Leveraging Fixed-Parameter Tractability for Robot Inspection Planning

Yosuke Mizutani[1], Daniel Coimbra Salomao[1], Alex Crane[1], Matthias Bentert[2], Pål Grønås Drange[2], Felix Reidl[3], Alan Kuntz[1], and Blair D. Sullivan[1]

[1] University of Utah, USA
[2] University of Bergen, Norway
[3] Birkbeck, University of London, UK

Abstract. Autonomous robotic inspection, where a robot moves through its environment and inspects points of interest, has applications in industrial settings, structural health monitoring, and medicine. Planning the paths for a robot to safely and efficiently perform such an inspection is an extremely difficult algorithmic challenge. In this work we consider an abstraction of the inspection planning problem which we term Graph Inspection. We give two exact algorithms for this problem, using dynamic programming and integer linear programming. We analyze the performance of these methods, and present multiple approaches to achieve scalability. We demonstrate significant improvement both in path weight and inspection coverage over a state-of-the-art approach on two robotics tasks in simulation, a bridge inspection task by a UAV and a surgical inspection task using a medical robot.

1 Introduction

Inspection planning, where a robot is tasked with planning a path through its environment to sense a set of points of interest (POIs) has broad potential applications. These include the inspection of surfaces to identify defects in industrial settings such as car surfaces [3], urban structures [5], and marine vessels [14], as well as in medical applications to enable the mapping of subsurface anatomy [6,7] or disease diagnosis.

Consider the demonstrative medical example of diagnosing the cause of pleural effusion, a medical condition in which a patient's pleural space—the area between the lung and the chest wall—fills with fluid, collapsing the patient's lung [25,20,28]. Pleural effusion is a symptom, albeit a serious one, of one of over fifty underlying causes, and the treatment plan varies significantly, depending heavily on which of the underlying conditions has caused the effusion. To diagnose the underlying cause, physicians will insert an endoscope into the pleural space and attempt to inspect areas of the patient's lung and chest wall. Automated medical robots have been proposed as a potential assistive technology with great promise to ease the burden of this difficult diagnostic procedure. However, planning the motions to inspect the inside of a patient's body with a medical robot, or indeed *any environment with any robot* is an extremely challenging problem.

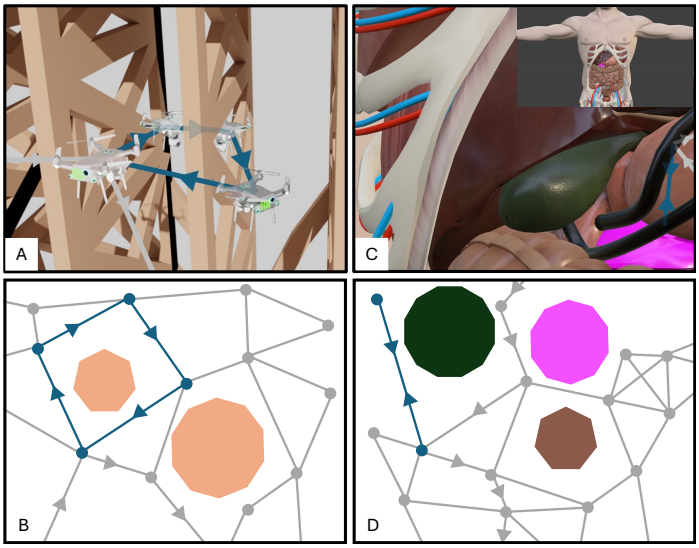

Fig. 1: Inspection planning, in contrast to traditional motion planning, may necessitate leveraging cycles and backtracking on graphs embedded in the robot's configuration space. This necessitates computing a walk (rather than a path) on a graph. (A) A quadrotor, while inspecting a bridge for potential structural defects, may need to circle around obstacles, (B) leveraging a cycle in its c-space graph (teal). (C) A medical endoscopic robot (black) may need to move into and then out of an anatomical cavity to, e.g., visualize the underside of a patient's gallbladder (green), (D) requiring backtracking in its c-space graph (teal).

Planning a motion for a robot to move from a single configuration to another configuration is, under reasonable assumptions, known to be PSPACE-hard [19]. Inspection planning extends this typical motion planning problem by requiring the traversal of multiple configurations. The planned route may need to include complexities such as tracing back to where the robot has already been and/or traversing circuitous routes. Consider Fig. 1, where examples are given of inspections that necessitate circuitous paths or backtracking during inspection. While the specific examples given are intuitive in the robots' workspaces, cycles and backtracking may be required in the c-space graph in ways that don't manifest intuitively in the workspace as well.

Further, it is almost certainly not sufficient to only consider the ability to inspect the POIs, but one must also consider the *cost* of the path taken to inspect them. This is because while inspecting POIs may be an important objective, it is not the only objective in real robotics considerations; unmanned aerial vehicles (UAVs) must operate within their battery capabilities, and medical robots must consider the time a patient is subjected to a given procedure.

The state-of-the-art in inspection planning, presented by Fu *et al.* [17] and named IRIS-CLI, casts this problem as an iterative process with two phases. In

the first phase, a rapidly-exploring random graph (RRG) [22] is constructed. In this graph, each vertex represents a possible configuration of the robot, edges indicate the ability to transition between configuration states, and edge weights indicate the cost of these transitions. Additionally, every vertex is labeled with the set of POIs which may be inspected by the robot when in the associated configuration state. In the second phase, a walk is computed in this graph with the dual objectives of (a) inspecting all POIs and (b) minimizing the total weight (cost) of the walk. By repeating both phases iteratively, Fu *et al.* are able to guarantee asymptotic optimality[4] of the resulting inspection plan.

In this work we focus on improving the second phase. We formulate this phase as an algorithmic problem on edge-weighted and vertex-multicolored graphs, which we call Graph Inspection (formally defined in Section 2). Graph Inspection is a generalization of the well-studied Traveling Salesperson (TSP) problem[5] [2]. As such, it is deeply related to the rich literature on "color collecting" problems studied by the graph algorithms community.

Graph Inspection is closely related to the Generalized Traveling Salesperson Problem (also known as Group TSP), in which the goal is to find a "Hamiltonian cycle visiting a collection of vertices with the property that exactly one vertex from each [color] is visited" [29]. If each vertex can belong to several color classes, the instance can be transformed into an instance of GTSP [24,13]. However, in the Graph Inspection problem, we do not demand that a vertex cannot be visited several times; indeed, we expect that to be the case for many real-world cases. Rice and Tsotras [30] gave an exact algorithm for GTSP in $\mathcal{O}^*(2^k)$ time[6], and although being inapproximable to a logarithmic factor [32], they gave an $O(r)$-approximation in running time $\mathcal{O}^*(2^{k/r})$ [31].

Two other related problems are the T-Cycle problem, and the Maximum Colored (s,t)-Path problem[7]. Björklund, Husfeldt, and Taslaman provided an $\mathcal{O}^*(2^{|T|})$ randomized algorithm for the T-Cycle problem, in which we are asked to find a (simple) cycle that visits all vertices in T [4]. In the Maximum Colored (s,t)-Path problem, we are given a vertex-colored graph G, two vertices s and t, and an integer k, and we are asked if there exists an (s,t)-path that collects at least k colors, and if so, return one with minimum weight. Fomin et al. [16] gave a randomized algorithm running in time $\mathcal{O}^*(2^k)$ for this problem. Again, in both of these problems a crucial restriction is the search for *simple* paths or cycles.

Though Graph Inspection is distinct from the problems mentioned above, we leverage techniques from this literature to propose two algorithms which can solve Graph Inspection optimally[8]. First, in Section 3.1 we show that while Graph Inspection is NP-hard (as a TSP generalization), a dynamic

[4] See [17] for specific definition of asymptotic optimality in their case.

[5] Given an edge-weighted graph and a start-vertex s, compute a minimum-weight closed walk from s visiting every vertex exactly once.

[6] The $\mathcal{O}^*$ notation hides polynomial factors.

[7] Also studied under the names Tropical Path [8] and Maximum Labeled Path [10].

[8] Note that in this case, and subsequently in the paper unless otherwise indicated, 'optimal' refers to an optimal walk on the given graph and is distinct from the asymptotic optimality guarantees provided in [17].

programming approach can solve our problem in $2^{|\mathcal{C}|} \cdot \text{poly}(n)$ time and memory, where $|\mathcal{C}|$ is the number of POIs. Additionally, we draw on techniques used in the study of Traveling Salesperson [9] to provide a novel integer linear programming (ILP) formulation, which we describe in Section 3.2.

To deal with the computational intractability of Graph Inspection, Fu *et al.* took the approach of relaxing the problem to *near optimality*, enabling them to leverage heuristics in the graph search to achieve reasonable computational speed when solving the problem. We take two approaches. Using the ILP, we show that on several practical instances drawn from [17], Graph Inspection can be solved almost exactly in reasonable runtime. However, we note the ILP will not directly scale to very large graphs. For the dynamic programming routine, our approach is more nuanced: First, we note that as the running time and memory consumption of this algorithm is exponential only in the number of POIs, while remaining polynomial in the size of the graph, it can optimally solve Graph Inspection when only a few POIs are present. This situation naturally arises in some application areas, particularly in medicine when the most relevant anatomical POIs may be few and known in advance. When the number of POIs is large, we adapt the dynamic programming algorithm into a heuristic by selecting several small subsets of POIs in a principled manner (see Section 4.1), running the dynamic program independently for each small subset, and then "merging" the resulting walks (see Section 4.2). Though our implementation is heuristic, it is rooted in theory: it is possible to combine dynamic programming with a "partition and merge" strategy such that, given enough runtime, the resulting walk is optimal (see Appendix A).

We demonstrate the practical efficacy of our algorithms on Graph Inspection instances drawn from two scenarios which were used to evaluate the prior state-of-the-art planner IRIS-CLI [17]. The first is planning inspection for a bridge using a UAV (the "drone" scenario), and the second is planning inspection of the inside of a patient's pleural cavity using a continuum medical robot (the "crisp" scenario). We implemented our algorithms, DP-IPA (Dynamic Programming) and ILP-IPA (ILP), where IPA stands for Inspection Planning Algorithm. We show (see Section 5 and Fig. 6) that on Graph Inspection instances of sizes similar to those used by [17], ILP-IPA produces walks with lower weight and higher coverage than those produced by IRIS-CLI. Indeed, ILP-IPA can produce walks with perfect coverage, even on much larger instances. However, for these larger instances DP-IPA provides a compelling alternative, producing walks with much lower weight, with some sacrifice in coverage.

In summary, this work takes steps toward the application of Graph Inspection as a problem formalization for inspection planning, and importantly provides (i) multiple novel algorithms with quality guarantees, (ii) an extensive discussion of methods used to implement these ideas in practice, and (iii) reusable software which outperforms the state-of-the-art on two relevant scenarios from the literature.

2 Preliminaries

All graphs $G = (V, E)$ in this work are undirected, unless explicitly stated otherwise. We denote the vertices and edges of G by V and E, respectively. When it is clear which graph is referenced from context, we write $n = |V|$ for the number of vertices and $m = |E|$ for the number of edges in the graph. An edge-weight function w is a function $w\colon E \to \mathbb{R}_{\geq 0}$. A (simple) *path* $P = v_1, v_2, \ldots, v_t$ in $G = (V, E)$ is a sequence of vertices such that for every $i < t$, it holds that $v_i v_{i+1} \in E$ and no vertex appears more than once in P. If we relax the latter requirement, we call the sequence P a *walk*. A walk is *closed* if it starts and ends in the same vertex.

The *weight* of a walk is the sum of weights of its edges. For vertices $u, v \in V$, we let $d(u, v)$ denote the *distance*, that is, the minimum weight of any path between u and v. For a set $S \subseteq V$, let $d(v, S) = d(S, v)$ be $\min_{u \in S} d(u, v)$. Moreover, we denote the graph induced by S by $G[S]$ and we use $G - S$ as a shorthand for $G[V \setminus S]$. When S only contains a single vertex v, then we also write $G - v$ instead of $G - \{v\}$. We refer to the textbook by Diestel [12] for an introduction to graph theory.

We use $\chi(v)$ to denote the *colors* (or *labels*) of a vertex (which, with nuance described below, correspond to POIs in the robot's workspace), and we write $\chi(S)$ to denote $\bigcup_{v \in S} \chi(v)$. For a set S, we write 2^S for the power set of S. We next define the main problem we investigate in this paper.

Graph Inspection

Input:	An undirected graph $G = (V, E)$, a set $\mathcal{C}$ of colors, an edge-weight function $w\colon E \to \mathbb{R}_{\geq 0}$, a coloring function $\chi\colon V \to 2^{\mathcal{C}}$, a start vertex $s \in V$, and an integer t.
Problem:	Find a closed walk $P = (v_0, v_1, \ldots, v_p)$ in G with $v_0 = v_p = s$ and $\lvert\bigcup_{i=1}^{p} \chi(v_i)\rvert \geq t$ minimizing $\sum_{i=1}^{p} w(v_{i-1}v_i)$.

Note that t is the minimum number of colors to collect. For the sake of simplicity, we may assume that G is connected, $t \leq |C|$, and $\chi(s) = \emptyset$.

Parameterized complexity. A *parameterized* problem is a tuple (I, k) where $I \in \Sigma^*$ is the input instance for a set of strings Σ^* and $k \in \mathbb{N}$ is a *parameter*. A parameterized problem is *fixed-parameter tractable* (FPT) if there exists an algorithm solving every instance (I, k) in $f(k) \cdot \text{poly}(|I|)$ time, where f is any computable function. We refer to the textbook by Cygan et al. [11] for an introduction to parameterized complexity theory. The goal of parameterized algorithms is to capture the exponential (or even more costly) part of the problem complexity within a parameter, making the rest of the computation polynomial in the input size.

3 Graph Search

In this section, we present two algorithms for solving Graph Inspection, along with strategies for finding upper and lower bounds on the optimal solution.

3.1 Dynamic Programming Algorithm

We begin by establishing that Graph Inspection is fixed-parameter tractable with respect to the number of colors by giving a dynamic programming algorithm we refer to as DP-IPA.

Theorem 1. *Graph Inspection can be solved in* $\mathcal{O}((2^{|\mathcal{C}|}(n+|\mathcal{C}|)+m+n\log n)n)$ *time, where* $n = |V|$, $m = |E|$, *and* $\mathcal{C}$ *is the set of colors.*

Proof. Recall that we may assume $\chi(s) = \emptyset$. Otherwise, we can collect all colors in $\chi(s)$ for free; removing the colors $\chi(s)$ from the coloring function and decreasing t by $|\chi(s)|$ gives an equivalent instance. We solve Graph Inspection using dynamic programming. First, we compute the all-pairs shortest paths of G in $\mathcal{O}(nm + n^2 \log n)$ by n calls of Dijkstra's algorithm ($\mathcal{O}(m + n \log n)$ time) using a Fibonacci heap. Note that the new distance function w' is complete and metric. Hence, we may assume that an optimal solution collects at least one new color in each step (excluding the last step where it returns to s). Hence, we store in a table $T[v, S]$ with $v \in V \setminus \{s\}$ and $S \subseteq \mathcal{C}$, where S contains at least one color in $\chi(v)$, the length of a shortest walk that starts in s, ends in v, and collects (at least) all colors in S. We fill T by increasing size of S by the recursive relation:

$$T[v, S] = \begin{cases} \infty & \text{if } S \cap \chi(v) = \emptyset \\ \min_{u \in V} T[u, S \setminus \chi(v)] + w'(uv) & \text{otherwise.} \end{cases}$$

Therein, we assume that $T[s, S] = 0$ if $S = \emptyset$ and $T[s, S] = \infty$, otherwise. We will next prove that the table is filled correctly. We do so via induction on the size of S. To this end, assume that T was computed correctly for all entries where the respective set S has size at most i. Now consider some entry $T[v, S]$ where $S \cap \chi(v) \neq \emptyset$ and $|S| = i + 1$. Let ℓ' be the value computed by our dynamic program and let opt be the length of a shortest walk that starts in s, ends in v, and collects all colors in S. It remains to show that $\ell' = \text{opt}$ and to analyze the running time. We first show that $\ell' \leq \text{opt}$. To this end, let $W = (s, v_1, \ldots, v_p = v)$ be a walk of length opt that collects all colors in S. If $p = 1$, then $S \subseteq \chi(v)$ and $\ell' \leq T[s, \emptyset] + w'(sv) = w'(sv)$. Moreover, the shortest path from s to v has length $w'(sv)$ and hence $\ell' \leq w'(sv) \leq \text{opt}$. If $p > 1$, then $W' = (s, v_1, \ldots, v_{p-1})$ is a walk from s to v_{p-1} that collects all colors in $S' = S \setminus \chi(v)$. Hence, by construction $T[v_{p-1}, S'] \leq \text{opt} - w'(v_{p-1}v)$ and hence $\ell' = T[v, S] \leq T[v_{p-1}, S'] + w'(v_{p-1}v) \leq \text{opt}$.

We next show that $\ell' \geq \text{opt}$. To this end, note that whenever $T[v, S]$ is updated, then there is some vertex u such that $T[v, S] = T[u, S \setminus \chi(v)] + w'(uv)$ (where possibly $u = s$ and $S \setminus \chi(v) = \emptyset$). By induction hypothesis, there is a walk

from s to u that collects all colors in $S \setminus \chi(v)$ of length $T[u, S \setminus \chi(v)]$. If we add vertex v to the end of this walk, we get a walk of length ℓ' that starts in s, ends in v, and collects all colors in S. Thus, opt $\leq \ell'$.

After the table is completely filled for all color sets S with $|S| \leq t$, then we just need to check whether there exists a vertex $v \in V \setminus \{s\}$ and a set S with $|S| = t$ such that $T[v, S] + w'(vs) \leq \ell$. Note that the number of table entries is $2^{|\mathcal{C}|}n$, computing one table entry takes $\mathcal{O}(n + |\mathcal{C}|)$ time, and the final check in the end takes $\mathcal{O}(2^{|\mathcal{C}|}n)$ time. Thus, the overall running time of our algorithm is in $\mathcal{O}(nm + n^2 \log(n) + 2^{|\mathcal{C}|}(n + |\mathcal{C}|)n) = \mathcal{O}((2^{|\mathcal{C}|}(n + |\mathcal{C}|) + m + n \log n)n)$. □

3.2 Integer Linear Programming Algorithm

In this section, we present ILP-IPA, an Integer Linear Programming (ILP) formulation of the problem, inspired by the flow-based technique for TSP [9]. As a (trivially checkable) precondition, we require that a solution walk includes at least two vertices. We observe that there is a simpler formulation if the input is a complete metric graph; however, in practice, creating the completion and applying this approach significantly degrades performance because of the runtime's dependence on the number of edges.

Our ILP formulation for Graph Inspection can be found in Fig. 2. Intuitively, the *flow amount* at a vertex encodes the number of occurrences of the vertex in a walk. Constraints (1a) and (1b) implement flow conditions, and (2a) and (2b) ensure that the flow originates at vertex s. An edge included in a solution and not touching s emits 2 *charges*, and the charges are distributed among the edge's endpoints. If every solution edge is part of a walk from s, then a charge consumption at each vertex can be slightly less than 2 per incoming flow. There are $\mathcal{O}(|\mathcal{C}| + m)$ constraints if $t = |\mathcal{C}|$, and $\mathcal{O}(|\mathcal{C}|m)$ constraints if $t < |\mathcal{C}|$.

Correctness. Before showing the correctness of the ILP formulation, we characterize solution walks for Graph Inspection. In the following, we view a solution walk as a sequence of directed edges. For a walk $P = v_0 v_1 \dots v_\ell$, we write $|P|$ for the number of edges in P, i.e. $|P| = \ell$, and $E(P)$ for the set of *directed* edges in the walk, i.e. $E(P) = \{v_{i-1}v_i \mid 1 \leq i \leq \ell\}$. We write $w(P)$ for the length of the walk, that is, $w(P) := \sum_{uv \in E(P)} w(u, v)$. We now prove a simple lemma.

Lemma 1. *For any feasible instance of* Graph Inspection*, there exists an optimal solution without repeated directed edges.*

Proof. Assume not, and let $P = sP_1uvP_2uvP_3s$ be a solution minimizing $|P|$. Consider a walk $P' = sP_1u\overline{P_2}vP_3s$, where $\overline{P_2}$ is the reversed walk of P_2. Since $|P'| < |P|$ and both visit the same set of vertices, we have that $w(P') > w(P)$ by our choice of P. However, $w(P') = w(sP_1u) + w(u\overline{P_2}v) + w(vP_3s) \leq w(sP_1u) + w(u, v) + w(vP_2u) + w(u, v) + w(vP_3s) = w(P)$, a contradiction. □

The following is a simple observation.

Variables & Objective

$x_{u,v}, x_{v,u} \in \{0,1\}$ for $uv \in E$	*$x_{u,v}$ is 1 iff the directed edge uv is present in the solution flow.*
$y_{e,v} \geq 0$ for $e \in E, v \in e$	*Amount of charges the edge e emits to its endpoint v.*
$z_c \in \{0,1\}$ for $c \in \mathcal{C}$	*Only used if $t < \lvert\mathcal{C}\rvert$. Value 1 iff color c is collected.*
$\min \sum_{uv \in E} w(u,v)(x_{u,v} + x_{v,u})$	*Minimize weighted cost of walk.*

Flow conditions

1a	$\sum_{u \in N(v)} x_{u,v} - x_{v,u} = 0$ for every $v \in V$	*Flow preservation.*
1b	$\sum_{u \in N(s)} x_{s,u} \geq 1$	*Outgoing flow at source s.*

Prohibit closed flow disjoint from s

2a	$y_{uv,u} + y_{uv,v} = 2(x_{u,v} + x_{v,u})$ for every $uv \in E(G - s)$	*Solution edges not incident to s emit charges.*
2b	$\sum_{u \in N(v) \setminus \{s\}} y_{uv,v} \leq \left(2 - \frac{2}{2n-3}\right) \cdot \sum_{u \in N(v)} x_{u,v}$ for every $v \in V \setminus \{s\}$	*Vertices can consume slightly less than two charges per incoming flow.*

Count number of collected colors

Case $t = |C|$

3a	$\sum_{v \in \chi^{-1}(c)} \sum_{u \in N(v)} x_{u,v} \geq 1$ for every $c \in C$	*Collect every color.*

Case $t < |\mathcal{C}|$

3b	$z_c \leq \sum_{v \in \chi^{-1}(c)} \sum_{u \in N(v)} x_{u,v}$ for every $c \in C$	*Walk must cross vertex of color c to collect c.*
3c	$\sum_{c \in C} z_c \geq t$	*Collect at least t colors.*

Fig. 2: ILP-IPA, an ILP for the Graph Inspection problem.

Observation 2. *Given an instance of* Graph Inspection*, there exists a closed walk P of length $\hat{w}$ visiting vertices $V' \subseteq V(G)$ if and only if there exists a connected Eulerian multigraph $G' = (V', E')$ such that $\hat{w} = \sum_{uv \in E'} w(uv)$, where E' is the multiset of the edges in P.*

This leads to a structural lemma about solutions.

Lemma 2. *For any feasible instance of* Graph Inspection*, there exists an optimal solution with at most* $2n-2$ *edges. This bound is tight.*

Proof. Let P be an optimal solution with the minimum number of edges. From Observation 2, we may assume there exists a connected Eulerian multigraph H that encodes P. Since H is connected, it has a spanning tree T as a subgraph. Let $H' = H - E(T)$. If H' contains a cycle C, then $H - C$ is also connected and Eulerian, as removing a cycle from a multigraph does not change the parity of the degree of each vertex. Hence, there exists a shorter solution P' that is an Eulerian tour in $H - C$, a contradiction. Knowing that both T and H' are acyclic, we have $|E(H)| = |E(T)| + |E(H')| \leq 2n-2$. This is tight whenever G is a tree where all leaves have a unique color. □

Now we are ready to prove the correctness of the ILP formulation.

Theorem 3. *The ILP formulation in Fig. 2 is correct.*

Proof. We show that we can translate a solution for Graph Inspection to a solution for the corresponding ILP and vise versa.

For the forward direction, let $P = v_0 v_1 \dots v_\ell$ with $v_0 = v_\ell = s$ be a solution walk with $\ell \geq 2$ collecting at least k colors. From Lemma 1, we may assume that there are no indices i, j such that $i < j$ and $v_i v_{i+1} = v_j v_{j+1}$. For constraint (1), we set $x_{u,v} = 1$ if $uv \in E(P)$ and 0 otherwise. It is clear to see that all flow conditions are satisfied. Moreover, observe that for any vertex $v \in V(G) \setminus \{s\}$, the flow amount $\sum_{u \in N(v)} x_{u,v}$ corresponds to the number of occurrences of v in P, which we denote by $\deg_P(v)$.

Next, if $|P| = 2$, then constraint (2) is trivially satisfied by setting $y_{e,v} = 0$ for all e, v. Otherwise, let P' be a continuous part of P such that s appears only at the beginning and at the end. Then, P' contains $|P'|-2$ edges that do not touch s and emit two charges each. We know that $|P'| - 1 = \sum_{v \in V(P') \setminus \{s\}} \deg_{P'}(v)$. For a directed edge $e \in E(P')$ and its endpoint $v \in e$, let $y^{(P')}_{e,v}$ be part of $y_{e,v}$ charged only by P'. We distribute the charges by setting $y^{(P')}_{v'_{i-1}v'_i, v'_{i-1}} = 2 - \frac{2i}{|P'|-1}$ and $y^{(P')}_{v'_{i-1}v'_i, v'_i} = \frac{2i}{|P'|-1}$ for every $1 \leq i < |P'|$, where $P' = v'_0 v'_1 \dots v'_{|P'|}$ with $v'_0 = v'_{|P'|} = s$.

Note that $\sum_{u \in N(v) \setminus \{s\}} y^{(P')}_{uv,v} = \deg_{P'}(v) \cdot \frac{2(|P'|-2)}{|P'|-1} = (2 - \frac{2}{|P'|-1}) \cdot \deg_{P'}(v) \leq (2 - \frac{2}{2n-3}) \cdot \deg_{P'}(v)$ for every $v \in V(P') \setminus \{s\}$. The last inequality is due to Lemma 2. This inequality still holds when we concatenate closed walks P' from s since $\sum_{u \in N(v) \setminus \{s\}} y_{uv,v} = \sum_{P'} \sum_{u \in N(v) \setminus \{s\}} y^{(P')}_{uv,v}$ and $\sum_{u \in N(v)} x_{u,v} = \sum_{P'} \deg_{P'}(v)$. Constraint (2) is now satisfied.

Finally, in order to collect colors $\chi(v)$, there most be an edge uv in the solution. Notice that constraint (3b) encodes this and constraint (3c) ensures that we collect at least k distinct colors. Finally, observe that the objective is properly encoded.

For the backward direction, we show that there cannot be a closed flow, i.e. *circulation*, avoiding s. For the sake of contradiction, let C be such a circulation. Then, since $x_{u,v} = 1$ for every $uv \in E(C)$, we have $\sum_{uv=e\in E(C)} y_{e,u} + y_{e,v} = 2|E(C)|$. This is considered as the total charge emitted from C, and it must be consumed by the vertices in C. We have $\sum_{v\in V(C)} \sum_{u\in N(v)} y_{uv,u} + y_{uv,v} \geq 2|E(C)|$, and by the pigeonhole principle, there must be a vertex $v \in V(C)$ such that its charge consumption is at least $\deg(v)$, violating constraint (2b). Hence, there must be a closed walk from s that realizes a circulation obtained by ILP. From constraint (3), the walk also collects at least k colors. □

Solution recovery. A closed walk in a multigraph is called an *Euler tour* if it traverses every edge of the graph exactly once. A multigraph is called *Eulerian* if it admits an Euler tour. It is known that a connected multigraph is Eulerian if and only if every vertex has even degree [15] and given an Eulerian multigraph with m edges, we can find an Euler tour in time $\mathcal{O}(m)$ [21].

Given a certificate of an optimal solution for the aforementioned ILP, we construct a solution walk as follows. First, let D be the set of directed edges uv such that $x_{u,v} = 1$. Next, we find an Euler tour P starting from s using all the edges in D. Then, P is a solution for Graph Inspection.

3.3 Upper and Lower Bounds

When evaluating solutions, having upper and lower bounds on the optimal solution provides useful context. For Graph Inspection, a polynomial-time computable lower bound follows directly from the LP relaxation of the ILP in Section 3.2. For an upper bound, we consider Algorithm ST (Algorithm 1), which uses a 2-approximation algorithm for Steiner Tree[9] [23] as a subroutine. The algorithm proceeds by first choosing the vertices closest to s collecting t colors and then finding a Steiner tree of those vertices. A closed walk can be obtained by using each edge of the Steiner tree twice.

Algorithm 1: Algorithm ST

```
S ← {s}
while |χ(S)| < t do
    // Choose the vertex with a new color closest to s.
    S ← S ∪ {arg min_{v∈V} d(s, v) | χ(v) \ χ(S) ≠ ∅}
Compute a 2-approximation T for Steiner Tree on G with terminals S
Construct a closed walk from s using the all edges in T
```

Theorem 4. Algorithm ST *returns a closed walk collecting at least t colors with length at most $t \cdot \mathrm{opt}$, where* opt *denotes the optimal walk length. The algorithm runs in time $\mathcal{O}(tm \log(n + t))$.*

[9] The Steiner Tree problem takes a graph G and a set of vertices S (called *terminals*) and asks for a minimum-weight tree in G that spans S.

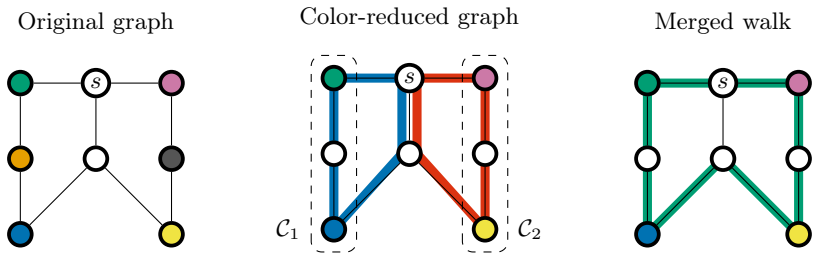

Fig. 3: An illustration of the partition-and-merge framework. The color set in the original graph (left) is reduced to 2 color sets $\mathcal{C}_1$ and $\mathcal{C}_2$, each of which contains 2 colors (middle). For each color set, we find an optimal walk collecting all colors in the set, resulting in the blue and red walks. Those walks are merged into the green walk, collecting the same colors in the color-reduced graph (right).

Proof. Since the algorithm returns a walk including all vertices in S, it collects at least t colors. Let $d_c = \min_{v \in \chi^{-1}(c)} d(s, v)$ for every $c \in \mathcal{C}$. Then, let $\tilde{d}$ be the t-th smallest such value, and due to Steps 1-3, for every $u \in S$, we have $d(s, u) \leq \tilde{d}$. Since $|S| \leq t + 1$, the weight of the minimum Steiner tree is at most $t\tilde{d}$, which results in that the length ℓ' of the walk returned by our algorithm is at most $2t\tilde{d}$. Now, suppose that P is an optimal walk of length opt collecting at least t colors $\mathcal{C}'$. Then, it is clear to see that $\mathrm{opt} \geq 2 \cdot d_c$ for any $c \in \mathcal{C}'$. From $|\mathcal{C}'| \geq t$, we have $\mathrm{opt} \geq 2\tilde{d}$, which implies $\ell' \leq t \cdot \mathrm{opt}$.

We next analyze the running time. Steps 1-3 takes $\mathcal{O}(m \log n + tn \log t)$ time for sorting vertices and computing the union of colors. Step 4 can be done by computing the transitive closure on S, which takes $\mathcal{O}(tm \log n)$ time. Step 5 takes $\mathcal{O}(n + m)$ time, so the overall running time is in $\mathcal{O}(tm \log(n + t))$.

Lastly, we show that this bound cannot be smaller. Let G be a star $K_{1,t+1}$ with s being the center with no colors. One leaf u has t colors, and each of the other t leaves has a unique single color. Every edge has weight 1. The optimal walk is (s, u, s) and has length 2, whereas the algorithm may choose $V \setminus \{u\}$ as S. This gives a walk of length $2t$. □

4 Graph Simplification

This section introduces strategies for transforming our exact algorithms into heuristics with improved scalability, including principled sub-sampling of colors (POIs) and creating plans by merging walks which inspect different regions of the graph. Fig. 3 illustrates this idea, the partition-and-merge framework.

4.1 Color Reduction

The algorithms of Sections 3.1 and 3.2 give us the ability to exactly solve Graph Inspection, but the running time is exponential in the number of colors (i.e., POIs). In some applications, this may not be prohibitive. In surgical robotics, a

Algorithm 2: GreedyMD.

Input : $\mathcal{C}, \chi_0, f$, and a positive integer $k \leq |\mathcal{C} \setminus \chi_0|$.
Output : $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| = k$.

$\mathcal{C}' \leftarrow \chi_0$. `// Every walk collects the colors visible from` s.
while $|\mathcal{C}'| < k + |\chi_0|$ **do**
  `// Choose the most dissimilar color.`
  $\mathcal{C}' \leftarrow \mathcal{C}' \cup \{\text{argmax}_{c \in \mathcal{C}} \min_{c' \in \mathcal{C}'} f(c, c')\}$
$\mathcal{C}' \leftarrow \mathcal{C}' \setminus \chi_0$
return $\mathcal{C}'$.

doctor may identify a small number of POIs which must be inspected to enable surgical intervention. However in other settings, it is unrealistic to assume that the number of colors is small. In the datasets we explore in Section 5 for instance, the POIs are drawn from a mesh of the object to be inspected, and we have no a priori information about the relative importance of inspecting individual POIs.

To deal with this challenge, we find a "representative" set $\mathcal{C}' \subseteq \mathcal{C}$ of colors, with $|\mathcal{C}'| = k$ small enough that our FPT algorithms run efficiently on the instance in which vertex colors are defined by $\chi(v) \cap \mathcal{C}'$ for each vertex v. We can then find a minimum-weight walk P on the color-reduced instance and reconstruct the set of inspected colors by computing $\bigcup_{v \in P} \chi(v)$.

Formally, we assume that there exists some function $f \colon \mathcal{C}^2 \to \mathbb{R}_{\geq 0}$ which encodes the "similarity" of colors, that is, for colors c_1, c_2, c_3, if $f(c_1, c_2) < f(c_1, c_3)$, then c_1 is more similar to c_2 than it is to c_3. In this paper, the function f is always a Euclidean distance, but we emphasize that our techniques apply also to other settings. For example, one may imagine applications in which POIs are partitioned into categorical *types*, and it is desirable that some POIs of each type are inspected. In this case, one could define f as an indicator function which returns 0 or 1 according to whether or not the input colors are of the same type. The core idea behind our methods is to select a small set of colors having *maximum dispersal*, meaning that as much as possible, every color in $\mathcal{C}$ should be highly similar (according to the function f) to at least one representative color in $\mathcal{C}'$.

We evaluate four algorithms for this task. The baseline (which we call Rand) selects colors uniformly at random. This is the strategy employed by IRIS-CLI when needed [17]. The second (called GreedyMD—MD for Maximum Dispersal) is a greedy strategy based on the Gonzalez algorithm for k-center [18]; this algorithm is described in more detail in Algorithm 2, where we set $\chi_0 = \chi(s)$. The final two algorithms (MetricMD, OutlierMD) are modified versions of this strategy. The interested reader is referred to Appendix B for a detailed description and the results of our comparative study. We note that all of our algorithms outperform the baseline Rand in terms of the resulting coverage. We perform our final comparisons (see Section 5) using GreedyMD for color reduction.

4.2 Merging Walks

When using DP-IPA or ILP-IPA on a color-reduced graph, the computed walk is minimum weight for the reduced color set, but the corresponding walk in the

original graph may not collect many additional colors. To increase the coverage in the original graph, we merge two or more walks into a single closed walk. Now, the challenge is how to keep the combined walk short. Suppose we have a collection of W solution walks $\{P_i\}$ and want a combined walk P that visits all vertices in $\bigcup_i V(P_i)$. We model this task as the following problem[10].

Minimum Spanning Eulerian Subgraph

Input: A loopless connected Eulerian multigraph G and edge weights $w : E \to \mathbb{R}_{\geq 0}$, where E denotes the edges in G's underlying simple graph.

Problem: Find a spanning subgraph G' of G such that G' is a connected Eulerian multigraph minimizing the weight sum, *i.e.* $\sum_{e \in E(G')} w(e)$.

We showed in Section 3.2 (see Observation 2) that each solution walk for Graph Inspection is an Eulerian tour in a multigraph. We hence use these two characterizations interchangeably. Unfortunately, this problem is NP-hard as we can see by reducing from Hamiltonian Cycle, which asks to find a cycle visiting all vertices in a graph. Given an instance G with n vertices of Hamiltonian Cycle, we duplicate all the edges in G so that the graph becomes Eulerian. If we set a unit weight function w for $E(G)$, i.e. $w(e) = 1$ for every $e \in E(G)$, then G has a Hamiltonian cycle if and only if (G, w) has a spanning subgraph of weight n.

In this paper, we propose and evaluate three simple heuristics for Minimum Spanning Eulerian Subgraph: ConcatMerge, GreedyMerge, and ExactMerge. ConcatMerge simply concatenates all walks. Since all walks start and end at vertex s, their concatenation is also a closed walk. In Appendix A, we give an algorithm which uses ConcatMerge and solves Graph Inspection optimally. We implemented a simplified version which is better by a factor of n in both running time and memory usage. GreedyMerge, detailed in Appendix C, is a polynomial-time heuristic including simple preprocessing steps for Minimum Spanning Eulerian Subgraph. At a high level, GreedyMerge builds a minimum spanning tree and removes as many redundant cycles as possible from the rest. ExactMerge is an exact algorithm using the ILP formulation for Graph Inspection.

Algorithm ExactMerge**.** We construct an instance of Graph Inspection by taking the underlying simple graph of the instance G of Minimum Spanning Eulerian Subgraph. We pick an arbitrary vertex $s \in V(G)$ as the starting vertex, and set unique colors to the other vertices. After formulating the ILP for Graph Inspection with $t = n - 1$ (collecting all colors) as in Section 3.2, we add the following constraints: $x_{u,v} + x_{v,u} \leq 1$ for every edge $uv \in E(G)$ with multiplicity 1. Lastly, we map a solution for the ILP to the corresponding multigraph. This multigraph should be spanning as we collect all colors in Graph

[10] An underlying simple graph of a multigraph is obtained by deleting loops and replacing multiedges with single edges.

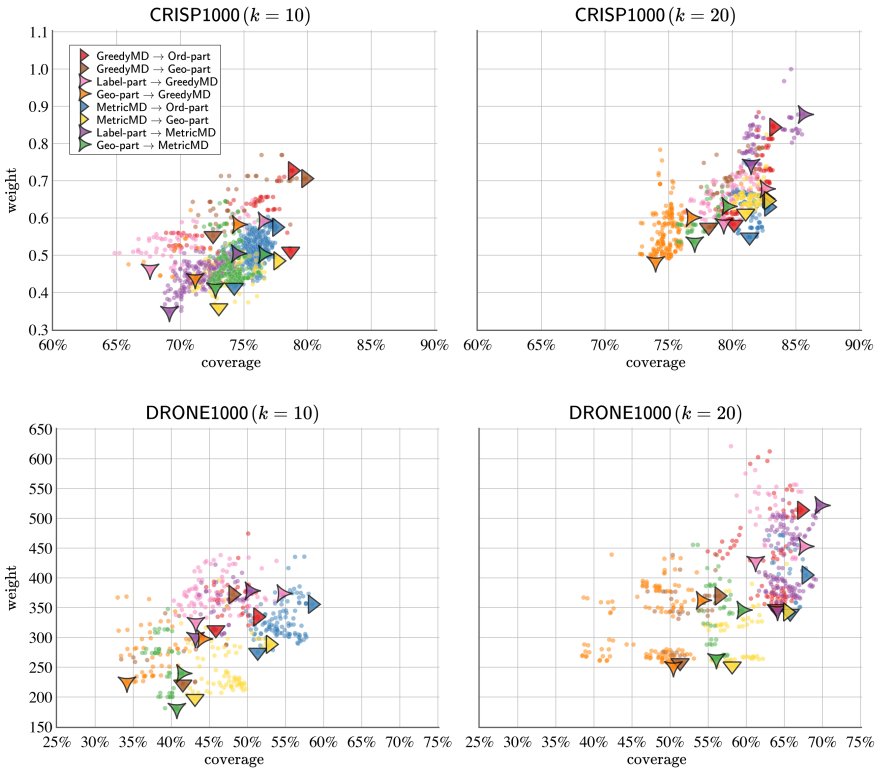

Fig. 4: Selected results of color partitioning experiment on datasets CRISP1000 and DRONE1000 with $k \in \{10, 20\}$. Each data point represents a solution computed using DP-IPA and ExactMerge. For each combination of color reduction method and partitioning strategy, we highlight the solutions with maximum coverage (rightward arrow) and with minimum weight (downward arrow).

Inspection, and it is by definition Eulerian. If each optimal solution contains at most $2n-2$ edges (which we have already shown; see Lemma 2) and W is constant, then our ILP formulation has $\mathcal{O}(n)$ variables. Thus (unlike the original instance of Graph Inspection) we can often quickly solve the walk merging problem exactly.

4.3 Partitioning Colors

Because we want to combine multiple ($W > 1$) walks to form our solution, it is useful to first partition the colors. This way, each independently computed walk collects (at least) some disjoint subset of colors. We propose two algorithms. The first (which we call Ord-part, short for *ordered* partitioning) takes an ordered color set $\mathcal{C}$ as input and partitions it sequentially, i.e., by selecting the first $|\mathcal{C}|/W$ colors as one subset, the second $|\mathcal{C}|/W$ colors as another, and so on. The second (called Geo-part, short for *geometric partitioning*) executes GreedyMD

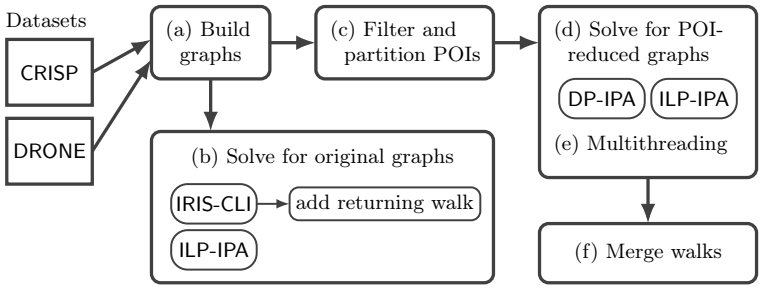

Fig. 5: Overview of our experiment pipeline.

with parameter W, and then partitions $\mathcal{C}$ by assigning each color to the most similar (according to the function f) of the W selected "representatives".

We also tested whether to perform color partitioning *before* or *after* color reduction. In the former case, the full color set $\mathcal{C}$ is partitioned by one of the algorithms described above[11], and then color reduction is performed on each subset. In the latter, color reduction is performed to obtain $W \cdot k$ colors, and then these colors are partitioned into W sets of size k using one of the algorithms described above. In Fig. 4 we display the results of our partitioning experiments on the instances used in [17], one for a surgical inspection task (CRISP1000) and another for a bridge inspection task (DRONE1000); results for extended datasets are deferred to Fig. 11. We now draw attention to two trends. First, we note that while using Ord-part before color reduction performs well in terms of coverage, particularly for the larger k values, we believe that this result is confounded somewhat by non-random ordering of the POIs in the input data. That is, we conjecture that the POIs arrive in an order which conveys some geometric information. Second, we note that while MetricMD seems to outperform GreedyMD (in terms of coverage) as a color reduction strategy for DRONE1000 with $k = 10$, this effect is lessened when $k = 20$. We believe that this trend is explainable, as for small k values the greedy procedure may select *only* peripheral POIs, while a larger k enables good representation of the entire space, including a potentially POI-dense "core" of the surface to be inspected. Given the complexity of the comparative results presented in Fig. 4, we favor the simplest, most generalizable, and most explainable strategy. For this reason, the experiments of Section 5 are performed using GreedyMD to reduce colors before partitioning using Ord-part.

5 Empirical Evaluation

To assess the practicality of our proposed algorithms, we ran extensive experiments on a superset of the real-world instances used in [17]. Fig. 5 shows an overview of the experiment pipeline. We first built RRGs using IRIS-CLI, originating from the

[11] In Fig. 4, Ord-part is referred to as Label-part when it is performed before color reduction, to emphasize that in this case the partitioning is based on the (potentially not random) sequence of POI labels given as input.

CRISP and DRONE datasets (Fig. 5 (a)). We tested IRIS-CLI and ILP-IPA on these instances with no additional color reduction. As IRIS-CLI iteratively outputs an s-t walk for some vertex $t \in V(G)$, we completed each walk with the shortest t-s path[12] to ensure a fair comparison while still giving as much credit as possible to IRIS-CLI (Fig. 5 (b)). For DP-IPA, we filter and partition POIs to obtain 3 sets of k POIs, where $k = 10, 20$ (Fig. 5 (c)). Then, we ran DP-IPA to exactly solve Graph Inspection for POI-reduced instances. In addition, we ran ILP-IPA for comparing color reduction/partitioning algorithms (Fig. 5 (d)) and measured speedups of those algorithms with different number of threads (Fig. 5 (e)). Lastly, we merged the walks using our algorithms to construct a "combined" closed walk (Fig. 5 (f)). Here we define the "search time" for the combined walk as the total of the search times of single-run walks plus the time taken for merging walks[12]. Except in experiment (e), we set the time limit of each algorithm to 900 seconds (15 minutes), and used 80 threads for DP-IPA and ILP-IPA.

We tested on four Graph Inspection instances, two of which replicate the instances used in [17]. The first dataset, CRISP, is a simulation for medical inspection tasks of the Continuum Reconfigurable Incisionless Surgical Parallel (CRISP) robot [1,26]. The dataset simulates a scenario segmented from a CT scan of a real patient with a pleural effusion—a serious medical condition that can cause the collapse of a patient's lung. The second, DRONE, is an infrastructure inspection scenario, in which a UAV with a camera is tasked with inspecting the critical structural features of a bridge. Its inspection points are the surface vertices in the 3D mesh model of a bridge structure used in [17]. To match the experiments in [17], we used IRIS-CLI to build RRGs with $n_{\text{build}} = 1000$ and, for CRISP, uniformly randomly selected 4200 POIs. We call these instances CRISP1000 and DRONE1000. Also, for each dataset, we built RRGs with $n_{\text{build}} = 2000$ (denoted CRISP2000 and DRONE2000). Appendix D details our graph instances and experiment environment. Code and data to replicate all experiments are available at [27].

Comparison to IRIS-CLI. First we compare the overall performance of our proposed algorithms to that of IRIS-CLI. In this experiment, we ran IRIS-CLI with all original instances, ILP-IPA with all original instances and $t = \frac{i}{10} \cdot |\mathcal{C}|$ for $5 \leq i \leq 10$, and DP-IPA with POI-reduced instances accompanied by walk-merging strategies, GreedyMD (MetricMD in Appendix Fig. 10), Ord-part *after* color reduction, and ExactMerge with $k \in \{10, 20\}$. We additionally computed the upper and lower bounds from Section 3.3 for all possible k values.

Fig. 6 plots the coverage and weight of each solution obtained within the time limit. IRIS-CLI achieved around 87% coverage on both CRISP instances. ILP-IPA outperformed IRIS-CLI on CRISP1000 by providing (i) for $t = 0.8 \cdot |\mathcal{C}|$, slightly better coverage paired with a 30% reduction in weight, and (ii) for $t = |\mathcal{C}|$, perfect coverage with only a 16% increase in weight. On CRISP2000, ILP-IPA failed to find a solution except with $t = |\mathcal{C}|$. Meanwhile, DP-IPA was competitive with IRIS-CLI, finding walks with moderate reductions in weight at the expense of slightly reduced coverage (83%). The differences between IRIS-CLI and our algorithms are

[12] The time taken for augmenting and merging walks was negligible.

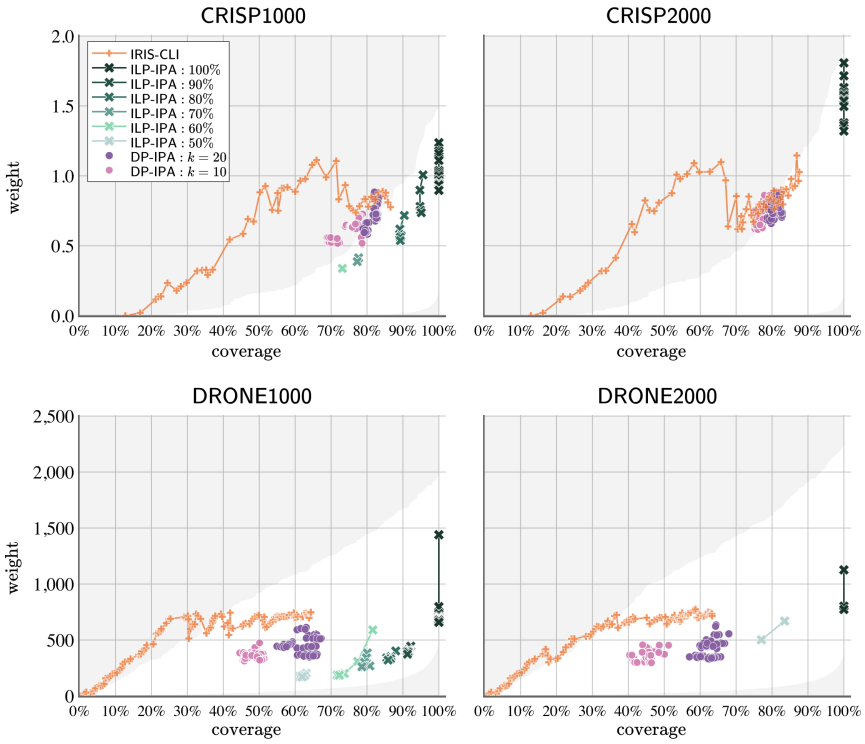

Fig. 6: Performance of IRIS-CLI, ILP-IPA and DP-IPA (with GreedyMD) on DRONE and CRISP benchmarks. Each data point represents a computed inspection plan; coverage is shown as a percentage of all POIs in the input graph. The area shaded in gray is outside the upper/lower bounds given in Section 3.3.

more significant on DRONE, where DP-IPA (with $k = 20$) outperformed IRIS-CLI by providing more coverage (68% vs. 64%) while reducing weight by over 50%. ILP-IPA outperformed IRIS-CLI by even larger margins on DRONE1000, but did not produce many solutions within the time limit on DRONE2000.

To summarize, ILP-IPA is the most successful on smaller instances, and works with various values of t. With larger graphs, ILP-IPA is more likely to time out when $t < |\mathcal{C}|$ (as described in Section 3.2, the ILP formulation in this case is more involved). DP-IPA is more robust on larger instances, outperforming IRIS-CLI in terms of solution weight while providing similar coverage.

Upper and lower bounds. First, we observe that the curvatures of upper bounds (recall Section 3.3) are quite different in CRISP and DRONE. We believe the geometric distribution of POIs explains this difference; with CRISP, the majority of POIs are close to the POIs seen at the starting point, which leads to concave upper-bound curves. DRONE, on the other hand, exhibits a linear

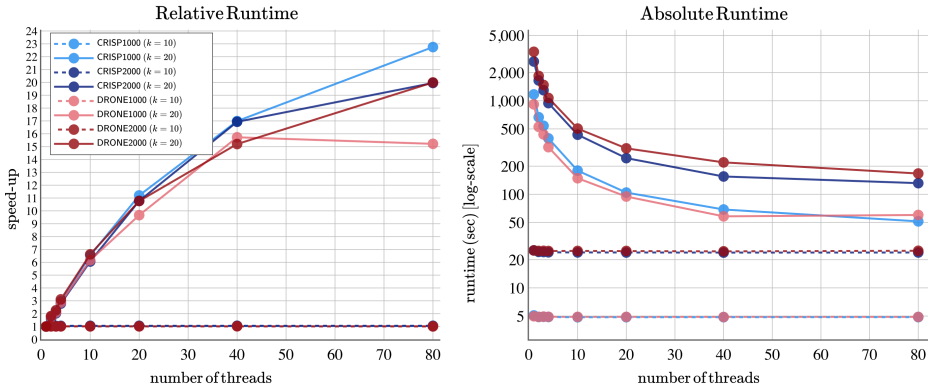

Fig. 7: Relative (left) and absolute (right) runtimes for DP-IPA with 1-80 threads. The relative runtime (speed-up) is with respect to a single thread.

trend in upper bounds because POIs are more evenly distributed in the 3D space, and the obtained solutions are far from these bounds. On the lower bound side, we obtain little insight on CRISP but see that on DRONE, it allows us to get meaningful bounds on the ratio of our solution's weight to that of an optimal inspection plan. For example, the lower bounds with $t = |\mathcal{C}|$ for DRONE1000 and DRONE2000 are 466.74 and 364.72, respectively. The best weights by ILP-IPA are 658.35 and 773.39, giving approximation ratios of 1.4 and 2.1 (respectively).

Multithreading Analysis. Our implementations of DP-IPA and ILP-IPA both allow for multithreading, but IRIS-CLI cannot be parallelized without extensive modification (i.e., the algorithm is inherently sequential). Fig. 7 illustrates order-of-magnitude runtime improvements for DP-IPA when using multiple cores. Analogous results for ILP-IPA are deferred to Appendix D.1.

Empirical Analysis of Walk-Merging Algorithms. The results reported in Fig. 6 are all computed using ExactMerge for walk merging. We also evaluated the effectiveness of ConcatMerge and GreedyMerge in terms of both runtime and resulting (merged) walk weight. We observed that while GreedyMerge is a heuristic, it produces walks of nearly optimal weight with negligible runtime increase as compared to ConcatMerge. Meanwhile, the runtime of ExactMerge was always within a factor of two of ConcatMerge; given the small absolute runtimes ($<$0.1 seconds in all cases), we chose to proceed with ExactMerge to minimize the weight of the merged walk. Complete experimental results are shown in Appendix D.

Comparing ILP-IPA and DP-IPA. In this work we have contributed two new Graph Inspection solvers, namely DP-IPA and ILP-IPA. We conclude this section with a brief discussion of their comparative strengths and weaknesses.

As discussed previously, for small graphs (e.g., CRISP1000 and DRONE1000) ILP-IPA is clearly the best option, as it provides higher coverage with less weight. However, DP-IPA performs better when the graph is large. In particular, if in some application minimizing weight is more important than achieving perfect coverage, then DP-IPA is preferable in large graphs. One might ask whether this trade-off (between weight and coverage) can also be tuned for ILP-IPA by setting $t < |\mathcal{C}|$, but we emphasize that in practice this choice significantly increases the runtime of ILP-IPA, such that it is impractical on large graphs. This is clear from the data presented in Fig. 6, and is also detailed in Appendix D.2. We observe that ILP-IPA becomes less competitive with DP-IPA as n and k grow.

6 Conclusion

In this work, we took tangible and meaningful steps toward mapping the Graph Inspection planning problem in robotics to established problems (e.g. Generalized TSP). We presented two algorithms, DP-IPA and ILP-IPA, to solve the problem under this abstraction, based on dynamic programming and integer linear programming. We presented multiple strategies for leveraging these algorithms on relevant robotics examples lending insight into the choices that can be made to use these methods in emerging problems. We then evaluated these methods and strategies on two complex robotics applications, outperforming the state of the art.

Our approach of creating several reduced color sets and merging walks offers a new paradigm for leveraging algorithms whose complexity has high dependence on the number of POIs, and opens the door for future exploration. We plan to see how these methods perform and scale with more than three walks. Further, it remains to implement these algorithms on real-world, physical robots and inspection tasks.

Acknowledgements

This work was supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 819416), by the Gordon & Betty Moore Foundation's Data Driven Discovery Initiative (award GBMF4560 to Blair D. Sullivan), and by the National Science Foundation (award ECCS-2323096 to Alan Kuntz).

References

1. Anderson, P.L., Mahoney, A.W., Webster, R.J.: Continuum reconfigurable parallel robots for surgery: Shape sensing and state estimation with uncertainty. IEEE robotics and automation letters **2**(3), 1617–1624 (2017)
2. Applegate, D.L., Bixby, R.E., Chvatál, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study, vol. 17. Princeton University Press (2006)
3. Atkar, P.N., Greenfield, A., Conner, D.C., Choset, H., Rizzi, A.A.: Uniform coverage of automotive surface patches. The International Journal of Robotics Research **24**(11), 883–898 (2005)

4. Björklund, A., Husfeldt, T., Taslaman, N.: Shortest Cycle Through Specified Elements. In: Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 1747–1753. Society for Industrial and Applied Mathematics (Jan 2012)
5. Cheng, P., Keller, J., Kumar, V.: Time-optimal UAV trajectory planning for 3D urban structure coverage. In: 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 2750–2757 (2008)
6. Cho, B.Y., Hermans, T., Kuntz, A.: Planning Sensing Sequences for Subsurface 3D Tumor Mapping. In: 2021 International Symposium on Medical Robotics (ISMR). pp. 1–7 (Nov 2021)
7. Cho, B.Y., Kuntz, A.: Efficient and Accurate Mapping of Subsurface Anatomy via Online Trajectory Optimization for Robot Assisted Surgery. In: IEEE International Conference on Robotics and Automation (ICRA). pp. 15478–15484 (May 2024)
8. Cohen, J., Italiano, G.F., Manoussakis, Y., Thang, N.K., Pham, H.P.: Tropical paths in vertex-colored graphs. Journal of Combinatorial Optimization **42**(3), 476–498 (Oct 2021)
9. Cohen, N.: Several Graph problems and their Linear Program formulations. Research report, INRIA (2019)
10. Couëtoux, B., Nakache, E., Vaxès, Y.: The Maximum Labeled Path Problem. Algorithmica **78**(1), 298–318 (May 2017)
11. Cygan, M., Fomin, F.V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: Parameterized algorithms, vol. 4. Springer (2015)
12. Diestel, R.: Graph theory, Graduate Texts in Mathematics, vol. 173. Springer-Verlag, Berlin, third edn. (2005)
13. Dimitrijević, V., Šarić, Z.: An efficient transformation of the generalized traveling salesman problem into the traveling salesman problem on digraphs. Information Sciences **102**(1), 105–110 (1997)
14. Englot, B., Hover, F.: Planning complex inspection tasks using redundant roadmaps. In: Robotics Research : The 15th International Symposium ISRR. pp. 327–343. Springer International Publishing (2017)
15. Euler, L.: Solutio problematis ad geometriam situs pertinentis. Commentarii academiae scientiarum Petropolitanae pp. 128–140 (1741)
16. Fomin, F.V., Golovach, P.A., Korhonen, T., Simonov, K., Stamoulis, G.: Fixed-Parameter Tractability of Maximum Colored Path and Beyond. In: Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 3700–3712. Society for Industrial and Applied Mathematics (Jan 2023)
17. Fu, M., Salzman, O., Alterovitz, R.: Computationally-efficient roadmap-based inspection planning via incremental lazy search. In: 2021 IEEE International Conference on Robotics and Automation (ICRA). pp. 7449–7456. IEEE (2021)
18. Gonzalez, T.F.: Clustering to minimize the maximum intercluster distance. Theoretical computer science **38**, 293–306 (1985)
19. Halperin, D., Salzman, O., Sharir, M.: Algorithmic Motion Planning. In: Handbook of Discrete and Computational Geometry. Chapman and Hall/CRC, 3 edn. (2017)
20. Harris, R.J., Kavuru, M.S., Mehta, A.C., Medendorp, S.V., Wiedemann, H.P., Kirby, T.J., Bice, T.W.: The impact of thoracoscopy on the management of pleural disease. Chest **107**(3), 845–852 (1995)
21. Hierholzer, C., Wiener, C.: Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. Mathematische Annalen **6**(1), 30–32 (1873)
22. Karaman, S., Frazzoli, E.: Sampling-based algorithms for optimal motion planning. The international journal of robotics research **30**(7), 846–894 (2011)

23. Kou, L., Markowsky, G., Berman, L.: A fast algorithm for Steiner trees. Acta informatica **15**, 141–145 (1981)
24. Lien, Y.N., Ma, E., Wah, B.W.S.: Transformation of the generalized traveling-salesman problem into the standard traveling-salesman problem. Information Sciences **74**(1), 177–189 (1993)
25. Light, R.W.: Pleural diseases. Lippincott Williams & Wilkins (2007)
26. Mahoney, A.W., Anderson, P.L., Swaney, P.J., Maldonado, F., Webster, R.J.: Reconfigurable parallel continuum robots for incisionless surgery. In: 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 4330–4336. IEEE (2016)
27. Mizutani, Y., Salomao, D.C., Crane, A., Bentert, M., Drange, P.G., Reidl, F., Kuntz, A., Sullivan, B.D.: Accompanying source code. https://github.com/TheoryInPractice/robotic-brewing/
28. Noppen, M.: The utility of thoracoscopy in the diagnosis and management of pleural disease. In: Seminars in respiratory and critical care medicine. vol. 31, pp. 751–759 (2010), tex.number: 06 tex.organization: Thieme Medical Publishers
29. Pop, P.C., Cosma, O., Sabo, C., Sitar, C.P.: A comprehensive survey on the generalized traveling salesman problem. European Journal of Operational Research **314**(3), 819–835 (2024)
30. Rice, M.N., Tsotras, V.J.: Exact graph search algorithms for generalized traveling salesman path problems. In: Klasing, R. (ed.) Experimental Algorithms. pp. 344–355. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
31. Rice, M.N., Tsotras, V.J.: Parameterized algorithms for generalized traveling salesman problems in road networks. In: Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems. pp. 114–123. SIGSPATIAL'13, Association for Computing Machinery, New York, NY, USA (Nov 2013)
32. Safra, S., Schwartz, O.: On the complexity of approximating TSP with neighborhoods and related problems. computational complexity **14**(4), 281–307 (Mar 2006)

A Walk Merging: Optimality in the Limit

We argue that our strategy for merging walks is a simplification of an algorithm that is optimal in the limit, given sufficient runtime. Note that the dynamic program behind Theorem 1 is optimal (always produces a walk of minimum weight that collects the given colors). A very simple strategy that is also optimal is to select an arbitrary permutation $(c_1, c_2, \ldots, c_k)$ of the colors and then compute a shortest walk that collects all colors in the respective order. If we repeat this for each possible permutation, then at some point, we will find an optimal solution.

We now observe that computing a walk that collects all colors in the guessed order can be computed in polynomial time by the following dynamic program T that stores for each vertex v and each integer $i \in [k]$ the length of a shortest walk between s and v that collects the first i colors in the guessed order. Therein, we use a second table D.

$$D[v,i] = \begin{cases} T[u,i-1] + \mathrm{dist}(u,v) & \text{if } c_i \in \chi(v) \\ \infty & \text{else} \end{cases}$$
$$T[v,i] = \min_{u \in V} D[u,i] + \mathrm{dist}(u,v)$$

We mention that we assume that $\mathrm{dist}(v,v) = 0$ for each vertex v.

We now modify the above strategy to achieve a better success probability than when permutations are chosen randomly. Instead of guessing the entire sequence of colors, we guess *buckets* of colors, that is, a sequence of c sets for some integer c that form a partition of the set of colors into sets of size k/c (appropriately rounded, for this presentation, we will assume that k is a multiple of c). Note that there are $k!/((k/c)!)^c$ possible guesses for such buckets. For each bucket, let S_i be the set of colors in the bucket. We can now compute for each pair u, v of vertices a shortest walk between u and v that collects all colors in S_i by running the dynamic program behind Theorem 1 for color set S_i from all vertices $u \in V$. Let this computed value be $S[u, v, S_i]$. Given a guess for a sequence of buckets, we can now compute an optimal solution corresponding to this guess by modifying the above dynamic program as follows.

$$T'[v,1] = S[s,v,S_1]$$
$$T'[v,i] = \min_{u \in V} T[u,i-1] + S[u,v,S_i] \text{ if } i > 1$$

Note that $T'[v,i] \leq T[v,ci]$ if the sequence used to compute T corresponds to the set of buckets used to compute T'. This algorithm again achieves optimality in the limit (that is, given enough runtime, it will find a minimum weight walk).

In order to avoid computing S for all pairs of vertices, we decided to implement a simplification where we only compute $S'[S_i] = S[s,s,S_i]$. This version is not guaranteed to find an optimal solution like the full DP above, as there are examples where no optimal solution returns back to s before the very end. As an example, consider a star graph where s is a leaf and each other leaf has a unique color. However, this simplification is faster by a factor of n and uses a factor of n less memory while performing well in practice.

B Color Reduction

In this section, we describe our color reduction strategies in more detail and provide empirical data comparing their effectiveness.

Initializing GreedyMD**.** The Gonzalez [18] algorithm on which GreedyMD is based requires that the set $\mathcal{C}'$ be initialized with at least one color. The simplest strategy is to simply add a uniformly randomly selected color to $\mathcal{C}'$ and then proceed with the algorithm. In practice, we found it more effective to set $\mathcal{C}' = \chi(s) \neq \emptyset$ (see Algorithm 2). That is, we begin by adding some color which is visible from the source vertex s. We then greedily add k more colors. At the end of the algorithm, we return $\mathcal{C}' \setminus \chi(s)$. Intuitively, the justification for this approach is that we collect the colors $\chi(s)$ "for free" since every solution walk begins at s. Consequently, we do not need to ensure that $\mathcal{C}'$ is representative of these colors, or of colors which are very similar to them. Empirically, we found that this initialization strategy significantly improved the coverage of the resulting solutions.

Introducing MetricMD **and** OutlierMD**.** A potential shortcoming of GreedyMD is that it favors outlier colors. That is, because at each iteration it chooses the color which is most dissimilar to the previously selected colors, we can be sure that outlier colors which are very dissimilar to every other color will be selected. This may be undesirable for two reasons. First, if the similarity function f is correlated to colors being visible from the same vertices, then discarding outliers from $\mathcal{C}'$ may improve coverage (as computed on $\mathcal{C}$). Second, if the similarity function $f(c_1, c_2)$ is correlated to the shortest distance between vertices labeled with c_1 and c_2, then discarding outliers from $\mathcal{C}'$ may reduce the weight needed for a walk collecting all colors in $\mathcal{C}'$.

We designed and tested two strategies to mitigate these effects. The first, OutlierMD (see Algorithm 3), uses an additional scaling parameter $r \geq 1$. GreedyMD is used to form a representative color set $\mathcal{C}'$ of size rk. Next, $\mathcal{C}'$ is partitioned into rk clusters by assigning each color $c \in \mathcal{C}$ to a cluster uniquely associated with the representative $c' \in \mathcal{C}'$ to which c is most similar. Finally, we return the k colors in $\mathcal{C}'$ associated with the largest clusters.

The second strategy, MetricMD, assumes that our colors are embeddable in a metric space. This is true, for example, when colors represent positions in $\mathbb{R}^3$ on some surface mesh of the object to be inspected. In this case, we begin by using GreedyMD to find a representative colors set $\mathcal{C}'$ of size k. Next, we perform k-means clustering on $\mathcal{C}$, using $\mathcal{C}'$ as the initial centroids. The resulting centroids are positions in space, and may not perfectly match the positions of any colors. To deal with this, we simply choose the colors closest to the centroids, and return these as our representative color set. See Algorithm 4.

Empirical Evaluation of Color Reduction Schemes. We experimentally evaluated our color reduction schemes (GreedyMD, OutlierMD, and MetricMD)

Algorithm 3: OutlierMD.

Input : $\mathcal{C}, \chi_0, f$, a positive integer $k \leq |\mathcal{C} \setminus \chi_0|$, and $r \in \mathbb{R}_{\geq 1}$.
Output : $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| = k$.

$\mathcal{C}' \leftarrow \mathsf{GreedyMD}(\mathcal{C}, \chi_0, f, \lfloor rk \rfloor)$
$X_{c'} = \{c'\}\ \forall\ c' \in \mathcal{C}'$ `// Initialize clusters for each` $c' \in \mathcal{C}'$
for $c \in \mathcal{C}$ **do**
  Add c to $X_{\mathrm{argmin}_{c' \in \mathcal{C}'} f(c,c')}$ `// Cluster` $\mathcal{C}$
Sort elements of $\mathcal{C}' : c'_1, c'_2, \ldots c'_{rk}$ according to the size of $X_{c'_i}$ (descending).
return $c'_1, c'_2, \ldots c'_k$.

Algorithm 4: MetricMD.

Input : $\mathcal{C}, \chi_0, f$, and a positive integer $k \leq |\mathcal{C} \setminus \chi_0|$.
Output : $\mathcal{C}' \subseteq \mathcal{C}$ with $|\mathcal{C}'| = k$.

$\mathcal{C}' \leftarrow \mathsf{GreedyMD}(\mathcal{C}, \chi_0, f, k)$
$S \leftarrow k\text{-}\mathsf{Means}(\mathcal{C} \setminus \chi_0, \mathcal{C}')$ `// Run` k-Means `on` $\mathcal{C} \setminus \chi_0$ `with initial centroids` $\mathcal{C}'$.
$S' \leftarrow \emptyset$
for $s \in S$ **do**
  `// For each centroid, choose the closest color.`
  $S' \leftarrow S' \cup \{\mathrm{argmin}_{c \in \mathcal{C}' \setminus S'} d(c, s)\}$
return S'.

along with the baseline Rand on each of our four datasets, with $k \in \{10, 20\}$. To perform the evaluation, each algorithm was used to create a representative set of k colors, and then DP-IPA was used to solve Graph Inspection on the color-reduced graphs. We chose DP-IPA rather than ILP-IPA for comparison because the former is the solver which needs color reduction to compute walks on our datasets (i.e., ILP-IPA can run on the DRONE and CRISP datasets without any color reduction). The results of these experiments are displayed in Fig. 8. In each plot displayed in Fig. 8, every colored dot represents a Graph Inspection solution computed by DP-IPA after color reduction performed by either Rand, GreedyMD, OutlierMD, or MetricMD. For each color reduction strategy, the solution with the highest coverage (in the original, non-color-reduced graph) is indicated with a rightward-pointing arrow, and the solution with minimum weight is indicated with a downward-facing arrow.

Because our color reduction schemes are designed to ensure good coverage, we are primarily interested in comparing the coverage of solutions. The results indicate that, in general, our strategies outperform the baseline Rand in terms of coverage, often by large margins. In terms of coverage, the comparative performances of GreedyMD, MetricMD, and OutlierMD are somewhat difficult to disentangle. Given that GreedyMD is the simplest of the three, the most explainable, and the most generalizable (it does not require a metric embedding

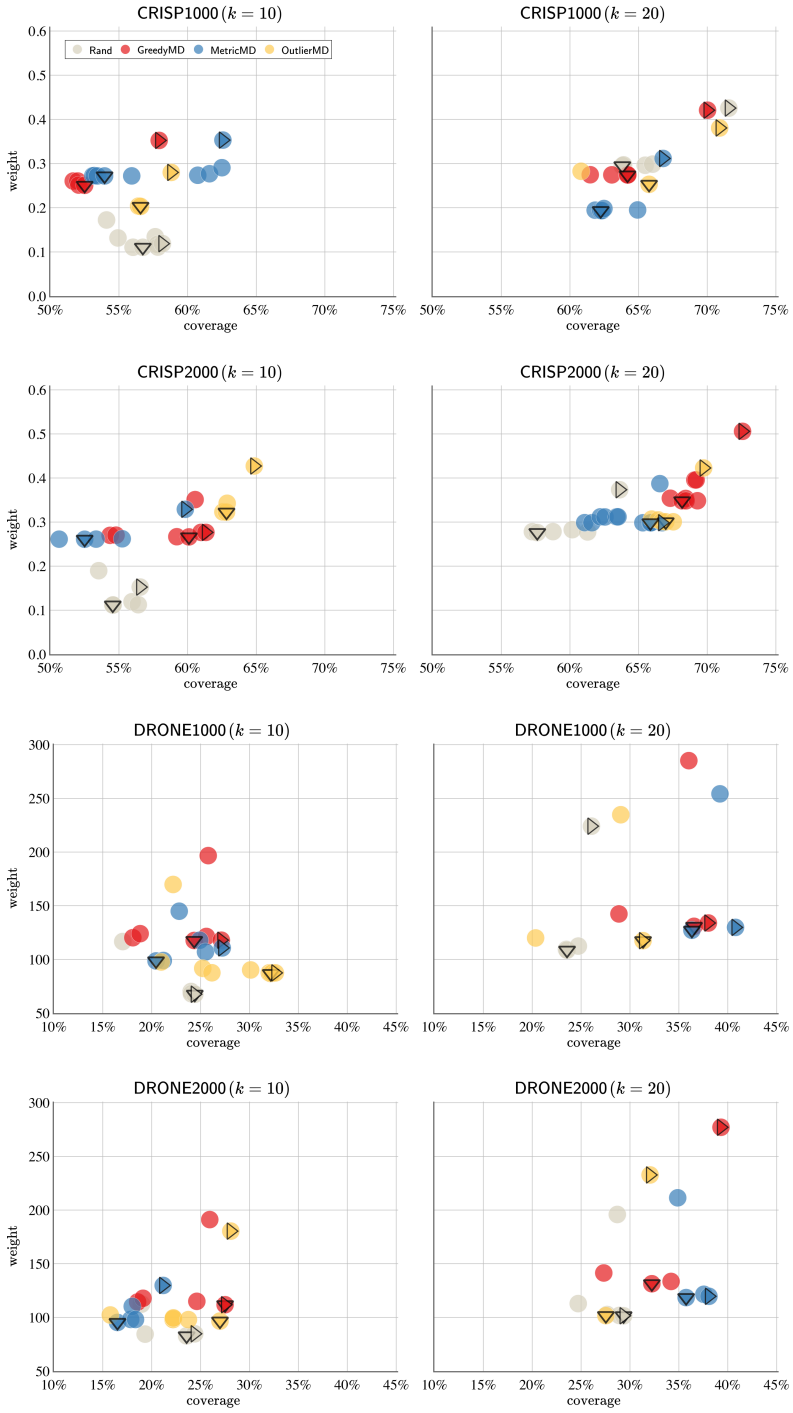

Fig. 8: Results of color reduction experiments for Rand, GreedyMD, OutlierMD, and MetricMD. Each datapoint represents a Graph Inspection solution generated by DP-IPA. For each algorithm, the solution with the highest coverage (computed in the original, non-color-reduced graph) is marked with a rightward-pointing arrow, and the minimum weight solution is marked with a downward-pointing arrow.

or any parameter tuning), we favor it for future experiments. However, we leave as an interesting open direction to perform a more extensive comparison of these methods on a larger data corpus.

We conclude this section with a discussion of the weight of solutions. We are not surprised that our strategies tend to produce higher-weight solutions than Rand since we are optimizing for coverage even if it means requiring that a solution walk visit outlier POIs. In particular, it is expected that GreedyMD performs poorly with respect to solution weight, since it explicitly favors outlier POIs. In applications where the weight of the solution is of paramount interest, even at the expense of lowering coverage, the aforementioned extended comparison of color reduction strategies may be of particular interest.

C Walk-Merging Algorithms

In this section, we present preprocessing steps for the Minimum Spanning Eulerian Subgraph and the GreedyMerge algorithm in detail.

Preprocessing for Minimum Spanning Eulerian Subgraph**.** We say an edge set $\tilde{E} \subseteq E(G)$ is *undeletable* if there is an optimal solution for Minimum Spanning Eulerian Subgraph including all the edges in $\tilde{E}$. We apply the following rules as preprocessing.

Rule 1 If there is an edge with multiplicity at least 4, then decrease its multiplicity by 2. This makes the multiplicity of every edge either 1, 2 or 3.

Rule 2 If there is an edge cut $e_1, e_2 \in E(G)$ of size 2, then mark e_1 and e_2 as undeletable. For example, this includes (but is not limited to) the following:
- edges incident to a vertex with degree 2.
- an edge with multiplicity 2 that forms a bridge in the underlying simple graph of G.

Algorithm GreedyMerge**.** Consider the following heuristic for Minimum Spanning Eulerian Subgraph.

Algorithm 5: GreedyMerge

1 Apply Rule 1 exhaustively.
2 Construct a minimum spanning tree S of G using a known algorithm.
3 Greedily find a maximal cycle packing $\mathcal{C}$ in $G - S$ as follows:
   1. Let U be the empty multigraph with $V(G)$
   2. Iteratively add every edge $e \in E(G - S)$ to U in order of nonincreasing weights. If e creates a cycle C in U, add C to $\mathcal{C}$ and remove C from U.

return $G - \bigcup_{C \in \mathcal{C}} E(C)$

This algorithm runs in $\mathcal{O}(m \log n)$ time. In practice, we apply (part of) Rule 2 to find undeletable edges and include them in S at step 2. To prove

the correctness of data reduction rules and algortihms, we start by a simple, well-known observation on Eulerian graphs.

Observation 5. *Let C be a closed walk in an Eulerian multigraph G. If $G - C$ is connected, then $G - C$ is also Eulerian.*

Proof. Removing a closed walk does not change the parity of the degree at each vertex. If all vertices in G have even degrees, so do those in $G - C$, which implies that $G - C$ is Eulerian if it is connected. □

We continue with the analysis of our two reduction rules.

Lemma 3. *Rule 1 is safe.*

Proof. By Lemma 1, there must be an optimal solution with edge multiplicity at most 2. From Observation 5, if there is an edge with multiplicity at least 4, then removing 2 of them (which can be seen as a closed walk) results in a connected Eulerian multigraph. □

Lemma 4. *Rule 2 is safe.*

Proof. Let $S \subset V(G)$ be a nonempty vertex set such that e_1, e_2 separates S from $V(G) \setminus S$. Then, any closed walk that visits $V(G)$ must contain at least one edge from S to $V(G) \setminus S$ and another from $V(G) \setminus S$ to S. Hence, both e_1 and e_2 must be in any solution. □

We conclude this section by analyzing GreedyMerge.

Theorem 6. GreedyMerge *correctly outputs a (possibly suboptimal) solution for* Minimum Spanning Eulerian Subgraph *in $\mathcal{O}(m \log n)$ time.*

Proof. We need to show that GreedyMerge outputs a spanning Eulerian subgraph G' (in this context, a *subgraph* is also a multigraph) of G. From step 2, we know that S is a spanning subgraph of G. From step 3, we have that $E(C) \subseteq E(G - S)$ for every $C \in \mathcal{C}$. From step 4, G' is clearly a subgraph of G, and from $\bigcup_{C \in \mathcal{C}} E(C) \subseteq E(G - S)$, we have $E(G') \supseteq E(S)$, and hence G' is spanning. Knowing that G' is connected and from Observation 5, a multigraph constructed by removing any closed walk remains Eluerian. Hence, G' is a spanning Eluerian subgraph of G.

We next analyze the running time. Rule 1 can be executed exhaustively in $\mathcal{O}(m)$ time by iterating over all edges. At this point, $|E(G)| \leq 3 \cdot \binom{n}{2} \leq 2n^2$. Now let us analyze the rest of the steps in GreedyMerge. The running time of step 2 is the same as that of Kruskal's algorithm, which is $\mathcal{O}(m \log m) \subseteq \mathcal{O}(m \log(2n^2)) = \mathcal{O}(m \log n)$. Similarly, step 3 has the same running time as we iteratively examine edges and check for connectivity. Hence, the overall running time is in $\mathcal{O}(m \log n)$. □

D Experiment Details and Supplemental Results

The following table summarizes the test instances we used for our experiment.

dataset		CRISP		DRONE	
n_{build}		1,000	2,000	1,000	2,000
name		CRISP1000	CRISP2000	DRONE1000	DRONE2000
number of vertices (n)		1,006	2,005	1,002	2,001
number of edges (m)		18,695	41,506	19,832	44,089
number of colors		4,200	4,200	3,204	3,254
number of colors at the starting vertex		535	540	10	10
number of colors at a vertex	min	0	0	0	0
	mean	183.39	175.71	22.67	19.16
	max	855	876	129	129
	stdev	179.02	170.75	24.61	22.41
edge weight	min	0.000002	0.000000	0.51	0.43
	mean	0.006971	0.005275	4.61	3.91
	max	0.060926	0.060926	18.51	18.51
	stdev	0.005354	0.004271	1.86	1.58
minimum spanning tree weight		1.109606	1.637212	1875.53	3118.65
diameter	unweighted	7	8	6	7
	weighted	0.136846	0.138467	48.24	49.73

Table 1: Corpus of test instances for Graph Inspection.

Experiment Environment. We implemented our code with C++ (using C++17 standard). We ran all experiments on identical hardware, equipped with 80 CPUs (Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz) and 191000 MB of memory, and running Rocky Linux release 8.8. We used Gurobi Optimizer 9.0.3 as the ILP solver, parallelized over CPUs.

D.1 Multithreading Analysis

Here, we present results of our multithreading experiments for ILP-IPA (visualized in Fig. 9). We configured the Gurobi ILP solver to output each feasible solution as soon as it is found, so to understand the impact of multithreading we are interested in determining the lowest-weight feasible solution identified in a given time limit, for various thread counts. Predictably, the general trend is clear: multi-threaded implementations provide lower-weight solutions faster than single threaded solutions.

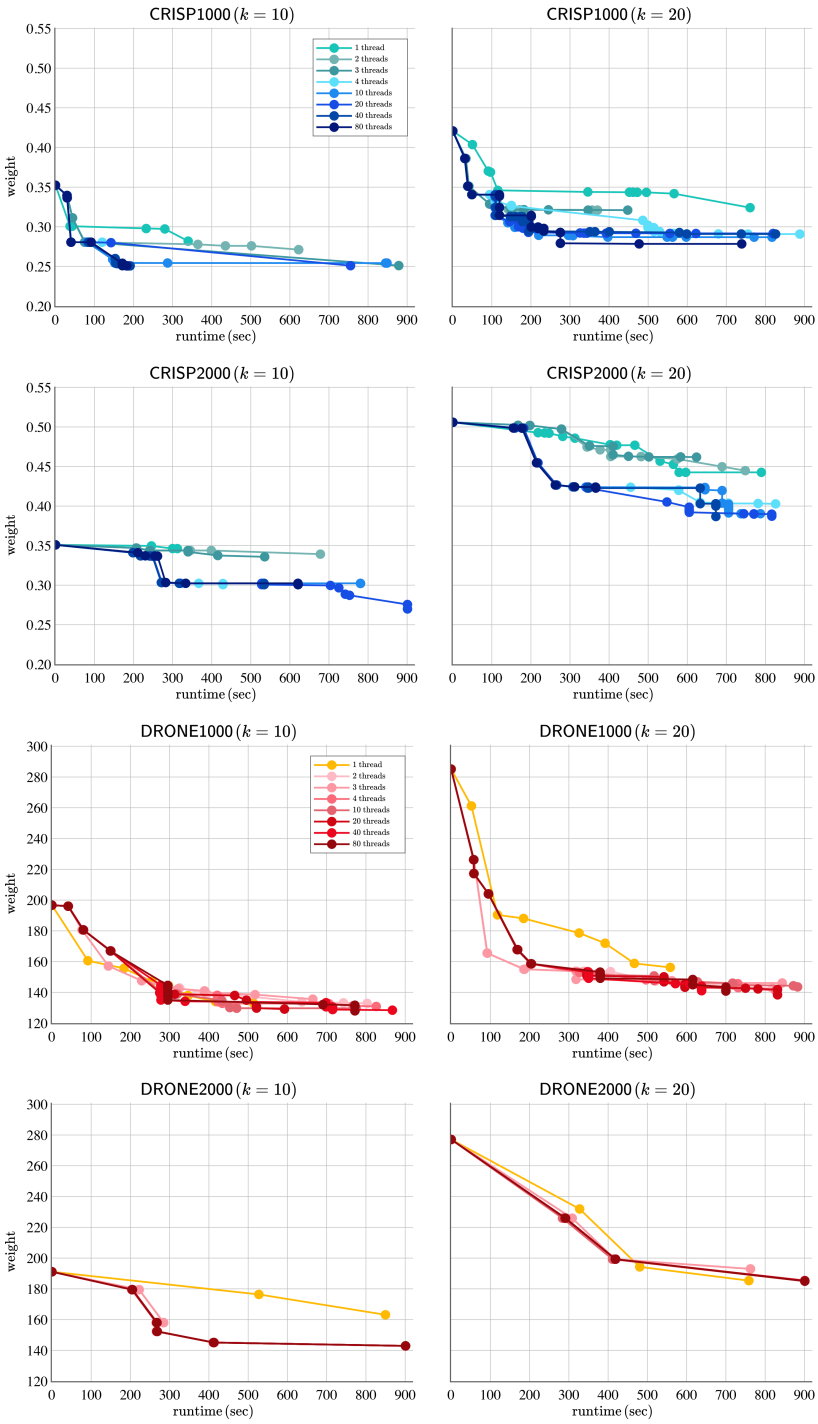

Fig. 9: Results of multithreading experiments for ILP-IPA, executed on datasets CRISP (top two rows) and DRONE (bottom two rows), with $k = 10$ (left column) or $k = 20$ (right column).

D.2 Additional Empirical Results & Figures

Based on our findings in Appendix B, we also compared all three inspection planning algorithms when DP-IPA is paired with MetricMD color reduction (instead of GreedyMD, as shown in Fig. 6). The results are shown in Fig. 10.

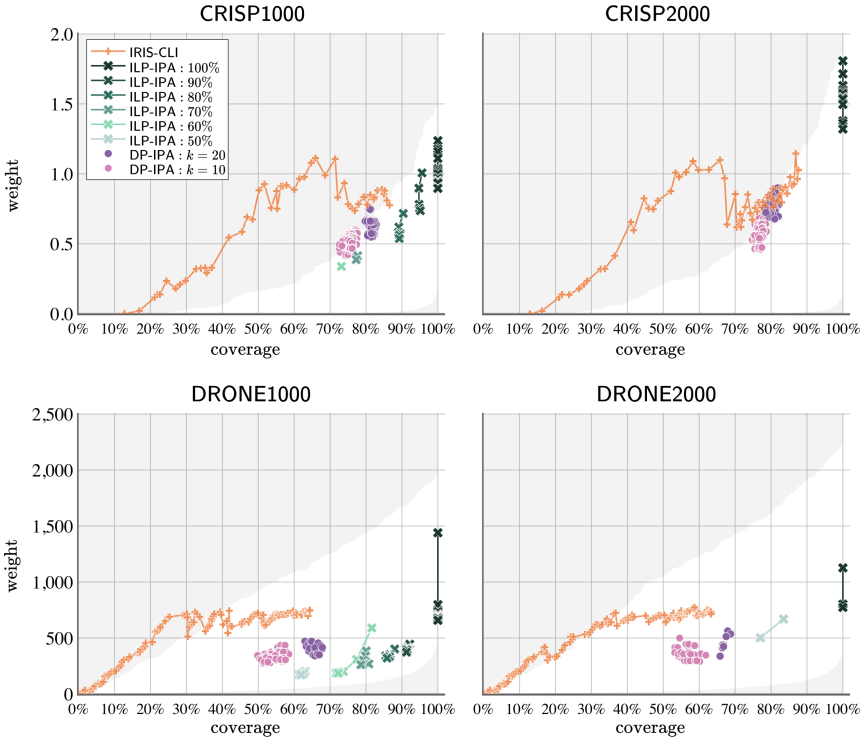

Fig. 10: Performance of IRIS-CLI, ILP-IPA and DP-IPA (with MetricMD) on DRONE and CRISP benchmarks. Each data point represents a computed inspection plan; coverage is shown as a percentage of all POIs in the input graph. The area shaded in gray is outside the upper/lower bounds given in Section 3.3.

They are qualitatively quite similar, but we observe that this approach has the disadvantage of requiring the POI similarity function to be a metric.

We showed the results of applying our color partitioning methods in combination with GreedyMD and MetricMD for the 1000-node graphs in Fig. 8; the analogous results for the 2000-node graphs are included below in Fig. 11.

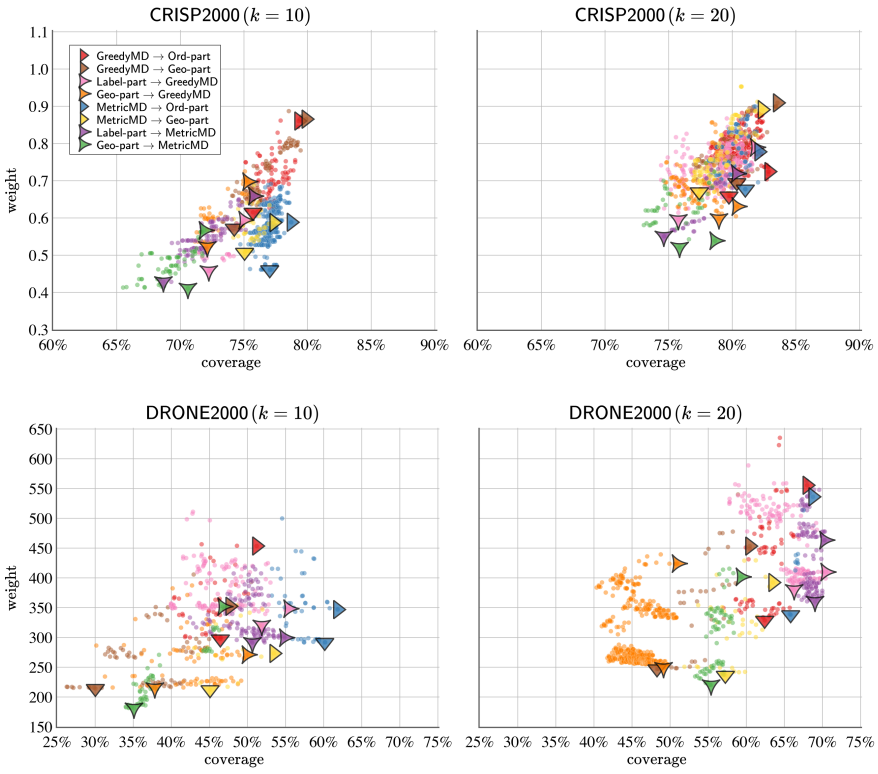

Fig. 11: Results of our color partitioning experiments for datasets CRISP2000 and DRONE2000 with $k \in \{10, 20\}$. Every data point represents a solution computed using DP-IPA and ExactMerge. For each combination of color reduction and color partitioning strategies, the solution with maximum coverage is indicated with a rightward-pointing arrow, and the solution with minimum weight is indicated with a downward-pointing arrow.

After applying color reduction and partitioning, we have a set of walks that need merging, as discussed in Section 4.2 with additional details in Appendix C. The empirical results of comparing these approaches are shown in Fig. 12.

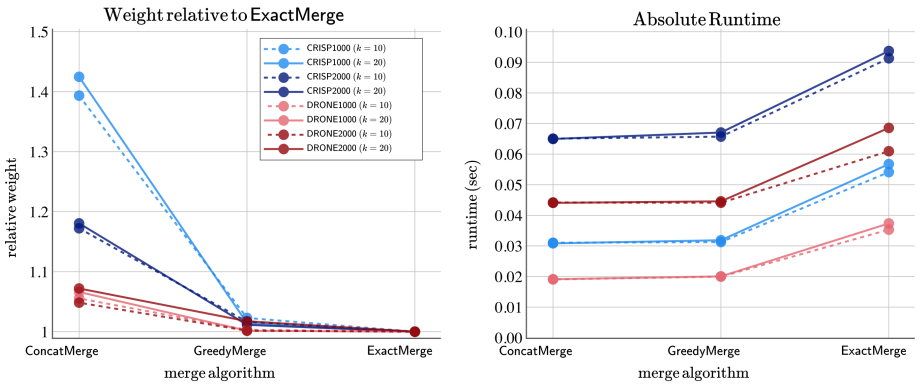

Fig. 12: Experimental results comparing ConcatMerge, GreedyMerge, and ExactMerge. On the left, the weight of the combined walk is compared (relative to ExactMerge). On the right, runtimes are compared. In all experiments, three walks were combined.

Table 2 compares the solution weights generated by DP-IPA and ILP-IPA. In this table, both DP-IPA and ILP-IPA are used to produce solutions on color-reduced graphs. The table shows solution weights by ILP-IPA relative to the optimal weights that DP-IPA produces. The trend is clear: ILP-IPA becomes less competitive with DP-IPA as n (n_{build}) and k grow.

		CRISP			DRONE		
n_{build}	k	mean	min	max	mean	min	max
$1,000$	10	1.03	1.00	1.16	1.04	1.00	1.13
	20	1.04	1.00	1.31	1.05	1.00	1.18
$2,000$	10	1.12	1.00	1.38	1.17	1.03	1.38
	20	1.16	1.01	1.34	1.19	1.02	1.51

Table 2: Comparison of solution weights generated by ILP-IPA, relative to the lowest weight produced by DP-IPA; both solvers had a 15-minute timeout.