Taniar D, Harabor DD (2018) Fast k-nearest neighbor on a navigation mesh. In: Symposium on Combinatorial Search (SoCS)
46. Zhou R, Hansen EA (2004) Structured duplicate detection in external-memory graph search. In: AAAI, pp 683–689
47. Zhou R, Hansen EA (2006) Breadth-first heuristic search. Artificial Intelligence 170(4):385–408

Appendix

A Theorems

Theorem 1 (Consistency is a necessary and sufficient condition) *Let Φ be a heuristic aggregation function. If Φ is consistent, then for any OMSPP instance and tuple of consistent heuristics $\mathbf{h}$, kA^*_Φ is admissible. If Φ is not consistent, then there exists an OMSPP instance and a tuple of consistent heuristics such that kA^*_Φ is not admissible.*

Proof First, we assume that Φ is consistent and show that for any OMSPP instance and for any tuple of consistent heuristics $\mathbf{h}$, kA^*_Φ is admissible. Assume that kA* chooses to expand a goal $t_i \in \{t_1, \dots t_k\})$ via a path p. Applying Lemma 1 to t_i, we obtain that there exists $n \in$ Open such that $g(n) + d(n, t_i) = d(s, t_i)$. Since t_i is expanded before n, we have

$$g(t_i) + \Phi(\mathbf{h}(t_i)) = F_\Phi(t_i) \le F_\Phi(n) = g(n) + \Phi(\mathbf{h}(n)) \tag{8}$$

Since all heuristics are consistent, we have that

$$\forall j : h_j(n) \le d(n, t_i) + h_j(t_i) \tag{9}$$
$$\forall j : h_j(n) - h_j(t_i) \le d(n, t_i) \tag{10}$$
$$\max(\mathbf{h}(n) - \mathbf{h}(t_i)) \le d(n, t_i) \tag{11}$$

Now, since Φ is consistent, then

$$\Phi(\mathbf{h}(n)) - \Phi(\mathbf{h}(t_i)) \le \max(\mathbf{h}(n) - \mathbf{h}(t_i)) \tag{12}$$
$$\Phi(\mathbf{h}(n)) - \Phi(\mathbf{h}(t_i)) \le d(n, t_i) \qquad \text{(due to (11))} \tag{13}$$
$$\Phi(\mathbf{h}(n)) \le d(n, t_i) + \Phi(\mathbf{h}(t_i)) \tag{14}$$
$$g(n) + \Phi(\mathbf{h}(n)) \le g(n) + d(n, t_i) + \Phi(\mathbf{h}(t_i)) \tag{15}$$
$$F_\Phi(n) \le d(s, t_i) + \Phi(\mathbf{h}(t_i)) \qquad \text{(by definition of } F_\Phi \text{ and } n) \tag{16}$$
$$F_\Phi(t_i) \le d(s, t_i) + \Phi(\mathbf{h}(t_i)) \qquad \text{(due to (8))} \tag{17}$$
$$g(t_i) + \Phi(\mathbf{h}(t_i)) \le d(s, t_i) + \Phi(\mathbf{h}(t_i)) \qquad \text{(by definition of } F_\Phi) \tag{18}$$
$$g(t_i) \le d(s, t_i) \tag{19}$$

By definition of d, we know that $d(s, t_i) \le g(t_i)$. Therefore, $g(t_i) = d(s, t_i)$, as required.

Second, we assume that Φ is not consistent, and show that there exists an OMSPP instance and a tuple of consistent heuristics such that kA^*_Φ is not admissible. To do it, we prove that there exists values for $\mathbf{u}$, u_i, $\mathbf{v}$, and δ such that: (1) the conditions in Lemma 2 are satisfied, (2) the graph described in Lemma 2 has non-negative edges, and (3) the heuristics in the resulting graph are still consistent.

Step #1. We define the values of $\mathbf{u}$, u_i, $\mathbf{v}$, and δ as follows. Since Φ is not consistent, there exists vectors $\mathbf{v}$, $\mathbf{u}$, and an index i such that $u_i = 0$ and $\Phi(\mathbf{v}) - \Phi(\mathbf{u}) > \max(\mathbf{v} - \mathbf{u})$. δ is

defined to be a value between $\Phi(\mathbf{u})$ and $\max(\mathbf{v}-\mathbf{u})$, i.e., $\Phi(\mathbf{v})-\Phi(\mathbf{u}) > \delta > \max(\mathbf{v}-\mathbf{u})$.
Step #2. All edges in the graph are non-negative: $\delta > 0$ because $u_i = 0$, the domain of Φ is vectors of non-negative values (Definition 5), $v_i \geq 0$, and hence $\max(\mathbf{v}-\mathbf{u}) \geq v_i - u_i \geq 0$. Also, $\epsilon > 0$, by definition of δ.
Step #3. To conclude the proof, it remains to observe that the heuristics involved are consistent. This is indeed the case because for all i', we have $h_{i'}(n) - h_{i'}(t_i) = v_{i'} - u_{i'} \leq \max(\mathbf{v}-\mathbf{u}) < \delta = d(n, t_i)$ and therefore $h_{i'}(n) \leq d(n,t_i) + h_{i'}(t_i)$. □

A *subconvex combination* of a vector $\mathbf{v}$ is a weighted sum $\sum_i \alpha_i \cdot v_i$ such that every coefficients α_i is non-negative and their sum is at most one, i.e., $\sum_i \alpha_i \leq 1$. The ith *order statistic* of a vector $\mathbf{v}$ is the ith smallest value of $\mathbf{v}$ and is denoted $v_{(i)}$. For instance, if $\mathbf{v} = \langle 3,1,0,1\rangle$, then $v_{(1)} = v_3 = 0$, $v_{(2)} = v_{(3)} = v_2 = v_4 = 1$ and $v_{(4)} = v_1 = 3$.

Proposition 3 *If a heuristic aggregation function Φ can be expressed as a subconvex combination of its input vector and its order statistics, then Φ is consistent.*

Proof Let Φ be a heuristic aggregation function that can be expressed as

$$\Phi(\mathbf{v}) = \sum_{i=1}^{k} \alpha_i v_i + \sum_{i=1}^{k} \alpha_{k+i} v_{(i)} \tag{20}$$

where $(\alpha_i)_{1\leq i\leq 2k}$ is a family of non-negative coefficients such that $\sum_{i=1}^{2k} \alpha_i \leq 1$.

For any two vectors $\mathbf{v}$ and $\mathbf{w}$, it is immediate that for any index i, we have $v_i - w_i \leq \max(\mathbf{v}-\mathbf{w})$. Let us show that $v_{(i)} - w_{(i)} \leq \max(\mathbf{v}-\mathbf{w})$. Define $V_{<i} = \{j | v_j < v_{(i)}\}$ and $W_{>i} = \{j | w_j > w_{(i)}\}$. By definition of the order statistics, $|V_{<i}| \leq i-1$ and $|W_{>i}| \leq k-i$. It follows that $|V_{<i} \cup W_{>i}| \leq k-1$. Thus, there exists an index l_i such that $l_i \notin V_i \cup W_i$. By construction of V_i and W_i, deduce that $v_{(i)} \leq v_{l_i}$ and that $w_{l_i} \leq w_{(i)}$ and conclude that $v_{(i)} - w_{(i)} \leq v_{l_i} - w_{l_i} \leq \max(\mathbf{v}-\mathbf{w})$.

For any two vectors $\mathbf{v}$ and $\mathbf{w}$, we have

$$\Phi(\mathbf{v}) - \Phi(\mathbf{w}) = \sum_{i=1}^{k} \alpha_i v_i + \sum_{i=1}^{k} \alpha_{k+i} v_{(i)} - \sum_{i=1}^{k} \alpha_i w_i - \sum_{i=1}^{k} \alpha_{k+i} w_{(i)} \tag{21}$$

$$= \sum_{i=1}^{k} \alpha_i (v_i - w_i) + \alpha_{i+k}(v_{(i)} - w_{(i)}) \tag{22}$$

$$\leq \sum_{i=1}^{2k} \alpha_i \max(\mathbf{v}-\mathbf{w}) \qquad (\alpha_i \geq 0) \tag{23}$$

$$\leq \max(\mathbf{v}-\mathbf{w}) \sum_{i=1}^{2k} \alpha_i \tag{24}$$

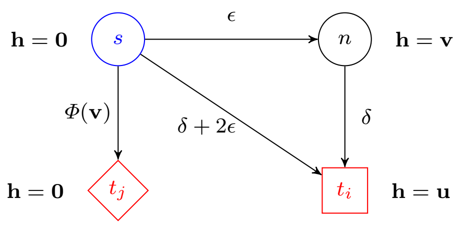
$$\leq \max(\mathbf{v}-\mathbf{w}) \qquad (\sum_{i=1}^{2k} \alpha_i \leq 1) \tag{25}$$

Therefore, Φ is consistent.□

Corollary 1 *Minimum, maximum, mean, median, and projection of a single element in a vector are all consistent heuristic aggregation functions.*

Proof Each of these functions can be expressed as a subconvex combination of the input vector and its order statistics: projecting the i^{th} element is $\Phi(\mathbf{v}) = v_i$; the *mean* is $\Phi(\mathbf{v}) = \frac{v_1+\cdots+v_k}{k}$; the *median* is $\Phi(\mathbf{v}) = \frac{1}{2}(v_{(\frac{k}{2})} + v_{(\frac{k+1}{2})})$; the *maximum* is $\Phi(\mathbf{v}) = v_{(k)}$; and the *minimum* is $\Phi(\mathbf{v}) = v_{(1)}$. Thus, according to Theorem 3 they are all consistent. □

Proposition 4 *A heuristic aggregation function Φ is not consistent if it can be expressed as $\Phi(\mathbf{v}) = \sum_i \alpha_i v_i$ where all coefficients α_i are non-negative and their sum is larger than one.*



Node	g	$\Phi(\mathbf{h})$	F_Φ
t_i	$\delta+2\epsilon$	$\Phi(\mathbf{u})$	$\Phi(\mathbf{v})-\epsilon$
t_j	$\Phi(\mathbf{v})$	0	$\Phi(\mathbf{v})$
n	ϵ	$\Phi(\mathbf{v})$	$\Phi(\mathbf{v})+\epsilon$

Fig. 9 A generic counter-example for kA$^*_\Phi$. The start is s, nodes t_i and t_j are goals where $i \neq j$. For any $\mathbf{v}$, $\mathbf{u}$, i, δ, and ϵ such that (1) $u_i = 0$, (2) $\delta > 0$, (3) $\Phi(\mathbf{v}) > \Phi(\mathbf{u}) + \delta$, and (4) $\epsilon = \frac{\Phi(\mathbf{v})-\Phi(\mathbf{u})-\delta}{3} > 0$, running kA$^*_\Phi$ will return a suboptimal path to t_i. To see this, the table on the right shows the g, $\Phi(\mathbf{h})$, and F_Φ values for n, t_i, and t_j after expanding s. The optimal path to t_i is through n and costs $\delta+\epsilon$, but kA$^*_\Phi$ will expand t_i before n, returning a path of length $\delta+2\epsilon$ to t_i, which is suboptimal.

Proof Consider the vectors $\mathbf{v} = \mathbf{1}$ and $\mathbf{u} = \mathbf{0}$. Their difference $\mathbf{v}-\mathbf{u} = \sum_i \alpha_i(1-0) = \sum_i \alpha_i > 1 = \max(\mathbf{v}-\mathbf{w})$. Thus, Φ is not consistent.□

Corollary 3 Sum *is not a consistent heuristic aggregation functions.*

Proof The sum can be expressed as $\Phi(\mathbf{v}) = \sum_i v_i$, i.e., all the coefficients are one. Therefore, their sum is over one and due to Observation 4, this means sum is not consistent.□

Lemma 2 (Generic example of inadmissible kA*) *For the OMSPP instance defined by the graph depicted in Figure 9, a start vertex s, and goals t_i and t_j, if (1) $u_i = 0$, (2) $\delta > 0$, (3) $\Phi(\mathbf{v}) > \Phi(\mathbf{u}) + \delta$, and (4) $\epsilon = \frac{\Phi(\mathbf{v})-\Phi(\mathbf{u})-\delta}{3} > 0$, then running* kA$^*_\Phi$ *will find a suboptimal path to t_i.*

Proof In the first iteration of kA$^*_\Phi$, the vertex s will be expanded, adding to Open the vertices n, t_i, and t_j, with g values ϵ, $\delta+2\epsilon$, and $\Phi(\mathbf{v})$, respectively, and F_Φ values $\epsilon+\Phi(\mathbf{v})$, $\delta+2\epsilon+\Phi(\mathbf{u})$, and $\Phi(\mathbf{v})$, respectively. To prove Lemma 2, we show below that kA$^*_\Phi$ will expand t_i first and then t_j, halting afterwards without expanding n. The optimal path to t_i goes through n, and thus it will not be found by kA$^*_\Phi$, as required. To show that t_i is expanded first, we express $F_\Phi(t_i)$ in terms of ϵ and $\Phi(\mathbf{v})$ as follows:

$$F_\Phi(t_i) = \delta + 2\epsilon + \Phi(\mathbf{u}) \tag{26}$$
$$= \delta + 3\epsilon - \epsilon + \Phi(\mathbf{u}) \tag{27}$$
$$= \delta + \Phi(\mathbf{v}) - \Phi(\mathbf{u}) - \delta - \epsilon + \Phi(\mathbf{u}) \tag{28}$$
$$= \Phi(\mathbf{v}) - \epsilon \tag{29}$$

The table in the righthand side of Figure 9 lists the F_Φ values of n, t_i, and t_j expressing only with $\Phi(\mathbf{v})$ and ϵ, showing that indeed kA$^*_\Phi$ will halt before expanding n, as required. □

Theorem 2 (Sufficient condition for avoiding re-expansions) *Let Φ be a heuristic aggregation function. If Φ is re-expansion-avoiding, then for any OMSPP instance and tuple of consistent heuristics* $\mathbf{h}$, *when* kA$^*_\Phi$ *expands a node n then $g(n) = d(s,n)$.*

Proof Assume that kA* chooses to expand a node $m \in$ Open. Applying Lemma 1 to m, we obtain that there exists $n \in$ Open such that $g(n) + d(n,m) = d(s,m)$. Since m is expanded before n, we have

$$g(m) + \Phi(\mathbf{h}(m)) = F_\Phi(m) \leq F_\Phi(n) = g(n) + \Phi(\mathbf{h}(n)) \tag{30}$$

Since all heuristics are consistent, we have that

$$\forall j : h_j(n) \leq d(n,m) + h_j(m) \tag{31}$$
$$\forall j : h_j(n) - h_j(m) \leq d(n,m) \tag{32}$$
$$\max(\mathbf{h}(n) - \mathbf{h}(m)) \leq d(n,m) \tag{33}$$

Now, since Φ is re-expansion-avoiding, then

$$\Phi(\mathbf{h}(n)) - \Phi(\mathbf{h}(m)) \leq \max(|\mathbf{h}(n) - \mathbf{h}(m)|) \tag{34}$$
$$\Phi(\mathbf{h}(n)) \leq d(n,m) + \Phi(\mathbf{h}(m)) \qquad \text{(due to (33))} \tag{35}$$
$$g(n) + \Phi(\mathbf{h}(n)) \leq g(n) + d(n,m) + \Phi(\mathbf{h}(m)) \tag{36}$$
$$F_\Phi(n) \leq d(s,m) + \Phi(\mathbf{h}(m)) \qquad \text{(by definition of } F_\Phi \text{ and } n) \tag{37}$$
$$F_\Phi(m) \leq d(s,m) + \Phi(\mathbf{h}(m)) \qquad \text{(due to (30))} \tag{38}$$
$$g(m) + \Phi(\mathbf{h}(m)) \leq d(s,m) + \Phi(\mathbf{h}(m)) \qquad \text{(by definition of } F_\Phi) \tag{39}$$
$$g(m) \leq d(s,m) \tag{40}$$

By definition of d, we know that $d(s,m) \leq g(m)$. Therefore, $g(m) = d(s,m)$, as required.

Theorem 3 (Admissibility is a necessary and sufficient condition) *Let Φ be a heuristic aggregation function. (1) If Φ is admissible then for any OMSPP instance and for any tuple of admissible heuristics $\mathbf{h}$, kA^*_Φ is admissible. (2) If Φ is not admissible, then there exists an OMSPP instance and a tuple of admissible heuristics such that kA^*_Φ is not admissible.*

Proof First, we assume that Φ is admissible and show that for any OMSPP instance and for any tuple of admissible heuristics $\mathbf{h}$, kA^*_Φ is admissible. Assume that kA^*_Φ chooses to expand a goal $t_i \in \{t_1, \dots t_k\}$. Due to Lemma 1, there exists $n \in \textsc{Open}$ such that $g(n) + d(n,t_i) = d(s,t_i)$.

$$h_i(n) \leq d(n,t_i) \qquad (h_i \text{ is admissible}) \tag{41}$$
$$\Phi(\mathbf{h}(n)) \leq \min_j h_j(n) \leq d(n,t_i) \qquad (\Phi \text{ is admissible}) \tag{42}$$
$$g(n) + \Phi(\mathbf{h}(n)) \leq g(n) + d(n,t_i) = d(s,t_i) \qquad \text{(by definition of } n) \tag{43}$$
$$F_\Phi(t_i) \leq F_\Phi(n) \leq d(s,t_i) \qquad (t_i \text{ is expanded before } n) \tag{44}$$
$$g(t_i) + \Phi(\mathbf{h}(t_i)) \leq d(s,t_i) \qquad \text{(by definition of } F) \tag{45}$$
$$g(t_i) \leq d(s,t_i) \qquad (\Phi \text{ is non-negative}) \tag{46}$$

By definition of d, we know that $d(s,t_i) \leq g(t_i)$. Therefore, $g(t_i) = d(s,t_i)$, as required.

Second, we assume that Φ is not admissible, and show that there exists an OMSPP instance and a tuple of admissible heuristics such that kA^*_Φ is not admissible. This part relies again on Lemma 2. To do it, we prove that there exists values for $\mathbf{u}$, u_i, $\mathbf{v}$, and δ such that: (1) the conditions in Lemma 2 are satisfied, (2) the graph described in Lemma 2 has non-negative edges, and (3) the heuristics in the resulting graph are still admissible.
Step #1. We define the values of $\mathbf{u}$, u_i, $\mathbf{v}$, and δ as follows. Let $\mathbf{u} = \mathbf{0}$ and $i = \arg\min \mathbf{v}$. δ is defined to be a value between $\Phi(\mathbf{v})$ and $\min \mathbf{v} = v_i$, i.e., $\Phi(\mathbf{v}) > \delta > \min \mathbf{v}$.
Step #2. $\delta > 0$ since $\delta > \min \mathbf{v}$, and all vectors are non-negative. Following, $\epsilon > 0$ by definition and $\Phi(\mathbf{v}) > \delta\, 0$.
Step #3. To conclude the proof, it remains to observe that the heuristics involved are admissible. This is indeed the case because we have $h_{t_i}(A) \leq \min \mathbf{v} \leq \delta = d(A,B)$ and therefore $h_{t_i}(A) \leq d(A,B)$.□

Theorem 4 *If there exists a heuristic h_i that is not admissible, and the heuristic aggregation function Φ is not the constant 0, then there exists a OMSPP instance and a tuple of arbitrary heuristics such that kA^*_Φ is not admissible.*

Proof This proof follows the same format as the proofs of Theorems 1 and 3. We rely on Lemma 2, showing that if there exists h_i that is not admissible and Φ is not always zero then there exists values for $\mathbf{u}$, u_i, $\mathbf{v}$, and δ such that: (1) the conditions in Lemma 2 are satisfied, and (2) the graph described in Lemma 2 has non-negative edges.
Step #1. We define the values of $\mathbf{u}$, u_i, $\mathbf{v}$, and δ as follows. Since Φ is not the constant function 0, there exists a vector $\mathbf{v}$, such that $\Phi(\mathbf{v}) > 0$, Let $\mathbf{u} = \mathbf{0}$ and δ be a value between $\Phi(\mathbf{v})$ and 0, i.e., $\Phi(\mathbf{v}) > \delta > 0$.
Step #2. It trivially holds that $\Phi(\mathbf{v})$, δ, and ϵ are all non-zero. □

Theorem 7 *Let Φ be a heuristic aggregation function. (1) If $\Phi = \min$, for every OMSPP instance and tuple of consistent heuristics, kA^*_Φ never expands any node that is a surplus node for $k{\times}A^*$. (2) If Φ is admissible but it is not $\min$, then there exists an OMSPP instance, a tuple of consistent heuristics, and a node A that is expanded by kA^*_Φ and is a surplus node for $k{\times}A^*$. (3) If $\Phi = \max$, then there exists an OMSPP instance, a tuple of consistent heuristics, and a node A that is a expanded by kA^*_Φ and is a surplus node for $k{\times}A^*$.*

Proof First, we prove that $\mathrm{kA}^*_{\min}$ does not expand any surplus node. Let n be a node expanded by $\mathrm{kA}^*_{\min}$ in some OMSPP instance. Let t_i be such that $h_{t_i}(n) = \min h_{t_i}(n)$. Applying Lemma 1 to t_i, we obtain $n_i \in$ Open such that $g(n_i) + d(n_i, t_i) = d(s, t_i)$. For any node p on the path from s to n, we have the following derivation.

$$\begin{aligned}
h_{t_i}(p) &\le d(p,n) + h_{t_i}(n) && (h_{t_i} \text{ is consistent}) && (47)\\
h_{t_i}(p) &\le d(p,n) + \Phi(\mathbf{h}(n)) && (\text{from the choice of } t_i) && (48)\\
g(p) + h_{t_i}(p) &\le g(p) + d(p,n) + \Phi(\mathbf{h}(n)) && && (49)\\
g(p) + h_{t_i}(p) &\le g(n) + \Phi(\mathbf{h}(n)) && (\text{by definition of } p) && (50)\\
g(p) + h_{t_i}(p) &\le F(n) && (\text{by definition of } F) && (51)\\
g(p) + h_{t_i}(p) &\le F(n_i) = g(n_i) + \Phi(\mathbf{h}(n_i)) && (n \text{ is expanded}) && (52)\\
g(p) + h_{t_i}(p) &\le g(n_i) + h_i(n_i) && (\text{by definition of } \Phi) && (53)\\
g(p) + h_{t_i}(p) &\le g(n_i) + d(n_i, t_i) && (h_{t_i} \text{ is admissible}) && (54)\\
g(p) + h_{t_i}(p) &\le d(s, t_i) && (n_i \text{ is chosen via Lemma 1}) && (55)
\end{aligned}$$

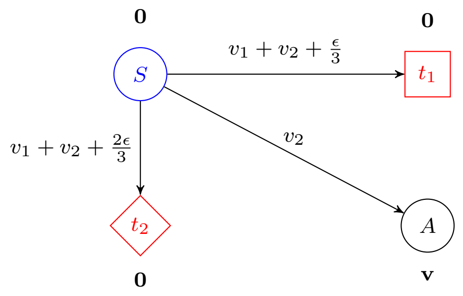
Thus, n is not surplus w.r.t Π_i. Therefore, n is not surplus for the OMSPP.

Second, we prove that if there exists $\mathbf{v}$ such that $\Phi(\mathbf{v}) < \min \mathbf{v}$, then there is an OMSPP instance, a tuple of consistent heuristics, and a surplus node A such that kA^*_Φ explores A. Let $\mathbf{v}$ such that $\Phi(\mathbf{v}) < \min \mathbf{v}$. Assume without loss of generality that $v_1 = \min \mathbf{v}$ and $v_2 = \max \mathbf{v}$. Let $\epsilon = \Phi(\mathbf{v}) - v_1 < 0$. Consider the OMSPP on Figure 10. Since $\epsilon < 0$, we have $F(A) < F(t_2) < F(t_1)$. Therefore, kA^*_Φ expands A. On the other hand, $f_1(A) > f_1(t_1)$ and $f_2(A) > f_2(t_2)$, so A is a surplus node. Thus kA^*_Φ expands a surplus node.

Third, if $\Phi = \max$ then we provide an OMSPP instance, a tuple of consistent heuristics, and a surplus node A such that kA^*_Φ explores A. Consider the instance in Figure 11.□

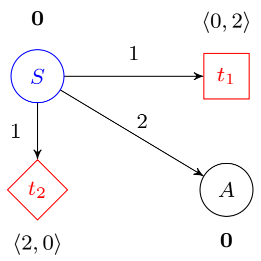
Theorem 8 *Let Φ be a heuristic aggregation function. (1) If Φ is admissible, then for any OMSPP instance and for any tuple of admissible heuristics $\mathbf{h}$, kA^*_Φ expands all the nodes that are surely expanded by $k{\times}A^*$. (2) If Φ is not admissible, then there exists an OMSPP instance and a tuple of consistent heuristics such that kA^*_Φ does not expand some nodes that are surely expanded by $k{\times}A^*$.*

Proof First, we assume that Φ is admissible and show that for any OMSPP instance and for any tuple of admissible heuristics $\mathbf{h}$, kA^*_Φ expands all surely expanded nodes. Let n be a node that is surely expanded w.r.t. some goal t_i. That is, n is reachable from s by a path of nodes with f_i values lower than $d(s,t_i)$ and $f_i(n) < d(s,t_i)$. Φ is admissible so for any node m along this path we have $\Phi(\mathbf{h}(m)) \le \min \mathbf{h}(m) \le h_{t_i}(m)$. Thus, the F values of the nodes along this path must also be lower than $d(s,t_i)$. As all edges in the underlying graph have non-negative cost, $h_{t_i}(t_i) = 0$ and therefore $F(t_i) = d(s,t_i)$. Hence, the minimal F value in Open is $d(s,t_i)$ when t_i is expanded. Thus, all the nodes along the path to n and n itself must have already been expanded, as all of them have F values smaller than $d(s,t_i) = F(t_i)$.



Node		$A^*_{t_1}$		$A^*_{t_2}$		kA^*_{Φ}	
	g	h_1	f_1	h_2	f_2	$\Phi(\mathbf{h})$	F
S	0	0	0	0	0	0	0
t_1	$v_1 + v_2 + \frac{\epsilon}{3}$	0	$v_1 + v_2 + \frac{\epsilon}{3}$	0	$v_1 + v_2 + \frac{\epsilon}{3}$	0	$v_1 + v_2 + \frac{\epsilon}{3}$
t_2	$v_1 + v_2 + \frac{2\epsilon}{3}$	0	$v_1 + v_2 + \frac{2\epsilon}{3}$	0	$v_1 + v_2 + \frac{2\epsilon}{3}$	0	$v_1 + v_2 + \frac{2\epsilon}{3}$
A	v_2	v_1	$v_1 + v_2$	v_2	$2v_2$	$\Phi(\mathbf{v})$	$v_1 + v_2 + \epsilon$

Fig. 10 Generic example with consistent heuristics. We assume that $v_1 \leq v_2$ and ϵ is defined as $\epsilon = \Phi(\mathbf{v}) - v_1$. The vector of heuristic values is displayed next to each node.



Node		$A^*_{t_1}$		$A^*_{t_2}$		$kA^*_{\max}$	
	g	h_1	f_1	h_2	f_2	$\max(\langle h_1, h_2 \rangle)$	F
S	0	0	0	0	0	0	0
t_1	1	0	1	2	3	2	3
t_2	1	2	3	0	1	2	3
A	2	0	2	0	2	0	2

Fig. 11 Max example with consistent heuristics where $kA^*_{\max}$ expands a surplus node.

Second, we assume that Φ is not admissible, and show that there exists an OMSPP instance, a tuple of consistent heuristics, and a surely expanded node A such that kA^*_{Φ} does not expand A. Assume that Φ is not admissible, then $\exists \mathbf{v}, \Phi(\mathbf{v}) > \min(\mathbf{v})$. Assume without loss of generality that $v_1 = \min \mathbf{v}$ and $v_2 = \max \mathbf{v}$. Let $\epsilon = \Phi(\mathbf{v}) - v_1$. Consider the OMSPP on Figure 10. On the one hand, since $\epsilon > O$, the F-value for A is larger than the F value for t_1 and t_2. Thus, A is not expanded by kA^*_{Φ}. On the other hand, A is a surely expanded node in the SPP instance with goal t_1, so it is a surely expanded node in the OMSPP instance. Therefore kA^*_{Φ} does not expand all surely expanded nodes.□