Paths, Proofs, and Perfection: Developing a Human-Interpretable Proof System for Constrained Shortest Paths

Konstantin Sidorov,[1] **Gonçalo Homem de Almeida Correia,** [2]
Mathijs de Weerdt, [1] **Emir Demirović** [1]

[1] Algorithmics Group, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology
[2] Faculty of Civil Engineering and Geosciences, Delft University of Technology
{k.sidorov, g.correia, m.m.deweerdt, e.demirovic}@tudelft.nl

Abstract

People want to rely on optimization algorithms for complex decisions but verifying the optimality of the solutions can then become a valid concern, particularly for critical decisions taken by non-experts in optimization. One example is the shortest-path problem on a network, occurring in many contexts from transportation to logistics to telecommunications. While the standard shortest-path problem is both solvable in polynomial time and certifiable by duality, introducing *side constraints* makes solving and certifying the solutions much harder. We propose a proof system for constrained shortest-path problems, which gives a set of logical rules to derive new facts about feasible solutions. The key trait of the proposed proof system is that it specifically includes high-level graph concepts within its reasoning steps (such as connectivity or path structure), in contrast to using linear combinations of model constraints. Using our proof system, we can provide a step-by-step, human-auditable explanation showing that the path given by an external solver cannot be improved. Additionally, to maximize the advantages of this setup, we propose a proof search procedure that specifically aims to find small proofs of this form using a procedure similar to A* search. We evaluate our proof system on constrained shortest path instances generated from real-world road networks and experimentally show that we may indeed derive more intepretable proofs compared to an integer programming approach, in some cases leading to much smaller proofs.

1 Introduction

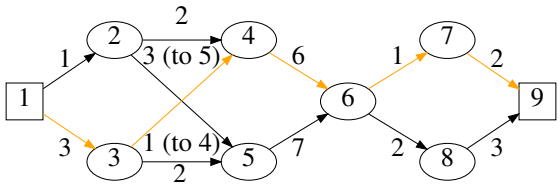
Combinatorial optimization has achieved remarkable success in various practical domains, including supply chain management, network design, and logistics. Despite most of such problems being NP-complete, many approaches can efficiently solve them in practice (Korte and Vygen 2000). The downside of this success is that the software for solving these problems is typically very sophisticated, and testing such software is a non-trivial task (Gillard, Schaus, and Deville 2019). Even modern optimization software may occasionally produce incorrect output, e.g., by falsely claiming infeasibility (Gocht, McCreesh, and Nordström 2022; Cheung, Gleixner, and Steffy 2017).

One way to address this issue is to augment the software output with a *certificate* – a logical derivation supporting the optimality/infeasibility claim. One of the most celebrated examples of this idea is proof logging in Boolean SAT solving, achieving major successes in, among other domains, software verification and combinatorial designs (Biere, Heule, and van Maaren 2021). Related approaches are known for mixed-integer programming (Cheung, Gleixner, and Steffy 2017) and pseudo-Boolean solving (Gocht, McCreesh, and Nordström 2020).

Proof systems of this type are generic enough to support various application domains. This ends up being not only the key strength of this approach but also its main weakness: proofs produced by such means, while exhaustive, also tend to be exhausting, meaning that such proofs may contain lengthy sequences of inferences not obvious to a human observer. In particular, the work (Heule, Kullmann, and Marek 2016) that resolves the Boolean Pythagorean triples problem was billed as the "largest proof ever" shortly after its publication (Lamb 2016), referring to the fact that the proof *exceeds 200 terabytes* in disk space.

For a simple example supporting this concern, consider a problem of finding the shortest path from vertex 1 to vertex 9 through the graph shown in Figure 1 *that goes through either vertex* 3 *or vertex* 5, therefore adding an extra constraint to the classical shortest-path problem.

Figure 1: Running example of a constrained shortest path problem. Orange edges constitute the optimal path $1 \to 3 \to 4 \to 6 \to 7 \to 9$.



This problem can be successfully modeled with, for example, an integer programming paradigm, giving a path 13 units long highlighted in the figure. Additionally, given the optimal path, we can build its optimality certificate with respect to the integer programming model. However, by inspecting such proof one may rightfully notice that this problem has much more "graph" intuition than the underlying integer programming model – e.g., the concepts of distance or

connectivity get "lost in translation". This leads to a hypothesis that choosing a more specific problem class can lead to more succinct and maybe even more interpretable proof.

In this work, we address this concern for a more general version of the problem, namely *constrained shortest path problems* in which the path must satisfy some arbitrary set of conditions (in the example above such condition would read "go through some vertex from the set $\{3, 5\}$"). While this problem domain is relatively simple to describe, it is compelling for evaluation due to (a) its practical relevance and (b) its computational hardness (this is an NP-complete problem as a generalization of the traveling salesman problem). By keeping a degree of freedom in the choice of the feasible set, we obtain a problem class that is more narrowly defined than for typical certification approaches while also being representative of more nuanced optimization problems.

We design a proof system for the constrained shortest path problems that exploits knowledge specific to the graph-theory domain. One example is the distance reasoning which reduces the search space substantially by discarding vertices that are "too far" from the start and finish vertices. We also show that the proof structure that we propose satisfies the standard requirements for a proof system, i.e., soundness (no suboptimal solution can be proven) and completeness (any optimal solution can be proven).

This results in proofs that are easier to understand for humans, which has an important added benefit: it may increase trust in the optimization software. For example, when human operators receive the output of the combinatorial solver, they may either doubt some particular decisions (e.g., peculiar edge choices) or not believe the results outright. Having a more understandable proof may help in these situations.

Next, we propose an algorithm for proof search and evaluate it against the state-of-the-art implementation for integer programming (Cheung, Gleixner, and Steffy 2017), a more general-purpose proof system. The key outcome of our evaluation is that searching for proofs in the proposed system results in proofs that (1) consist of building blocks related to the graph theory domain and are therefore more interpretable to humans, and (2) are in some cases much shorter. Our explanation for that is the fact that individual proof steps contain more powerful reasoning rules for this problem class.

To summarize, the main contributions of this paper are a proof system for constrained shortest path problems and an approach for constructing optimality proofs for this problem class that produces concise proofs on a pool of road network instances with different constraint types. The rest of the paper is structured as follows. In Section 2, we introduce the prior work in certificate generation relevant to our research. We formally introduce the constrained shortest path problem in Section 3. Section 4 introduces the main contributions of this paper, viz. a proof system for such problem instances and an algorithm for constructing proofs in it. Further, we conduct an experimental evaluation of this approach in Section 5, showing that the resulting algorithm is consistently able to outperform the proof generation for integer programming in terms of the proof size. Finally, we conclude from the demonstrated results and outline the potential future work directions in Section 6.

2 Related work

We start with an introduction to the current approaches to proving the optimality of a solution given a problem instance. We review techniques used for generating proofs for integer programs, their connection to solving algorithms, and discuss domain-specific approaches oriented toward interpretability by human viewers.

Certificate generation for integer programming. The key idea behind optimality certification is to introduce a *proof system* which would be sound and complete while still being tractable to verify. To bring this idea closer to the developments in this paper, we introduce an example showing how it may look like in practice.

Example 1. *Consider the following integer program:*

$$\begin{aligned} & 2x + 3y \to \max \\ (1)\ & x + 2y \leq 3 \\ (2)\ & 4x + 5y \leq 10 \\ & x, y \in \mathbb{Z}_{\geq 0}. \end{aligned}$$

The optimal solution $(x, y) = (1, 1)$ has the objective value of 5. Optimality is typically confirmed by claiming an upper bound on the objective. However, this information alone is not sufficient – we require proof that the upper bound is 5.

This problem can be remedied by doing a few linear operations on inequalities and rounding off the results where needed, as illustrated by the example certificate below.

Example 1 (certificate)**.** *The certificate can be represented by the following sequence of operations:*

- $(3) = \frac{1}{3} \times (1) + \frac{2}{3} \times (2) = 3x + 4y \leq 7\frac{1}{3}$,
- $(4) = \lfloor (3) \rfloor = 3x + 4y \leq 7$ *– which is valid since the left-hand side in* (3) *is always integral,*
- $(5) = \frac{1}{2} \times (1) + \frac{1}{2} \times (4) = \underbrace{2x + 3y}_{\textit{objective}} \leq 5.$

The observations invoked in this example can be generalized to make a *cutting-plane proof system* (Cook, Coullard, and Turán 1987), which is both sound and complete for this problem class. This result, while reassuring, does not give much in the way of *finding* the necessary proof. Fortunately, this can be done while *solving* the problem instance, since the cutting-plane method (Gilmore and Gomory 1961), one of the earliest approaches to solving integer programming instances, can be seen as a systematic way of applying cutting plane steps until an optimal solution can be recovered purely by *linear* programming.

Modern integer programming solvers, however, largely follow *branch-and-cut* scheme which embeds cutting plane generation in the branch-and-bound framework (Conforti, Cornuéjols, and Zambelli 2014). To handle the challenge of expressing branch-and-bound steps via cutting-plane reasoning, Cheung et al. introduce a more elaborate proof system in (Cheung, Gleixner, and Steffy 2017) that subsumes cutting-plane reasoning while also supporting tree search reasoning. Other related works include pseudo-Boolean

certificates (Gocht, McCreesh, and Nordström 2020) and propositional logic (Wetzler, Heule, and Hunt 2014).

Explaining the solutions of satisfaction problems. One shortcoming of using the cutting-plane method for certifying the optimality is that the individual steps found in the resulting proof could be seen by a human observer as lengthy and unobvious. We review approaches striving to output certificates understandable to a human observer.

Approaches based on unsatisfiable sets. A common approach to analyzing unsatisfiable instances is to find a subset of constraints that comprise the unsatisfiable instance. Such sets are also known as *minimal unsatisfiable sets* (MUS); a common problem-independent way to find one such set is QuickXplain, as described by Junker in (Junker 2001). Senthooran et al. (Senthooran et al. 2021) have developed a framework for generating MUSes more flexibly, e.g., by generating multiple MUSes or restoring feasibility.

These approaches work well if the model constraints are (a) simple enough to admit meaningful decomposition and (b) complex enough to encode meaningful domain concepts. While there are examples of applied domains where both of these assumptions hold (e.g., plant location), these approaches tend to have trivial/incomprehensible output when either of these conditions fails. For instance, applying this approach to the problem from Figure 1 would yield a response that amounts to solving the same problem on the same graph with a few edges removed.

Sequential explanations. In (Bogaerts, Gamba, and Guns 2021), Bogaerts et al. has considered the problem of walking a user through the solution of a logical puzzle. The main objective of this study is to deliver a complete yet simple explanation of the solution process. The explanations are modeled with sequences of inference steps equipped with a complexity measure. For evaluation, the paper defines one such measure by assigning complexity to individual steps and defining a sequence complexity by aggregating them, e.g., by averaging. Aside from the greedy construction of the explanation, the algorithm also attempts to make nested explanations, i.e., sequences of inferences that prove a single fact by contradiction. To this end, it searches the nested explanations of facts and adds them as soon as they score better than the "parent" explanation.

This scheme is shown to work well for puzzle-like satisfaction problems. However, one distinct trait of this problem class is that puzzles (by design) do not require heavy case enumeration – which is important for our setup since combinatorial optimization problems typically require extensive branching to be solved. While the concept of nested explanations addresses this concern for the few cases when the branching is indeed required, it is not clear whether this idea can scale to more "branched" reasoning processes.

To conclude, the existing research concerning the certification of optimality/infeasibility claims can be largely partitioned into (a) approaches working in general-purpose modeling setups (e.g., integer programming) but providing little from the "interpretability" point of view, and (b) approaches having "explainability-preserving" mechanisms but working for specific domains, which oftentimes lack the solving complexity typical of industrial decision-making problems. In this work, we produce an approach attempting to bridge the gap between proof completeness and domain complexity. To this end, we design a certification approach for a constrained shortest path problem, a problem class that is wide enough to expose patterns typical for more nuanced problem types but narrow enough to have a meaningful, high-level description.

3 Constrained shortest path problem

The shortest path problem is a well-known problem in graph theory that can be formulated as finding a path in a directed weighted graph G between vertices s and t such that the sum of edge weights is minimized. Despite having a combinatorial structure, this particular problem can be solved efficiently using Dijkstra's algorithm (Dijkstra 1959).

However, for some important practical scenarios this optimization problem is not adequate – for example, when not every path in G is feasible according to some external constraints. These variations are commonly known as *constrained shortest path problems*, such as in (Lozano and Medaglia 2013), meaning that there are some additional requirements for the path to be feasible. For many practical variations, such as resource constraints on paths (Lozano and Medaglia 2013) or enforcing usage of some vertices (de Uña et al. 2016), finding the shortest path under this set of restrictions is known to be an NP-complete problem.

In this work we define the class of constrained shortest path problems as follows:

Definition 1. *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph with weight function $w : E \to \mathbb{R}_{\geq 0}$ and s, t be some fixed vertices of $\mathcal{G}$. Also, let $\phi : 2^{\mathcal{V}} \times 2^{\mathcal{E}} \to \{0, 1\}$ be a* side constraint predicate *that defines which paths in $\mathcal{G}$ are feasible. The* constrained shortest path problem *given by $(\mathcal{G}, s, t, \phi)$ is a problem that requires finding a subgraph $G = (V, E) \subseteq \mathcal{G}$ with smallest total weight $\sum_{e \in E} w(e)$ such that G is a simple path satisfying the side constraint $\phi(V, E)$.*

4 Proof system for constrained shortest paths

Proof system structure. Considering that the optimization problem has a graph structure, we can make a hypothesis that optimality proofs for the shortest path problems become more concise by using knowledge specific to graph theory. In the next example, we expand upon the sample problem to demonstrate what this reasoning may look like.

Example 2. *Consider the problem in Figure 1 and observe the following:*

1. *No path of length less than* 13 *can go through vertex* 8 *– since it would have to cover at least the distance from* 1 *to* 8 *(which is 11) and the distance from* 8 *to* 9 *(which is 3), and adding the distance bounds gives* $13 \geq 12$*. Thus, the problem instance remains equivalent even if we remove vertex* 8 *from consideration.*
2. *Any path between vertices* 1 *and* 9 *must include vertex* 6 *since removing it disconnects the graph. Thus, the problem instance remains equivalent if we only consider paths* with *vertex* 6*.*

To generalize this, we use graph-theoretic reasoning to progressively "narrow down" the graph by either removing

its vertices or edges or by enforcing them to be in the path. We formalize this idea by the following definition.

Definition 2. *An* interval $I = [G_\perp, G^\top]$ *is a set of graphs that are bounded (in a set-theoretical sense) between graphs* $G_\perp$ *and* $G^\top$*, i.e.,* $I = \{G : G_\perp \subseteq G \subseteq G^\top\}$*. The graphs* $G_\perp$ *and* $G^\top$ *are called the* bottom graph *and* top graph*.*

Continuing the Example 2, suppose that we maintain an interval $[G_\perp, G^\top]$ as a proof state – starting from the interval with $G_\perp$ being an empty graph and $G^\top$ being the original graph $\mathcal{G}$. In that case:

- the reasoning from step 1 can be reflected by *removing* vertex 8 from $G^\top$ – narrowing the feasible set to a set of paths *not* going (among others) to the vertex 8,
- the reasoning from step 2 can be reflected by *adding* a vertex 6 to $G_\perp$ – narrowing the feasible set to a set of paths going through (among others) the vertex 6.

Thus, the graphs $G_\perp$ and $G^\top$ in the interval $[G_\perp, G^\top]$ can be interpreted as the graph that must be included in *any* feasible path and the graph that contains *all* feasible paths.

Note, however, that any correct reasoning of this form *cannot exclude short infeasible paths*, i.e., if the path G is shorter than some path $G^* = (V^*, E^*)$ claimed to be optimal and $G \in I$ for the current interval I then it should stay in any interval $J \subseteq I$ that we get by "narrowing" reasoning. We reflect this insight using the following definition.

Definition 3. *A graph* $G = (V, E)$ *is called a* candidate path *with respect to* $\mathcal{G}, s, t$ *and optimal solution* G^* *if it satisfies the following conditions: (1) it is a subgraph of* $\mathcal{G}$*, (2) it is a simple path from* s *to* t*, and (3) its length* $\sum_{e \in E} w(e)$ *is less than the length of* G^**.*

Now, keeping in mind the link between proving optimality and solving problem instances, we consider how to develop an algorithm for solving constrained shortest-path problem instances. For instance, we could suggest the following:

(i) At the start, we need to consider all feasible paths from s to t – which in our notation corresponds to the interval spanning from an empty graph to the original graph $\mathcal{G}$.

(ii) If we at some point consider the paths from an interval I, we may "simplify" I by removing some graph elements and enforcing others – resulting in a new interval J such that $J \subseteq I$. (We have introduced earlier some procedures for doing that in polynomial time, such as removing disconnected vertices – the main point here is that these procedures should be correct and efficient.)

(iii) If there is no valid procedure that can reduce an interval I further, we can choose an arbitrary vertex or edge and branch on it, i.e., solving two new problems, one for inclusion and one for exclusion of the element. For example, branching on vertex v creates *two* child intervals J^+, J^- such that J^+ differs from I by adding v to the bottom graph while J^- differs from I by removing v from the top graph. Note that this procedure can be applied regardless of the actual interval (as long as it does not collapse to a single graph) and yields two child intervals that cover the original interval, i.e., $J^+ \cup J^- = I$.

(iv) Finally, if we have an interval I such that every path in it is infeasible (e.g., I does not contain any of the mandatory vertices), we can safely dispose of it. Here we assume that there is a polynomial-time procedure for checking infeasibility – e.g., for a mandatory vertex constraint this may be a procedure that reports infeasibility if there is a mandatory set V^+ such that $V \cap G^\top = \emptyset$. In particular, we do not need the full resolution procedure for that problem – if the problem is infeasible but our procedure does not report it that simply implies that the proof would have to enumerate a few more cases.

The next definition puts all of these points together to produce a formal definition of the proof that we use further on:

Definition 4. *Let* $(\mathcal{G}, s, t, \phi)$ *a constrained shortest path problem and* $G^* = (V^*, E^*)$ *be a path claimed to be optimal. Suppose that there is a predicate* $\Phi(I)$ *on intervals* I *that can be computed in polynomial time such that* $\Phi(I)$ *being false implies that there is no path* $G \in I$ *such that* $\phi(G)$ *(while the converse may not necessarily hold) and* $\Phi([G, G]) = \phi(G)$ *for any path* G*. Then a* proof $\mathcal{P}$ *is a rooted tree* $(\mathcal{I}, \mathcal{A})$ *where each vertex in* $\mathcal{I}$ *is associated with an interval,* $\mathcal{A}$ *is the set of directed edges between them and:*

(i) its root is an interval $I_{root} = [G_0, \mathcal{G}]$ *where* G_0 *is a graph with vertices* $\{s, t\}$ *and no edges, called* trivial root*;*

(ii) child intervals contract with respect to their parents, i.e., if interval $I \in \mathcal{I}$ *then all of its children* J *are its subsets:*

$$\bigcup_{J:(I,J)\in\mathcal{A}} J \subseteq I;$$

(iii) candidate paths (as understood in Definition 3) are retained in child intervals, i.e., for any path G

$$\begin{cases} G \text{ is a candidate path} \\ G \in I \in \mathcal{I} \end{cases} \implies \exists J : (I, J) \in \mathcal{A}, G \in J.$$

(iv) leaf intervals are Φ*-infeasible: if* I *is a leaf of* $\mathcal{P}$ *then* $\Phi(I)$ *is false.*

Soundness and completeness of the proof system. We show that the previously defined proofs are complete, i.e., that if the claimed solution G^* is indeed optimal then there is a proof of this fact in the sense of Definition 4.

Lemma 1. *Let* $(\mathcal{G}, s, t, \phi)$ *be a constrained shortest path problem with an optimal solution* $\mathcal{G}^*$*. Then there exists a proof* $\mathcal{P}$ *satisfying all requirements of Definition 4.*

Conversely, the proof system we introduced is sound, meaning that the existence of proof implies optimality.

Lemma 2. *Let* $(\mathcal{G}, s, t, \phi)$ *be a constrained shortest path problem and* $\mathcal{G}^*$ *be a feasible path. Suppose that there is a proof* $\mathcal{P}$ *compliant with Definition 4. Then* G^* *is an optimal solution of a constrained shortest path problem.*

The proofs of both lemmas can be found in Appendix A; the intuition is that the proof system is expressive enough to admit proof by exhaustion – a proof that systematically goes over each feasible path – but not expressive enough to construct a "bogus" proof, since the true optimal path should find its way to a leaf node.

Thus we can conclude that Definition 4 indeed encodes a concept of proof for the domain of constrained shortest path problems. However, this definition gives little in the way of *constructing* the proofs – in particular, it is not clear how can the proof graph be "grown" out of the fixed source interval. On the other hand, we know that proving the optimality is often tightly connected with the search – in fact, it can be seen as a problem of constructing a valid and succinct search tree. Thus we introduce next the building blocks of a search tree (as imagined in Example 2) into our proof system concept.

Reasoning steps. A proof in our system consists of two reasoning types: *linear* reasoning, which narrows down an interval by adding components to its bottom graph or removing components from its top graph, and *case-based* reasoning, which takes a vertex/edge, constructs two proofs for it being present/absent, and merges them into a new proof tree.

Now, we formally introduce each of the reasoning types, starting with the linear reasoning part.

Definition 5. *Given a constrained shortest path problem $(\mathcal{G}, s, t, \phi)$ and an optimal solution G^*, a* reasoning rule *is a mapping between intervals of $\mathcal{G}$ that is (1) contracting: if interval I is mapped to interval J then $J \subseteq I$, (2) retaining candidate paths: if interval I is mapped to interval J and G is a candidate path such that $G \in I$ then $G \in J$.*

This definition allows some freedom in choosing the concrete set of reasoning rules. As an example, our *out-of-range* reasoning removes vertices v if every path that includes vertex v going from the starting vertex to the end vertex has a greater or equal cost to the optimal distance – for the list of rules used during the proof generation, see Appendix B.

As for the case-based reasoning, that can be done by branching on vertices and edges of the original graph – this gives two procedures for "growing" a tree of intervals from a leaf $[G_\perp, G^\top]$, viz. *branch on vertex* and *branch on edge*.

Formally, branching on the vertex (edges are handled symmetrically) starts with an interval $I = [G_\perp, G^\top]$ and, given a vertex $v \in G^\top$, connects two new (child) intervals to I. One of them enforces a vertex, i.e., $I \to [G_\perp + v, G^\top]$, while the other one removes it, that is, $I \to [G_\perp, G^\top - v]$.

Thus, we have a "restricted" version of the proof system from Definition 4 where proofs have the same properties but can be constructed from I_{root} by either applying reasoning steps or executing branching. Soundness in this setup is obvious, while completeness depends on the particular choice of reasoning steps; for further details, see Appendix A.

Proof generation. The proof structure that we have discussed so far largely follows the typical solving procedure: we start with an original problem, simplify it, split it into two new problems (or less), and repeat until all of the cases were accounted for. However, there is a notable difference between these two processes: the shape of the search tree is irrelevant when *solving* the problem but is very important for *certification* since the more compact proof tree shape leads to a faster certification.

One corollary of that is that in our setup we can assume that (a) we already have an optimal path G^* and (b) we need to show that fact in the most succinct way possible. To address these points, we use the search strategy similar to A* search on graphs as shown in Algorithm 1. In short, this procedure repeatedly walks the graph of intervals (as implemented in Algorithm 2) – favoring the ones leading to proofs with the smallest number of leaves – until the first new interval is encountered when it extends the "known" part of the graph and starts a new walk. The procedure terminates as soon as the complete proof can be reconstructed. Note that this does not necessarily produce the smallest proof, however in experimentation we show this works reasonable well.

Algorithm 1: A proof search procedure

Data: Constrained shortest path problem instance $Q = (\mathcal{G}, s, t, \phi)$
Data: Claimed optimal path G^* or $\emptyset$ if the instance is claimed to be infeasible
Result: Proof tree $\mathcal{P}$ certifying the optimality/unsatisfiability

```
// See also Definition 6
```

$r \leftarrow (R, \text{Simplify}(R))$;
$\mathcal{T} \leftarrow (\{r\}, \emptyset)$;
while $\neg$*Proven*$(\mathcal{T}, r)$ **do**
  $\mathcal{T} \leftarrow \text{Walk}(\mathcal{T}, Q, G^*)$;
end
return Extract$(\mathcal{T})$;

Algorithm 2: Transition walk procedure

Data: Transition DAG $\mathcal{T}$ with root node r
Data: Constrained shortest path problem instance $Q = (\mathcal{G}, s, t, \phi)$
Data: Claimed optimal path G^* or $\emptyset$ if the instance is claimed to be infeasible
Result: Transition DAG $\mathcal{T}' \supset \mathcal{T}$ with additional interval pair vertices

$v \leftarrow r$;
while v *is not a leaf* **do**
  $t \leftarrow \arg\min_{t:(v,t)\in\mathcal{T}} \text{Bound}(v') + \text{Bound}(v'')$;
  $v \leftarrow \arg\max_{v:(t,v)\in\mathcal{T}} \text{Bound}(v)$;
end

```
// Create all valid transitions
```

for $t \leftarrow$ *vertices and edges not fixed in* v **do**
  $\mathcal{T} \leftarrow \mathcal{T} \cup (\{t\}, (v, t))$;
end
return $\mathcal{T}$;

The search procedure maintains a special data structure for keeping track of visited graph intervals – we introduce its key properties in the following definition.

Definition 6. *A* transition DAG $\mathcal{T}$ *is a directed acyclic graph with the following structure:*

- *vertices are composed of interval pairs (I_{orig}, I_{red}) and* transition vertices $t = \mathcal{V} \cup \mathcal{E}$ *corresponding to branching on vertices and edges;*
- *in each interval pair vertex, the reduced interval can be constructed from the original one with reasoning rules;*

- *the source vertex is an interval pair containing I_{root},*
- *all edges go either from interval vertices to transition vertices or in the opposite direction;*
- *for each interval pair there are either no outgoing edges or there is an outgoing edge for each graph vertex v in $G^\top \setminus G_\perp$ and for each edge e in $G^\top \setminus G_\perp$, where $I_{red} = [G_\perp, G^\top]$;*
- *for each transition node there are exactly two children obtained by branching on the corresponding vertex/edge.*

As with the A* search scheme, one of the key ingredients here is the bounding procedure. The bounding function $\text{Bound}(I)$ is implemented by the following proposition:

Proposition 1. *Given a interval pair $I = [G_\perp, G^\top]$ and $\mathcal{P}$ any valid proof tree with I being one of its vertices, the subtree of $\mathcal{P}$ rooted at I has at most Bound$(I) = 2^{|E^\top| - |E_\perp|}$, where $E_\perp$ and $E^\top$ are the edge sets of $G^\perp$ and $G^\top$.*

If additionally a transition t is chosen with J' and J'' are the two children of I (we simplify here the interval pair to I_{red}), then the bound I is at most Bound(J') + Bound(J'').

The naïve implementation of this procedure requires adding very large numbers while evaluating the available transitions. To alleviate the numeric issues arising from that, we use the *log-sum-exp* trick. In our context, this means using the *binary logarithms* of width bounds rather than the bounds themselves and, if the log-bounds for child intervals are n and m with $n > m$, evaluating the log-bound for the parent interval as $\log_2(2^n + 2^m) = n + \log_2(1 + 2^{-(n-m)})$.

Finally, the termination criterion is defined as follows: an interval pair vertex v in a transition DAG $\mathcal{T}$ is considered proven if it has a transition with two proven children or it is itself a terminal vertex (either if ϕ is false for any feasible graph or if the bounds are inconsistent):

$$\text{Proven}(v) = \text{Terminal}(v) \vee (\text{Proven}(v') \wedge \text{Proven}(v''))$$

for some v', v'', t such that $v \to t$, $t \to v'$, $t \to v''$ in $\mathcal{T}$.

Bounding heuristics. One notable feature of the description above is that the side constraint ϕ is *treated as a black box*, i.e., it is only used to check whether the interval I can be considered a leaf node in the proof. This leads to a hypothesis that the proof generation process could be aided by tracing the "progress" with respect to the side constraint satisfaction in the bounding function.

One way in which we can estimate such progress is by considering how many cases do we still have to consider – for example, the condition that can be simplified to $\phi(G) = [v \in G]$ can be treated as a simpler one than $\phi(G) = [v \in G \wedge (u \in G \vee w \in G)]$. For the former side constraint, we have to evaluate two distinct cases – $v \in G$ and $v \notin G$, while for the latter one we have to consider four distinct cases: $v \notin G$, $\{u, v\} \subseteq G$, $\{w, v\} \subseteq G$ and $v \in G \wedge \{w, v\} \notin G$.

To formalize that intuition, let $\text{BoundSide}(I; \phi)$ be a function that gives a number of cases to consider for some representation of ϕ. (See Appendix E for the implementation of BoundSide functions for different side constraint types.) We have considered the following modification of our approach:

- *Pure* approach uses $\text{Bound}(I)$ as its upper bound, completely ignoring the side constraint.
- *Lex* approach uses a pair of $\text{BoundSide}(I; \phi)$ and $\text{Bound}(I)$ as its upper bound, meaning that the search algorithm first resolves the side constraint and then switches to proving the optimality knowing that the side constraint has been proven.
- *Weighted* approach uses $(2^K)\text{BoundSide}(I; \phi) \times \text{Bound}(I)$ in the leaf nodes (or $K \log \text{BoundSide}(I; \phi) + \log \text{Bound}(I)$ in log-space) and adds up *these* numbers while going up from the leaves. This prioritizes the side constraint resolution by assigning higher weights to the leaves that have not yet resolved their side constraint.

5 Experimental results

We evaluate how often our approach is able to construct optimality certificates and compare the size of the resulting certificates with the state-of-the-art certified MIP approach (Cheung, Gleixner, and Steffy 2017). The full source code bundle for this section can be found at (Sidorov et al. 2023b).

We evaluate our proofs with respect to the *width*, i.e., the number of leaves in the proof tree. We also considered *depth*, however, we found a strong correlation between the two. Note that the time to produce proofs is not our focus. Nevertheless, our approach is comparable to the baseline – see Appendix H for more details.

Experimental setup. For the purposes of this section we use two different types of side constraints:

1. *Mandatory vertices* (de Uña et al. 2016). All side constraints have form $V^+ \cap G \neq \emptyset$, i.e., the path should go through one of the vertices in $V^+ = \{v_1, \ldots, v_k\}$.
2. *Resource constraints* (Lozano and Medaglia 2013). All side constraints have form $\sum_{e \in E} r(e) \leq R$ for some $R \in \mathbb{R}_{\geq 0}$ and vector $r : \mathcal{E} \to \mathbb{R}_{\geq 0}$.

Existing benchmarks fall into one of two categories: (1) real-world networks, with network sizes far beyond the reach of the optimal methods (e.g., (Leskovec and Krevl 2014)), and (2) artificial benchmarks which generate graphs according to some predefined distribution (such as those used in (Lozano and Medaglia 2013)), typically missing the structure of the real-world graphs. We therefore decided to generate our own instances based on real-world road network data using OpenStreetMap data (Boeing 2017). The main idea is to extract a city road network graph, then sample start and end vertices uniformly among those reachable within some time frame; for the full description of the process, see Appendix C. This procedure generates realistic networks, i.e., obtained from actual road network data rather than from a synthetic process (in contrast to procedures from (Lozano and Medaglia 2013) or (Beasley and Christofides 1989)). On the other hand, graphs in the output have controllable size, i.e., can become large or small enough by varying data generation parameters (in contrast to datasets like (Leskovec and Krevl 2014)).

We solve each instance with the baseline approach (see further) and retain only those for which we were able to obtain an optimal solution within a time limit of 60 seconds, chosen as to create non-trivial instances that are practically feasible to evaluate. This left us with a pool of instances

having 1,105 samples for mandatory vertex constraint (55% feasible) and 1,145 instances for resource constraint (72% feasible). The resulting graphs have on average 217.5 vertices and 488.7 edges, with the mean edge/vertex ratio being 2.23; the ranges for these values are up to 661 vertices, 1,386 for edges and 2.75 ratio vertex/edge. The full dataset can be found in (Sidorov et al. 2023a).

Baseline approach. Modeling and solving the problems using off-the-shelf solvers (MIP, pseudo-Boolean, and SAT) was unsuccessful – the approaches could not scale past very small instances and proof certificates were enormous even after trimming (Heule, Hunt, and Wetzler 2013). To rectify this, we use the integer programming model with *subtour elimination* constraints (Dantzig, Fulkerson, and Johnson 1954), leading to the following iterative approach: (1) Solve the model with all sub-tour elimination constraints derived earlier (initially there are none), and (2) For each set of vertices that forms a cycle in the intermediate solution, add the corresponding subtour elimination constraint, and return to step 1. The approach terminates once an optimal path is found. The optimality certificate (Cheung, Gleixner, and Steffy 2017) from the last iteration is the certificate for the *original* problem, from which we extract the binary tree corresponding to the proof having the smallest width from the certificate. See Appendix D for full specification.

Computational results. To address the first goal of our experiments, Table 1 shows the success rates (percentages of proofs constructed with our approach) per side constraint type, per bounding mode. The results suggest that the proposed approach is able to keep up with the baseline. Surprisingly, the success rates suggest that employing the side constraint information turns out to be *counterproductive* for the task of producing proof within an allotted time limit.

Table 1: Success rates on different side constraint types and bounding modes.

	Mandatory	**Resource**
Pure	96.29%	92.40%
Lex	93.57%	42.88%
Weighted	95.93%	72.93%

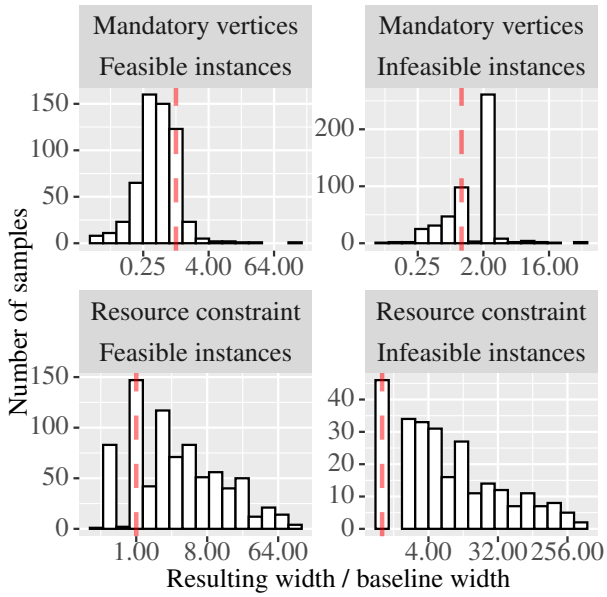
For the second goal, we compare the metric values for instances *solved both by the baseline and by the proposed approach*. A summary is given in Figure 2.

Mandatory vertex instances. The pure version of our approach is able to consistently produce shorter proofs for feasible instances – the proof is smaller in 72.8% of the cases. This is a large reduction compared to the baseline. A possible explanation for this is that *out-of-range* reasoning step is very efficient in this setting, accounting for at least half of the reasoning steps in 93% of the instances.

For the infeasible instances, our approach manages to also provide shorter proofs in 22% cases, however, this is due to the fact that the baseline is often able to provide a refutation close to the root. Out-of-range reasoning here is less efficient, with the median percentage being 37% of used inference steps.

Resource constraint instances. Resource constraint does not exhibit improvement in the proof size, which is caused by the fact that the baseline solves most ($> 95\%$) of such instances with at most one level of branching. This is somewhat expected, since these instances contain a knapsack-like structure, which are better dealt with using integer programming techniques. Nevertheless, the proof found by our approach are reasonably sized that could be used for human inspection. For more details, see Appendix G.

Figure 2: Distribution of the ratio of produced proof depth and baseline proof depth; lower is better, red dashed line corresponds to parity between our approach and the baseline.



6 Conclusions

We propose an approach for generating proofs of optimality for constrained shortest-path problem instances. The proposed approach produces proofs that are based on graph concepts that may be easier to audit from a human perspective and are in some cases much more compact compared to their integer programming counterparts. This is largely due to our reasoning steps encapsulating more powerful reasoning.

One way of improving this scheme is to *optimize* the proofs – currently, our search terminates on finding a valid proof, however, it is easy to imagine that some unfortunate sequence of decisions may lead to an unnecessarily long proof. The challenge is to design branching rules that focus on producing short proofs rather. While conventional solvers benefit from decades of continuous improvement, most of that research has been focused on efficiency rather than interpretability or the proof size.

Another promising idea is to expand our approach to other settings by designing individual reasoning rules should be (1) interpretable, (2) non-trivial, and (3) efficiently discoverable. Global constraints from constraint programming have similar characteristics and may serve as an initial inspiration.

7 Acknowledgements

Konstantin Sidorov is supported by the TU Delft AI Labs programme as part of the XAIT lab.

References

Beasley, J. E.; and Christofides, N. 1989. An Algorithm for the Resource Constrained Shortest Path Problem. *Networks*, 19(4): 379–394.

Biere, A.; Heule, M.; and van Maaren, H., eds. 2021. *Handbook of Satisfiability*. Number volume 336 in Frontiers in Artificial Intelligence and Applications. Amsterdam ; Washington, DC: IOS Press, second edition edition. ISBN 978-1-64368-160-3.

Boeing, G. 2017. OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks. *Computers, Environment and Urban Systems*, 65: 126–139.

Bogaerts, B.; Gamba, E.; and Guns, T. 2021. A Framework for Step-Wise Explaining How to Solve Constraint Satisfaction Problems. *Artificial Intelligence*, 300: 103550.

Cheung, K. K. H.; Gleixner, A.; and Steffy, D. E. 2017. Verifying Integer Programming Results. In Eisenbrand, F.; and Koenemann, J., eds., *Integer Programming and Combinatorial Optimization*, volume 10328, 148–160. Cham: Springer International Publishing. ISBN 978-3-319-59249-7 978-3-319-59250-3.

Conforti, M.; Cornuéjols, G.; and Zambelli, G. 2014. *Integer Programming*, volume 271 of *Graduate Texts in Mathematics*. Cham: Springer International Publishing. ISBN 978-3-319-11007-3 978-3-319-11008-0.

Cook, W.; Coullard, C. R.; and Turán, G. 1987. On the Complexity of Cutting-Plane Proofs. *Discrete Applied Mathematics*, 18(1): 25–38.

Dantzig, G.; Fulkerson, R.; and Johnson, S. 1954. Solution of a Large-Scale Traveling-Salesman Problem. *Journal of the Operations Research Society of America*, 2(4): 393–410.

de Uña, D.; Gange, G.; Schachte, P.; and Stuckey, P. J. 2016. A Bounded Path Propagator on Directed Graphs. In Rueher, M., ed., *Principles and Practice of Constraint Programming*, volume 9892, 189–206. Cham: Springer International Publishing. ISBN 978-3-319-44952-4 978-3-319-44953-1.

Dijkstra, E. W. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1): 269–271.

Gillard, X.; Schaus, P.; and Deville, Y. 2019. SolverCheck: Declarative Testing of Constraints. In Schiex, T.; and de Givry, S., eds., *Principles and Practice of Constraint Programming*, volume 11802, 565–582. Cham: Springer International Publishing. ISBN 978-3-030-30047-0 978-3-030-30048-7.

Gilmore, P. C.; and Gomory, R. E. 1961. A Linear Programming Approach to the Cutting-Stock Problem. *Operations Research*, 9(6): 849–859.

Gocht, S.; McCreesh, C.; and Nordström, J. 2020. VeriPB: The Easy Way to Make Your Combinatorial Search Algorithm Trustworthy. In *Workshop From Constraint Programming to Trustworthy AI at the 26th International Conference on Principles and Practice of Constraint Programming (CP'20). Paper Available at http://www. Cs. Ucc. Ie/Bg6/Cptai/2020/Papers/CPTAI_2020_paper_2. Pdf.*

Gocht, S.; McCreesh, C.; and Nordström, J. 2022. An Auditable Constraint Programming Solver. In Solnon, C., ed., *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, volume 235 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 25:1–25:18. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-240-2.

Heule, M. J. H.; Hunt, W. A.; and Wetzler, N. 2013. Trimming While Checking Clausal Proofs. In *2013 Formal Methods in Computer-Aided Design*, 181–188. Portland, OR: IEEE. ISBN 978-0-9835678-3-7.

Heule, M. J. H.; Kullmann, O.; and Marek, V. W. 2016. Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer. In Creignou, N.; and Le Berre, D., eds., *Theory and Applications of Satisfiability Testing – SAT 2016*, volume 9710, 228–245. Cham: Springer International Publishing. ISBN 978-3-319-40969-6 978-3-319-40970-2.

Junker, U. 2001. Quickxplain: Conflict Detection for Arbitrary Constraint Propagation Algorithms. In *IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, volume 4. Citeseer.

Korte, B.; and Vygen, J. 2000. *Combinatorial Optimization*, volume 21 of *Algorithms and Combinatorics*. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-662-21708-5.

Lamb, E. 2016. Two-Hundred-Terabyte Maths Proof Is Largest Ever. *Nature*, 534(7605): 17–18.

Leskovec, J.; and Krevl, A. 2014. SNAP Datasets: Stanford Large Network Dataset Collection.

Lozano, L.; and Medaglia, A. L. 2013. On an Exact Method for the Constrained Shortest Path Problem. *Computers & Operations Research*, 40(1): 378–384.

Senthooran, I.; Klapperstueck, M.; Belov, G.; Czauderna, T.; Leo, K.; Wallace, M.; Wybrow, M.; and de la Banda, M. G. 2021. Human-Centred Feasibility Restoration. In Michel, L. D., ed., *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 49:1–49:18. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-211-2.

Sidorov, K.; Correia, G.; de Weerdt, M.; and Demirović, E. 2023a. Constrained shortest path instance dataset. https://doi.org/10.5281/zenodo.10402259.

Sidorov, K.; Correia, G.; de Weerdt, M.; and Demirović, E. 2023b. Paths, Proofs and Perfection source code bundle. https://doi.org/10.5281/zenodo.10402563.

Simplemaps. 2023. World Cities Database. https://simplemaps.com/data/world-cities. Accessed: 2023-08-15.

Wetzler, N.; Heule, M. J. H.; and Hunt, W. A. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In Hutchison, D.; Kanade, T.; Kittler, J.;

Kleinberg, J. M.; Kobsa, A.; Mattern, F.; Mitchell, J. C.; Naor, M.; Nierstrasz, O.; Pandu Rangan, C.; Steffen, B.; Terzopoulos, D.; Tygar, D.; Weikum, G.; Sinz, C.; and Egly, U., eds., *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561, 422–429. Cham: Springer International Publishing. ISBN 978-3-319-09283-6 978-3-319-09284-3.

A Soundness and completeness proofs

Proof of Lemma 1. Let $\text{CAND}(\mathcal{G}, s, t)$ be a set of all candidate paths through $\mathcal{G}$ from s to t and consider the following proof:

$$\mathcal{P} = (\mathcal{I}, \mathcal{A}),$$
$$\mathcal{I} = \{R\} \cup \{[P, P] : P \in \text{CAND}(\mathcal{G}, s, t)\},$$
$$\mathcal{A} = \{(R, [P, P]) : P \in \text{CAND}(\mathcal{G}, s, t)\},$$

where R is the trivial source. (In other words, the proof $\mathcal{P}$ considers all possible candidate paths and shows their infeasibility, one path at a time.)

This proof satisfies all four requirements: it is a tree with a proper root, a set of all paths is a subset of the root interval and all candidate paths are kept in proof interval by construction. Finally, for every leaf interval $L = [P, P]$ the condition $\Phi(L)$ does not hold because otherwise $\phi(P) = \Phi(L)$ would be true, contradicting the optimality of G^*. □

Proof of Lemma 2. For a proof by contradiction, suppose that there is a feasible path G^+ shorter than G^*. That would, in particular, mean that G^+ is a candidate path with respect to G^*. Since G^+ is included in the root (Definition 4.(i)) and gets preserved in some of its descendants (Definition 4.(iii)), then in some of the descendants of that descendant, etc., we can find a leaf interval $[I, J]$ such that $G^+ \in [I, J]$. Since $\mathcal{P}$ is a proof and $[I, J]$ is a leaf interval, no graph belonging to it should satisfy the condition ϕ – otherwise $\Phi(I)$ would have to be true by Definition 4.(iv). That, however, contradicts the fact that G^+ is a feasible solution of the constrained shortest path problem. □

B Reasoning rules used for proof generation

In the following list, each of the rules maps an interval $[G_\perp, G^\top]$ to an interval $[H_\perp, H^\top]$:

- *Unique incoming edge*. Let $e = (u, v) \in G_\perp$. Then any other edge $e' = (w, v) \in G^\top$ entering v should be removed from $G^\top$.
- *Unique outgoing edge*. Defined symmetrically.
- *Out-of-range vertex removal*. Let v is a vertex in an instance such that $d(s, v) + d(v, t) \geq L^*$, where $d(u, v)$ is the shortest distance from u to v over $G^\top$, and L^* is the distance bound from the instance definition. In that case, exclude v from $G^\top$.
- *Cycle prevention*. Let (u, v) be an edge in $G^\top$ such that u is reachable from v in $G_\perp$. In that case, exclude (u, v) from $G^\top$.
- *Bridge inclusion*. Let v be a vertex in $G_\perp$. Suppose that an edge $e = (u, v) \in G^\top$ is the only incoming edge to the vertex v. Then $G_\perp$ can be extended to include v.

In the next lemma, we show that this system (or, rather, its strict subset) is enough to enforce completeness.

Lemma 3. *Let $(\mathcal{G}, s, t, \phi)$ be a constrained shortest path problem with an optimal solution $\mathcal{G}^*$. Then there exists a proof $\mathcal{P}$ satisfying all requirements of Definition 4 constructed using only the following procedures:*

- *branching on edge,*
- *unique incoming/outgoing reasoning steps,*
- *out-of-range vertex removal reasoning step.*

Proof. Similarly to Lemma 1, we can start the construction by building a tree having I_{root} as its root interval and repeating the following steps:

- Find a leaf interval $I = [G_\perp, G^\top]$ such that $G_\perp \neq G^\top$ and $\Phi(I)$ is true. Terminate if no such interval exists.
- Remove all available edges in child intervals by invoking unique incoming/outgoing reasoning steps to each of them independently.
- Remove all available vertices in child intervals by invoking out-of-range vertex removal,
- Branch from I on some of the edges $e \in G$.

This proof satisfies Definition 4.(i) and 4.(ii) by construction. Definition 4.(iii) also holds since reasoning steps do not remove candidate paths while branching steps do not remove any paths, since the child intervals cover the parent interval.

Finally, to prove Definition 4.(iv) by contradiction, suppose that $I = [G_\perp, G^\top]$ such that $\Phi(I)$ is true. Now $G_\perp \neq G^\top$ would contradict the termination condition in the proof construction, so we assume that $G_\perp = G^\top$. In that case, $G_\perp$ is a simple path. Indeed, s and t must be connected – otherwise the out-of-range step makes the interval empty. Also, for each vertex in $G_\perp$, there is only one incoming and one outgoing edge (except for start and finish vertices, which only have an outgoing and an incoming edge respectively) – any other possibility is ruled out by unique incoming/outgoing rules. But in that case we have $\phi(G_\perp) = \Phi(I)$, implying that $G_\perp$ is a feasible simple path shorter than G^*, contradicting its optimality. □

It should be noted, however, that having "additional" rules, while not influencing completeness, can be beneficial for generating more compact proofs.

C Dataset generation

We obtain the graphs $\mathcal{G}$ with marked start/finish vertices (s, t) using the following procedure:

1. Sample all cities in the Netherlands mentioned in World Cities Database (Simplemaps 2023) and extract the road network for each of them using OSMnx library (Boeing 2017).
2. For each of the edges, evaluate its travel time by car in seconds and store it as edge weight.
3. In the resulting network, sample starting vertex s uniformly, then sample finish node t uniformly among vertices $\{v : A \leq d(s, v) \leq B\}$ for some fixed range $[A..B]$. (This work uses $[A..B] = [120..180]$.)
4. Retain only vertices v for which the bound $d(s, v) + d(v, t) \leq M d(s, t)$ holds for some fixed M. (This work uses $M = 1.5$.)

After that, we build the side constraint $\mathcal{G}$ as follows:

- For mandatory vertices domain the mandatory sets are chosen among the vertices *not* on the (unconstrained) shortest path. For every instance, at least one set is chosen; the number of additional sets follows a geometric distribution with $p = 2/3$. For each set, its size is chosen uniformly between 1 and 3 vertices; after the set cardinality is chosen the set itself is constructed by uniformly sampling a random subset with the required size.
- For resource constraint domain the resource values $r(e)$ are sampled uniformly from range $[50, 100]$ for edges e on the unconstrained shortest path G^0 and from range $[10, 20]$ elsewhere. The resource bound R is then fixed to $\frac{1}{2}\sum_{e\in G^0} r(e)$. Finally, for each instance, we use 4 distinct resource constraints.

D Integer programming model for solving the constraint shortest path problem

Given a constrained shortest path problem $(\mathcal{G}, s, t, \phi)$, the optimal solution G^* can be recovered from the following 0—1 linear program:

$$\sum_{e\in\mathcal{E}} w(e)x^E(e) \to \min$$

$$\sum_{u:(u,v)\in\mathcal{E}} x^E(u,v) - \sum_{u:(v,u)\in\mathcal{E}} x^E(v,u) = 0 \,\forall v \in \mathcal{V}\setminus\{s,t\}$$

$$\sum_{u:(u,s)\in\mathcal{E}} x^E(u,s) - \sum_{u:(s,u)\in\mathcal{E}} x^E(s,u) = -1$$

$$\sum_{u:(u,t)\in\mathcal{E}} x^E(u,t) - \sum_{u:(t,u)\in\mathcal{E}} x^E(t,u) = 1$$

$$\sum_{u:(v,u)\in\mathcal{E}} x^E(v,u) = x^V(v) \quad \forall v \in \mathcal{V}$$

$$\sum_{e=(u,v):u,v\in S} x^E(e) \le |S| - 1 \quad \forall S \subseteq \mathcal{V}$$

$$\phi(x^V, x^E)$$

$$x^E(e), x^V(v) \in \{0,1\} \quad \forall v \in \mathcal{V}, e \in \mathcal{E}.$$

E Side constraint bounding

Mandatory vertices

Let ϕ be defined by sets $V_1, \ldots, V_m$ of mandatory sets and $I = [G_\perp, G^\top]$ be the current interval. Without loss of generality, assume that all sets $V_k \subseteq G^\top$ (all other vertices can be removed) but none of the sets V_k intersects with $G_\perp$ (otherwise, the set V_k is guaranteed to be visited and thus can be eliminated). Then the side constraint is bounded by the following recursive procedure: bound is 1 if $V_k = \emptyset$ for some k or if $m = 0$, otherwise

$$\text{BoundSide}(I; V_1, \ldots, V_m) =$$

$$\text{BoundSide}(I_v^+; V_1, \ldots, V_m) + \text{BoundSide}(I_v^-; V_1, \ldots, V_m)$$

where v is the vertex occurring the most in sets V_k and $I_v^\pm$ are the intervals obtained by adding and removing the vertex v.

Table 2: Distribution of differences between produced proof depth and baseline proof depth (mandatory vertices, feasible instances)

	Min	25%	Median	75%	Max
Pure	-15	-3	-2	0	18
Lex	-15	-3	-2	0	2
Weighted	-15	-3	-2	0	18

Resource constraints

Given a resource constraint $\sum_{e\in E} r(e) \le R$, the bound is estimated by a recursive procedure in which $\text{BoundSide}(I; R) = 1$ if $R \ge \sum_{e\in\mathcal{E}} r(e)$, $R \le 0$ or $E = \emptyset$; otherwise,

$$\text{BoundSide}(I; R) =$$

$$\text{BoundSide}(I_e^+; R - r(e)) + \text{BoundSide}(I_e^-; R)$$

for the edge e with the largest resource consumption $r(e)$.

Similar to the dynamic programming algorithm for the knapsack problem, this recurrence is problematic for larger values of R. We remedy this with a trick used to solve the knapsack problem approximately, viz. we choose a predefined number of buckets B (we use $B = 16$) and divide the resource consumptions from range $[0..R]$ into B equally sized buckets.

F Experimental infrastructure

We have run our experiments on Dell Precision T5820 with the following configuration:

- *CPU*: Intel® Xeon® processor W-2223 3.6GHz 3,9GHz Turbo, 4C, 8.25M Cache, HT, (120W) DDR4-2666.
- *Operating system*: Ubuntu 22.04.2 LTS.
- *Memory*: 16GB (2 x 8 GB) 2933 MHz DDR4 ECC RDIMM.

Aside from the dependencies listed in the code appendix, we have also used the version of SCIP 8.0.0.1 from (Cheung, Gleixner, and Steffy 2017).

G Proof size comparison

Mandatory vertices

Tables 2 and 3 contain information about the feasible runs – here and in the further tables, we include the minimum/maximum values, median as well as the first and third quartiles. As can be seen, the proofs that were found for the feasible runs are typically more concise than the baseline proofs. That being said, lex-bounding is able to handle outliers better despite being the worst-performing method in terms of its success rate.

The infeasible runs are mentioned in Tables 4 and 5. The comparison is more consistent in this set of runs, differing only in the "upper" outlier – as before, lex-bounding has the most solid guarantees, followed by weighted bounding and then by pure bounding.

Table 3: Distribution of relative differences between produced proof widths and baseline proof widths (mandatory vertices, feasible instances)

	Min	25%	Median	75%	Max
Pure	0.37	1	3	8	124.5
Lex	0.5	1	1	1.5	4
Weighted	0.37	1	2	4	19

Table 4: Distribution of differences between produced proof depth and baseline proof depth (mandatory vertices, infeasible instances)

	Min	25%	Median	75%	Max
Pure	-9	0	1	1	13
Lex	-9	0	1	1	2
Weighted	-9	0	1	1	9

Resource constraints

Tables 6 and 7 tackle the feasible instances for resource constraints. While here, as expected from the plot in the main text, the differences are more favoring to the baseline, the earlier patterns regarding the bounding approaches hold here as well.

Finally, Tables 6 and 7 address the infeasible instances for resource constraints.

Evaluation of lex-bounding

The experimental results suggest that the lex-bounding, despite being the least successful in *finding* a proof, turns out to *bound* the proof size relatively well. Figure 3 repeats the evaluation from the main text for lex-bounding.

H Search duration comparison

Though the focus of the work was on the proof *size*, the proposed approach also happens to find proofs for an amount of time comparable to the baseline. Table 10 lists the percentage of instances (per constraint type, per bounding mode) where our approach finds the proof faster than the baseline. Instances with mandatory vertex constraints exhibit uniform performance across the bounding modes. On the other hand, the elapsed time on resource-constrained instances is substantially influenced by the bounding type. In particular, lex-bounding ends up not only bounding the proof size better than the others but also does a much better job at matching the baseline duration than the other bounding types.

Table 5: Distribution of relative differences between produced proof widths and baseline proof widths (mandatory vertices, infeasible instances)

	Min	25%	Median	75%	Max
Pure	-10	0	2	5	19
Lex	-1	0	0	1	3
Weighted	-11	0	1	3	9

Table 6: Distribution of differences between produced proof depth and baseline proof depth (resource constraint, feasible instances)

	Min	25%	Median	75%	Max
Pure	-10	0	2	5	19
Lex	-1	0	0	1	3
Weighted	-11	0	1	3	9

Table 7: Distribution of relative differences between produced proof widths and baseline proof widths (resource constraint, feasible instances)

	Min	25%	Median	75%	Max
Pure	0.37	1	3	8	124
Lex	0.5	1	1	1.5	4
Weighted	0.37	1	2	4	19

Table 8: Distribution of differences between produced proof depth and baseline proof depth (resource constraint, infeasible instances)

	Min	25%	Median	75%	Max
Pure	0	1	3	8	23
Lex	0	0	1	2	3
Weighted	0	1	2	3	9

Table 9: Distribution of relative differences between produced proof widths and baseline proof widths (resource constraint, infeasible instances)

	Min	25%	Median	75%	Max
Pure	1	2	4	18.25	380
Lex	1	1	2	3	5
Weighted	1	2	3	5	21

Figure 3: Distribution of the ratio of produced proof depth and baseline proof depth for lex-bounding; lower is better, red dashed line corresponds to parity between our approach and the baseline.

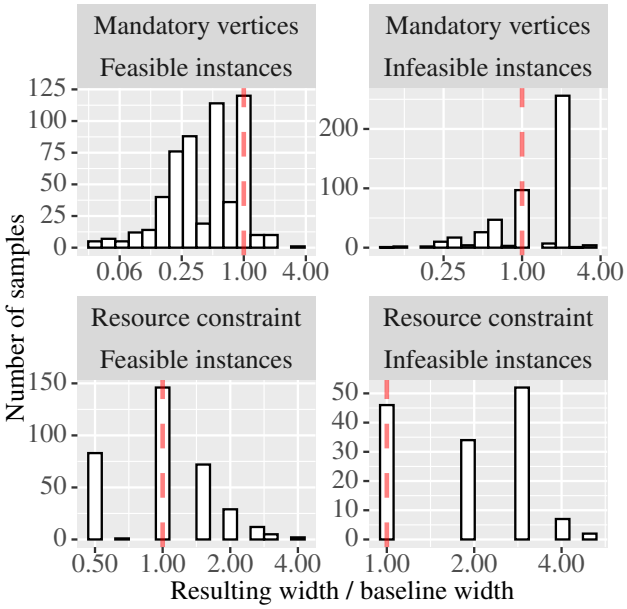


Table 10: Percentage of instances with lower proof production times

	Mandatory	**Resource**
Pure	50.15%	34.09%
Lex	50.61%	46.63%
Weighted	49.19%	25.79%

Figure 4 shows the distribution for the duration of our approach relative to the baseline. One of the striking features of this histogram is that there are quite a few instances where our approach was not only better than the baseline but better by *orders of magnitude*. On the other hand, the converse behavior – our approach taking orders of magnitude *more* time – is much rarer, with the "typical" slow-down factors peaking around $2\times$ to $3\times$.

In all of the comparisons in this section, we only consider the instances that took more than a second to solve by either of the approaches. Table 11 lists the remaining number of instances per each of the cases. It is worth mentioning that the resource constraint comparisons are done on much smaller samples, which is caused by the fact that many resource-constrained instances are solved in the root node by both the baseline and our approach, making any substantial comparison on them impossible.

Figure 4: Distribution of the ratio of search time of our approach and the baseline.

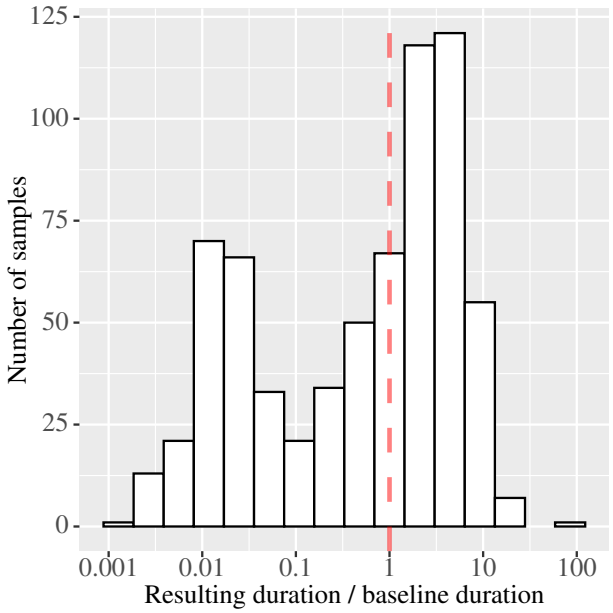


Table 11: Number of instances used for the duration comparisons

	Mandatory	**Resource**
Pure	678	572
Lex	652	208
Weighted	681	477