

# The Hamiltonian Cycle and Travelling Salesperson Problems with Traversal-Dependent Edge Deletion

Sarah Carmesin,<sup>1</sup> David Woller,<sup>2,3</sup> David Parker,<sup>4</sup> Miroslav Kulich,<sup>2</sup> Masoumeh Mansouri<sup>1</sup>

<sup>1</sup> School of Computer Science, University of Birmingham, United Kingdom

<sup>2</sup> Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University in Prague, Czech Republic

<sup>3</sup> Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, Czech Republic

<sup>4</sup> Department of Computer Science, University of Oxford, United Kingdom

sxc1431@student.bham.ac.uk, wolledav@cvut.cz, david.parker@cs.ox.ac.uk, kulich@cvut.cz, m.mansouri@bham.ac.uk

## Abstract

Variants of the well-known Hamiltonian Cycle and Travelling Salesperson problems have been studied for decades. Existing formulations assume either a static graph or a temporal graph in which edges are available based on some function of time. In this paper, we introduce a new variant of these problems inspired by applications such as open-pit mining, harvesting and painting, in which some edges become deleted or untraversable depending on the vertices that are visited. We formally define these problems and provide both a theoretical and experimental analysis of them in comparison with the conventional versions. We also propose two solvers, based on an exact backward search and a meta-heuristic solver, and provide an extensive experimental evaluation.

## 1 Introduction

Finding a closed loop on a graph where every vertex is visited exactly once is a Hamiltonian Cycle Problem (HCP), and its corresponding optimization problem in a weighted graph is a Travelling Salesperson Problem (TSP). Variants of the HCP and the TSP have been studied for decades. However, the wealth of research on this topic does not cover problems where the availability of an edge in a graph depends on the vertices already visited. This specific type of *dynamic* graph is relevant to many real-world applications, such as open-pit mining, harvesting and painting.

For instance, consider the mining inspired example shown in Figure 1, where the graph depicts a representation of a mining field and each vertex is a place to be drilled by a drilling machine. The problem is to find a route such that each vertex is visited and drilled exactly once, i.e., an instance of a HCP/TSP. However, in this problem, drilling at a vertex creates a pile of rubble, which not only makes traversing that vertex again impossible but also affects the availability of some edges around it. For example, as depicted in Figure 1(a), when vertex  $C$  is drilled, indicated by a red circle, the rubble obstructs three edges,  $BD$ ,  $CD$  and  $DA$ , which are all deleted, whereas a different traversal only results in the removal of edge  $DA$ , as shown in Figure 1(b).

To model a graph that changes due to the path of already visited vertices, as exemplified in the scenario above, we introduce a new class of graphs, called *Self-Deleting* (SD). Us-

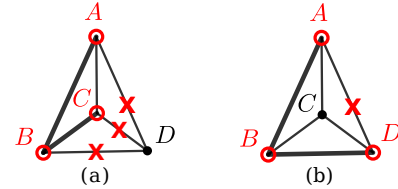


Figure 1: Representation of a mining example, where due to different traversals, indicated with thicker edges, in (a) and (b) different edges are deleted.

ing this class, we formally define two new problem variants: the Hamiltonian Cycle Problem with Self-Deleting graphs (HCP-SD), and the Travelling Salesperson Problem with Self-Deleting graphs (TSP-SD). We then compare, both theoretically and experimentally, HCP-SD and TSP-SD with the conventional versions. In particular, we identify how a self-deleting graph compares to a standard graph in terms of shortest paths, and determine where HCP and HCP-SD are equivalent. We also statistically analyse, using the graph's average vertex degree, the threshold point near which the most expensive instances of HCP and HCP-SD are located. Finally, we propose two solvers, based on an exact backward search and a meta-heuristic solver. The performance of each is extensively evaluated through experiments with a dataset based on standard TSPLIB instances as well as randomly generated datasets catering for the specificity of these new variants.

## 2 Related Work

There is a large body of research on the HCP, the TSP and their variants. As mentioned, this paper focuses on a particular type of HCP and TSP where the edges become deleted or untraversable depending on the vertices visited. None of the existing variants of these problems with dynamic graphs has this property. In a TSP on temporal networks, an edge's weight and/or availability changes with respect to some notion of time (Aaron, Krizanc, and Meyerson 2014; Michail and Spirakis 2016), and the unavailable edges can reappear again, as opposed to HCP-SD where the deleted edges are never re-enabled. The other difference is that the weight or availability of an edge in a temporal network changes with time and not due to the way the graph is traversed.

The Covering Canadian Traveller Problem (CCTP) (Liao and Huang 2014) is to find the shortest tour visiting all vertices where the availability of an edge is not known in advance. The traveller only discovers whether an edge is available once reaching one of its end vertices. The availability of an edge is set in advance and not dependent on the traversal.

The Sequential Ordering Problem (SOP), sometimes known as precedence constraint TSP (Chan 1993), is the problem of finding a minimal cost tour through a graph subject to certain precedence constraints (Escudero 1988). These constraints are given as a separate acyclic-directed graph. In the SOP, the precedence relation is solely between vertices, however, in our problem we have precedence relations between vertices and edges. Therefore, SOP is a special case of our problem, and we prove this formally in Lemma 4.

The Minimum Latency Problem (MLP) (Blum et al. 1994; Mikula and Kulich 2022) is a variant of the TSP where the cost of visiting a node depends on the path that a traveller takes. Given a weighted graph and a path, the latency of a vertex  $v$  on that path is defined as the distance travelled on that path until arriving at  $v$  for the first time. The goal of the MLP is to find a tour over all vertices such that the total latencies are minimal. Similarly, in our problem the availability of an edge depends on the path taken. However, in a MLP, the graph never changes and the latencies are the result of a simple sum.

On the HCP, some theoretical analysis focuses on investigating conditions, e.g., vertex degree (Dirac 1952; Ore 1960), under which a graph contains a Hamiltonian cycle. For instance, Pósa (Pósa 1976) and Komlós and Szemerédi (Komlós and Szemerédi 1983) proved that there is a sharp threshold for Hamiltonicity in random graphs as the edge density increases. An intuitive approach to finding a Hamiltonian cycle is to use a depth-first-search (DFS). Rubin (Rubin 1974) introduced some rules to prune the search tree. His rules do not improve the worst-case computation time  $O(n!)$ , where  $n$  is the number of vertices, however statistical analysis has shown that using such criteria improves the average computation time (Vandegriend 1999; Slegers and Berg 2021).

In terms of applications of TSP in automated planning, different variants have been used in coverage route planning (Applegate et al. 2011), e.g., for an autonomous lawnmower (Arkin, Fekete, and Mitchell 2000), or for autonomous drilling of a PCB (Grötschel, Jünger, and Reinelt 1991). Those most relevant to this paper are coverage planning problems whose environments change due to the coverage actions by agents, e.g., robots, that operate within them. The open-pit mining scenario described earlier is an example of such a coverage planning problem for which a specialised solver for the mining case is proposed by (Mansouri, Lagriffoul, and Pecora 2017). Autonomous harvesting is another instance where heavy vehicles should not pass through the areas already harvested to avoid soil compaction. The harvested areas also limit the mobility of harvesting machines, hence affecting the reachability among the nodes representing areas to be harvested. Ullrich, Hertzberg, and Stiene (Ullrich, Hertzberg, and Stiene 2014) formulate this

application as an optimization problem for which a specialised solver is also proposed. In both cases described above, the authors did not study the theoretical underpinning of the problem, nor provide a general solution that can easily be employed for other instances of problems with traversal-dependent edge deletion.

### 3 Problem statement

In this section, we formally define self-deleting graphs and introduce the corresponding notions of walks and paths. We then proceed to give a formal definition of the HCP-SD and the TSP-SD problems.

**Definition 1.** A *self-deleting graph*  $S$  is a tuple  $S = (G, f)$  where  $G = (V, E)$  is a simple, undirected graph and  $f : V \rightarrow 2^E$ . The function  $f$  specifies for every vertex  $v \in V$  which edges  $f(v)$  are deleted from  $E$  if the vertex  $v$  is processed. We refer to  $f$  as the *delete-function*.

If a vertex  $v$  is *processed*, we delete edges  $f(v)$  from  $G$ . For a self-deleting graph  $S$  and set  $X \subset V$  of vertices, the *residual graph*  $G_X$  of  $S$  after processing  $X$  is defined as:

$$G_X = G \setminus \bigcup_{v \in X} f(v).$$

We call a simple path  $p = (v_1, \dots, v_x)$  in a self-deleting graph  $f$ -*conforming* if for every  $1 \leq i < x$  the edge  $e_i = \{v_i, v_{i+1}\}$  is in the residual graph  $G_{\{v_1, \dots, v_i\}}$ . An  $f$ -conforming simple path  $p$  traverses the graph  $G$  while processing every vertex on  $p$  when it is visited.

In contrast to a path, vertices on a walk can be visited more than once. For a walk on a self-deleting graph, a vertex is processed when it is visited for the last time. Formally, we call a walk  $w = (v_1, \dots, v_x)$   $f$ -*conforming* if for every  $1 \leq i < x$  the edge  $e_i = \{v_i, v_{i+1}\}$  is in the residual graph  $G_{\{v_1, \dots, v_i\} \setminus \{v_{i+1}, \dots, v_x\}}$ .

Following standard terminology we call a sequence of vertices  $c = (v_1, \dots, v_x, v_1)$  an  $f$ -conforming cycle if  $(v_1, \dots, v_x)$  is an  $f$ -conforming path and the edge  $\{v_x, v_1\}$  exists in the residual graph  $G_c$ . Then, a *Hamiltonian cycle* of self-deleting graph  $S$  is an  $f$ -conforming cycle that contains all vertices of  $S$  exactly once.

**Problem 1.** Given a self-deleting graph  $S = (G, f)$ , the *Hamiltonian Cycle Problem on Self-Deleting graphs (HCP-SD)* is to find a Hamiltonian cycle on  $S$ .

**Problem 2.** Given a self-deleting graph  $S = (G, f)$ , the *weak Hamiltonian Cycle Problem on Self-Deleting graphs (weak HCP-SD)* is to find an ( $f$ -conforming) closed walk on  $S$  that contains every vertex at least once.

**Observation 1.** Every Hamiltonian cycle of  $S$  is a Hamiltonian cycle of  $G$ .

This implies that the HCP-SD is at least as hard as finding a Hamiltonian path.

Using a weighted graph as the underlying graph of a self-deleting graph we can define optimization problems on self-deleting graphs as follows.

**Problem 3.** Given a self-deleting graph  $S = (G, f)$ , where  $G$  is a weighted graph, the *Travelling Salesperson Problem*

on self-deleting graphs (TSP-SD) is to find a shortest Hamiltonian cycle on  $S$ .

**Problem 4.** Given a self-deleting graph  $S = (G, f)$ , the *weak Travelling Salesperson Problem on self-deleting graphs (weak TSP-SD)* is to find a shortest ( $f$ -conforming) closed walk on  $S$  that contains every vertex at least once.

#### 4 Properties of self-deleting graphs

In this section, we provide some formal analysis of self-deleting graphs, in comparison to static graphs. First, we analyse path segments in self-deleting graphs.

**Lemma 1.** *Let  $S = (G, f)$  be a self-deleting graph and  $p = (v_1, \dots, v_x)$  be an  $f$ -conforming path of  $S$ . For every  $1 \leq i < j \leq x$  it holds that the path segment  $p_{i,j} = (v_i, \dots, v_j)$  is an  $f$ -conforming path of  $S_{i,j} = (G', f)$  where  $G'$  is the induced subgraph of  $G$  on the vertices  $(v_i, \dots, v_j)$ .*

*Proof.* Let  $p = (v_1, \dots, v_x)$  be an  $f$ -conforming path of  $S$  and let  $1 \leq i < j \leq x$ . By definition, the path segment  $p_{i,j} = (v_i, \dots, v_j)$  is an  $f$ -conforming path of  $S_{i,j} = (G', f)$  if for every  $i \leq k < j$  it holds that the edge  $e_k = \{v_k, v_{k+1}\}$  is in the residual graph  $G'_{\{v_i, \dots, v_k\}}$ . We now show that, for every  $i \leq k < j$ , the edge  $e_k = \{v_k, v_{k+1}\}$  is in the residual graph  $G'_{\{v_i, \dots, v_k\}}$ .

Assume for a contradiction there is a  $k$  with  $i \leq k < j$  where  $e_k \notin G'_{\{v_i, \dots, v_k\}}$ . There are two possible reasons for this.

1.  $e_k \notin E(G')$ : Since  $e_k$  is in  $E(G)$ ,  $G'$  cannot be an induced subgraph of  $G$  and we have a contradiction.
2.  $e_k$  gets deleted by some  $f(v_y)$ ,  $i \leq y \leq k$ : If  $e_k \in \cup_{i \leq y \leq k} f(v_y)$  then  $e_k \in \cup_{1 \leq y \leq k} f(v_y)$  and therefore  $e_k \notin G_{\{v_1, \dots, v_k\}}$ . Since  $e_k \notin G_{\{p_1, \dots, p_k\}}$  the path  $p$  is not  $f$ -conforming and we again arrive at a contradiction.

Since both cases yield a contradiction, the lemma holds.  $\square$

Let  $p = (v_1, \dots, v_x)$  be an  $f$ -conforming path from vertex  $v_1$  to vertex  $v_x$  and let  $|p|$  denote the length of the path  $p$ . We call  $p$  a *shortest  $f$ -conforming path* from  $v_1$  to  $v_x$  if for every other  $f$ -conforming path  $\hat{p} = (v_1, \dots, v_x)$  from  $v_1$  to  $v_x$  it holds that  $|p| \leq |\hat{p}|$ .

**Lemma 2.** *Let  $p = (v_1, \dots, v_k)$  be a shortest  $f$ -conforming path from  $v_1$  to  $v_k$  on a self-deleting graph  $S = (G, f)$ . The following two statements hold:*

1. *For every  $1 < i < k$  it holds that the path  $p_i = (v_1, \dots, v_i)$  is not necessarily a shortest  $f$ -conforming path in  $S$ .*
2. *It further holds that the path  $\hat{p}_i = (v_i, \dots, v_k)$  is a shortest  $f$ -conforming path from  $v_i$  to  $v_k$  in the self-deleting graph  $S' = (G_{\{v_1, \dots, v_i\}}, f)$ .*

*Proof.* Let  $p = (v_1, \dots, v_k)$  be a shortest  $f$ -conforming path from  $v_1$  to  $v_k$  on a self-deleting graph  $S = (G, f)$ . For any  $1 < i < k$  we denote the path segment of  $p$  from  $v_1$  to  $v_i$  by  $p_i$  and the path segment from  $v_i$  to  $v_k$  by  $\hat{p}_i$ . Due to Lemma 1, the path segments  $p_i$  and  $\hat{p}_i$  are  $f$ -conforming. We now prove the two statements separately.

1. A shortest  $f$ -conforming path from  $v_1$  to  $v_i$  could contain a vertex  $v_j$  for which  $f(v_j)$  deletes an edge needed in the second part  $\hat{p}_i$  of the  $f$ -conforming path  $p$ . So,  $p_i$  is not necessarily a shortest  $f$ -conforming path from  $v_1$  to  $v_i$ .
2. We now prove that the path  $\hat{p}_i = (v_i, \dots, v_k)$  is a shortest  $f$ -conforming path from  $v_i$  to  $v_k$  in the self-deleting graph  $S' = (G_{\{v_1, \dots, v_i\}}, f)$ . For a contradiction, assume there is a  $1 < i < k$  with a shortest  $f$ -conforming path  $w$  from  $v_i$  to  $v_k$  with  $|w| < |\hat{p}_i|$  in  $S'$ . We then consider the following two cases.
  - (a)  $w$  and  $p_i$  do not share a vertex, as depicted in Fig. 2(a). If this is the case, then the path from  $v_1$  to  $v_k$  that consists of the path  $p_i$  and the path  $w$  is shorter than the path  $p$ . This is a contradiction.
  - (b)  $w = (v_i, w_1, \dots, w_l, v_k)$  and  $p_i$  share a vertex  $v_x = w_y$ , as depicted in Fig. 2(b). Then the path  $(v_1, \dots, v_x, w_{y+1}, \dots, v_k)$  is shorter than  $p$ . This is a contradiction.  $\square$

Lemma 2 indicates the inherent difference between static and self-deleting graphs. In static graphs, every segment of a shortest path is a shortest path. This fact is exploited by different algorithms, often based on dynamic programming, for path finding in static graphs, e.g. Dijkstra's algorithm (Dijkstra 1959). As a consequence, these types of algorithms cannot easily be applied to self-deleting graphs.

**Lemma 3.** *Let  $S = (G, f)$  be a self-deleting graph where, for every vertex  $v \in V(G)$ ,  $f(v)$  deletes only edges that are incident to  $v$ , then the Hamiltonian path problem on self-deleting graphs is equivalent to the Hamiltonian path problem on directed graphs.*

*Proof.* We construct a corresponding directed graph  $D = (V, A)$ , to a self-deleting graph  $S = (G, f)$ , where  $f(v)$  deletes only edges incident to  $v$ , as follows.

$$V(D) = V(G),$$

$$A(D) = \{(v, w) \mid \{v, w\} \in E(G) \wedge \{v, w\} \notin f(v)\}$$

(Here  $(v, w)$  describes the directed arc from  $v$  to  $w$ , while  $\{v, w\}$  describes the undirected edge between  $v$  and  $w$ .)

Another way to explain this construction is as follows. We make  $G$  a directed graph in which each edge is replaced by two arcs in opposite directions. For every vertex  $v$  we then delete all outgoing arcs corresponding to an edge in  $f(v)$ .

We now prove that a path  $p$  is  $f$ -conforming in  $S$  if and only if  $p$  is a path in  $D$ .

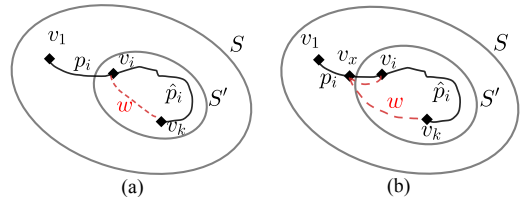


Figure 2: Illustrations for the proof of Lemma 2: If the path  $w$  is shorter than the path  $\hat{p}_i$  then  $p$  was not a shortest path.

$\Rightarrow$ : If the path  $p = (v_1, \dots, v_k)$  is  $f$ -conforming, it holds by definition that for every  $1 \leq i < k$  the edge  $\{v_i, v_{i+1}\}$  is in the residual graph  $G_{\{v_1, \dots, v_i\}}$ . Since  $\{v_i, v_{i+1}\} \in G_{\{v_1, \dots, v_i\}}$  it holds that  $\{v_i, v_{i+1}\} \in E(G)$ . Also since  $\{v_i, v_{i+1}\} \in G_{\{v_1, \dots, v_i\}}$  it holds that the edge  $\{v_i, v_{i+1}\} \notin f(v_x)$  with  $1 \leq x \leq i$ , so the edge  $\{v_i, v_{i+1}\}$  is in particular not in  $f(v_i)$ . Therefore the arc  $(v_i, v_{i+1})$  is in  $A(D)$ .

$\Leftarrow$ : Now, let  $p = (v_1, \dots, v_k)$  be a simple path in  $D$ . So, for every  $1 \leq i < k$  the arc  $(v_i, v_{i+1})$  is in  $A(D)$ . For every arc  $a = (v, w) \in A(D)$  it holds that the edge  $e = \{v, w\}$  is in  $E(G)$  and  $e \notin f(v)$ . Since  $(v_i, v_{i+1})$  is in  $A(D)$ , it follows that  $\{v_i, v_{i+1}\} \notin f(v_i)$ . Since no other vertex  $v_n$  with  $n < i$  is incident to  $e$  it follows that  $\{v_i, v_{i+1}\} \notin f(v_m)$  for every  $m \leq i$ . So  $\{v_i, v_{i+1}\} \in G_{v_1, \dots, v_i}$ . Therefore  $p$  is  $f$ -conforming in  $S$ .

Every path through  $D$  is an  $f$ -conforming path through  $S$  and vice-versa. So the Hamiltonian path problem on  $S$  is equivalent to the Hamiltonian path problem on  $D$ .  $\square$

A *sequential ordering problem (SOP)* is defined as a graph  $G = (V, E)$  accompanied by a precedence graph  $P$ . The precedence graph  $P$  is a directed graph defined on the same set of vertices  $V$ . It represents the precedence relation between the vertices of  $G$ . An edge from  $v_i$  to  $v_j$  in  $P$  implies that  $v_i$  must precede  $v_j$  in any path through  $G$ . The problem is to find a Hamiltonian path in  $G$  that does not violate the precedence relation given by  $P$ .

**Lemma 4.** *To every sequential ordering problem SOP there is a corresponding self-deleting graph  $S_{SOP}$  such that a path  $p$  is a solution to SOP if and only if  $p$  is a Hamiltonian path of  $S_{SOP}$ .*

*Proof.* Let a SOP be given by the graph  $H$  and the precedence graph  $P$ . Let  $pre(v) \subseteq V(H)$  be the set of vertices that precede  $v$  in  $P$ , formally  $pre(v) = \{w \mid (w, v) \in A(P)\}$ . We construct the corresponding self-deleting graph  $S_{SOP} = (G, f)$  as follows.

$$G = H,$$

$$f(v) = \bigcup_{w \in pre(v)} \{e \in E(G) \mid e \text{ incident to } w\}$$

$\Rightarrow$ : Let  $p = (v_1, \dots, v_k)$  be a path in  $H$  that satisfies the precedence relations given in  $P$ . So, for every  $1 \leq i < j < k$  the vertices  $v_j$  and  $v_{j+1}$  are not required to precede  $v_i$ . Thus, the edges  $(v_j, v_i)$  and  $(v_{j+1}, v_i)$  are not in  $P$  and  $v_j, v_{j+1} \notin pre(v_i)$ . So by construction of  $f$  the edge  $(v_j, v_{j+1})$  does not get deleted by any  $f(v_i)$  for  $1 \leq i \leq j$ . This implies that the edge  $(v_j, v_{j+1})$  is in the residual graph  $G_{\{v_1, \dots, v_j\}}$ . Thus,  $p$  is  $f$ -conforming in  $S_{SOP}$ .

$\Leftarrow$ : If the path  $p = (v_1, \dots, v_k)$  is  $f$ -conforming in  $S_{SOP}$  it holds per definition that for every  $1 \leq i < k$  the edge  $(v_i, v_{i+1})$  is in the residual graph  $G_{\{v_1, \dots, v_i\}}$ . Thus, it holds that the edge  $(v_i, v_{i+1})$  has not been deleted by any vertex  $v_x$  with  $1 \leq x \leq i$ . It follows that  $v_i$  and  $v_{i+1}$  are not in  $pre(v_x)$  with  $1 \leq x \leq i$ . Thus,  $v_i$  and  $v_{i+1}$  are not required to be visited before  $v_x$  with  $1 \leq x \leq i$  and the path  $p$  satisfies the precedence conditions in  $P$ . It is therefore a valid path in  $H$ .

We proved that any valid path in a SOP is  $f$ -conforming in the corresponding self-deleting graph and vice-versa. This holds in particular for Hamiltonian paths.  $\square$

## 5 Exact and heuristic solvers

Next, we describe two solvers for the HCP-SD and TSP-SD problems: one that produces an *exact* solution and one which relies on *heuristics*.

### Exact solvers

An intuitive approach to solving the HCP on a self-deleting graph  $S$  is to employ a DFS in a forward-search manner: starting with some vertex  $p_1$ , we delete all edges in  $f(p_1)$  in  $G$ , then choose a neighbour  $p_2$  of  $p_1$  as the next vertex on the path and repeat until the path is a Hamiltonian cycle or the current path cannot be extended, in which case we backtrack. This approach can be improved with methods used in algorithms for Hamiltonian cycles in conventional graphs, namely graph/search-tree pruning, as introduced by (Rubin 1974; Vandegriend and Culberson 1998). Their algorithms identify edges that *must* be in a Hamiltonian cycle, e.g., edges incident to a vertex of degree 2, and employ these required edges to improve the average runtime of a forward DFS. However, even with these pruning rules, the algorithm fails to detect paths that cannot be extended to a Hamiltonian cycle early. This is due to the fact that the edge deletion is traversal dependent.

Since failures occurring at a late stage are often due to the choices at an earlier stage of the search, we propose a *backward* search algorithm, shown in Algorithm 1. This takes advantage of the late failures to greatly reduce the size of the search tree. Instead of exploring the path from a start vertex and deleting edges subsequently, Algorithm 1 starts by deleting all edges that would get deleted at some point. It then explores the graph in a backward fashion, adding edges according to visited vertices as follows. During this backward exploration of the graph, edges are added, so searching for required edges, as is done in conventional forward DFS for Hamiltonian cycles, is not possible.

The first call of *backwardSearch* receives a single start vertex as the path and the self-deleting graph. During the repeated calls of *backwardSearch*, the path grows backwards, so the first call will be with  $path = (v_1)$ , the next with  $path = (v_n, v_1)$ , then  $path = (v_{n-1}, v_n, v_1)$  and so forth. During each call of *backwardSearch* the residual graph  $R$  with respect to  $path$  is calculated (line 1). In line 2 follows a goal check where it is first verified whether the path has the correct length and if so, whether the missing edge between both end vertices exists (line 3). If the initial check fails, the algorithm calculates the set  $SV$  of vertices that are candidates for the second vertex in the Hamiltonian path in line 9. If all the candidates are already on  $path$  the path cannot be extended to a Hamiltonian cycle. We check this condition in line 10. In line 13 the set  $N$  of neighbours of the first vertex of the current  $path$  in  $R$  is calculated. For every neighbour, *backwardSearch* is called with an extended path until one of them returns a Hamiltonian cycle.

**Lemma 5.** *Let  $S = (G, f)$  be a self-deleting graph. If there*

---

Algorithm 1: Backward search algorithm for finding a Hamiltonian cycle in a self-deleting graph

---

**Input:** Current path, the self-deleting graph

**Output:** Hamiltonian cycle of  $S$  or failure

**Function:** backwardSearch( $path, S = (G, f)$ )

```

1:  $R \leftarrow G \setminus \{e \in f(v) \mid v \notin (path \setminus path.last)\}$ 
2: if  $|path| = |V(G)|$  then
3:   if  $(path.last, path.first) \in E(R)$  then
4:     return  $[path.last] + path$ 
5:   else
6:     return failure
7:   end if
8: else
9:    $SV \leftarrow \{v \in V(G) \mid (path.last, v) \in E(G) \wedge$ 
     $(path.last, v) \notin f(path.last)\}$ 
10:  if  $SV \setminus path = \emptyset$  then
11:    return failure
12:  else
13:     $N \leftarrow \{v \mid (path.first, v) \in E(R) \wedge v \notin path\}$ 
14:    for  $v \in N$  do
15:       $result \leftarrow backwardSearch([v] + path, S)$ 
16:      if  $result \neq failure$  then
17:        return result
18:      end if
19:    end for
20:    return failure
21:  end if
22: end if

```

---

exists at least one Hamiltonian cycle in  $S$ , then the backward search finds a Hamilton cycle.

*Proof.* We prove by contradiction: Assume there exists a Hamiltonian self-deleting graph  $S = (G, f)$ , where the algorithm returns *failure*. Let  $n = |V(G)|$  and  $P = (p_1, \dots, p_{n+1})$  with  $p_1 = p_{n+1}$  a Hamilton cycle of  $S$ . We analyse certain function calls to prove the lemma.

If  $backwardSearch((p_2, \dots, p_{n+1}), S)$  is called, line 1 calculates the residual graph  $R = G \setminus f(p_1)$ . Since  $|(p_2, \dots, p_{n+1})| = n$ , line 3 triggers. The algorithm then checks whether the edge  $(p_1, p_2)$  exists in  $R$ . If so,  $P$  is returned, which is a contradiction since we assumed *failure* is returned. However, if there is no edge  $(p_1, p_2)$  in  $R$  then  $p$  is not a Hamiltonian cycle, contradicting the assumption.

Therefore  $backwardSearch((p_2, \dots, p_{n+1}), S)$  is never called. So there is a largest number  $2 \leq x \leq n$  for which  $backwardSearch((p_x, \dots, p_{n+1}), S)$  is never called. We analyse the call  $backwardSearch((p_{x+1}, \dots, p_{n+1}), S)$ .

In line 1 the residual graph  $R = G_{V(G) \setminus \{p_{x+1}, \dots, p_n\}}$  is calculated. Since  $|(p_{x+1}, \dots, p_{n+1})| < |V(G)|$ , the algorithm continues in line 8. In line 9 the set  $SV$  of candidates for the second vertex on the Hamiltonian cycle starting in  $p_1$  is calculated. Since  $P$  is a Hamiltonian cycle the set contains at least  $p_2$ . And since the current path is  $p_{x+1}, \dots, p_n$  with  $x \geq 2$ ,  $p_2$  is not in path and the if-condition in line 10 fails.

We continue in line 13. Here, the list of neighbours  $N$  of current first vertex in  $R$  that are not already on the path is

calculated. We now consider two cases:

(a)  $p_x \notin N$ :  $N$  contains all neighbours of  $p_{x+1}$  in  $R$ . So if  $p_x \notin R$  then there is no edge between  $p_x$  and  $p_{x+1}$  in the residual graph after processing  $p_1, \dots, p_x$ . Thus,  $p$  is no Hamiltonian cycle, a contradiction. So (b) must hold.

(b)  $p_x \in N$ : The only reason for not calling  $backwardSearch((p_x, \dots, p_{n+1}), S)$  is that another call like  $backwardSearch((y, p_{x+1}, \dots, p_{n+1}), S)$  with  $y \in N$  does not return failure. Thus the algorithm finds another Hamiltonian cycle, this again is a contradiction.

Since we always arrive at a contradiction, the assumption does not hold. Thus, if  $S$  is Hamiltonian the algorithm finds a Hamiltonian cycle.  $\square$

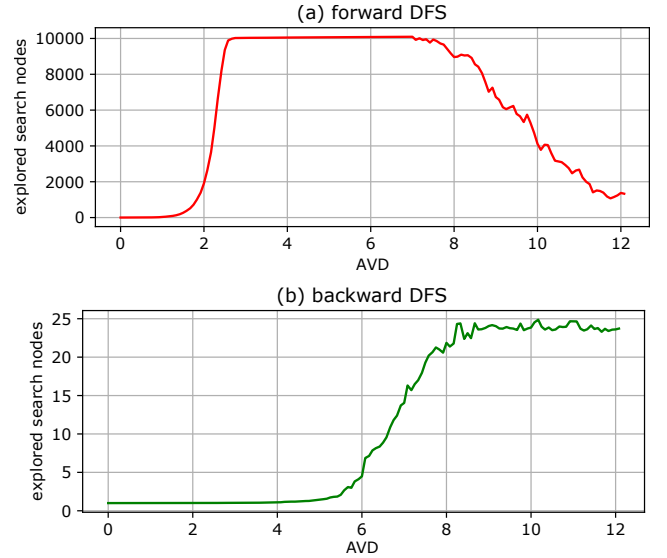


Figure 3: Explored nodes in the forward-DFS and backward-DFS on the dataset random24-100

In order to investigate the behaviour of both exact algorithms, we first need to define the Average Vertex Degree (AVD) for a self-deleting graph. AVD is a metric commonly used in the analysis of static graphs for the HCP. Let  $k$  be the number of times an edge  $e$  appears in the delete function  $f$ . The probability that the edge will be deleted after processing any  $l$  vertices from  $V$  in arbitrary order is given by  $p(e, l) = 1 - \prod_{i=0}^{k-1} \frac{(n-l)-i}{n-i}$ . Then, the expected “static” AVD of  $S$  after processing any  $l$  vertices can be determined as  $\delta(l) = (n-1) - \frac{2}{n} \sum_{e \in E} p(e, l)$ . From here, we can define the AVD of  $S$  as  $\frac{1}{n} \sum_{l=1}^n \delta(l)$ .

A dataset random24-100 of 14400 random self-deleting graphs with 24 vertices was generated in order to compare both exact algorithms. The delete function  $f$  was sampled uniformly randomly with overlapping of  $f(v)$  for two distinct  $v$  allowed. In terms of AVD, the dataset uniformly covers the interval from 0 to 12. In an experimental comparison between backward and forward search, both solving the same dataset random24-100 and capped at 10000 expanded search nodes, the backward search performs much better. It was able to solve all instances and on average was able to

identify a Hamiltonian instance after 27.9 explored nodes and a non-Hamiltonian instance after 1.6 explored nodes. The forward search failed to find a solution within the limit for most instances. The diagrams in Figure 3 show the average explored nodes by which either algorithm was able to decide the instance or the limit was reached.

Figure 4(a) shows the percentage of infeasible instances decided by the backward search at various search depths while using the same random24-100 dataset. Infeasible instances with AVD less than 3 are detected instantly at depth 1. The hardest instances to detect are located between AVDs 6 and 7. Above 7, the dataset does not contain any infeasible instances. Finally, more than 80% of infeasible instances are detected at depth 10, less than half of  $|V|$ .

Figure 4(b) illustrates how the percentage of detected infeasible instances at various depths depends on  $|V|$ . At a fixed depth, the percentage unsurprisingly decreases with increasing  $|V|$ , but even for  $|V| = 200$  about 50% instances are detected at depth 10. Interestingly, the percentage increases when using a relative depth and close to 100% infeasible instances are detected at depth  $0.2|V|$ , when  $|V| > 100$ . This experiment indicates that the backward search algorithm's ability to detect infeasible instances of HCP-SD early on in the search improves with increasing  $|V|$  and, consequently, the algorithm may be scalable enough to find feasible solutions even for instances with  $|V|$  of practical interest.

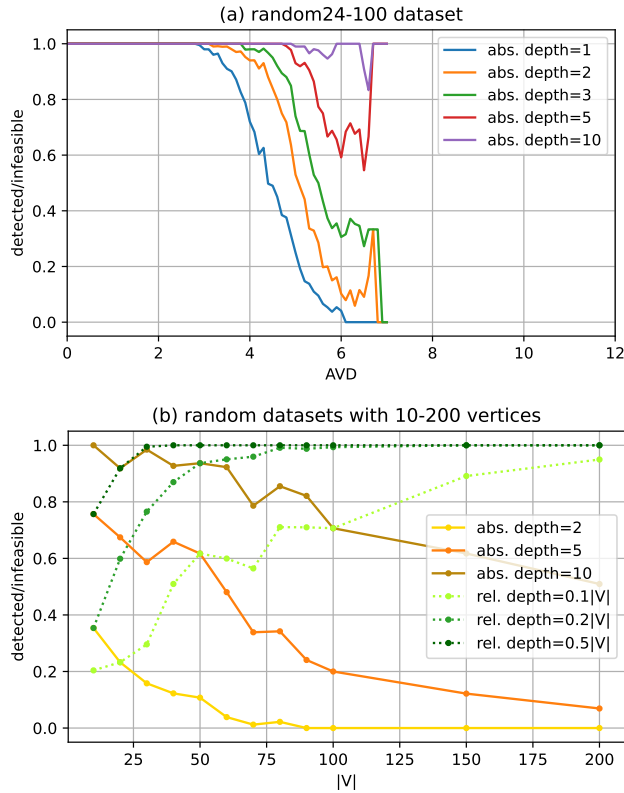


Figure 4: Backward search behaviour

## Heuristic solver

The proposed exact solver is likely to provide limited scalability when addressing *optimization* problems due to its exhaustive nature. Also, finding near-optimal solutions is often sufficient in practical applications, therefore, heuristic algorithms may be the only computationally feasible approach to obtain them. A common procedure is to design a problem-specific metaheuristic algorithm, that is tailored to a particular application. Various heuristic approaches were successfully applied to problems related to the TSP-SD, such as metaheuristics based on local search (Helsgaun 2017), evolutionary optimization (Sabar et al. 2019) or swarm optimization (Xiang et al. 2020).

In this paper, we use a generic metaheuristic solver for problems with permutative representation (Woller, Hrazdřra, and Kulich 2022), so that we can remain application agnostic regarding multiple variants of TSP-SD. The solver implements several high-level metaheuristics and also a bank of low-level local search operators, perturbations and construction procedures. These can be readily applied to various problems, whose solution can be encoded as a sequence of potentially recurring nodes. The only user requirement is to specify a set of nodes  $A$ , lower and upper bounds  $L, U$  of the frequency of their occurrence in a solution sequence  $x = (x_1, x_2, \dots, x_n)$ , where  $x_i \in A$ ; a fitness function  $f(x)$  and an aggregation of penalty functions  $g(x)$ . The bounds are always respected by the solver, whereas the penalty functions are treated as soft constraints. Their purpose is to direct the search process towards valid solutions. TSP-SD can be described in the solver formalism as follows:

$$\begin{aligned}
 A &= \{v_1, v_2, \dots, v_n\} = V(G), \\
 L &= (1, 1, \dots, 1) = U, \\
 f(x) &= \sum_{i=1}^n \|e_i\|, \\
 g(x) &= \sum_{i=1}^n g_i(x), \text{ where} \\
 g_i(x) &= \begin{cases} 0, & \text{if } e_i \in E(G_{\{x_1, x_2, \dots, x_i\}}), \\ M, & \text{otherwise.} \end{cases}
 \end{aligned}$$

Here, the set of nodes to visit  $A$  corresponds to the set of vertices  $V(G)$ . Each node  $v_i$  has to be processed exactly once, thus  $L_i = U_i = 1$ . Then,  $e_i$  is the edge  $\{x_i, x_{i+1 \bmod n}\}$  and  $G_{\{x_1, x_2, \dots, x_i\}}$  is the residual graph after processing first  $i$  nodes in  $x$ . The goal is to minimize the total length of the cycle given by  $x$ . A large constant  $M$  is added to the penalty  $g(x)$  via  $g_i(x)$  for attempting to use an already deleted edge.

For the weak TSP-SD, both the set of nodes  $A$  and the respective bounds  $L, U$  are defined in the same way as in the TSP-SD, but the definition of  $f(x)$  and  $g_i(x)$  differs:

$$\begin{aligned}
 f(x) &= \sum_{i=1}^n \|p_i\|, \\
 g_i(x) &= \begin{cases} 0, & \text{if } p_i \text{ exists in } G_{\{x_1, x_2, \dots, x_i\}}, \\ M, & \text{otherwise.} \end{cases}
 \end{aligned}$$

Here,  $p_i$  is the shortest path from  $x_i$  to  $x_{i+1 \bmod n}$  in the residual graph  $G_{\{x_1, x_2, \dots, x_i\}}$ , which is found using A\*. Thus, the time complexity of weak TSP-SD fitness evaluation is higher than TSP-SD by  $\mathcal{O}(|E|)$ . Only the first and last vertex of  $p_i$  are processed. If  $p_i$  does not exist, a large constant  $M$  is added to the penalty  $g(x)$  via  $g_i(x)$ . The goal is to minimize the total length of the closed walk given by  $x$ .

## 6 Statistical analysis of HCP-SD

In this section, we investigate properties analogous to those previously studied in the literature for HCP, since they are crucial for understanding behaviour and evaluating the performance of the proposed solvers. For the HCP, the probability density function of a randomly generated graph being Hamiltonian was experimentally shown to be sigmoidally shaped around a certain threshold point (Sleegers and Berg 2021). This threshold corresponds to the graph’s AVD, for which the probability is approximately 0.5. Their experiments indicate that HCP instances close to this boundary are the most expensive to decide for various exact algorithms in terms of computational cost, although isolated clusters of hard instances were also identified far away from it. The location of this threshold has been proved to be  $\ln(V) + \ln(\ln(V))$ , which is called the Komlós-Szemerédi bound (Komlós and Szemerédi 1983).

First, we replicated the experiment from (Sleegers and Berg 2021), showing the probability density function of Hamiltonicity for a randomly generated graph with 24 vertices. For this purpose, we generated a dataset of 100 random graphs for every number of edges from 1 to 144, resulting in 14400 graphs with AVD ranging from 0 to 12. The HCP was decided for the whole dataset using the Concorde TSP solver and the result of the experiment is shown in Figure 5(a) - HCP (exact). The dataset random24-100 of 14400 random self-deleting graphs with 24 vertices was created analogously, covering the same range of AVDs. On this dataset, HCP-SD was decided with both an exact and heuristic solver and weak HCP-SD with a heuristic solver described in Section 5. The exact solver was always terminated after successfully deciding the problem, whereas the heuristic solver was terminated either after finding a feasible solution, or reaching a time budget of  $|V|$  seconds. Therefore, the heuristic solver’s results are suitable for assessing the solver’s properties, rather than reasoning about the problem itself. Figure 5(a) indicates that the probability density function of HCP-SD is shaped similarly to that of HCP but is steeper and the threshold point is located further to the right.

The weak HCP-SD appears to have similar properties, but there is no exact solver available, and using the heuristic solver may affect the location of the threshold point, as it may label a feasible instance as infeasible. We can see that instances with AVD less than 3 that were shown to be easy to decide for the exact solver in Figure 4(a), actually have zero probability of being Hamiltonian. Instances with AVD between 6 and 7, which were shown to be the hardest to decide, are located close to the HCP-SD Hamiltonicity threshold point. Thus, in a similar fashion to HCP, HCP-SD instances close to the threshold point are computationally harder for the exact solver.

Second, 12 more datasets of random self-deleting graphs with 10 to 200 vertices and uniformly randomly sampled  $f$  were generated to investigate the Hamiltonicity bound w.r.t. to  $|V|$  for both variants of HCP-SD. Each of these datasets was generated to cover an interval that contains the threshold point of both problems and consisted of 2500 instances, evenly distributed across the interval into groups of 50 instances with the same AVD. Again, the HCP-SD was decided with an exact and heuristic solver and the weak HCP-SD with a heuristic solver, and the location of the threshold point was determined for each dataset and problem. The locations of the threshold points are shown in Figure 5(b), thus showing a bound analogous to the Komlós-Szemerédi bound. The bound HCP-SD (exact) follows a sublinear, presumably logarithmic trend, similar to the Komlós-Szemerédi bound but faster growing. As for the weak HCP-SD, the heuristic data evidently do not provide an accurate estimate of the bound.

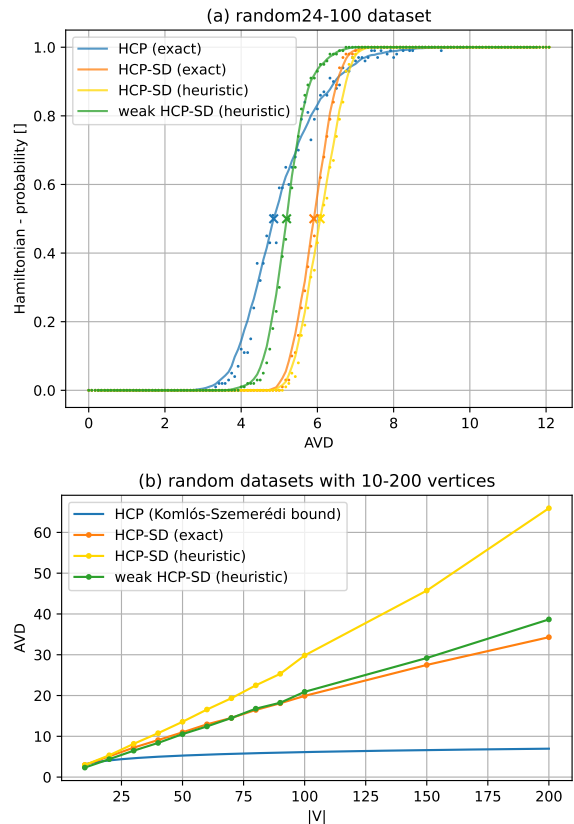


Figure 5: Comparison of Hamiltonicity bounds

The threshold points should never be higher than for the HCP-SD because all self-deleting graphs feasible in HCP-SD are also feasible in weak HCP-SD. The bound HCP-SD (heuristic) illustrates that the heuristic solver consistently struggles with finding feasible solutions close to the real Hamiltonicity bound, found by the exact solver.

## 7 TSP-SD solvers evaluation

So far, we have focused only on the results relevant to decision problems, but both proposed solvers are designed to address the formulated optimization problems as well. Each solver has unique properties that are investigated in a series of eight experiments on a newly created dataset<sup>1</sup>. The dataset consists of 11 instances of self-deleting graphs with a size ranging from 14 to 1084 vertices. The instances are selected from the TSPLIB library (Reinelt 1991), but a uniformly randomly generated delete function  $f$  is added. In terms of the AVD, most of the instances are generated close to the HCP-SD Hamiltonicity bound of the heuristic solver, so that they could be solved by the heuristic solver alone. The following naming format is used: *original\_name*| $V$ |-AVD.

The heuristic solver offers a portfolio of alternative components, each suitable for a different set of problems with permutative representation. The solver must be tuned to achieve the best performance for a specific problem. The tuning was carried out using the irace package (López-Ibáñez et al. 2016) with a tuning budget of 2500 experiments. The configuration obtained is shown in Table 1. The tuner selected the Basic Variable Neighborhood Search (VNS) (Hansen et al. 2019) to use as a high-level metaheuristic and the Pipe Variable Neighborhood Descent (VND) (Duarte et al. 2018) to control the local search. The results of the exact solver were generated on a dedicated machine with Ubuntu 18.04 OS, Intel Core i7-7700 CPU. Experiments using the heuristic solver were generated on an AMD EPYC 7543 CPU cluster. Each instance was solved once by the exact solver and 50 times by the heuristic solver, since it is stochastic. The heuristic solver always had a time budget of  $10|V|$  seconds per single run. We present the results in Tables 2, 3 and 4. Individual experiments are referred to by the column letter of the corresponding table. Finally, the relative improvement brought by an experiment B relative to an earlier experiment A in a particular instance  $i$  is calculated as  $100 \times (1 - \frac{obj_i(B)}{obj_i(A)})$ , where  $obj_i(A)$  is the objective value on  $i$  in  $A$ . This value is eventually averaged across the entire dataset.

The proposed backward search is introduced as a decision algorithm for the HCP-SD in Algorithm 1. To address the optimization problem TSP-SD, only a slight modification is required. The algorithm does not stop when the first valid solution is found (line 4). Instead, it continues to search until a given time limit is reached while storing the best solution found so far. Another minor modification is the order of expansion at line 14. In the default variant, the nodes  $v \in N$  are traversed in arbitrary order, determined by the iterator implementation of the set  $N$ . In the following experiments, a greedy expansion is also tested. In this variant, nodes  $v \in N$  are sorted according to their distance from *path.first* and expanded from closest to farthest.

Table 2 documents the performance of the exact TSP-SD solver. The backward search performs the path expansion in default order in experiments in columns A and B, whereas greedy expansion is used in experiments in columns C and

D. Column A presents the objective values and computation times needed to find the first valid solution of TSP-SD while using the default expansion. A solution is found within one second for instances with up to  $|V| = 202$  and within one minute for all instances in the dataset. The dataset contains two variants of the berlin52 instance with different values of AVD, from which the berlin52-10.4 instance is closer to the Hamiltonicity bound. Finding a valid solution for berlin52-10.4 requires 10 times more time than berlin52-13.2. Thus, AVD seems to be an important factor playing against the backward search. The scalability of the exact solver in this experiment is surprisingly good, as was already indicated in Figure 4(b).

In Table 2, column B, the exact solver was given a budget of 12 hours to solve the TSP-SD for each instance. The first three were solved to optimality, but the remaining eight reached the time limit. On average, the first valid solution was improved by 9.75%, but the improvement decreases with increasing instance size. In the case of the three largest instances, the improvement is only 1%. This experiment only confirms the expectation of poor scalability when using an exact approach in an optimization problem due to its exhaustive nature. Unlike in the previous experiment, the berlin52-10.4 variant was actually easier to solve when addressing the optimization problem, as the backward search tree is presumably pruned more with a lower AVD.

Table 2, column C, depicts the benefit of using the greedy expansion in the backward search. The computation times needed to find the first valid solution are slightly, but consistently better than with the default expansion. More importantly, the objective values are frequently more than ten times better than with the default expansion, which is a considerable improvement brought by a simple heuristic rule. On average, the first valid solutions found with the greedy expansion are better by 56% than with the default expansion. The improvement increases with increasing instance size and is around 90% for the four largest instances. As shown in Table 2, column D, increasing the time budget to 12 hours further improves the objective by 6% on average relative to the first valid greedy solutions. Similarly to random expansion, this improvement decreases with increasing instance size and is less than 1% for the largest instance.

Table 3, column A, presents the results of the heuristic solver alone on the TSP-SD. Each instance was solved 50 times with a time budget of  $10|V|$  seconds, e.g. 140 seconds for the burma14-3.1 instance. The optimal solution was found for the two smallest instances. However, the solver cannot find a valid solution every time and fails entirely to provide any valid solutions in all 50 runs for the berlin52-10.4 instance. In terms of solution quality, the best solutions found by the heuristic solver alone are worse by 26% on average than the first valid solutions found by the greedy exact solver. Furthermore, the mean success rate is only 62%. The heuristic solver is expected to converge faster than the exact solver, but presumably spends a large portion of the time budget on finding a valid initial solution instead. This assumption is confirmed in Table 3, column B, where the heuristic solver is initialized with the first valid solution found by the exact solver (Table 2, column C). Here, the

<sup>1</sup>All datasets and codes are publicly available at **blinded**



Component	Value
metaheuristic	basicVNS ( $k_{min} = 7, k_{max} = 10$ )
construction	nearestNeighbor
perturbation	randomMoveAll ( $allowInfeasible = true$ )
local search	pipeVND ( $firstImprove = true$ )
operators	centeredExchange ( $p \in \{1, 2, 3, 5\}$ ), moveAll ( $p \in \{2, 10\}$ ) relocate( $p \in \{1, 2, 3, 4, 5\}$ ), exchangeIds exchange( $p, q \in \{(1, 2), (2, 4), (3, 4)\}$ ) reverseExchange( $p, q \in \{(1, 2), (2, 2), (3, 3), (3, 4), (4, 4)\}$ )

Table 1: Heuristic solver - tuned configuration

Expansion Stop condition Instance ↓	default				greedy			
	first valid obj.	time (s)	12 hours obj.	time (s)	first valid obj.	time (s)	12 hours obj.	time (s)
burma14-3.1	55	<0.01	52	<0.01	52	<0.01	52	<0.01
ulysses22-5.5	174	<0.01	141	0.02	173	<0.01	141	0.02
berlin52-10.4	33388	0.37	23866	2942	29302	0.16	23866	2858
berlin52-13.2	28470	0.03	19417	43200	18461	<0.01	17938	43200
eil101-27.5	3447	0.10	3128	43200	1715	0.01	1642	43200
gr202-67.3	3073	0.48	2954	43200	934	0.08	862	43200
lin318-99.3	576916	1.43	560322	43200	116719	0.25	115058	43200
f417-160.6	510858	3.23	493671	43200	31387	1.05	29747	43200
d657-322.7	872446	8.85	860343	43200	98599	4.41	93668	43200
rat783-481.4	174085	14.30	172727	43200	15652	8.39	15300	43200
vm1084-848.9	8616499	45.46	8527195	43200	349923	35.81	348304	43200
	A		B		C		D	

Table 2: TSP-SD optimization results - exact solver

best solutions found by the warm-started heuristic solver in  $10|V|$  seconds are better by 5% on average than those obtained by the greedy exact solver in 12 hours and by 11.3% than the first valid solutions. Most importantly, the improvement does not decrease with increasing instance size and is consistent across the entire dataset. The previous two experiments reveal the drawbacks of both approaches: the exact solver scales poorly in the optimization problem, whereas the penalty-based heuristic solver does not provide a valid solution reliably. On the other hand, the exact solver provides valid solutions to all instances very fast, and the heuristic solver is much better at refining good-quality solutions. Therefore, using both solvers sequentially, i.e., implementation of a warm start optimization, combines the advantages of both.

Table 4 illustrates the benefit of relaxing TSP-SD to weak TSP-SD. Every solution to the TSP-SD is also valid for the weak TSP-SD, but the weak formulation might yield a better optimal value. On the other hand, the fitness evaluation in weak TSP-SD calculates the shortest paths  $p_i$  instead of reading the edge weights. Thus, the time complexity of the evaluation is higher by  $\mathcal{O}(|E|)$ , and the heuristic solver is drastically slower when solving the weak TSP-SD. The performance of the heuristic alone is shown in Table 4A. Regarding the success rate, the heuristic is significantly more successful than with TSP-SD, as the space of valid solutions in the weak TSP-SD formulation is much larger. In Table 4, column B, the best-known TSP-SD solution from the initialized heuristic solver (Table 3, column B) was used as an

initial solution. The experiment shows that only the TSP-SD solution of the smallest instance was not improved in the weak TSP-SD formulation. In the remaining instances, the weak TSP-SD solution is better by 7% on average than the best-known TSP-SD solution, so the relaxation is highly beneficial.

## 8 Conclusions

We introduce new variants of the Hamiltonian Cycle and the Travelling Salesperson Problems with self-deleting graphs, for which formal definitions, theoretical analyses and two solvers were proposed. In the future, we intend to investigate general heuristics for the proposed backward search. We also want to develop a new solver which works in the space of feasible solutions. Finally, we intend to study how to derive self-deleting graphs using motion planning techniques to determine which edges should be deleted.

## Acknowledgements

The research was supported by Czech Science Foundation Grant No. 23-05104S. The work of David Woller has also been supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS23/122/OHK3/2T/13. Computational resources were provided by the e-INFRA CZ project (ID:90140), supported by the Ministry of Education, Youth and Sports of the Czech Republic. Masoumeh Mansouri is a UK participant in Horizon Europe Project CONVINCENCE, and her work is supported by UKRI grant number 10042096.

Setup Instance ↓	heuristic only, $10 V $ seconds			exact init., $10 V $ seconds	
	min	mean±stdev	valid (%)	min	mean±stdev
burma14-3.1	52	52±0	100	52	52±0
ulysses22-5.5	141	144±8	47	141	166±5
berlin52-10.4	-	-	0	24456	25741±861
berlin52-13.2	18304	19192±648	40	17263	17835±277
eil101-27.5	1532	1728±87	51	1394	1513±55
gr202-67.3	1184	1352±87	78	812	849±11
lin318-99.3	189225	198324±8171	11	110698	110888±355
fl417-160.6	57686	68736±4830	95	27162	27259±140
d657-322.7	141030	150185±5227	100	85054	85347±162
rat783-481.4	21069	22078±619	100	13753	13833±115
vm1084-848.9	489491	513769±9452	100	325218	326067±503
	A			B	

Table 3: TSP-SD optimization results - heuristic solver

Setup Instance ↓	heuristic only, $10 V $ seconds			TSP-SD best init., $10 V $ seconds	
	min	mean±stdev	valid (%)	min	mean±stdev
burma14-3.1	52	52±0	100	52	52±0
ulysses22-5.5	129	129±1	100	129	129±0
berlin52-10.4	18701	20328±1174	100	18354	19740±480
berlin52-13.2	14579	15760±593	100	14838	16320±585
eil101-27.5	1313	1442±69	100	1240	1295±23
gr202-67.3	886	1060±122	100	779	790±2
lin318-99.3	135965	143259±5488	100	104422	104945±204
fl417-160.6	26035	26891±733	100	25976	26001±33
d657-322.7	96213	99730±1527	100	83402	83534±44
rat783-481.4	15072	15409±174	90	13599	13620±5
vm1084-848.9	352794	360779±3296	76	319335	319481±112
	A			B	

Table 4: weak TSP-SD optimization results - heuristic solver

## References

Aaron, E.; Krizanc, D.; and Meyerson, E. 2014. DMVP: foremost waypoint coverage of time-varying graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, 29–41. Springer.

Applegate, D. L.; Bixby, R. E.; Chvátal, V.; and Cook, W. J. 2011. *The traveling salesman problem*. Princeton university press.

Arkin, E. M.; Fekete, S. P.; and Mitchell, J. S. 2000. Approximation algorithms for lawn mowing and milling. *Computational Geometry*, 17(1-2): 25–50.

Blum, A.; Chalasani, P.; Coppersmith, D.; Pulleyblank, B.; Raghavan, P.; and Sudan, M. 1994. The minimum latency problem. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, 163–171.

Chan, D. 1993. Precedence constrained TSP applied to circuit board assembly and no wait flowshop. *The International Journal of Production Research*, 31(9): 2171–2177.

Dijkstra, E. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1: 269–271.

Dirac, G. A. 1952. Some theorems on abstract graphs. *Proceedings of the London Mathematical Society*, 3(1): 69–81.

Duarte, A.; Sánchez-Oro, J.; Mladenović, N.; and Todosi-jević, R. 2018. Variable Neighborhood Descent. In *Hand-*

*book of Heuristics*, 341–367. Springer International Publishing.

Escudero, L. F. 1988. An inexact algorithm for the sequential ordering problem. *European Journal of Operational Research*, 37(2): 236–249.

Grötschel, M.; Jünger, M.; and Reinelt, G. 1991. Optimal control of plotting and drilling machines: a case study. *Zeitschrift für Operations Research*, 35(1): 61–84.

Hansen, P.; Mladenović, N.; Brimberg, J.; and Pérez, J. A. M. 2019. Variable neighborhood search. In *Handbook of metaheuristics*, 57–97. Springer.

Helsgaun, K. 2017. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. Technical report, Roskilde: Roskilde University.

Komlós, J.; and Szemerédi, E. 1983. Limit distribution for the existence of Hamiltonian cycles in a random graph. *Discrete mathematics*, 43(1): 55–63.

Liao, C.-S.; and Huang, Y. 2014. The covering Canadian traveller problem. *Theoretical Computer Science*, 530: 80–88.

López-Ibáñez, M.; Dubois-Lacoste, J.; Pérez Cáceres, L.; Stützle, T.; and Birattari, M. 2016. The irace Package: Iterated Racing for Automatic Algorithm Configuration. *Operations Research Perspectives*, 3: 43–58.

- Mansouri, M.; Lagriffoul, F.; and Pecora, F. 2017. Multi vehicle routing with nonholonomic constraints and dense dynamic obstacles. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3522–3529. IEEE.
- Michail, O.; and Spirakis, P. G. 2016. Traveling salesman problems in temporal graphs. *Theoretical Computer Science*, 634: 1–23.
- Mikula, J.; and Kulich, M. 2022. Solving the traveling delivery person problem with limited computational time. *Central European Journal of Operations Research*, 1–31.
- Ore, O. 1960. Note on Hamilton Circuits. *The American Mathematical Monthly*, 67(1): 55–55.
- Pósa, L. 1976. Hamiltonian circuits in random graphs. *Discrete Mathematics*, 14(4): 359–364.
- Reinelt, G. 1991. TSPLIB—A traveling salesman problem library. *ORSA journal on computing*, 3(4): 376–384.
- Rubin, F. 1974. A search procedure for Hamilton paths and circuits. *Journal of the ACM (JACM)*, 21(4): 576–580.
- Sabar, N. R.; Bhaskar, A.; Chung, E.; Turky, A.; and Song, A. 2019. A self-adaptive evolutionary algorithm for dynamic vehicle routing problems with traffic congestion. *Swarm and evolutionary computation*, 44: 1018–1027.
- Sleegers, J.; and Berg, D. v. d. 2021. Backtracking (the) Algorithms on the Hamiltonian Cycle Problem. *arXiv preprint arXiv:2107.00314*.
- Ullrich, A.; Hertzberg, J.; and Stiene, S. 2014. ROS-Based Path Planning and Machine Control for an Autonomous Sugar Beet Harvester. In *Proceedings of International Conference on Machine Control & Guidance, (MCG-2014)*.
- Vandegriend, B. 1999. *Finding Hamiltonian cycles: algorithms, graphs and performance*. University of Alberta.
- Vandegriend, B.; and Culberson, J. 1998. The Gn, m phase transition is not hard for the Hamiltonian Cycle problem. *Journal of Artificial Intelligence Research*, 9: 219–245.
- Woller, D.; Hrazdíra, J.; and Kulich, M. 2022. Metaheuristic Solver for Problems with Permutative Representation. In *Intelligent Computing & Optimization*, 42–54. Cham: Springer International Publishing.
- Xiang, X.; Qiu, J.; Xiao, J.; and Zhang, X. 2020. Demand coverage diversity based ant colony optimization for dynamic vehicle routing problems. *Engineering Applications of Artificial Intelligence*, 91: 103582.