Computing a Minimum Color Path in Edge-Colored Graphs

Neeraj Kumar

Department of Computer Science,
University of California, Santa Barbara, USA
neeraj@cs.ucsb.edu

Abstract. In this paper, we study the problem of computing a *min-color* path in an *edge-colored* graph. More precisely, we are given a graph $G = (V, E)$, source s, target t, an assignment $\chi : E \to 2^{\mathcal{C}}$ of edges to a set of colors in $\mathcal{C}$, and we want to find a path from s to t such that the number of unique colors on this path is minimum over all possible s-t paths. We show that this problem is hard (conditionally) to approximate within a factor $O(n^{1/8})$ of optimum, and give a polynomial time $O(n^{2/3})$-approximation algorithm. We translate the ideas used in this approximation algorithm into two simple greedy heuristics, and analyze their performance on an extensive set of synthetic and real world datasets. From our experiments, we found that our heuristics perform significantly better than the best previous heuristic algorithm for the problem on all datasets, both in terms of path quality and the running time.

1 Introduction

An *edge colored* graph $G = (V, E, \mathcal{C}, \chi)$ comprises of an underlying graph $G = (V, E)$, and a set of colors $\mathcal{C}$ such that each edge $e \in E$ is assigned a subset $\chi(e) \subseteq \mathcal{C}$ of colors. For any path π in this graph, suppose we define its cost to be $|\chi(\pi)|$ where $\chi(\pi) = \bigcup_{e \in \pi} \chi(e)$ is the set of colors used by this path. In this paper, we study the natural problem of computing a path π from a source s to some target t such that its cost, that is the number of colors used by π, is minimized. The problem is known to be NP-hard, and by a reduction from Set-cover is also hard to approximate within a factor $o(\log n)$.

The problem was first studied by Yuan et al. [19] and was motivated by applications in maximizing the reliability of connections in mesh networks. More precisely, each network link is assigned one or more colors where each color corresponds to a given failure event that makes the link unusable. Now if the probability of all the failure events is the same, a path that minimizes the number of colors used has also the least probability of failure. Therefore, the number of colors used by a minimum color path can be used as a measure for 'resilience' of the network. This has also been applied in context of sensor networks [2] and attack graphs in computer security [14]. Apart from resilience, the minimum color path problem can also be used to model licensing costs in networks. Roughly speaking, each link can be assigned a set of colors based on the providers that operate the link, and a minimum color path then corresponds to a minimum number of licenses that are required to ensure connectivity between two given nodes. More generally, the problem applies to any setting where colors can be thought of as "services" and we only need to pay for the first usage of that service. The problem was also studied by Hauser [12] motivated by robotics applications. In such settings, colors are induced by geometric objects (obstacles) that block one or more edges in a path of the robot. Naturally, one would like to remove the minimum number of obstacles to find a clear path, which corresponds to the colors used by a minimum color path.

The problem has also gathered significant theoretical interest. If each edge of the graph is assigned exactly one color (called its *label*), the problem is called *min-label* path and was studied in [11]. They gave an algorithm to compute an $O(\sqrt{n})$-approximation and also show that it is hard to approximate within $O(log^c n)$ for any fixed constant c, and n being the number of vertices. Several other authors have also studied related problems such as minimum label spanning tree and minimum label cut [8,16]. The min-color path problem on *vertex-colored* graph was recently studied in [1] where they gave an $O(\sqrt{n})$-approximation algorithm. Indeed, one can transform an edge-colored graph into a vertex-colored graph by adding a vertex of degree two on each edge e and assigning it the set of colors $\chi(e)$. However, this does not gives a sublinear approximation in n as the number of vertices in the transformed graph can be $\Omega(n^2)$.

1.1 Our Contribution

In this work, we make progress on the problem by improving the known approximation bounds and by designing fast heuristic algorithms.

- By a reduction from the *minimum k-union* problem [6] which was recently shown to be hard to approximate within a factor of $O(n^{1/4})$, we show that min-color path cannot be approximated within a factor $O(n^{1/8})$ of optimum on edge-colored graphs. This also implies improved lower bounds for min-label path, as well as min-color path on *vertex-colored* graphs.
- We give an $O(n^{2/3})$-approximation algorithm for min-color path problem on edge-colored graphs. If the number of colors on each edge is bounded by a constant, the algorithm achieves an approximation factor of $O((\frac{n}{OPT})^{2/3})$, where OPT is the number of colors used by the optimal path.
- We translate the ideas from the above approximation algorithm into two greedy heuristics and analyze its performance on a set of synthetic and real-world instances [15]. Although similar greedy heuristics were proposed by the previous work [19] and have been shown to perform well on *randomly generated* colored graphs; a holistic analysis of such algorithms on more challenging and realistic instances seems to be lacking. We aim to bridge this gap by identifying the characteristics of challenging yet realistic instances that helps us design a set of synthetic benchmarks to evaluate our algorithms.

From our experiments, we found that our heuristics achieve significantly better performance than the heuristic from [19] while being significantly (up to 10 times) faster. We also provide an ILP formulation for the problem that performs reasonably well in practice. All source code and datasets have been made available online on github [17].

The rest of the paper is organized as follows. In Section 2, we discuss our hardness reduction. The details of $O(n^{2/3})-$approximation is given in Section 3. We discuss our heuristic algorithms and experiments in Section 4. For the rest of our discussion, unless stated otherwise, we will use the term colored graph to mean an *edge-colored* graph and our goal is to compute a min-color path on such graphs. On some occasions, we will also need to refer to the *min-label* path problem, which is a special case of min-color path when all edges have exactly one color.

2 Hardness of Approximation

By a simple reduction from SET-COVER, it is known that the min-color path problem is hard to approximate to a factor better than $o(\log n)$, where n is the number of vertices. This was later improved by Hassin et al. [11], where they show that the min-label path (and therefore the min-color path) problem is hard to approximate within a factor $O(\log^c n)$ for any fixed constant c. In this section, we work towards strengthening this lower bound. To this end, we consider the *minimum k-union* problem that was recently studied in [6].

In the *minimum k-union* problem, we are given a collection $\mathcal{S}$ of m sets over a ground set U and the objective is to pick a sub-collection $\mathcal{S}' \subseteq \mathcal{S}$ of size k such that the union of all sets in $\mathcal{S}'$ is minimized. The problem is known to be hard to approximate within a factor $O(m^{1/4})$. However, it is important to note that this lower bound is conditional, and is based on the so-called DENSE VS RANDOM conjecture being true [6]. The conjecture has also been used to give lower bound guarantees for several other problems such as Densest k-subgraph [3], Lowest Degree 2-Spanner, Smallest m-edge subgraph [5], and Label cover [7].

In the following, we will show how to transform an instance of minimum k-union problem to an instance of min-color path. More precisely, given a collection $\mathcal{S}$ of m sets over a ground set U and a parameter k, we will construct a colored graph $G = (V, E, \mathcal{C}, \chi)$ with two designated vertices s, t, such that a solution for min-color path on G corresponds to a solution of minimum k-union on $\mathcal{S}$ and vice versa. We construct G in three steps (See also Figure 1).

- We start with a path graph G' that has $m+1$ vertices and m edges. Next, we create $m-k+1$ copies of G' and arrange them as rows in a $(m-k+1)\times(m+1)$-grid, as shown in Figure 1.
- So far we only have horizontal edges in this grid of the form $(v_{ij}, v_{i(j+1)})$. Next, we will add *diagonal edges* of the form $(v_{ij}, v_{(i+1)(j+1)})$ that basically connect a vertex in row i to its right neighbor in row $i+1$.

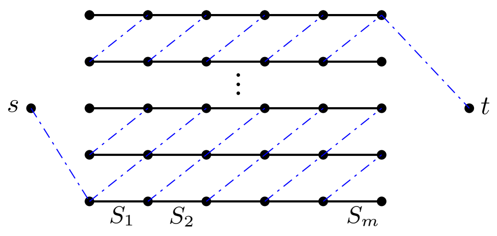

Fig. 1. Reducing minimum k-union to min-color path. The dashed edges are uncolored. The horizontal edge $(v_{ij}, v_{i(j+1)})$ is assigned color corresponding to the set S_j in all rows i.

– Finally, we add the vertices s, t and connect them to bottom-left vertex v_{11} and the top-right vertex $v_{(m-k+1)(m+1)}$, respectively.

We will now assign colors to our graph G. For the set of possible colors $\mathcal{C}$, we will use the ground set U and assign every subset $S_j \in \mathcal{S}$ to horizontal edges of G from left to right. More precisely,

– The diagonal edges of G are not assigned any color.
– Every horizontal edge that connects a node in column j to $j+1$ gets assigned the set of color S_j, for all $j \in 1, 2, \ldots, m$. That is $\chi(v_{ij}, v_{i(j+1)}) = S_j$, for all $i \in \{1, 2, \ldots, m-k+1\}$.

We make the following claim.

Lemma 1. *Assuming that minimum k-union problem is hard to approximate within a factor $O(m^{1/4})$ of optimal, the min-color path problem cannot be approximated within a factor of $O(n^{1/8})$, where m is the number of sets in the collection and n is the number of vertices in G.*

Proof. Consider any $s-t$ path π in G. Without loss of generality, we can assume that π is simple and moves monotonically in the grid. This holds because if π moves non-monotonically in the grid, then we can replace the non-monotone subpath by a path that uses the same (or fewer) number of colors. Now observe that in order to get from s to t, the path π must make a horizontal displacement of m columns and a vertical displacement of $m-k$ rows. Since the vertical movement is only provided by diagonal edges and π can only take at most $m-k$ of them, it must take k horizontal edges. If π is the path that uses minimum number of colors, then the sets S_j corresponding to the horizontal edges taken by π must have the minimum size union and vice versa.

Now suppose it was possible to approximate the min-color path problem within a factor $O(n^{1/8})$ of optimal. Then given an instance of minimum k-union, we can use the above reduction to construct a graph G that has $n = O(m^2)$ vertices and run this $O(n^{1/8})$ approximation algorithm. This will give us a path π that uses at most $O(m^{1/4})r$ colors, where r is the minimum number of colors used. As shown above, r must also be the number of elements in an optimal solution of minimum k-union. Therefore, we can compute a selection of k sets that have union at most $O(m^{1/4})$ times optimal, which is a contradiction. □

Note that we can also make G vertex-colored : subdivide each horizontal edge e by adding a vertex v_e of degree two, and assign the set of colors $\chi(e)$ to v_e. Observe that since the graph G we constructed above had $O(n)$ edges, the same lower bound also translates to min-color path on vertex-colored graph.

Corollary 1. *Min-color path on vertex-colored graphs is hard to approximate within a factor $O(n^{1/8})$ of optimal.*

One can also obtain a similar bound for min-label path (proof in Appendix).

Lemma 2. *Min-label path is hard to approximate within a factor $O((\frac{n}{OPT})^{1/8})$, where OPT is the number of colors used by the optimal path.*

3 An $O(n^{2/3})-$ Approximation Algorithm

In this section, we describe an approximation algorithm for our problem that is sublinear in the number of vertices n. Note that if the number of colors on each edge is at most one, there exists an $O(\sqrt{n})$ approximation algorithm [11]. However, their technique critically depends on the number of colors on each edge to be at most one, and therefore cannot be easily extended to obtain a sublinear approximation[1].

An alternative approach is to transform our problem into an instance of min-color path on vertex-colored graphs by adding a vertex of degree two on each edge e and assigning the colors $|\chi(e)|$ to this vertex. Applying the $O(\sqrt{|V|})$-approximation from [1], easily gives an $O(\sqrt{|E|})$-approximation for our problem, which is sub-linear in n if the graph is *sparse*, but can still be $\Omega(n)$ in the worst case. To address this problem, we apply the technique of Goel et al. [9] where the idea is to partition the graph G into *dense* and *sparse* components based on the degree of vertices (Step 1 of our algorithm). We consider edges in both these components separately. For edges in dense component, we simply discard their colors, whereas for edges in the sparse components, we use a pruning strategy similar to [1] to discard a set of colors based on their occurrence. Finally, we show that both these pieces combined indeed compute a path with small number of colors. We start by making a couple of simple observations that will be useful.

First, we assume that the number of colors used by the optimal path (denoted by k) is given to our algorithm as input. Since k is an integer between 1 and $|\mathcal{C}|$, it is easy to see that an α-approximation for this version gives an α-approximation for

[1] This is because if each edge has at most one color, the pruning stage in algorithm from [11] can be phrased as a maximum coverage problem, for which constant approximations are known. However even if number of colors is exactly two, the pruning stage becomes a variant of the *densest k-subgraph* problem which is hard to approximate within a factor of $\Omega(n^{1/4})$ of optimum.

min-color path. This holds as we can simply run the α-approximation algorithm $|\mathcal{C}|$ times, once for each value of k and return the best path found.

Now, since we have fixed k, we can remove all edges from the graph that contain more than k colors, as a min-color path will never use these edges. Since each edge in G now contains at most k colors, we have the following lemma.

Lemma 3. *Any $s-t$ path of length ℓ uses at most $k\ell$ colors and is therefore an ℓ-approximation.*

This suggests that if there exists a path in G of small length, we readily get a good approximation. Note that the diameter of a graph $G=(V,E)$ is bounded by $\frac{|V|}{\delta(G)}$, where $\delta(G)$ is the minimum degree over vertices in G. So if the graph is dense, that is, degree of each vertex is high enough, the diameter will be small, and any path will be a good approximation (Lemma 3).

We are now ready to describe the details of our algorithm. We outline the details for the most general case when the number of colors on each edge is bounded by a parameter $z \leq k$. If z is a constant, the algorithm achieves slightly better bounds.

Algorithm : Approximate k-Color Path The input to our algorithm is a colored graph $G=(V,E,\mathcal{C},\chi)$, the number of colors k and a threshold β (which we will fix later) for deciding if a vertex belongs to a dense component or a sparse component. Note that all edges of G have at most $z \leq k$ colors on them.

1. First, we will classify the vertices of G as lying in sparse or dense component. To do this, we include vertices of degree at most β to the *sparse* component and remove all edges adjacent to it. Now we repeat the process on the modified graph until no such vertex exists. Finally, we assign the remaining vertices to the *dense* component, and restore G to be the original graph.
2. For all edges $e=(u,v)$ such that both u,v lie in the dense component, discard its colors. That is set color $\chi(e)=\emptyset$.
3. Now, consider the set of edges that have at least one endpoint in the sparse component, call them *critical edges*. Note that the number of such edges is at most $n\beta$.
4. Remove every color c_i that occurs on at least $\sqrt{\frac{zn\beta}{k}}$ critical edges. That is, set $\chi(e)=\chi(e)\setminus\{c_i\}$, for all edges $e \in E$.
5. Let G' be the colored graph obtained after above modifications. Using $|\chi(e)|$ as weight of the edge e, run Dijkstra's algorithm to compute a minimum weight $s-t$ path π in G'. Return π.

It remains to show that the algorithm above indeed computes an approximately good path. We will prove this in two steps. First, we make the following claim.

Lemma 4. *The number of colors that lie on the path π in the modified colored graph G' is at most $\sqrt{zkn\beta}$.*

Proof. Observe that each color appears on no more than $\sqrt{\frac{zn\beta}{k}}$ edges of G'. Now consider the optimal path π^* in G that uses k colors. Since each of these k colors contribute to the weight of at most $\sqrt{\frac{zn\beta}{k}}$ edges of π^*, the weight of the path π^* in G' is at most $(k \cdot \sqrt{\frac{zn\beta}{k}}) = \sqrt{zkn\beta}$. Therefore, the minimum weight $s-t$ path π will use no more than $\sqrt{zkn\beta}$ colors. □

Lemma 5. *The number of colors that lie on the path π in the original colored graph G is $O(\frac{zn}{\beta} + \sqrt{zkn\beta})$.*

Proof. To show this, we will first bound the number of colors of π that we may have discarded in Steps 2 and 4 of our algorithm.

Consider a connected dense component C_i. Now let G_i be the subgraph induced by vertices in C_i. Since the degree of each vertex in G_i is at least β, the diameter of G_i is at most $\frac{n_i}{\beta}$, where n_i is the number of vertices in the component C_i. Observe that since the weight of all edges of C_i is zero in G', we can safely assume that π only enters C_i at most once. This holds because if π enters and exits C_i multiple times, we can simply find a shortcut from the first entry to last exit of weight zero, such a shortcut always exists because C_i is connected. Therefore π contains at most $\frac{n_i}{\beta}$ edges and uses at most $\frac{zn_i}{\beta}$ colors in the component C_i. Summed over all components, the total number of colors discarded in Step 2 that can lie on π is at most $z \sum_i \frac{n_i}{\beta} \leq \frac{zn}{\beta}$. Next, we bound the number of colors discarded in Step 4. Observe that since each critical edge contains at most z colors, the total number of occurrences of all colors on all critical edges is $zn\beta$. Since we only discard colors that occur on more than $\sqrt{\frac{zn\beta}{k}}$ edges, the total number of discarded colors is bounded by $\left(zn\beta \Big/ \sqrt{\frac{zn\beta}{k}}\right) = \sqrt{zkn\beta}$.

Summing these two bounds with the one from Lemma 4, we achieve the claimed bound. □

The bound from Lemma 5 is minimized when $\beta = (\frac{zn}{k})^{1/3}$. This gives the total number of colors used to be $O((\frac{zn}{k})^{2/3}) \cdot k)$ and therefore, an approximation factor of $O((\frac{zn}{k})^{2/3})$. If the number of colors z on each edge is bounded by a constant, we get an approximation factor of $O((\frac{n}{k})^{2/3})$. Otherwise, we have that $z \leq k$, which gives an $O(n^{2/3})$-approximation.

Theorem 1. *There exists a polynomial time $O(n^{2/3})$-approximation algorithm for min-color path in an edge-colored graphs $G = (V, E, \mathcal{C}, \chi)$. If the number of colors on each edge is bounded by a constant, the approximation factor can be improved to $O((\frac{n}{OPT})^{2/3})$.*

4 Fast Heuristic Algorithms and Datasets

In this section we will focus on designing fast heuristic algorithms for the minimum color path problem. Given a colored graph $G = (V, E, \mathcal{C}, \chi)$, one natural heuristic is to use Dijkstra's algorithm as follows: simply replace the set of colors $\chi(e)$ on each edge e by their cardinalities $|\chi(e)|$ as weights and then compute a minimum weight path in this graph.

Building upon this idea, Yuan et al. [19] proposed a greedy strategy where they start with a path computed by Dijkstra's algorithm as above, and iteratively select the color that improves the path found so far by maximum amount. More precisely, for each color $c \in \mathcal{C}$, decrement the weight of each edge on which c occurs by one, and compute a path using Dijkstra's algorithm. Now, select the color that improves the path found so far by maximum amount in terms of number of colors used. Keep the weight of edges with selected color to their decremented value and repeat the process until the path can no longer be improved.

This heuristic was called Single-Path All Color Optimization Algorithm (SPACOA) in their paper and was shown to achieve close to optimal number of colors on *uniformly colored* random graphs [2]. We argue that although their heuristic performs well on such instances, there still is a need to design and analyze algorithms on a wider range of more realistic instances. This holds because of two reasons. First, in most practical applications where the min-color path is used, the distribution of colors is typically not uniform. For instance, in network reliability applications where colors correspond to a failure event, it is likely that a specific failure event is more common (occurs on more edges) than the other. Similarly, in a network topology setting [15], where colors correspond to ISPs, some providers have larger connectivity than the others. Second, in most of these applications, existence of an edge between two nodes typically depends on proximity of nodes (imagine wireless routers or sensor networks) which is also not accurately captured by random graphs.

Moreover, we note that due to their structural properties (such as small diameters) uniformly colored random graphs are not good instances to measure the efficacy of heuristic algorithms because on these instances the number of colors used by a color oblivious Dijkstra's algorithm is also quite close to optimal, and as such there is little room for improvement. (See also Table 1). In the next two sections, we aim to construct synthetic instances for the min-color path problem that are more challenging and at the same time realistic. Thereafter, we present a couple of greedy heuristic algorithms and analyze their performance on these synthetic and some real world instances. We will use the SPACOA heuristic from [19] as a benchmark for our comparisons. We begin by analyzing min-color paths in uniformly colored random graphs and explain why a color oblivious algorithm such as Dijkstra performs so well. This gives some useful insights into characteristics of hard instances.

4.1 Min-Color Path in Uniformly Colored Random Graphs

We begin by analyzing uniformly colored random graphs where given a random graph, colors are assigned uniformly to its edges [19]. That is, for each edge in

[2] we assume that the random graph is constructed under $G(n, p)$ model, that is an edge exists between a pair of vertices with probability p.

the graph, a color is picked uniformly at random from the set $\mathcal{C}$ of all colors, and assigned to that edge. We note that a colored graph G is likely to be a 'hard instance' for min-color path if there exists an $s-t$ path in G with small number of colors, and the expected number of colors on any $s-t$ path is much larger, so that a color oblivious algorithm is 'fooled' into taking one of these paths. We observe that in randomly generated colored graphs as above, this is quite less likely to happen, which is why the paths computed by a color oblivious Dijkstra's algorithm are still quite good.

To see this, consider an $s-t$ path π of length ℓ in G. Observe that since the colors are independently assigned on each edge, it is equivalent to first fix a path and then assign colors to its edges. Let $p_{i\pi}$ be the probability that color i appears on some edge of π. The probability that color i does not appear on any edge of π is $(1-\frac{1}{|\mathcal{C}|})^{\ell}$ and therefore $p_{i\pi} = 1-(1-\frac{1}{|\mathcal{C}|})^{\ell}$, for each color i. In other words, we can represent the occurrence of a color i on the path π by a Bernoulli random variable with a success probability $p_{i\pi}$. The number of colors on this path π will then correspond to the number of successes in $|\mathcal{C}|$ such trials, which follows the binomial distribution $B(|\mathcal{C}|, p_{i\pi})$. The expected number of colors on the path is given by $|\mathcal{C}|p_{i\pi}$ which clearly increases as the length of the path increases. The probability that the number of colors on π is k is given by $\binom{|\mathcal{C}|}{k} \cdot p_{i\pi}^{k} \cdot (1-p_{i\pi})^{|\mathcal{C}|-k}$. For example, if the number of colors $|\mathcal{C}| = 50$, then the probability that a path of length 20 uses a small number, say 5, colors is about 10^{-4}.

Therefore, in order to construct colored graph instances where there is significant difference between the paths computed by a color oblivious algorithm such as Dijkstra and the optimal path, we need to ensure that (a) there are a large number of paths between the source vertex s and destination vertex t (b) the vertices s and t are reasonably far apart. The first condition maximizes the probability that there will be an s-t path with a small number of colors. The second condition ensures that the expected number of colors on any s-t path is large, and it is quite likely that a color oblivious algorithm is fooled into taking one of these expensive paths.

4.2 Constructing Hard Instances

We construct our instances in two steps. First, we show how to construct the underlying graph $G = (V, E)$ and next describe how to assign colors on edges of G. We begin by assuming that unless otherwise stated, the vertices s and t are always assigned to be the pair of vertices that are farthest apart in G, that is, they realize the diameter of G. The idea now is to construct graphs that have large diameters (so that s, t are reasonably separated), are 'locally' dense (so that there is a large number of s-t paths) and capture application scenarios for min-color path problems.

- **Layered Graphs** These graphs comprise of n nodes arranged in a $k \times (n/k)$ grid. Each column consists of k nodes that form a layer and consecutive layers are fully connected. More precisely, a node v_{ij} in column j is connected to all nodes v_{lj+1} in the next column and the for all $l = \{1, \ldots, k\}$. All vertices in the first column are connected to the source s, and the last column are connected to t. Such graphs are known to appear in design of centralized telecommunication networks [10], task scheduling, or software architectures.

- **Unit Disk Intersection Graphs** These graphs comprise of a collection of n unit disks randomly arranged in a rectangular region. The graph is defined as usual, each disk corresponds to a vertex and is connected to all the other disks it intersects. Since the edges only exist between vertices that are close to each other, disk intersection graphs tend to have large diameters proportional to the dimensions of the region they lie in. These graphs appear quite frequently in ad-hoc wireless communication networks [13]
- **Road Networks** These graphs intend to capture applications of min-color path to transportation networks such as logistics, where colors may correspond to trucking companies that operate between certain cities, and one would like to compute a path with fewest number of contracts needed to send cargo between two cities. For these graphs, we simply use the well-known road network datasets such as the California road network from [18]. As one may expect, road networks also tend to have large diameters.

Next, we assign colors to edges of the graph G. To keep things simple, we will assign colors to edges of G independently. We consider edges of G one by one and assign them up to z colors, by sampling the set of colors z times. However, in order to also capture that some colors are more likely to occur than others, we sample the colors from a truncated normal distribution as follows. We start with a normal distribution with mean $\mu = 0.5$, standard deviation $\sigma = 0.16$ (so that $0 < \mu \pm 3\sigma < 1$) and scale it by the number of colors. Now we sample numbers from this distribution rounding down to the nearest integer. With high probability, the sampled color indices will lie in the valid range $[0, |\mathcal{C}|)$, otherwise the sample returns an empty color set.

Table 1. Number of colors used by Dijkstra vs best known solutions on various colored graph instances. Note the higher difference between Dijkstra and optimal values for our instances.

Instance	Dijkstra	Best known	Remarks
Layered	43.38	17.6	$k = 4$ nodes per layer
Unit-disk	34.66	13.88	$n = 1000$ random disks in a 10×100 rectangle
Road-network	366	246	$1.5M$ nodes, $2.7M$ edges, 500 colors
Uniform-col [19]	11.45	9.5	edges added with $p = \log n/2n$

Finally, for the sake of comparison, we also include the randomly generated colored graphs (Uniform-col) from [19]. The number of colors used by Dijkstra's algorithm and the optimal number of colors are shown in Table 1. For all the datasets except Road-network, the number of nodes is 1000, the number of colors is 50 and the number of samples per edge was 3. The reported values are averaged over 20 runs. For the Uniform-col instances, the probability of adding edges p

was chosen so that the difference between colors used by Dijkstra's algorithm and the optimal is maximized.

4.3 ILP Formulation

We will now discuss an ILP formulation to solve the min-color path problem exactly. Given a colored graph $G = (V, E, \mathcal{C})$, the formulation is straightforward. We have a variable c_i for each color $i \in \mathcal{C}$, and another variable e_j for each edge $j \in E$. The objective function can be written as:

$$\begin{aligned} \mathit{minimize} \sum_i c_i & \qquad \text{subject to} \\ c_i \geq e_j & \qquad i \in \chi(j) \;\; \text{(color } i \text{ lies on edge } j\text{)} \quad (1) \\ \sum_{j \in out(v)} e_j - \sum_{j \in in(v)} e_j = \left\{ \begin{array}{ll} 1, & v = s \\ -1, & v = t \\ 0, & v \neq s, t \end{array} \right\} & \qquad \forall v \in V \quad (2) \\ c_i, e_j \;\in\; \{0, 1\} & \end{aligned}$$

The first set of constraints (1) ensure that whenever an edge is picked, its colors will be picked as well. The second set of constraints (2) ensure that the set of selected edges form a path. We implemented the above formulation using Gurobi MIP solver (version 8.0.1) and found that they run surprisingly fast (within a second) on Uniform-col instances from Table 1. However, the solver tends to struggle even on small instances (about a hundred nodes) of all other datasets, suggesting that these instances are indeed challenging. In the next section, we will discuss a couple of heuristic algorithms that can compute good paths reasonably fast, and later in Section 4.5 compare their results with the optimal values computed by the ILP solver for some small instances.

4.4 Greedy Strategies

We begin by noting that a reasonably long path that uses small number of colors must repeat a lot of its colors. Therefore, the primary challenge is to identify the set of colors that are likely to be repeated on a path, and "select" them so that they are not counted multiple times by a shortest path algorithm. This selection is simulated by removing the color from the colored graph, so that subsequent runs of of shortest path algorithm can compute potentially better paths. Inspired from the approximation algorithm of Section 3, our first heuristic Greedy-Select simply selects the colors greedily based on the number of times they occur on edges of G and returns the best path found. We outline the details below.

Algorithm: **Greedy-Select** ***Colors*** The input to the algorithm is a colored graph $G = (V, E, \mathcal{C}, \chi)$ and it returns an $s - t$ path π.

1. Find an initial path π_0 by running Dijkstra's algorithm on G with weight of each edge $e = |\chi(e)|$. Let the number of the colors used by π_0 is K, an upperbound on number of colors our paths can use. Set the path $\pi = \pi_0$.
2. Initialize $i = 1$, and set $G_0 = G$ the original colored graph.
3. Remove the color $c_{\max}$ that appears on maximum number of critical edges of G_{i-1}. That is, set $\chi(e) = \chi(e) - \{c_{\max}\}\ \forall e \in E$. Let G_i be the colored graph obtained.
4. Compute the minimum weight path π_i in G_i with weight of each edge $e = |\chi(e)|$ using Dijkstra's algorithm.
5. Let K' be the number of colors on π_i in the *original* colored graph G. If $K' < K$, update the upperbound $K = K'$ and set $\pi = \pi_i$.
6. if $i < K$, set $i = i + 1$ and return to Step 3. Otherwise return path π.

Although the above algorithm runs quite fast and computes good paths, we can improve the path quality further by the following observation. Consider a color c_i that occurs on a small number of edges, then using an edge that contains c_i (unless absolutely necessary) can be detrimental to the path quality, as we may be better off picking edges with colors that occur more frequently. This suggests an alternative greedy strategy: we try to guess a color that the path is *not likely* to use, discard the edges that contain that color, and repeat the process until s and t are disconnected. This way we arrive at a small set of colors from the opposite direction, by iteratively discarding a set of 'expensive' candidates. To decide which color to discard first, we can again use their number of occurrences on edges of G – a small number of occurrences means a small number of edges are discarded and s-t are more likely to remain connected. However, we found that this strategy by itself is not as effective as Greedy-Select, but one can indeed combine both these strategies together into the Greedy-Prune-Select heuristic, that is a little slower, but computes even better paths.

Algorithm:* Greedy-Prune-Select *Colors The input is a colored graph $G = (V, E, \mathcal{C}, \chi)$, and a parameter *threshold* that controls the number of times we invoke Greedy-Select heuristic. The output is an $s - t$ path π.

1. For each color $c \in \mathcal{C}$, initialize *preference*(c) to be number of edges it occurs on. Initialize $i = 0$, $G_0 = G$ to be the initial graph, and $\mathcal{C}_0 = \mathcal{C}$ to be the initial set of candidate colors that can be discarded.
2. Run Greedy-Select on G_0 to find an initial path π_0 to improve upon. Record the number of edges $M = |E|$ in the graph at this point.
3. Repeat the following steps until $\mathcal{C}_i$ is empty:
   (a) Pick a color $c_i \in \mathcal{C}_{i-1}$ such that *preference*(c_i) is minimum, and remove all edges e such that $c_i \in \chi(e)$. Let G_i be the graph obtained, and $\mathcal{C}_i = \mathcal{C}_{i-1} \setminus \{c_i\}$.
   (b) If s, t are disconnected in G_i, restore the discarded edges. That is set $G_i = G_{i-1}$. Set $i = i + 1$ and return to Step 3.
   (c) Otherwise, remove all edges from G_i that do not lie in the same connected component as s, t. Update *preference* of all colors that lie on these discarded edges.

   (d) If the graph G_i has changed significantly, that is $M - |E_i| \geq \textit{threshold}$ or if this is the last iteration, run Greedy-Select again to compute the path π_i. Update $M = |E_i|$.
   (e) Set $i = i + 1$ and return to Step 3.
4. Return the path π_i that is best in terms of number of colors.

The running time is typically dominated by the number of calls to Greedy-Select. In our experiments, we set $\textit{threshold} = 0.25|E|$ which guarantees that we only make a small number of calls to Greedy-Select. Theoretically, Greedy-Select runs in $O(|\mathcal{C}| \cdot D)$ time, where D is the running time of Dijkstra's algorithm. An implementation of Greedy-Prune-Select using BFS to test connectivity runs in $O(|\mathcal{C}| \cdot (|V| + |E|)) + O(|\mathcal{C}| \cdot D)$ time, which is also $O(|\mathcal{C}| \cdot D)$. This is an order of magnitude better than SPACOA heuristic that has a worst-case running time of $O(|\mathcal{C}|^2 \cdot D)$.

4.5 Experiments and Results

We will now discuss the performance of above heuristic algorithms on our datasets. We compare our results with the values computed by the ILP solution (on small instances) and the SPACOA heuristic from [19]. In summary, we found that both our heuristics compute paths that are much better than SPACOA heuristic from [19], while also being significantly faster. The paths computed by Greedy-Prune-Select are almost always significantly better than Greedy-Select and the difference especially shows on larger datasets. The results are shown in Tables 2 to 5 averaged over five runs with the exception of real-world instances. Runtimes longer than one hour are marked with ∞. Some more experimental results with different color distributions and another synthetic dataset can found in Tables 6 and 7 in the Appendix. All code was written in C++ using the OGDF graph library [4] and executed on a standard linux machine (Ubuntu 16.04) running on Intel(R) Core(TM) i5-4460S CPU @ 2.90GHz with 16GB RAM.

Layered Graph Instances We run our algorithms on a 4×125 layered graph instance with 50 colors on a 4×2500 instance with 500 colors. As the number of layers grows from 125 to 2500, these instances get progressively challenging for the ILP solver due to a large number of candidate paths. As expected, the SPACOA runs really slow on large instances as it needs to try a lot of colors per iteration. There is reasonable difference between the quality of paths computed by Greedy-Prune-Select and Greedy-Select especially on larger instances. The results are shown in Table 2.

Unit-disk Instances We run our algorithms on two sets of instances, with 500 nodes (disks) in a 10×50 rectangle, and a 10^4 nodes in a 10×1000 rectangle. The rectangles are chosen narrow so that the graph has a large diameter. The behavior is quite similar to layered graphs. The results are shown in Table 3.

Table 2. Path quality and running time on layered graph instances.

Algorithm	Colors used	Time taken (ms)	Colors used	Time taken (ms)
	$4 \times 125 = 0.5k$ nodes		$4 \times 2500 = 10k$ nodes	
Dijkstra	36.8	0.6	441.8	23.6
SPACOA	33.6	65	396	127×10^3
Greedy-Select	18.2	12.6	185.6	3.5×10^3
Greedy-Prune-Select	17.2	49	173	12.5×10^3
ILP	16.4	707×10^3	∞	∞

Table 3. Path quality and running time on Unit disk graph instances.

Algorithm	Colors used	Time taken (ms)	Colors used	Time taken (ms)
	$4 \times 125 = 0.5k$ nodes		$4 \times 2500 = 10k$ nodes	
Dijkstra	28.8	1	357.8	38
SPACOA	23	124.8	333.6	41.4×10^3
Greedy-Select	14.2	13	145.6	4.7×10^3
Greedy-Prune-Select	13.4	55	134	17.6×10^3
ILP	12.6	1176×10^3	∞	∞

Real-world Instances Next, we focus on a couple of real-world examples. Our first instance is the California road network [18] that has $1.5M$ nodes and $2.7M$ edges. The graph however was not colored to begin with, so we color it artificially by assigning 500 colors from the truncated normal distribution as explained before. Our second instance is from the Internet Topology Zoo [15], a manually compiled dataset of connectivity of internet service providers over major cities of the world. We translate this to our colored graph model, the cities naturally correspond to nodes, providers correspond to colors, and a color is assigned to an edge if the corresponding provider provides connectivity between these two cities. This graph has $5.6k$ nodes, $8.6k$ edges and 261 colors, with an average of 1.44 colors per edge.

Table 4. Path quality and running time on some real world instances.

Algorithm	Colors used	Time taken (ms)	Colors used	Time taken (ms)
	CA Road Network		Internet topology	
Dijkstra	366	3.068×10^3	7	26
SPACOA	355	3.12×10^6	4	3111
Greedy-Select	251	0.73×10^6	5	29
Greedy-Prune-Select	246	2.71×10^6	4	286
ILP	∞	∞	4	1817

The road-network instance due to its size is challenging to all algorithms. On the other hand, the internet-topology dataset seems quite easy for all the instances. One possible explanation for this is that although the number of nodes is large, the graph has a lot of connected components and a small diameter. This limits the space of candidate paths making all algorithms (particularly the ILP solver) quite fast.

Uniform-Col Instances These instances are the same as one from [19] and have been mostly included for the sake of comparison. We run our algorithms on a Uniform-Col instance with 10^3 nodes and 50 colors, and another instance with 10^4 nodes and 500 colors.

Table 5. Path quality and running time on Uniform-Col instances.

Algorithm	Colors used	Time taken (in ms)	Colors used	Time taken (in ms)
	10^3 nodes		10^4 nodes	
Dijkstra	11.45	0.95	11.7	15
SPACOA	9.95	75.45	11.2	8664
Greedy-Select	10.4	9.2	11.5	150
Greedy-Prune-Select	10.3	46.3	11.4	1919
ILP	9.5	3913.8	-	-

The SPACOA heuristic performs marginally better than our heuristics on these examples. The primary reason for this is that the difference between optimal solution and that computed by Dijkstra's algorithm is really small (about 2), and the SPACOA heuristic typically overcomes this difference in just one iteration by trying all colors and picking the one that gives the best path. The cases in which SPACOA heuristic struggles to find good paths is when it has to try multiple iterations to bridge the gap between Dijkstra's algorithm and optimal, and gets stuck in a local minima. That is less likely to happen when the difference between optimal and Dijkstra value is small. This is also evident from the performance of SPACOA heuristic on union of random colored graphs (basically a set of concatenated Uniform-Col Instances) shown in Table 6 (in Appendix). In this case, the difference between optimal and Dijkstra value is larger and SPACOA heuristic fails to compute good paths.

5 Conclusion

In this paper, we made progress on the min-color path problem by showing that under plausible complexity conjectures, the problem is hard to approximate within a factor $O(n^{1/8})$ of optimum. We also provide a simple $O(n^{2/3})$-approximation algorithm and designed heuristic algorithms that seem to perform quite well in practice. A natural open question is to see if these bounds can be improved further. The log-density framework has been useful in designing tight approximation bounds for related problems such as minimum k-union [6] and densest k-subgraph [3]. It would be interesting to see if those techniques can be applied to min-color path.

References

1. Bandyapadhyay, S., Kumar, N., Suri, S., Varadarajan, K.: Improved Approximation Bounds for the Minimum Constraint Removal Problem. In: APPROX 2018. LIPIcs, vol. 116, pp. 2:1–2:19 (2018)
2. Bereg, S., Kirkpatrick, D.G.: Approximating barrier resilience in wireless sensor networks. In: ALGOSENSORS'09. pp. 29–40 (2009)
3. Bhaskara, A., Charikar, M., Chlamtac, E., Feige, U., Vijayaraghavan, A.: Detecting high log-densities: an $O(n^{1/4})$ approximation for densest k-subgraph. In: Proceedings of the 42nd STOC. pp. 201–210. ACM (2010)
4. Chimani, M., Gutwenger, C.: The Open Graph Drawing Framework (OGDF).
5. Chlamtac, E., Dinitz, M., Krauthgamer, R.: Everywhere-sparse spanners via dense subgraphs. In: Proceedings of the 53rd FOCS. pp. 758–767 (2012)
6. Chlamtáč, E., Dinitz, M., Makarychev, Y.: Minimizing the union: Tight approximations for small set bipartite vertex expansion. In: Proceedings of the 28th SODA. pp. 881–899 (2017)
7. Chlamtáč, E., Manurangsi, P., Moshkovitz, D., Vijayaraghavan, A.: Approximation algorithms for label cover and the log-density threshold. In: Proceedings of the 28th SODA. pp. 900–919 (2017)
8. Fellows, M.R., Guo, J., Kanj, I.: The parameterized complexity of some minimum label problems. Journal of Computer and System Sciences **76**(8), 727–740 (2010)
9. Goel, G., Karande, C., Tripathi, P., Wang, L.: Approximability of combinatorial problems with multi-agent submodular cost functions. In: Proceedings of the 50th FOCS. pp. 755–764 (2009)
10. Gouveia, L., Simonetti, L., Uchoa, E.: Modeling hop-constrained and diameter-constrained minimum spanning tree problems as steiner tree problems over layered graphs. Mathematical Programming **128**(1-2), 123–148 (2011)
11. Hassin, R., Monnot, J., Segev, D.: Approximation algorithms and hardness results for labeled connectivity problems. J. Comb. Optim. **14**(4), 437–453 (2007)
12. Hauser, K.: The minimum constraint removal problem with three robotics applications. The International Journal of Robotics Research **33**(1), 5–17 (2014)
13. Huson, M.L., Sen, A.: Broadcast scheduling algorithms for radio networks. In: Proceedings of MILCOM'95. vol. 2, pp. 647–651. IEEE (1995)
14. Jha, S., Sheyner, O., Wing, J.: Two formal analyses of attack graphs. In: Computer Security Foundations Workshop. pp. 49–63. IEEE (2002)
15. Knight, S., Nguyen, H.X., Falkner, N., Bowden, R., Roughan, M.: The internet topology zoo. IEEE Journal on Selected Areas in Communications **29**(9), 1765–1775 (2011)
16. Krumke, S.O., Wirth, H.C.: On the minimum label spanning tree problem. Information Processing Letters **66**(2), 81–85 (1998)
17. Kumar, N.: Minimum Color Path Experiments (Github Repository). `http://github.com/sud03r/min-color-path` (2019)
18. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics **6**(1), 29–123 (2009)
19. Yuan, S., Varma, S., Jue, J.P.: Minimum-color path problems for reliability in mesh networks. In: INFOCOM'05. vol. 4, pp. 2658–2669 (2005)

6 Appendix

6.1 Proof of Lemma 2

Suppose in the min-color path instance G from Lemma 1, we are also given as input OPT, the number of colors that the path can use. Then, we can easily remove all edges that have more than OPT number of colors as a min-color path will never use them. Now since $|\chi(e)| \leq OPT$, we can create an instance G' of min-label path by subdividing each edge into a path of at most OPT edges, each containing exactly one color. It is easy to see that a min-label path in G' corresponds to a min-color path in G and vice-versa. However, the number of vertices in G' is OPT times the number of vertices in G. It is easy to see that an $O((\frac{n}{OPT})^{1/8})$ approximation in this min-label path instance yields an $O(n^{1/8})$ approximation for the min-color path instance where OPT is given as input. Observe that since $OPT \in \{1, 2, \ldots, |\mathcal{C}|\}$, we can run this min-color path approximation and take the path that uses minimum number of colors over all $|\mathcal{C}|$ possible guesses for OPT, giving us an $O(n^{1/8})$ approximation for the general min-color path instance, which is a contradiction.

6.2 Union of Random Graphs

In this section, we intend to capture some random graph models similar to the previous work [19]. The intuition is that although graphs from applications are not inherently random, they can still be thought of as a union of random components that are connected together. This ensures that the diamater is large and there are fairly large number of s-t paths making these instances reasonably challenging. We consider a simple model which comprises of k random graphs each with n/k nodes concatenated in series. More precisely, we have k random graphs $G_1, G_2, \ldots, G_k$, and let s_i, t_i be the terminals of G_i (nodes that are farthest apart in G_i). We now simply connect these graphs in series along the terminals as follows : $(s_1 - G_1 - t_1) \ldots (s_i - G_i - t_i) \ldots (s_k - G_k - t_k)$ to obtain the graph G. The number of random graphs is always 20, and $n = 1k, 50$ colors for small instances, and $n = 10k, 500$ colors for large instances.

Table 6. Path quality and running time on union of Random Graphs, $k = 20$.

Algorithm	Colors used	Time taken (ms)	Colors used	Time taken (ms)
	$20 \times 50 = 1k$ nodes		$20 \times 500 = 10k$ nodes	
Dijkstra	39.8	1.4	210.2	24
SPACOA	35.7	161	189.4	47.3×10^3
Greedy-Select	31	38	179.4	3.4×10^3
Greedy-Prune-Select	29.8	100	150.6	13.3×10^3
ILP	29.7	980	-	-

6.3 Uniformly-Colored Synthetic Instances

In this section, we analyze how our heuristics perform with an alternative method of assigning colors. We pick two of our synthetic instances (namely layered, unit-disk) and assigning colors to the edges uniformly. That is for all edges $e \in E$, sample a color uniformly from the set of colors $\mathcal{C}$ and add it to $\chi(e)$. Repeat the sample three times to ensure an assignment of about 3 colors per edge. We only color large instances. Observe that since each color is now equally likely to occur, the paths are likely to have more colors now. Surprisingly enough, for layered graph instances, SPACOA heuristic generally fails to improve at all upon Dijkstra estimate. One possible explanation for that is discarding a single color does not improve a path at all in terms of number of colors and the algorithm gets stuck in a local minima. However, our heuristics continue to perform significantly better with Greedy-Prune-Select being the clear winner by a significant margin.

Table 7. Path quality and running time on uniformly-colored layered and unit-disk graph instances.

Algorithm	Colors used	Time taken (ms)	Colors used	Time taken (ms)
	Layered, $10k$ nodes		unit-disk, $10k$ nodes	
Dijkstra	500	23.4	480.6	35.8
SPACOA	499.6	7.0×10^3	462.2	59.2×10^3
Greedy-Select	358.2	12.5×10^3	296.4	9.2×10^3
Greedy-Prune-Select	328.8	22.4×10^3	263.4	29.7×10^3