OJMO

Open Journal of Mathematical Optimization

Giovanni Righini

Efficient optimization of the Held–Karp lower bound

Volume 2 (2021), article no. 9 (17 pages)

https://doi.org/10.5802/ojmo.11

Article submitted on April 27, 2020, revised on August 12, 2021,
accepted on October 22, 2021.

CENTRE MERSENNE

Open Journal of Mathematical Optimization is a member of the
Centre Mersenne for Open Scientific Publishing
http://www.centre-mersenne.org/

Efficient optimization of the Held–Karp lower bound

Giovanni Righini
University of Milan, Department of Computer Science
via Celoria 18, Milano
Italy
giovanni.righini@unimi.it

Abstract

Given a weighted undirected graph $G = (V, E)$, the Held–Karp lower bound for the Traveling Salesman Problem (TSP) is obtained by selecting an arbitrary vertex $\bar{p} \in V$, by computing a minimum cost tree spanning $V \backslash \{\bar{p}\}$ and adding two minimum cost edges adjacent to $\bar{p}$. In general, different selections of vertex $\bar{p}$ provide different lower bounds. In this paper it is shown that the selection of vertex $\bar{p}$ can be optimized, to obtain the largest possible Held–Karp lower bound, with the same worst-case computational time complexity required to compute a single minimum spanning tree. Although motivated by the optimization of the Held–Karp lower bound for the TSP, the algorithm solves a more general problem, allowing for the efficient pre-computation of alternative minimum spanning trees in weighted graphs where any vertex can be deleted.

Digital Object Identifier 10.5802/ojmo.11

Keywords Traveling salesman problem, Minimum spanning tree, Held–Karp lower bound, Union-Find data-structure.

1 Introduction and motivation

The Traveling Salesman Problem (TSP) is the well-known problem of computing a minimum cost Hamiltonian cycle on a given weighted graph [1, 5]. Since the problem is NP-hard, many techniques have been devised not only to solve it to optimality but also to compute upper and lower bounds to its optimal value. One of the most celebrated lower bounding techniques was proposed in 1970 by Held and Karp [6]; it is known to provide very tight lower bounds with limited computational effort and it is used as a sub-routine within state-of-the-art TSP solvers, like Concorde [1], and state-of-the-art TSP heuristics, like the Helsgaun implementation of the Lin–Kernighan algorithm [7]. The Held–Karp lower bound is still subject to investigation: besides being included in branch-and-bound algorithms for the symmetric TSP [9], its properties have been studied for the asymmetric TSP [10] and for the metric TSP [8]. Recent contributions include, for instance, hybridization with constraint programming [2] and randomized approximation algorithms [3].

The lower bounding procedure proposed by Held and Karp is based on Lagrangean relaxation: at each iteration the edge weights of the underlying graph $G = (V, E)$ are modified so that the costs of Hamiltonian cycles remain unaffected. Then, valid lower bounds for the TSP can be efficiently obtained from minimum spanning 1-trees, i.e. spanning trees with an additional edge, relying on the observation that a Hamiltonian cycle is a special case of a spanning 1-tree. This can be done in different ways: here below we list three of them (see Figure 1).

Lower bound LB_0**:**

- a minimum spanning tree T^* of G;
- a minimum cost edge not in T^*.

Lower bound $LB_H(\bar{\ell})$ (Helsgaun [7]):

- a minimum spanning tree T^* of G;
- a minimum cost edge $[\bar{\ell}, j] \notin T^*$, incident to a leaf $\bar{\ell}$ of T^*.

Lower bound $LB_{HK}(\bar{p})$ (Held and Karp [6]):

- a minimum spanning tree $T^*_{\bar{p}}$ of the subgraph induced by $V \backslash \bar{p}$, where $\bar{p}$ is an arbitrary selected vertex;
- two minimum cost edges of G with an endpoint in $\bar{p}$.

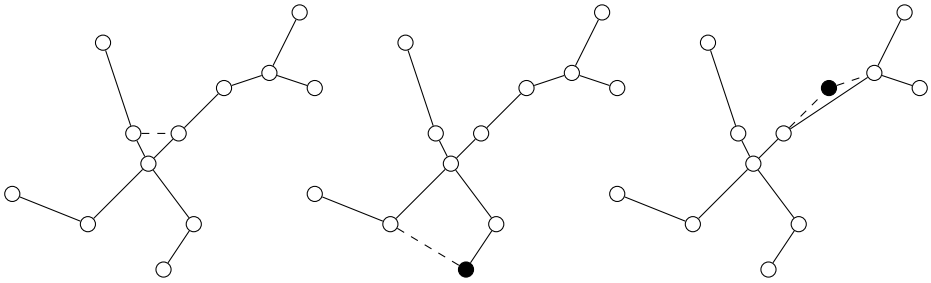


Figure 1 Left (LB_0): a minimum spanning 1-tree, made by a minimum spanning tree and a minimum cost edge (dashed). Center (LB_H): a Helsgaun 1-tree, made by a minimum spanning tree and a minimum cost edge (dashed) incident to a leaf (filled). Right (LB_{HK}): a Held–Karp 1-tree, made by a minimum spanning tree disregarding a vertex (filled) and two minimum cost edges incident to it (dashed). The comparison illustrates the difference between the three corresponding lower bounds.

It is immediate to observe that $LB_H(\bar{\ell}) \geq LB_0$ for any choice of a leaf $\bar{\ell} \in T^*$ and $LB_{HK}(\bar{p}) \geq LB_0$ for any choice of $\bar{p} \in V$, because $LB_H(\bar{\ell})$ and $LB_{HK}(\bar{p})$ are the costs of 1-trees of G, while LB_0 is the cost of a minimum 1-tree of G.

Both $LB_H(\bar{\ell})$ and $LB_{HK}(\bar{p})$ depend on the selection of a vertex: then, both can be maximized by selecting $\bar{\ell}$ and $\bar{p}$ in an optimal way. Let $LB_H^* = \max_{\bar{\ell} \in \Lambda(T^*)}\{LB_H(\bar{\ell})\}$, where $\Lambda(T^*)$ indicates the set of leaves of T^*, and $LB_{HK}^* = \max_{\bar{p} \in V}\{LB_{HK}(\bar{p})\}$. Since $LB_{HK}(\bar{p}) = LB_H(\bar{\ell})$ when $\bar{p} = \bar{\ell}$ and since $V \supset \Lambda(T^*)$, then it is guaranteed that $LB_H^* \leq LB_{HK}^*$.

Although LB_{HK}^* is more appealing than LB_H^* for the tightness of the TSP lower bound that can be achieved, the computation of LB_H^* is much easier: it just requires to select the leaf of T^* for which the second smallest cost of the incident edges is maximum. For this reason the Helsgaun lower bound is attractive for practical implementation purposes.

The main motivation of this paper is the efficient optimization of $LB_{HK}(\bar{p})$, which is less trivial: a naive procedure is to compute as many *alternative minimum spanning trees* as the number n of vertices of G, each one spanning G with the exception of a vertex $\bar{p} \in V$. This paper presents and analyzes an algorithm that allows to compute all the *alternative edges* that are needed to rebuild a minimum cost spanning tree when a vertex is deleted from G. This goal is achieved with the same worst-case computational complexity required by the computation of a single minimum spanning tree of G. As a by-product, the algorithm allows to efficiently compute LB_{HK}^*.

The algorithm takes in input a weighted graph, a minimum spanning tree and a sorted list of all edges and it returns an optimal set of alternative edges. In particular, it computes a vertex and a corresponding 1-tree that provide the largest Held–Karp lower bound LB_{HK}^*.

Paper outline

The next section gives an overview of the main idea behind the algorithm, with the aim of facilitating the reader. The full details are presented in Sections 3, 4 and 5. Section 3 illustrates the main data-structures, their properties, their initialization and some procedures for updating them. Section 4 describes the main algorithm (Algorithm 7). Section 5 describes how the maximum value LB_{HK}^* of the Held–Karp lower bound is obtained.

2 Overview of the algorithm

Consider an undirected graph $G = (V, E)$ with edge costs $c : E \mapsto \Re$; the number of vertices is n and the number of edges is m. A minimum spanning tree T^* of G is assumed to be known and the edges of E are assumed to having been sorted by non-decreasing cost in a previous execution of Kruskal algorithm.

When a vertex $p \in V$ is deleted, T^* is disconnected into a forest F_p^* made of as many connected components as the neighbors of p in T^*. In the remainder they are called *p-components*, to mean that they arise when p is deleted. When vertex $p \in V$ is deleted, *alternative edges* must be found to reconnect the p-components and they must be selected to produce a new minimum spanning tree T_p^*.

A trivial way to accomplish this task is to repeat, for each deleted vertex $p \in V$, te computation of a minimum spanning tree T_p^* with Kruskal or Boruvka algorithm after having restored the state of a suitable data-structure

to represent the p-components of F_p^*. The vertices in each p-component can be identified in $O(n)$ and this is the complexity to set up the state of a Union-Find data-structure (for readers unfamiliar with Union-Find, it is illustrated in detail in Appendix A). Then, the computation of an alternative minimum spanning tree with Kruskal algorithm requires $O(m + n \log d(p))$, where $d(p)$ indicates the degree of vertex $p \in V$. Hence, repeating this naive procedure for all vertices would require $O(mn + n^2 \log n)$.

To achieve a better worst-case time complexity one could work the other way around: all edges are scanned in non-decreasing order of their cost, as in Kruskal algorithm, and for each edge $[i, j] \notin T^*$ the algorithm searches the vertices p such that the edge $[i, j]$ reconnects two different p-components of F_p^*. For each vertex p and for each pair of p-components in F_p^*, the first edge that is found to reconnect them, certainly belongs to T_p^* owing to the edge ordering. In this way all forests F_p^* can be populated in parallel with alternative edges for all $p \in V$ until all alternative minimum spanning trees T_p^* are produced.

This search, if performed as described above, would be very inefficient. However, it can be restricted (and accelerated) as follows. Consider the minimum spanning tree T^* and an edge $[i, j] \notin T^*$. Adding edge $[i, j]$ to T^* defines a unique cycle $C(i, j)$. The vertices along $C(i, j)$, except i and j, are those for which $[i, j]$ is a candidate alternative edge: if a vertex $p \in V$ along $C(i, j)$ is deleted from T^*, edge $[i, j]$ reconnects two different p-components. On the contrary, for all vertices $p \in V$ not belonging to $C(i, j)$, edge $[i, j]$ cannot be a candidate alternative edge, because both i and j belong to the same p-component. To efficiently scan $C(i, j)$, the algorithm illustrated in the remainder skips in constant time all vertices p along $C(i, j)$ for which an alternative edge reconnecting the p-components of i and j has already been found. The effect is illustrated in Figure 2. When edge $[i', j']$ is considered, it can be an alternative edge for vertices a, b, c, d and e. Later on, when the more expensive edge $[i'', j'']$ is considered, it can be an alternative edge for vertices f, g, h and l, but there is no point in considering it for a, b, c, d and e again.

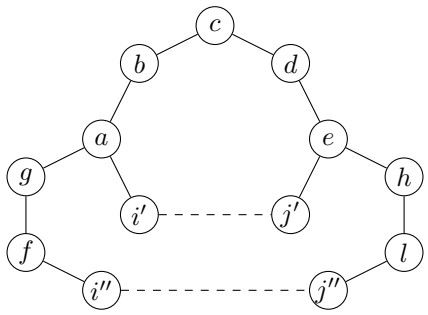


Figure 2 Cycles $C(i', j')$ and $C(i'', j'')$ partially overlap: since $[i', j']$ is cheaper than $[i'', j'']$, the vertices belonging to $C(i', j')$ can be skipped when $[i'', j'']$ is considered.

This idea allows to achieve an overall computational complexity $O(m + n \log n)$, since all operations executed when a candidate alternative edge is discarded give a contribution $O(1)$ for each edge, i.e. $O(m)$ overall, while all operations executed when a candidate alternative edge is inserted in one of the minimum alternative spanning trees give a contribution $O(\log n)$ for each insertion, i.e. $O(n \log n)$ overall.

3 The data-structures

This section describes the main data-structures used in the algorithm, their properties and their initialization. Some necessary terminology and definitions are also introduced.

3.1 The oriented minimum spanning tree

The minimum spanning tree T^* is initially oriented from an arbitrarily selected root vertex $r \in V$.

▶ **Definition 1.** The *predecessor* of any vertex $p \in V \backslash \{r\}$, indicated by $\mathsf{Pred}(p)$, is defined as the vertex adjacent to p along the path between p and r in T^*.

▶ **Property 2.** Since T^* is a spanning tree, there exists a unique path between any vertex $p \in V \backslash \{r\}$ and r; therefore the predecessor exists and is unique for each $p \in V \backslash \{r\}$.

$\mathsf{Pred}(r)$ is set to a null value to indicate that the root vertex r has no predecessor.

▶ **Definition 3.** On the oriented tree T^* the *depth* of any vertex $p \in V$, indicated by $\mathsf{Depth}(p)$, is defined as the number of edges between r and p.

From Definition 3 and Property 2 the following property follows.

▶ **Property 4.** For any two vertices u and v such that $v = \mathsf{Pred}(u)$, it holds $\mathsf{Depth}(u) = \mathsf{Depth}(v) + 1$.

On the oriented tree T^* two additional values $\mathsf{Dn}(p)$ and $\mathsf{Up}(p)$ are defined for each vertex $p \in V$. Consider a depth-first-search visit of T^* and let us define a *move* to happen every time an edge is traversed in any direction.

▶ **Definition 5.** We define $\mathsf{Dn}(r) = 1$. For all $p \in V \backslash \{r\}$, $\mathsf{Dn}(p)$ is the *progressive number of the move* that reaches p from $\mathsf{Pred}(p)$. For all $p \in V$, $\mathsf{Up}(p)$ is the *progressive number of the move* that reaches $\mathsf{Pred}(p)$ from p.

The following properties hold.

▶ **Property 6.**

i. The indices $\mathsf{Dn}(p)$ and $\mathsf{Up}(p)$ have unique values in T^*.
ii. Their values span the interval $[1, \ldots, 2n]$.
iii. For any vertex $v \in V$ and for each vertex $u \in V$ in the oriented subtree rooted at v, with $u \neq v$, $\mathsf{Dn}(v) < \mathsf{Dn}(u) < \mathsf{Up}(u) < \mathsf{Up}(v)$.

An example is illustrated in Figure 3.

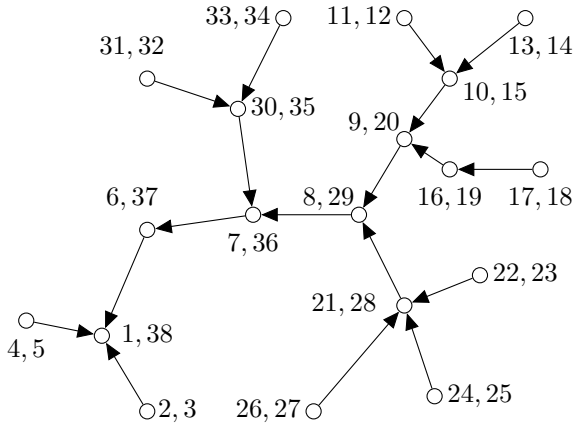


Figure 3 An oriented spanning tree with the values $(\mathsf{Dn}, \mathsf{Up})$ for each vertex.

Consider an edge $[i, j] \notin T^*$. When it is added to T^* a unique cycle, indicated by $C(i, j)$, is formed. Owing to the orientation of T^*, it is possible to define the *apex* of the cycle.

▶ **Definition 7.** The *apex* of a cycle $C(i, j)$ is the minimum depth vertex along it.

Owing to the orientation of T^* and to Property 6, the following properties hold.

▶ **Property 8.**

i. For any cycle $C(i, j)$, the apex exists and it is unique;
ii. if vertex p is the apex of $C(i, j)$, then $\mathsf{Dn}(p) \leq \min\{\mathsf{Dn}(i), \mathsf{Dn}(j)\}$ and $\mathsf{Up}(p) \geq \max\{\mathsf{Up}(i), \mathsf{Up}(j)\}$.

Let $\mathsf{SubTree}(p, i, j)$ be a boolean function that tests Property 8ii, to check whether an endpoint of edge $[i, j]$ is the apex of $C(i, j)$ or not.

A consequence of Definition 7 and Property 8 is the following.

▶ **Property 9.** Given any edge $[i, j] \notin T^*$, if a vertex p verifies $\mathsf{SubTree}(\mathsf{p}, \mathsf{i}, \mathsf{j})$ and it also belongs to $C(i, j)$, then it is the apex of $C(i, j)$.

Property 9 is exploited in Algorithm 8 and 9 described in Section 4.

Algorithm 1 The procedure that orients the minimum spanning tree.

```
 1: procedure Orient
 2:     for p ∈ V do
 3:         δ(p) ← ∅
 4:     for [i, j] ∈ T* do
 5:         δ(i) ← δ(i) ∪ {j}
 6:         δ(j) ← δ(j) ∪ {i}
 7:     r ← Select
 8:     Pred(r) ← 0
 9:     Depth(r) ← 0
10:     α ← 0
11:     DFS(r)
```

Algorithm 2 The recursive procedure to visit T^* in depth-first-search order.

```
 1: procedure DFS(p)
 2:     α ← α + 1
 3:     Dn(p) ← α
 4:     for k ∈ δ(p) do
 5:         if k ≠ Pred(p) then
 6:             Pred(k) ← p
 7:             Depth(k) ← Depth(p) + 1
 8:             DFS(k)
 9:     α ← α + 1
10:     Up(p) ← α
```

3.1.1 Procedure Orient

The values of Pred, Depth, Dn and Up are initially computed by the procedure Orient shown in Algorithm 1.

The procedure pre-computes the star $\delta(p)$ of each vertex $p \in V$, from the list of edges of T^* (lines 2-6). A root vertex r with no predecessor is arbitrarily selected and its predecessor and depth are set to 0 (lines 7-9). Then T^* is visited in depth-first-search order from vertex r with a call to the recursive procedure DFS (line 11).

A counter α counts every move in either direction along the edges of T^*. Every time a vertex p is reached from $\mathsf{Pred}(p)$ by a call to $\mathsf{DFS}(p)$, α is increased by 1 (line 2) and its value sets $\mathsf{Dn}(p)$ (line 3). Then all the neighbors of vertex p are visited (lines 4-8) with the only exception of $\mathsf{Pred}(p)$ (line 5). Before calling $\mathsf{DFS}(k)$ for a generic neighbor $k \in \delta(p)$, the algorithm sets $\mathsf{Pred}(k)$ at the current vertex p (line 6) and $\mathsf{Depth}(k)$ at the value $\mathsf{Depth}(p) + 1$ (line 7). When the neighborhood of p has been completely explored, α is increased again (line 9) and it sets $\mathsf{Up}(p)$ (line 10). Finally the search backtracks.

Complexity of Orient.

The computation of the star of each vertex (lines 2 to 6) takes $O(n)$, because T^* contains $n-1$ edges. All the operations on lines 7-10 of Orient can be done in $O(1)$. The time complexity taken by all instructions on lines 2, 3, 9 and 10 of DFS is $O(n)$, because α ranges from 1 to $2n$ by Property 6. The total number of iterations of the loop on lines 4-8 of DFS is twice the number of edges of T^*, i.e. $2(n-1)$, and the body of the loop (lines 6 and 7) is executed in $O(1)$. Therefore the time complexity for visiting T^* with DFS is $O(n)$.

3.2 Local subgraphs

▶ **Definition 10.** For each vertex $p \in V$ a *local subgraph* $\mathcal{G}(p) = (\mathcal{V}(p), \mathcal{T}(p))$ is defined: it has $|\mathcal{V}(p)| = d(p)$ vertices, where $d(p)$ is the degree of p in T^*.

The vertices in $\mathcal{V}(p)$ are in one-to-one correspondence with the neighbors of p in T^* and therefore to the p-components they belong to. Hence, with a little abuse of notation in the remainder the same indices are used to indicate the neighbors of vertex $p \in V$ and the corresponding vertices in the local subgraph $\mathcal{V}(p)$.

Edges in local subgraphs are called *links* in the remainder, in order to make the difference with the edges of graph $\mathcal{G}$.

The link set $\mathcal{T}(p)$ is a forest for each $p \in V$; it is initially empty; at the end of the execution, it is a spanning tree of $\mathcal{G}(p)$. Each time the algorithm detects an alternative edge $[i, j]$ to be used when vertex p is deleted, this piece of information is recorded in $\mathcal{T}(p)$ (this happens in Algorithm 9 and Algorithm 10 described in Section 4). Each link in $\mathcal{T}(p)$ is represented as a record with two fields: one is the link itself between two vertices of $\mathcal{G}(p)$, and the other is the alternative edge $[i, j]$ reconnecting the two corresponding p-components.

An example is shown in Figure 4. When vertex p is deleted from the spanning tree, edges $[i, j]$ and $[k, l]$ are used to reconnect the three resulting p-components at minimum cost. Two corresponding links $[a, b]$ and $[b, c]$ are inserted in the local subgraph $\mathcal{G}(p)$, forming a spanning tree $\mathcal{T}(p)$. Each link in $\mathcal{T}(p)$ has an associated alternative edge in G.

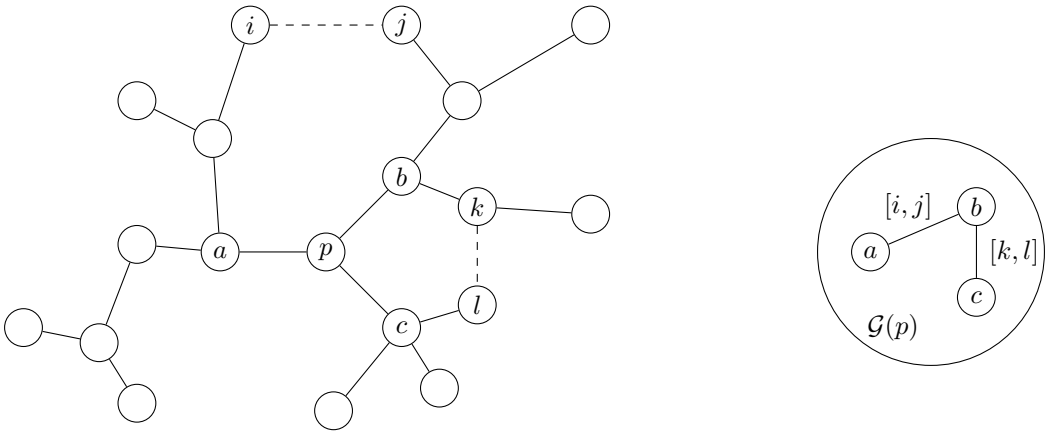


Figure 4 Left: the spanning tree T^* and two alternative edges (dashed). Right: two corresponding links $[a, b]$ and $[b, c]$ in the local subgraph $\mathcal{G}(p)$.

▶ **Remark 11.** The local subgraphs of the leaves of T^* have no links, since they contain a single vertex. Hence, they are not used in the algorithm.

▶ **Definition 12.** Links in $\mathcal{T}(p)$ incident to the local vertex corresponding to $\mathsf{Pred}(p)$ are *vertical links*; links in $\mathcal{T}(p)$ not incident to the local vertex corresponding to $\mathsf{Pred}(p)$ are *horizontal links*.

▶ **Remark 13.** The local forest $\mathcal{T}(r)$ cannot include vertical links, since r has no predecessor.

Figure 5 shows an example of vertical and horizontal links corresponding to a same alternative edge. When the candidate alternative edge $[i, j]$ is considered, a vertical link between the local vertices i and a is inserted in $\mathcal{G}(k)$, because k is a vertex between i and the apex; a horizontal link between the local vertices k and j is inserted in $\mathcal{G}(a)$, because a is the apex.

3.2.1 Local Union-Find data-structure

Since a spanning tree $\mathcal{T}(p)$ must be eventually computed for each local subgraph $\mathcal{G}(p)$, a Union-Find data-structure is kept for each vertex $p \in V$. It is made by an array of $d(p)$ linked lists. Each list $\mathcal{L}(p, k)$ is initialized with a single element $k \in \mathcal{V}(p)$, corresponding to a neighbor k of p in T^*. For each list two additional scalar values are also recorded: $\mathsf{Card}(p, k)$ records the cardinality of $\mathcal{L}(p, k)$; $\mathsf{Head}(p, k)$ records the head of the list to which the local vertex k belongs.

The operations executed on the Union-Find data-structure of vertex $p \in V$ when $\mathcal{T}(p)$ is enriched with a new link are the same as those reported in many textbooks on algorithms and data-structures (see for instance [4]). For readers unfamiliar with them, they are recalled in Appendix A, where the pseudo-code of the procedure Merge is also described. An example is shown in Figure 6.

3.2.2 Procedure Initialization

The procedure Initialization, shown in Algorithm 3, initializes the local data-structure of each vertex $p \in V$.

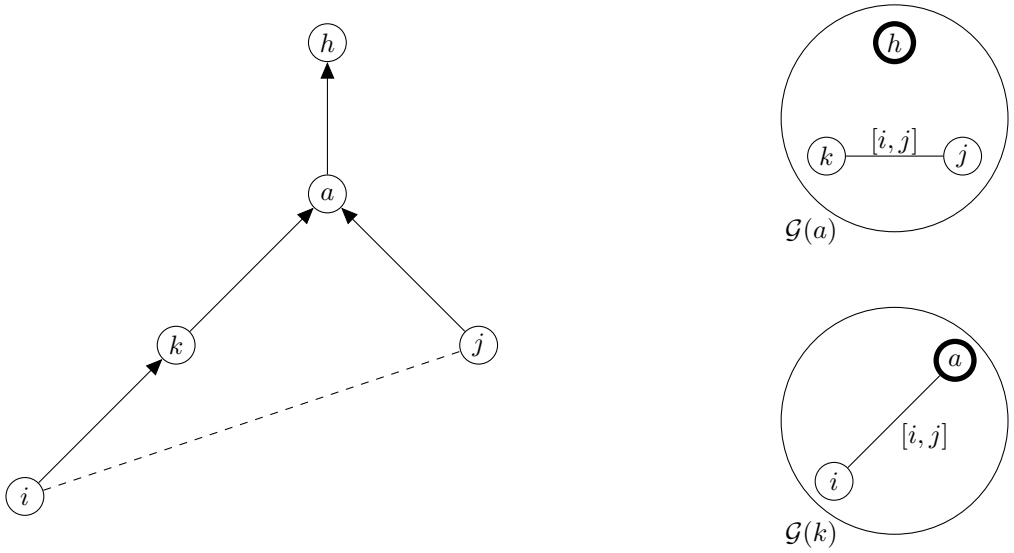
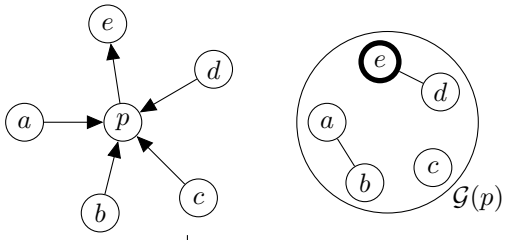


Figure 5 An example of links added to local subgraphs when a candidate alternative edge is considered. In each local subgraph, the vertex corresponding to the predecessor is drawn in thick line.



Vertex	Head	Card	$\mathcal{L}$
a	a	2	$\{a, b\}$
b	a	0	$\{\}$
c	c	1	$\{c\}$
d	d	2	$\{d, e\}$
e	d	0	$\{\}$

Figure 6 A vertex p with five neighbors (left), its local subgraph $\mathcal{G}(p)$ (center) and the corresponding state of the Union-Find data-structure (right). Oriented arcs indicate the predecessors in T^*. In $\mathcal{G}(p)$, the vertex corresponding to $e = \mathsf{Pred}(p)$ is indicated by a thick line. Local vertices a and b have been connected with a horizontal link; consequently, the lists of a and b have been merged in a single list, so that $\mathcal{L}(a) = \{a, b\}$, $\mathcal{L}(b) = \emptyset$, $\mathsf{Card}(a) = 2$, $\mathsf{Card}(b) = 0$ and $\mathsf{Head}(a) = \mathsf{Head}(b) = a$. The local vertex d has been connected to $e = \mathsf{Pred}(p)$ with a vertical link; consequently, the lists of d and e have been merged in a single list, so that $\mathcal{L}(d) = \{d, e\}$, $\mathcal{L}(e) = \emptyset$, $\mathsf{Card}(d) = 2$, $\mathsf{Card}(e) = 0$ and $\mathsf{Head}(d) = \mathsf{Head}(e) = d$.

The local vertex sets $\mathcal{V}(p)$ for all $p \in V$, initially empty (line 3), are populated in lines 5-6. The corresponding Union-Find data-structure is initialized in lines 7-12. The local forests $\mathcal{T}(p)$ are initially empty (line 14).

The value $\mathsf{AltTreeCost}(p)$ indicates the total cost of the alternative edges that have been selected to reconnect the p-components for each vertex $p \in V$. It is initialized at 0 on line 15.

Each spanning tree $\mathcal{T}(p)$ in the local graph of vertex p requires $d(p) - 1$ edges. Since $\sum_{p \in V} d(p) = 2(n-1)$, then the number of (not necessarily distinct) alternative edges that must be selected is $2(n-1) - n$, i.e. $n - 2$. This is the initial value of the variable μ, that counts the number of missing alternative edges (line 16). The search for alternative edges terminates as soon as $\mu = 0$.

Algorithm 3 The procedure that initializes the local subgraphs.

```
1:  procedure Initialization
2:      for p ∈ V do
3:          V(p) ← ∅
4:      for [i, j] ∈ T* do
5:          V(i) ← V(i) ∪ {j}
6:          V(j) ← V(j) ∪ {i}
7:          L(i, j) ← {j}
8:          Card(i, j) ← 1
9:          Head(i, j) ← j
10:         L(j, i) ← {i}
11:         Card(j, i) ← 1
12:         Head(j, i) ← i
13:     for p ∈ V do
14:         T(p) ← ∅
15:         AltTreeCost(p) ← 0
16:     μ ← n − 2
```

Complexity of Initialization**.**

The procedure Initialization contains three loops: the first and the third loop are repeated for each vertex in V, i.e. n times; the second loop is repeated for each edge of T^*, i.e. $n-1$ times. All operations within the loops require $O(1)$. Therefore Initialization takes $O(n)$.

3.3 Oriented paths and trees

Consider a generic edge $[i,j] \notin T^*$ and the corresponding cycle $C(i,j)$, whose apex is indicated as $\mathsf{apex}(i,j)$. Two oriented paths are defined as follows.

▶ **Definition 14.** The *oriented path* $P(i,j)$ goes from vertex i to $\mathsf{apex}(i,j)$ in T^*; the *oriented path* $P(j,i)$ goes from vertex j to $\mathsf{apex}(i,j)$ in T^*.

It is possible that one of the two paths does not exist, when i or j is the apex of $C(i,j)$, as illustrated in Figure 7.

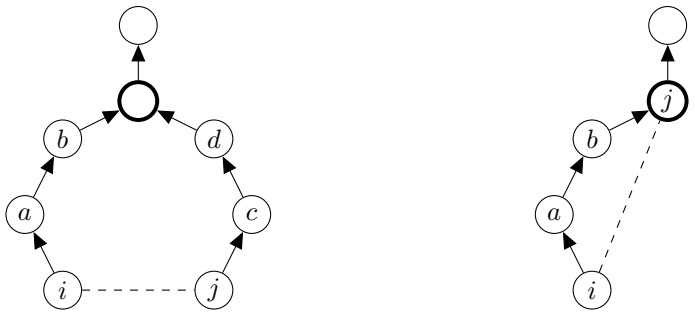


Figure 7 Left: edge $[i,j]$ corresponds to two oriented paths, one including a and b, the other including c and d. Right: edge $[i,j]$ corresponds to a single path, because one of its endpoints is the apex of $C(i,j)$. In both cases, the apex is indicated by a thick line. Oriented arcs indicate predecessors in T^*.

▶ **Definition 15.** The *internal vertices* of a non-empty oriented path $P(i,j)$ (or $P(j,i)$) are the vertices along it excluding the endpoints, i.e. vertex i (or j) and $\mathsf{apex}(i,j)$.

As observed in Section 2, $\mathsf{apex}(i,j)$ and the internal vertices of $P(i,j)$ and $P(j,i)$ are those for which edge $[i,j]$ can be used as an alternative edge. In the local subgraph of $\mathsf{apex}(i,j)$ edge $[i,j]$ corresponds to a horizontal link, while in the local subgraphs of the internal vertices along the two paths edge $[i,j]$ corresponds to a vertical link, as shown in Figure 5. For this reason, the apex is processed separately from the internal vertices along the two paths.

▶ **Definition 16.** The *root* of a non-empty oriented path $P(i,j)$ or $P(j,i)$ is its vertex t with $\mathsf{Depth}(t) = \mathsf{Depth}(\mathsf{apex}(i,j)) + 1$.

For each internal vertex p of an oriented path $P(i,j)$ edge $[i,j]$ reconnects two p-components, one of them containing i and the other one containing $\mathsf{Pred}(p)$ (and the same holds for $P(j,i)$ swapping i with j). For each candidate alternative edge $[i,j]$ the algorithm scans $P(i,j)$ from i and $P(j,i)$ from j up to their roots. Therefore all local forests of internal vertices of the two paths are considered to possibly insert a vertical link in each of them.

▶ **Definition 17.** Two oriented paths P' and P'' *overlap* if and only if they have at least an internal vertex in common.

When two (or more) oriented paths overlap, the algorithm merges them to form an *oriented tree*. The oriented trees made by the oriented paths have the following properties.

▶ **Property 18.**

i. Each vertex belongs to at most one oriented tree;
ii. each oriented tree has a unique *root*, that is the minimum depth root of its paths;
iii. there is at least one vertical link in the local forest of each vertex in an oriented tree.

Owing to this property, every time the algorithm scans a path and it detects that part of it overlaps with an existing oriented tree, the oriented tree is skipped in $O(1)$ and the scan resumes directly from its root.

For this purpose, for each vertex $p \in V$ a variable $\mathsf{Path}(p)$ records the index of the first path that introduces a vertical link in $\mathcal{G}(p)$.

▶ **Definition 19.** A path is *relevant* if and only if it inserts a vertical link in at least one subgraph.

▶ **Property 20.** Since the number of vertical links is bounded above by $n-2$ (the limit case occurs when T^* is a path), this is also the maximum number of relevant paths.

Every time a new path is scanned, a path counter π is increased by 1 and the path root is initialized at 0. Every time a vertical link is inserted in some local forest $\mathcal{T}(p)$, the path root is updated to p. At the end of the scan, if the path root is still equal to 0, then the path is discarded as non-relevant and the path counter π is decreased by 1. No update to any data-structure is done, while the currently scanned path has not yet been recognized as relevant.

3.4 The Tree-Union-Find data-structure

A suitable data-structure is used to allow for efficiently merging relevant oriented paths into oriented trees. This data-structure is indicated as Tree-Union-Find, to distinguish it from the Union-Find data-structures associated with local subgraphs.

The Tree-Union-Find data-structure includes an array TList of linked lists, an array TCard, an array THead and an array TRoot. Each component of such arrays corresponds to an oriented tree, i.e. a set of overlapping relevant oriented paths: $\mathsf{TList}(\tau)$ is the set of paths merged into the tree τ; $\mathsf{TCard}(\tau)$ is their number; $\mathsf{THead}(\tau)$ is the index of a representative path among those in $\mathsf{TList}(\tau)$; $\mathsf{TRoot}(\tau)$ is the root of tree τ, i.e. the minimum depth vertex among the roots of the paths in $\mathsf{TList}(\tau)$.

All arrays in the Tree-Union-Find data-structure are made by at most $n-2$ elements, because no more than $n-2$ relevant paths can exist, owing to Property 20.

Algorithm 4 shows the initialization of a new path. Algorithm 5 shows how non-relevant paths are detected and purged. Algorithm 6 illustrates the procedure that merges the subtrees containing two paths π' and π'': $\mathsf{TreeMerge}$ assigns index τ' to the tree with maximum cardinality and τ'' to the other one, according to the values of TCard (lines 2-7). The root of the resulting tree is selected as the one with minimum depth among the root of τ' and the root of τ'' (lines 8-9). Then tree τ'' is appended to tree τ', so that $\mathsf{THead}(\pi) \leftarrow \tau'$ for all paths π in $\mathsf{TList}(\tau'')$ (lines 10-11). Finally TCard and TList are updated, appending the shortest list to the longest one (lines 12-14).

4 The algorithm

After the introduction of the necessary definitions and properties and the description of the main data-structures, it is now possible to introduce the algorithm.

Algorithm 4 The algorithm that initializes a new oriented path.

```
1: procedure InitPath
2:     π ← π + 1
3:     TList(π) ← {π}
4:     TCard(π) ← 1
5:     THead(π) ← π
6:     TRoot(π) ← 0
```

Algorithm 5 The algorithm that deletes non-relevant paths.

```
1: procedure PurgePath(π)
2:     if TRoot(THead(π)) = 0 then
3:         π ← π − 1
4:         TList(π) ← ∅
```

Algorithm 6 The algorithm that merges two subtrees containing paths π' and π''.

```
1: procedure TreeMerge(π′, π″)
2:     if TCard(THead(π′)) > TCard(THead(π″)) then
3:         τ′ ← THead(π′)
4:         τ″ ← THead(π″)
5:     else
6:         τ′ ← THead(π″)
7:         τ″ ← THead(π′)
8:     if (Depth(TRoot(τ″)) < Depth(TRoot(τ′))) then
9:         TRoot(τ′) ← TRoot(τ″)
10:    for π ∈ TList(τ″) do
11:        THead(π) ← τ′
12:    TCard(τ′) ← TCard(τ′) + TCard(τ″)
13:    TCard(τ″) ← 0
14:    TList(τ′)) ← TList(τ′)) ∪ TList(τ″))
```

The main body of the algorithm is shown in Algorithm 7. The complexity of each step is indicated on the corresponding line and it is proven in the remainder. The algorithm takes in input a graph $G = (V, E)$, an edge weighting function c, a minimum spanning tree T^* and a sorted list $\mathcal{S}$ of all edges $e \in E$. It returns the optimal vertex p^* and the corresponding 1-tree T' providing the largest Held–Karp lower bound LB^*_{HK}.

Algorithm 7 The main algorithm.

```
1: procedure Main(G, c, T*, S)
2:     // Find all alternative edges //
3:     Orient                                              ▷ O(n)
4:     Initialization                                      ▷ O(n)
5:     Search                                              ▷ O(m + n log n)
6:     // Compute the maximum Held–Karp lower bound LB*_HK //
7:     SelectVertex                                        ▷ O(m)
8:     Return(p*, T′, LB*_HK)
```

Procedures Orient and Initialization have been already described in Section 3. Procedure SelectVertex is described in the next Section; it is used to optimize the Held–Karp lower bound.

This Section describes in detail the procedure Search, that computes n alternative minimum spanning trees.

The Search procedure, shown in Algorithm 8, is made by a main loop (lines 3-18) whose body is divided into three parts. At each iteration of the main loop a candidate alternative edge is considered and all the corresponding vertical and horizontal links are inserted. The loop is repeated until all necessary links have been inserted; this is detected from the test on the number of missing edges in the local subgraphs: $\mu = 0$ (line 18).

Algorithm 8 The procedure that detects all alternative edges.

```
1:  procedure Search
2:      π ← 0
3:      repeat
4:          /* Edge selection */
5:          repeat
6:              [i, j] ← Extract(S)
7:          until [i, j] ∉ T*
8:          /* Vertical links */
9:          u ← i
10:         if ¬SubTree(i, i, j) then
11:             PathScan(i, j, u)
12:         v ← j
13:         if ¬SubTree(j, i, j) then
14:             PathScan(j, i, v)
15:         /* Horizontal links */
16:         if ¬SubTree(i, i, j) ∧ ¬SubTree(j, i, j) ∧ then(Stop(u) ∨ Stop(v))
17:             ProcessApex(u, v, i, j)
18:     until μ = 0
```

The first part of the body of the main loop (lines 4-7) is a secondary loop in which the next candidate alternative edge (i.e. an edge not in T^*) is extracted from the sorted edge list $\mathcal{S}$. The candidate alternative edge is indicated by $[i, j]$.

In the second part (lines 8-14) Search scans the paths $P(i, j)$ and $P(j, i)$ to possibly insert vertical links. Two indices u and v are used to identify the current vertex on the paths: u along $P(i, j)$ and v along $P(j, i)$. The operations on the two paths are symmetrical and they are grouped in the procedure PathScan that is called twice (lines 11 and 14). PathScan is called only if the corresponding path exists; this is the reason for tests on lines 10 and 13. SubTree(p, i, j) is a boolean function that tests Property 8 (ii), to check whether an endpoint of edge $[i, j]$ is the apex of $C(i, j)$ or not. In this way empty paths are disregarded.

The effect of this second part is twofold: first, to insert all possible vertical links in local forests; second, to indicate how the search along each path terminates. This is represented by the value of the variables Stop(u) and Stop(v). Consider the side of vertex i. If Stop(u) is true, the current vertex u is within the cycle $C(i, j)$; hence, when the loop is over, Pred(u) is apex(i, j). If Stop(u) is false, the path on the side of i has been merged with a pre-existing oriented tree, whose root is at apex(i, j) or above (i.e. closer to the root) and therefore Pred(u) is out of $C(i, j)$. The same holds symmetrically for Stop(v) on the side of j.

In the third part (lines 15-17) a horizontal link is possibly inserted in the local forest of apex(i, j) by a call to procedure ProcessApex (line 17). However, this is done only if the apex is different from i and j (i.e. both oriented paths exist) and if at least one of the two current vertices u and v is within the cycle $C(i, j)$. If both u and v have reached the apex, then both the apex(i, j)-components of i and j are already connected with that of Pred(apex(i, j)) and therefore no horizontal link must be inserted in $\mathcal{T}$(apex(i, j)).

4.1 Procedure PathScan and vertical links

For each side of the cycle $C(i, j)$ the current vertex is initialized at an endpoint of the candidate alternative edge $[i, j]$ (lines 9 and 12). The variable Stop is initialized at true (line 2) and a new path is initialized by a call to InitPath (line 3) so that its index is π.

The current vertex, indicated by w, corresponds either to u or to v, depending on the call on line 11 or 14 of Search. The test on line 4 detects whether the predecessor of the current vertex w is at a larger depth then apex(i, j). As soon as Pred(w) reaches the apex or above, the loop ends, because vertical links must be inserted only in internal vertices of the oriented path.

Within the loop (lines 4-22) at each iteration an attempt is made to insert a vertical link between w and Pred(p) in $\mathcal{G}(p)$, where $p =$ Pred(w) (line 5). For this purpose the local Union-Find data-structure of vertex p is tested to check whether w and Pred(p) are already connected or not (line 6).

Algorithm 9 The procedure that inserts vertical links.

```
1:  procedure PathScan(i, j, w)
2:      Stop(w) ← true
3:      InitPath
4:      while ¬SubTree(Pred(w), i, j) do
5:          p ← Pred(w)
6:          if (Head(p, w)) ≠ Head(p, Pred(p)) then
7:              /* Insert a vertical link */
8:              T(p) ← T(p) ∪ {([w, Pred(p)], [i, j])}
9:              AltTreeCost(p) ← AltTreeCost(p) + c(i, j)
10:             μ ← μ − 1
11:             Merge(p, w, Pred(p))
12:             TRoot(THead(π)) ← p
13:             w ← p
14:         if Path(p) = 0 then
15:             Path(p) ← π
16:         else
17:             /* Skip Path(p) up to its root */
18:             if TRoot(THead(π)) ≠ 0 then
19:                 TreeMerge(π, Path(p))
20:             w ← TRoot(THead(Path(p)))
21:             if SubTree(w, i, j)) then
22:                 Stop(w) ← false
23:     PurgePath(π)
```

If the test succeeds (i.e. w and $\mathsf{Pred}(p)$ are not connected), a vertical link is inserted and several update operations are done (lines 7-13). In particular, a new element is inserted in $\mathcal{T}(p)$ as a record with two fields: the link is $[w, \mathsf{Pred}(p)]$, while the alternative edge is $[i, j]$ (line 8); the total cost of the alternative edges $\mathsf{AltTreeCost}(p)$ is increased by $c(i, j)$ (line 9); the number of missing alternative edges is decreased by 1 (line 10). Then the local Union-Find data-structure of vertex p is updated, merging the connected components of w and $\mathsf{Pred}(p)$ (line 11) with a call to Merge (see Appendix A for the pseudo-code). When a vertical link is inserted, the path is relevant and its root is updated to vertex p where the vertical link has been inserted (line 12). Finally the current vertex w is moved one step forward along the path to $p = \mathsf{Pred}(w)$ (line 13).

That a vertical link has been inserted or not, two situations can occur: either no vertical link has been inserted in $\mathcal{T}(p)$ in previous iterations (and in this case $\mathsf{Path}(p) = 0$) or a vertical link has already been inserted while scanning a path (and in this case the index of that path has been recorded in $\mathsf{Path}(p)$). In the former case, detected by the test on line 14, $\mathsf{Path}(p)$ is set to the index π of the current path (line 15) and no other operation is required. In the latter case, the current vertex w is moved directly to the root of the oriented subtree containing the path $\mathsf{Path}(p)$ and the subtree containing the current path π is merged with the subtree containing path $\mathsf{Path}(p)$. The TreeMerge procedure is called (line 19) only if π has already been detected as relevant (line 18). The current vertex is updated on line 20. Consequently, a new test is done to check whether w is still below the apex of $C(i, j)$ (line 21). If not, Stop is set to false (line 22).

When the loop is over, π is checked again: if it is non-relevant then it is purged (line 23).

4.2 Procedure ProcessApex and horizontal links

The procedure $\mathsf{ProcessApex}(u, v, i, j)$, shown in Algorithm 10, inserts a horizontal link in the local forest of $\mathsf{apex}(i, j)$ if and only if the $\mathsf{apex}(i, j)$-components of i and j are not already connected in $\mathcal{T}(\mathsf{apex}(i, j))$. If they are already connected, the procedure has no effect.

To check whether the horizontal link can be inserted, it is necessary to know the indices of the two vertices adjacent to the apex along $P(i, j)$ and $P(j, i)$. These correspond to u and v when $\mathsf{Stop}(u)$ and $\mathsf{Stop}(v)$ are true: if $\mathsf{Stop}(u) \wedge \mathsf{Stop}(v)$, then $\mathsf{Pred}(u) = \mathsf{Pred}(v) = \mathsf{apex}(i, j)$. However, this is not the case when Stop is false. Hence, the apex, indicated by p in Algorithm 10, is found as the predecessor of the current vertex for which Stop is true (lines 2-5). At least one of $\mathsf{Stop}(u)$ and $\mathsf{Stop}(v)$ is guaranteed to be true, owing to the test on line 16 of Search.

If one of the two current vertices, say u, has been moved to the apex or above, i.e. $\mathsf{Stop}(u)$ is false, then the p-component of i is already connected with the p-component of $\mathsf{Pred}(p)$ in $\mathcal{T}(p)$. Hence a test on the local Union-Find data-structure of vertex p must be done to check whether v and $\mathsf{Pred}(p)$ are connected or not. This is the reason for setting a variable u' to u if $\mathsf{Stop}(u)$ is true and to $\mathsf{Pred}(p)$ if $\mathsf{Stop}(u)$ is false (lines 6-9). The same is done for v' (lines 10-13).

If the test on the Union-Find data-structure succeeds (line 14), then a horizontal link is inserted; otherwise the procedure terminates with no effect.

Before inserting the horizontal link, it is necessary to find the index of both vertices adjacent to the apex p along $P(i,j)$ and $P(j,i)$. They are not both available if Stop is false for one of the two current vertices. Therefore, u or v is reset to the position just below the apex, in case it is not (lines 15-16 and 17-18). This is done by a procedure Find, that exploits the values of Up and Dn of all vertices adjacent to p. W.l.o.g. assume $\mathsf{Stop}(u)$ be false. The execution of $\mathsf{Find}(p,i)$ implies a search among the vertices of $\mathcal{V}(p)$ to find the vertex which lies on the path between i and p. This requires to find the (unique) vertex $u \in \mathcal{V}(p)$ such that $(\mathsf{Dn}(u) \leq \mathsf{Dn}(i)) \wedge (\mathsf{Up}(u) \geq \mathsf{Up}(i))$.

Once u and v have been set to the neighbors of the apex p along $P(i,j)$ and $P(j,i)$, the horizontal link between them is finally inserted in $\mathcal{T}(p)$. The two fields of the new element in $\mathcal{T}(p)$ record the link $[u,v]$ and the alternative edge $[i,j]$ (line 19); the cost of $\mathcal{T}(p)$ is increased by $c(i,j)$ (line 20); the number μ of missing edges is decreased by 1 (line 21). Finally the Union-Find data-structure of the local subgraph $\mathcal{G}(p)$ is updated by a call to Merge (line 22) to connect the components of u and v (see Appendix A for the pseudo-code).

Algorithm 10 The procedure that inserts horizontal links.

```
1:  procedure ProcessApex(u, v, i, j)
2:      if (Stop(u)) then
3:          p ← Pred(u)
4:      else
5:          p ← Pred(v)
6:      if (Stop(u)) then
7:          u' ← u
8:      else
9:          u' ← Pred(p)
10:     if (Stop(v)) then
11:         v' ← v
12:     else
13:         v' ← Pred(p)
14:     if (Head(p, u') ≠ Head(p, v')) then
15:         if ¬Stop(u) then
16:             u ← Find(p, i)
17:         if ¬Stop(v) then
18:             v ← Find(p, j)
19:         T(p) ← T(p) ∪ {[u, v], [i, j]}
20:         AltTreeCost(p) ← AltTreeCost(p) + c(i, j)
21:         μ ← μ − 1
22:         Merge(p, u, v)
```

Complexity of Search**.**

The number of iterations of the outer loop of Search (lines 3-18) is $O(m)$; therefore the time taken by all constant time operations in Search (including $\mathsf{SubTree}$ that takes constant time), $\mathsf{PathScan}$ and $\mathsf{ProcessApex}$ (including $\mathsf{InitPath}$ and $\mathsf{PurgePath}$), yield an overall $O(m)$ contribution to the time complexity.

The loop on lines 5-7 can be implemented so that it takes $O(1)$ time per iteration, i.e. $O(m)$ overall. This requires to sort the edges of T^* with the same criterion used to sort the edges in $\mathcal{S}$ (non-decreasing cost plus some additional lexicographic criterion to break ties). This pre-sorting step requires $O(n \log n)$ because T^* contains $n-1$ edges. In this way the test on line 7 is a comparison between the edge $[i,j]$ extracted from $\mathcal{S}$ and the first

edge of T^* not yet extracted from $\mathcal{S}$ and it takes constant time. Hence, the pre-sorting step (not indicated in the pseudo-code) and the loop on lines 5-7 require $O(m + n \log n)$ time.

The block of operations on lines 7-13 of PathScan is executed $O(n)$ times, because a vertical link is inserted each time and the number of possible vertical links is $O(n)$. Since all operations excepted Merge have $O(1)$ complexity, their overall contribution to the time complexity of the algorithm is $O(n)$.

The test on line 14 of PathScan can succeed at most n times. Therefore the total contribution to the time complexity of the $O(1)$ operation on line 15 is $O(n)$.

The block of operations on lines 17-22 of PathScan takes $O(1)$ with the exception of the contribution of TreeMerge. The complexity analysis requires to distinguish the case in which π is non-relevant ($\mathsf{TRoot}(\mathsf{THead}(\pi)) = 0$) from the case in which π is relevant ($\mathsf{TRoot}(\mathsf{THead}(\pi)) > 0$).

An empty path can exist only twice for each candidate alternative edge, once for each side of $C(i, j)$. Therefore the execution of this block with non-relevant π can occur $O(m)$ times. In these cases TreeMerge is not executed and thus the total contribution is $O(m)$.

The number of relevant paths in the Tree-Union-Find data-structure is $O(n)$, because all roots are different. Therefore, a call to TreeMerge can occur at most $O(n)$ times. Hence, the number of times the block is executed with relevant π is $O(n)$ and therefore the total contribution of these iterations is $O(n)$ plus the contribution of TreeMerge.

The number of times the loop on lines 4-22 of PathScan is executed is the sum of both types of iterations, i.e. with relevant π and non-relevant π. Therefore the tests on lines 6 and 14 are executed $O(m + n)$ times.

The procedure ProcessApex is called at most once for each edge, i.e. $O(m)$ times. Therefore all $O(1)$ operations on lines 2-13 contribute $O(m)$.

The test on line 14 can succeed $O(n)$ times, because the number of horizontal links that can be inserted is $O(n)$. Excluding the contribution of Merge and Find, the total contribution of the operations on lines 15-22 to the time complexity is $O(n)$.

The execution of $\mathsf{Find}(p, i)$ implies a search among the vertices of $\mathcal{V}(p)$ (i.e. the neighbors of p in T^*) to find the vertex which lies on the path between p and i on T^*, i.e. the (unique) vertex u satisfying $(\mathsf{Dn}(u) \leq \mathsf{Dn}(i)) \wedge (\mathsf{Up}(u) \geq \mathsf{Up}(i))$. This task can be accomplished in $O(\log d(p))$ (which is not worse than $O(\log n)$) by binary search, because the vertices of $\mathcal{V}(p)$ are sorted by their values of Dn and Up after the execution of Orient. Since the number of calls to Find is $O(n)$, as observed above, the overall contribution of Find to the total time complexity is $O(n \log n)$.

At each call of Merge two lists in the local subgraph $\mathcal{G}(p)$ of some vertex $p \in V$ are merged. This can happen $O(n)$ times overall, because $O(n)$ alternative edges must be found. For the well-known property of the Union-Find data-structure (recalled in Appendix A) for each local graph with $d(p)$ vertices, the time taken by the update operations is $O(d(p) \log d(p))$. Summing up these contributions over all vertices results in an $O(n \log n)$ contribution to the complexity of the whole algorithm, because $\sum_{p \in V} d(p) \log d(p) \leq \sum_{p \in V} d(p) \log n = \log n \sum_{p \in V} d(p) = 2|T^*| \log n = 2(n-1) \log n$.

The total time taken by TreeMerge to merge trees is $O(n \log n)$, because of the properties of Union-Find: every time two or more trees are merged, their lists are merged so that the shortest one is appended to the longest one. This guarantees that no representative is updated more than $\log n$ times since the number of oriented trees is $O(n)$. Note that to achieve this result, it is necessary to accept that the root of an oriented tree does not necessarily belong to its representative path.

The conclusion of this analysis is that the worst-case time complexity of Search is $O(m + n \log n)$ which is the same of a single run of Kruskal algorithm or Prim algorithm implemented with Fibonacci heaps.

5 Computation of the optimal Held–Karp lower bound LB^*_{HK}

If the computation of the n alternative minimum spanning trees is aimed at the optimization of the Held–Karp lower bound $LB_{HK}(\bar{p})$, the procedure SelectVertex is executed, as shown in Algorithm 7, to compute the most profitable vertex $\bar{p}$, the resulting 1-tree and the corresponding value LB^*_{HK}.

The procedure SelectVertex, illustrated in Algorithm 11, finds the two minimum cost edges $e_1(p)$ and $e_2(p)$ incident in each vertex p (their costs are indicated by c_1 and c_2) and evaluates what the resulting 1-tree lower bound would be if p were removed from T^*, replacing its star with the alternative edges recorded in $\mathcal{T}(p)$ plus the two edges $e_1(p)$ and $e_2(p)$. The vertex p^* providing the largest value is returned, together with the corresponding

value z, indicating the best lower bound that can be achieved, LB^*_{HK}. The value $\mathsf{StarCost}(p)$ indicates the total cost of the edges of T^* incident to each vertex $p \in V$.

In the last two lines of Algorithm 11 the optimal Held–Karp 1-tree T' is produced. Each element in $\mathcal{T}(p)$ is stored as a record with two fields, one indicating the link in the local forest and the other indicating the corresponding alternative edge. In the pseudo-code shown in Algorithm 11 the notation $\widehat{\mathcal{T}}(p)$ indicates the set of alternative edges corresponding to the elements of $\mathcal{T}(p)$.

Complexity of SelectVertex.

The computation of $\mathsf{StarCost}(p)$ for each vertex requires to scan each edge of the graph twice. Hence its complexity is $O(m)$. The operation of deleting from T^* the edges incident to p^* (line 23) takes $O(n)$. The operation of merging the two lists of edges T^* and $\widehat{\mathcal{T}}(p)$ (line 23) takes $O(n)$. Therefore the overall complexity of SelectVertex is $O(m)$.

Algorithm 11 The procedure that computes the optimal Held–Karp lower bound.

```
 1: procedure SelectVertex
 2:     z ← −∞
 3:     for p ∈ V do
 4:         StarCost(p) ← 0
 5:         c1 ← ∞
 6:         c2 ← ∞
 7:         for k ∈ δ(p) do
 8:             StarCost(p) ← StarCost(p) + c(p, k)
 9:             if c(p, k) < c1 then
10:                 c2 ← c1
11:                 c1 ← c(p, k)
12:                 e2 ← e1
13:                 e1 ← [p, k]
14:             else
15:                 if c(p, k) < c2 then
16:                     c2 ← c(p, k)
17:                     e2 ← [p, k]
18:         if AltTreeCost(p) − StarCost(p) + c1 + c2 > z then
19:             z ← AltTreeCost(p) − StarCost(p) + c1 + c2
20:             p* ← p
21:             e1* ← e1
22:             e2* ← e2
23:     T' ← T*\δ(p*) ∪ T̂(p*) ∪ {e1*, e2*}
24:     Return(p*, T', z)
```

6 Conclusions

The value of the 1-tree-based lower bound for the TSP can be optimized, not only following Helsgaun's idea of optimally selecting a leaf of a minimum spanning tree and an additional minimum cost edge incident to it, but also by improving upon the original idea by Held and Karp of selecting a vertex $\bar{p}$ excluded from a minimum spanning tree and two additional minimum cost edges incident to it. The algorithm presented here shows that the vertex $\bar{p}$ that Held and Karp selected in an arbitrary way can be selected in an optimal way with the same worst-case time complexity required to compute a minimum spanning tree on the same weighted graph, i.e. $O(m + n \log n)$; this is also the time complexity required by each iteration of the Held–Karp method when the vertex $\bar{p}$ is selected in an arbitrary way. The resulting lower bound is guaranteed to be larger than or equal to the lower bound provided by Helsgaun's method.

The advantage of LB^*_{HK} with respect to LB^*_H obviously depends on the specific instance. Just to have a rough estimate of the expected improvement that LB^*_{HK} can provide, some limited computational tests were

also carried out. Two data-sets were generated at random. Data-set A is made of 100 random graphs with 100 vertices and density 15%; edge weights were generated with uniform probability distribution in the range $[0, 1000]$. Data-set B is made of 100 Euclidean instances; each of them is defined by 100 points placed at random in a square of size 100, with uniform probability distribution for both coordinates; graphs are complete and edge weights correspond to Euclidean distances.

Results are reported in Table 1. Column $\overline{LB}_{HK}$ indicates the average value of LB_{HK}, i.e. the expected value of the lower bound when the vertex $\bar{p}$ is chosen at random as suggested by Held and Karp. Column LB^*_H reports the optimized value of the 1-tree lower bound with Helsgaun method. Column LB^*_{HK} reports the optimized value of the Held–Karp lower bound. The rightmost columns reports the number of times LB^*_{HK} was found to be tighter than LB^*_H. All values have been rounded to the closest integer. The experimental comparison shows that not only LB^*_{HK} dominates LB^*_H but it is strictly better for more than half of the instances. The improvement of LB^*_{HK} over LB^*_H is instance-dependent: in our tests it is zero or negligible for some instances, significant for others. The improvement of LB^*_{HK} over $\overline{LB}_{HK}$ is always relevant (about $10 - 20\%$). Computing time was negligible and therefore it is not reported in Table 1.

Data-set	$\overline{LB}_{HK}$	LB^*_H	LB^*_{HK}	n. impr.
A	8990	11686	11759	55/100
B	530	595	597	63/100

Table 1 Comparison between optimized and non-optimized 1-tree-based lower bounds.

It should be remarked that the algorithm illustrated in this paper computes a lower bound to the TSP, not a solution: it is conceived as a sub-routine within a more complex algorithm, such as a branch-and-bound algorithm, devised to solve some combinatorial problem implying Hamiltonian cycles. Therefore, a meaningful evaluation of the practical usefulness of the algorithm can be obtained only by evaluating the effect it produces on the computing time of the algorithm in which it is embedded. Hence the efficient code implementation of this algorithm can be a research topic deserving investigation.

Independently of the application to the Held–Karp lower bound, this paper provides an $O(m + n \log n)$ algorithm to pre-compute the necessary alternative edges, so that a minimum spanning tree can be immediately restored in a graph if a vertex is deleted (e.g. in case of node failure in a telecommunication network).

Acknowledgements

While developing his master thesis, Luca Diedolo made some preliminary experiments comparing LB^*_{HK} with LB^*_H, triggering the study presented in this paper. Two anonymous reviewers and the editor provided very useful and detailed comments and suggestions.

A Union-Find

The Union-Find data-structure is used to represent a partition of a ground set N into subsets that are iteratively merged until a single set coinciding with the ground set is obtained. The Union-Find data-structure allows to efficiently perform two operations:

- retrieving which is the subset in which a given element $k \in N$ is contained;
- merging two subsets in a single one.

A list $\mathcal{L}(k)$ is associated with each element $k \in N$. Each list has a distinguished element acting as the head of the list. The Union-Find data-structure also includes two vectors, indicated by Head and Card, with as many components as the number of elements in the ground set. For each $k \in N$, $\mathsf{Head}(k)$ indicates the element that is the head of the list containing k, while $\mathsf{Card}(k)$ indicates the number of elements in $\mathcal{L}(k)$. Initially $\mathcal{L}(k) = \{k\}$, $\mathsf{Card}(k) = 1$ and $\mathsf{Head}(k) = k$ forall $k \in N$.

The list to which a given element $k \in N$ belongs can be retrieved in $O(1)$, because it is indicated by $\mathsf{Head}(k)$. Hence, the test whether two elements $u \in N$ and $v \in N$ belong to the same subset or not translates into testing whether $\mathsf{Head}(u) = \mathsf{Head}(v)$.

When two distinct lists with heads in $u \in N$ and $v \in N$ must be merged, the list with smallest cardinality is appended to the list with largest cardinality. This is done by updating pointers in $O(1)$, but it requires to also update Head and Card. Assuming $\mathcal{L}(v)$ is appended to $\mathcal{L}(u)$, the values of Card are updated as follows:

$\mathsf{Card}(u) \leftarrow \mathsf{Card}(u) + \mathsf{Card}(v)$ and $\mathsf{Card}(v) \leftarrow 0$. Then, $\mathsf{Head}(k)$ must be set to u for all elements $k \in \mathcal{L}(v)$. This operation takes linear time, for each iteration. However, appending the shortest list to the longest one guarantees that the value of Card for all elements in the shortest list doubles at least. Hence, no element is subject to the update operation more than $\log n$ times, where $n = |N|$. Therefore the update operations require at most $O(\log n)$ for each element, i.e. $O(n \log n)$ overall.

Procedure Merge is called by PathScan and ProcessApex, when a link is inserted in a local forest. It is illustrated in Algorithm 12.

Algorithm 12 The procedure that merges two lists in the Union-Find data-structure of a local subgraph $\mathcal{G}(p)$.

procedure $\mathsf{Merge}(p, u, v)$
  if $\mathsf{Card}(p, \mathsf{Head}(p, u)) < \mathsf{Card}(p, \mathsf{Head}(p, v))$ **then**
    $k' \leftarrow v$
    $k'' \leftarrow u$
  else
    $k' \leftarrow u$
    $k'' \leftarrow v$
  for $k \in \mathcal{L}(p, \mathsf{Head}(p, k''))$ **do**
    $\mathsf{Head}(p, k) \leftarrow \mathsf{Head}(p, k')$
  $\mathsf{Card}(p, \mathsf{Head}(p, k')) \leftarrow \mathsf{Card}(p, \mathsf{Head}(p, k')) + \mathsf{Card}(p, \mathsf{Head}(p, k''))$
  $\mathsf{Card}(p, \mathsf{Head}(p, k'')) \leftarrow 0$
  $\mathcal{L}(p, \mathsf{Head}(p, k')) \leftarrow \mathcal{L}(p, \mathsf{Head}(p, k')) \cup \mathcal{L}(p, \mathsf{Head}(p, k''))$

References

1 David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton Series in Applied Mathematics. Princeton University Press, 2006.

2 Pascal Benchimol, Jean-Charles Régin, Louis-Martin Rousseau, Michel Rueher, and Willem-Jan van Hoeve. Improving the Held and Karp Approach with Constraint Programming. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. CPAIOR 2010*, volume 6140 of *Lecture Notes in Computer Science*, pages 40–44. Springer, 2010.

3 Chandra Chekuri and Kent Quanrud. Approximating the Held-Karp Bound for Metric TSP in Nearly-Linear Time. In *Proceedings of the* 58^th *Annual IEEE Symposium on Foundations of Computer Science*, pages 789–800. IEEE Computer Society, 2017.

4 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

5 Gregory Gutin and Abraham P. Punnen, editors. *The Traveling Salesman Problem and Its Variations*. Springer, 2007.

6 Michael Held and Richard M. Karp. The Traveling-Salesman Problem and Minimum Spanning Trees. *Oper. Res.*, 18:1138–1162, 1970.

7 Keld Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *Eur. J. Oper. Res.*, 126:106–130, 2000.

8 Christine L. Valenzuela and Antonia J. Jones. Estimating the Held-Karp lower bound for the geometric TSP. *Eur. J. Oper. Res.*, 102:157–175, 1997.

9 Ton Volgenant and Roy Jonker. A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation. *Eur. J. Oper. Res.*, 9:83–89, 1982.

10 David P. Williamson. Analysis of the Held-Karp lower bound for the asymmetric TSP. *Oper. Res. Lett.*, 12:83–88, 1992.