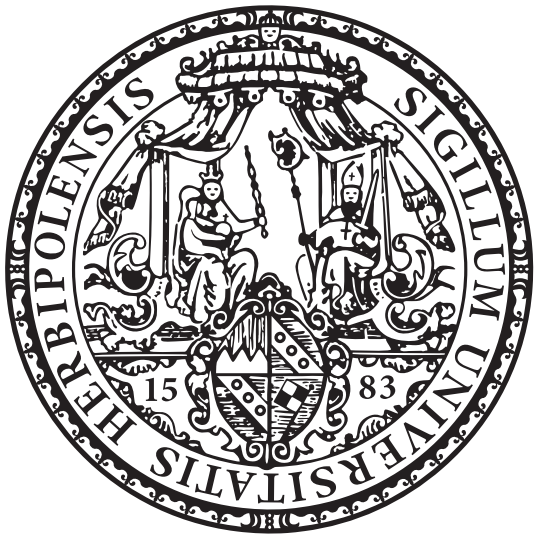
Bachelor Thesis

Experimental Evaluation of Exact Coloring Algorithms for Mixed Graphs

Maximilian Schramm

Date of Submission:	March 26, 2026
Revised:	May 04, 2026
Advisors:	Prof. Dr. Alexander Wolff
	Antonio Lauerbach, M. Sc.



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Abstract

Graph coloring is a well-studied problem in graph theory with applications in frequency assignment, air traffic flow management, and scheduling. Mixed graph coloring generalizes this problem to mixed graphs, which have undirected edges and directed arcs within the same graph. This additional complexity allows us to model more complicated concepts. Mixed graphs have applications such as edge routing when drawing graphs, and the unit-time scheduling problem. As with any problem that has real-world applications, we want to minimize the time it takes to find a solution, which requires theoretical and experimental effort. We build on the theoretical work of Lauerbach, who introduced two coloring algorithms for mixed graphs in his master's thesis (2026). We experimentally analyze the runtime of his algorithms. We also analyze Integer Linear Programming and Boolean Satisfiability formulations of the problem in the same way. Finally, we present a compact heuristic for choosing the fastest algorithm for coloring a given mixed graph, depending on the graph's properties.

Zusammenfassung

Graphenfärben ist ein klassisches Problem der Graphentheorie mit Anwendungen in Bereichen wie der Zuweisung von Frequenzen, der Regelung von Flugverkehr und der Ablaufplanung. Gemischte Graphen generalisieren Graphen, indem sie sowohl gerichtete als auch ungerichtete Kanten in einem Graphen zulassen. Gleichfalls generalisiert das Färben gemischter Graphen das Färben ungerichteter Graphen. Diese zusätzlichen Bedingungen ermöglichen das Modellieren komplexerer Zusammenhänge, wie beim Zeichnen von Graphen und im Unit-Time Scheduling Problem. Wie bei jedem Problem mit praktischen Anwendungen ist es von Interesse, die Zeit zu minimieren, die wir für die Lösungsfindung aufwenden müssen. Dafür muss sowohl theoretischer, als auch experimenteller Aufwand geleistet werden. Wir bauen auf der theoretischen Arbeit von Lauerbach auf, der zwei Algorithmen zum Färben von gemischten Graphen vorstellt, indem wir die Laufzeit seiner Algorithmen experimentell analysieren. Um die Ergebnisse in Kontext zu setzen, analysieren wir außerdem Implementierungen des Problems unter Verwendung von ganzzahliger linearer Programmierung und Boolean Satisfiability. Wir erstellen aus unseren Ergebnissen eine Heuristik, die es ermöglicht basierend auf den Eigenschaften eines gemischten Graphen den schnellsten Färbungsalgorithmus zu wählen.

Contents

1 Introduction

The mixed graph coloring problem is an important extension of the graph coloring problem. Graph coloring has applications in assigning frequencies in cellular networks [MS10], air traffic flow management [BB04], register allocation during code compilation [DWELM99], and many other fields. Generalizing this problem to mixed graphs, which have undirected edges and directed arcs, allows for additional precedence constraints to model more complicated relations, such as the unit scheduling problem [SDA00], edge routing when drawing graphs [GMR$^+$23], and general scheduling with precedence.

As with any problem that has real world applications, we want to minimize the time it takes to find a solution. This requires theoretical effort to lower asymptotic runtime by improving or finding new algorithms, as well as experimental effort to study the runtime-behavior of algorithms in practice. Mixed graph coloring is NP-hard [Kar72], which justifies using exponential-time methods to approach the problem. This thesis conducts experiments based on the theoretical work of Lauerbach [Lau25] to study the runtime behavior of select mixed-graph-coloring algorithms.

1.1 Related Work

In the following, we give a short overview of the history of mixed graph coloring, list a selection of mixed and undirected coloring benchmarks, and briefly mention some other extensions to the graph coloring problem apart from mixed coloring.

Mixed Graph Coloring

Mixed graph coloring was first defined in 1976 by Sotskov and Tanaev [ST76] in a different form than the one used in this thesis. For arcs (u, v), they require that the color of u is less than or equal to the color of v. In 1997, Hansen, Kuplinsky, and De Werra [HKD97] defined the version of mixed graph coloring studied in this thesis, where arcs impose a strict inequality. Sotskov's [Sot20] historical review gives a more detailed history of mixed graph coloring than the scope of this thesis allows.

A recent approach presented by Lauerbach [Lau25] adapted Lawler's graph-coloring algorithm [Law76] to mixed graphs, providing an $\mathcal{O}(2.4423^n)$-time algorithm which uses exponential space. Lauerbach also used a divide-and-conquer approach to obtain a $\mathcal{O}(4.8107^n)$-time polynomial-space algorithm. These two algorithms are studied experimentally in this thesis. Additionally, he provides polynomial-space algorithms for MixedColoring with specific numbers of colors. Up to 6 colors, they are asymptotically faster than his adaptation of Lawler's algorithm.

Mixed and Undirected Coloring Benchmarks

Sotskov, Dolgui, and Werner [SDW01] benchmark multiple branch-and-bound mixed-coloring algorithms on randomly generated instances of the unit-time job-scheduling problem. They test for exact solutions up to 200 vertices, and for approximations up to 900 vertices. Kouider and Haddadène [KH21] introduce and benchmark a branch-and-bound algorithm for unit-time job-scheduling, which they model as mixed coloring. Kouider, Haddadène, Samia, and Oulamara [KHOO15] benchmark mixed integer linear programs with a tabu-search approach on mixed coloring instances, reduced from instances of the unit-time job-scheduling problem. They obtain their instances from the Jobshop1 [MV] job-scheduling problem dataset by discarding the time components. While our benchmarks are not limited to the unit-time job-scheduling problem, we combine these two approaches for obtaining benchmark instances. We randomly generate mixed graphs so that we have a large number of samples. We also convert the Jobshop1 instances into mixed graphs to confirm that our results hold for real-world applications.

Gualandi and Chiarandini maintain a benchmark set of undirected graphs for coloring [GC26], collected from the DIMACS benchmark set. It is used extensively for evaluating graph-coloring algorithms and is considered a standard benchmark set [MT10] in the field. It is, for example, used by Cazenave, Negrevergne, and Sikora [CNS21] to evaluate the performance of Monte Carlo search for undirected graph coloring, and by Méndez-Díaz and Zabala [MDZ06] to benchmark a branch-and-cut algorithm designed to avoid duplicate branches resulting from symmetrical colorings.

Other Extensions of Graph Coloring

Beyond mixed coloring, there are further extensions of the coloring problem. In *selective coloring* [DERT15], given a graph with a partition of its vertex set into several clusters, one vertex per cluster should be selected such that the chromatic number of the subgraph induced by the selected vertices is minimal. This coloring problem has applications in routing and wavelength assignment in optical networks.

In *weighted coloring* [WCP+20], a graph with weighted vertices is colored, i.e. partitioned into (disjoint) independent sets, such that the sum of the cost of independent sets, given as the maximum weight in each independent sets, is minimal. This models real-world scenarios where the vertices are not equally important.

List coloring [ERT79], also called choosability, is a coloring problem where each vertex is restricted to a list of colors. The lists have equal length, but can contain different colors. A graph is *k-choosable* if, for lists of length k, no matter what colors they contain, the graph can be colored.

1.2 Contribution

This thesis builds on Lauerbach's theoretical work by implementing both of his mixed-graph-coloring algorithms and measuring their runtime experimentally. To put their runtimes into perspective, we additionally solve the mixed coloring problem by modeling

it as an Integer Linear Program (ILP) which is solved using Gurobi [Gur26] and as a Boolean Satisfiability (SAT) problem which is solved using the Z3 Solver [DMB08].

The remainder of the thesis is structured as follows. We first introduce the concepts and notations used throughout this thesis; see Chapter 2. This includes many properties that we use throughout the thesis, as well as a precise definition of the mixed coloring problem.

In Chapter 3 we introduce the four mixed-coloring algorithms of which we analyze the runtime in this thesis. We create variants of each algorithm, that is, different ways of executing them, and make predictions about how these variants perform.

Following this is Chapter 4, the heart of the thesis, where we describe the experiments we perform, and analyze their results. We begin in Section 4.1 by describing the mixed graphs that we use for experimentation. This includes an algorithm to generate random graphs while controlling some properties of the resulting graph, as well as a method of obtaining mixed graphs from a known set of job-scheduling problems. We additionally describe how we distribute certain properties among a set of graphs to ensure that experiments are representative of as many graphs as possible and provide a final list of all benchmark sets we use for experimentation in section 4.1.3. In Section 4.2, we experimentally determine how to minimize the runtime of each algorithm and provide a list of the findings. Then, in Section 4.3, we experimentally compare the runtime of our algorithms depending on multiple graph properties, and summarize, in Section 4.3.4, our findings into a heuristic for choosing an algorithm for mixed graph coloring depending on the graph's properties.

We conclude the thesis with an outlook on open problems; see Chapter 5.

2 Preliminaries

Mixed Graph An *undirected graph* G consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$, with $E(G) \subseteq \binom{V(G)}{2}$. A *directed graph* G consists of a set of vertices $V(G)$ and a set of *arcs* $A(G)$, with $A(G) \subseteq V(G)^2$. A *mixed graph* G is a combination of these types of graphs. It consists of a set of vertices $V(G)$, a set of edges $E(G)$, and a set of arcs $A(G)$.

For conciseness we refer to something that can be either an edge or an arc as a *connection*. A connection $[u, v]$ is undirected and can refer either to the arc (u, v), the arc (v, u), or the edge $\{u, v\}$. This concept is useful in cases such as connectedness of a graph, where it does not matter whether a connection is an edge or an arc.

We denote the number of vertices in G, called its *size*, as $n(G)$. We only write $V(G)$, $n(G)$, etc. when it is unclear which graph is referred to. For conciseness, if there is only one graph relevant to the current context, we simply write V, n, etc. when referring to that graph's components and properties. We usually refer to mixed graphs simply as graphs, while explicitly mentioning when a graph is undirected.

Simple Graphs An (undirected) graph is simple if, for every unordered pair of vertices u, v, there is at most one connection $[u, v]$, and there are no loops, i.e. a connection $[u, u]$ from a vertex u to itself.

Vertex Degree In a graph G, if there is a connection $[u, v]$, we say that u and v are *incident* to this connection. An arc (u, v) is directed from its *tail* u to its *head* v. We say that it is *outbound* from u and *inbound* to v. The *degree* $d_G(u)$ of a vertex u in graph G is the number of connections to which u is incident. The *outdegree* $d_G^+(u)$ of a vertex u is the number of its outbound arcs. The *indegree* $d_G^-(u)$ of a vertex u is the number of arcs of its inbound arcs.

A vertex set S is incident to a connection $[u, v]$ if exactly one of u, v is in S. An arc (u, v) is outbound from a vertex set S if $u \in S$ and $v \notin S$, and inbound to S if $u \notin S$ and $v \in S$. For a vertex set S we define the degree $d_G(S)$, outdegree $d_G^+(S)$, and indegree $d_G^-(S)$, as the number of incident connections, outbound arcs, and inbound arcs, respectively.

Adjacency & Neighbors In a graph G, a vertex v is *adjacent* to a vertex u if u and v are incident to the same connection. The *open neighborhood* of u, denoted $N_G(u)$, is the set of all vertices adjacent to u. The *closed neighborhood* of u, denoted $N_G[u]$, is the set of u itself and all vertices adjacent to u.

Maximal Independent Sets In a graph G, an *independent set* I is a set of vertices where no pair of vertices is adjacent. An independent set is *maximal* if we cannot add any vertex to the set such that it remains independent. We denote with $\mathrm{MIS}(G)$ the set of maximal independent sets of G, and with $\mathrm{MIS}^\circ(G)$ the set of maximal independent sets of G with indegree 0.

Walks, Paths & Cycles A *walk* of length k in a graph G is a sequence $\langle u_0, \ldots, u_k \rangle$ of vertices such that there is a connection $[v_{i-1}, v_i]$ in G for each $i \in 1, \ldots, k$. A walk is *directed* if each such connection is an arc (v_{i-1}, v_i) directed in order of the walk, and *undirected* if each connection is an edge. A walk is *closed* if $v_0 = v_k$, otherwise it is *open*. A *path* is a walk without repeated vertices, except that v_0 may be equal to v_k. A closed path is also called a *cycle*.

Connected Graphs An (undirected) graph is connected if, for every pair of vertices u, v, there is a path from u to v. Note that a walk from u to v implies a path from u to v. A connected graph with n vertices must have at least $n - 1$ connections.

Trees An (undirected) graph is a *tree* if every unordered pair of vertices u, v is connected by exactly one path. Note that this makes trees simple, as duplicate edges would create duplicate paths between their vertices. Equivalently, an (undirected) graph that is connected and has $n - 1$ edges is a tree.

Partitions A *proper partition* $\langle V_1, \ldots, V_k \rangle$ of a mixed graph G is a partition of the vertex set such that there is no arc outbound from a set with a higher index and inbound to a set with a lower index, i.e. there is no arc $(u, v) \in A$ with $u \in V_i$, $v \in V_j$, and $i > j$.

Subgraphs We say that G' is a *subgraph* of a graph G, denoted $G' \subset G$, if it holds that $V(G') \subset V(G)$, $E(G') \subset E(G)$, and $A(G') \subset A(G)$. For a subset of vertices $S \subset V$, we denote the subgraph induced by S, $\left(S, E \cap \binom{S}{2}, A \cap S^2\right)$, as $G[S]$. For a set of vertices S, we write $G - S$ instead of $G[V(G) \setminus S]$.

MixedColoring A *coloring* of a mixed graph G is an assignment of a *color*, a natural number, to each vertex $u \in V$. It is a *proper* coloring if no two vertices which share an edge have the same color

$$\forall \{u, v\} \in E \colon \mathrm{color}(u) \neq \mathrm{color}(v) \tag{2.1}$$

and for every arc (u, v), the color of u is less than the color of v.

$$\forall (u, v) \in A \colon \mathrm{color}(u) < \mathrm{color}(v) \tag{2.2}$$

A *k-coloring* is a coloring that uses k colors. A graph is *k-colorable* if a proper k-coloring for G exists. The *chromatic number* of G, denoted $\chi(G)$, is the smallest k such that G

is k-colorable. A proper $\chi(G)$-coloring of G is an *optimal* coloring. We call the problem of finding an optimal coloring or the chromatic number of a graph MixedColoring.

For the purpose of this thesis we consider only graphs which are simple, connected, and contain no directed cycles because:

- Non-simple connections would either make no difference to MixedColoring, or make MixedColoring impossible:
  - Duplicate edges and arcs have no effect on coloring, as they just impose the same constraint multiple times.
  - An edge and an arc between the same pair of vertices is equivalent to only an arc, as $<$ implies $\neq$.
  - Opposing arcs make coloring impossible, as one vertex' color cannot be both larger and smaller than another vertex' color.
  - Loops make coloring impossible, as a vertex cannot have a different color than itself.
- A disconnected graph can be split into its connected subgraphs which can be colored independently.
- A graph with a directed cycle is not colorable, as observed by Hansen et al. [HKD97, Proposition 1 (ii)].

Therefore MixedColoring on a graph that does not have all of these properties is either impossible or can easily be reduced to MixedColoring on graphs that do have them.

Transitive Reduction & Closure If a graph G has a directed path from a vertex u to a vertex v that does not include the connection $[u, v]$, then this connection is *redundant* for MixedColoring, i.e. its presence or absence does not affect the result. This is because a directed path from u to v constrains the color of v to be larger than the color of u. As such, neither an arc (u, v) nor an edge $\{u, v\}$ changes the constraints on colors. Note that an arc (v, u) can not exist as it, combined with the directed path from u to v, would form a directed cycle and we do not consider such graphs.

The *transitive reduction* of a graph G, denoted as G^-, is obtained by removing every redundant connection from G. The *transitive closure* of a graph G, denoted as G^+, is obtained by adding every redundant arc to G. If there is a redundant edge, then it is replaced by an arc, as we only consider simple graphs. We refer to the transitive reduction and transitive closure of a graph as its *transitive variants*. Note that, because we only add or remove redundant connections, the transitive variants of G have the same proper colorings and chromatic number as G.

3 Algorithms

In this section we introduce the implementations of the MixedColoring algorithms we evaluate in this thesis: `Lawler`, `Polyspace`, `ILP`, and `SAT`. Each algorithm can be run in different ways, which we call *variants* of the algorithm. For example, `Polyspace` can be run with different values of α, which changes how it partitions graphs. We additionally introduce an algorithm which finds all maximal independent sets with indegree 0 of a given graph because both `Lawler` and `Polyspace` require it as a subroutine.

Furthermore, we call changes that we make to an algorithm's input graph *modifications*. We study the effect of modifications that do not change the graph's proper colorings or chromatic number, such as coloring the graph's transitive closure instead of the original graph, on the runtime of algorithms in chapter 4. We call the combination of an algorithm's variant and modifications to the input graph a *configuration* of an algorithm.

`ILP` and `SAT` require a lower and upper bound $\chi_{\min}, \chi_{\max}$ on the chromatic number χ of their input graph. In this thesis we simply use the bounds $\chi_{\min} = 1, \chi_{\max} = n$. While there are better bounds, they are not within the scope of this thesis, though certainly an interesting topic for further research.

3.1 Maximal Independent Sets

Algorithm 1: An algorithm that constructs all maximal independent sets of an undirected graph with smarter branching than the brute force approach.

Input: Undirected graph G
Output: Set of maximal independent sets with indegree 0 of G

```
1 MIS(G):
2     if |V| = 0 then
3         return ∅
4     u = vertex of minimum degree in G
5     M = ∅
6     foreach v ∈ N[u] do
7         M ∪= {I ∪ {v} | I ∈ MIS(G − N[v])}
8     return M
```

Both `Lawler` and `Polyspace` use a subroutine which finds all maximal independent sets with indegree 0 of a given graph. For this, we use a well known [FK10, Fig. 1.2]

algorithm that enumerates maximal independent sets on undirected graphs in $\mathcal{O}^*(3^{n/3})$-time. We restate it here as Algo 1 for convenience. Instead of the brute force approach where every subset of V is tested for being a maximal independent set, this algorithm picks the vertex u of minimum degree in G and branches into $d_G(u)$ cases based on the easily verifiable property that a neighbor v of a vertex u can not be in the same independent set as u.

To obtain only the maximal independent sets with indegree 0 of a mixed graph G, we run Algo 1 on the graph $G[S]$ induced by the set S of all vertices $u \in V$ with indegree 0. This gives the desired result as per Lemma 3.1. Note that $G[S]$ does not contain arcs. If there were some $u, v \in S$ such that an arc (u, v) existed, then v would have $d_G^-(v) \geq 1$ and would not be in S. Therefore we can treat $G[S]$ as an undirected graph and Algo 1 can take it as input.

Lemma 3.1. *The independent sets with indegree 0 of G are equivalent to the independent sets of $G[S]$ with $S = \{u \in V \mid d_G^-(u) = 0\}$.*

Proof. $G[S]$ is obtained by removing from G all vertices which do not have indegree 0 and all connections that these vertices are incident to. We demonstrate that neither vertices with indegree greater than 0, nor connections that such a vertex is incident to, affect the construction of an independent set.

An independent set I has indegree 0 iff it only consists of vertices with indegree 0.

- $\Leftarrow$ If an independent set consists only of vertices with indegree 0, then no arc can be inbound to any vertex of the set, and therefore the set has indegree 0.
- $\Rightarrow$ If an independent set I has indegree 0, then no arc can be inbound to the I, that is, outbound from a $u \notin I$ and inbound to a $v \in I$. As there can be no connections between vertices of an independent set, there can also be no arc outbound from a $u \in I$ and inbound to a $v \in I$. Therefore, no $v \in I$ can have inbound arcs, meaning that all $v \in I$ have indegree 0.

As such, the candidate vertices for independent sets with indegree 0 of G are the same as for independent sets of $G[S]$.

Note additionally that all connections $[u, v]$ that we remove have at least one vertex u with $d_G^-(u) > 0$, which cannot be in $\text{MIS}^\circ(G)$ or $\text{MIS}(G[S])$ regardless of connections, otherwise they would not be removed. Since a connection only imposes the restriction that at most one of its vertices may be in an independent set, a connection where one vertex cannot be in the independent set for another reason does not impose any restriction. Therefore, removing these connections does not change independent sets with indegree 0.

□

3.2 `Lawler`

The algorithm `Lawler` is an adaptation of Lawler's Coloring algorithm to Mixed-Coloring by Lauerbach. It is based on a property of undirected graphs, where there exists an optimal coloring in which one color class is a maximal independent set. This

property does not hold in mixed graphs, but Lauerbach generalizes it to mixed graphs by restricting the color class to a maximal independent set with indegree 0. From this, he obtains a recursive formula for the chromatic number of mixed graphs [Lau25, Lemma 3.1].

$$\chi(G) = \begin{cases} 0 & \text{if } G = \emptyset \\ 1 + \min_{I \in \text{MIS}^\circ(G)}(\chi(G - I)) & \text{otherwise} \end{cases} \tag{3.1}$$

The resulting algorithm is a dynamic program that incrementally constructs the set L_k of all k-colorable subgraphs of G by adding to each $(k-1)$-colorable subgraph $S \in L_{k-1}$ every maximal independent set I with indegree 0 of the remaining graph $G-S$. Duplicate subgraphs are removed. Lauerbach obtains an $\mathcal{O}(2.44225^n)$ bound on the asymptotic runtime of this algorithm [Lau25, Theorem 3.3].

`Lawler` only has variants based on the choice of MIS°-implementation.

Algorithm 2: `Lawler`. An adaptation of Lawler's Coloring algorithm to MixedColoring by Lauerbach

Input: Graph G
Output: Chromatic number χ of G

```
1 LawlerColoring(G):
2     L_0 = {∅}
3     for k = 1, ..., n do
4         L_k = ∅
5         foreach S ∈ L_{k-1} do
6             foreach I ∈ MIS°(G − S) do
7                 if S ∪ I = V then
8                     return k
9                 L_k ∪= {S ∪ I}
```

3.3 `Polyspace`

`Polyspace` is a branching divide-and-conquer MixedColoring algorithm by Lauerbach. It is based on a property of mixed graphs where for a mixed graph G and all proper partitions $\langle S, I, T\rangle$ of G with $|S| < n\alpha, |T| \leq n(1-\alpha)$, and $I \in \text{MIS}^\circ(G - S)$ it holds that $\chi(G) \leq \chi(G[S]) + 1 + \chi(G[T])$. For at least one such partition, it holds that $\chi(G) = \chi(G[S]) + 1 + \chi(G[T])$ [Lau25, Lemma 3.4].

Using this property, the algorithm first constructs every $S \subset V$ with $|S| < n\alpha$ and recursively colors $G - S$. Then for each S it finds all $I \in \text{MIS}^\circ(G - S)$. T is obtained as $T = V \setminus (S \cup I)$ for each combination of S and I and also colored recursively. Finally the minimum of $\chi(G[S]) + 1 + \chi(G[T])$ is found across all combinations of S and T. This minimum is the chromatic number of $\chi(G)$.

Lauerbach obtains an $\mathcal{O}(4.81069^n)$ bound on the asymptotic runtime of this algorithm [Lau25, Theorem 3.5] by setting $\alpha \approx 0.563964$, suggesting that this value of α may also lead to good runtimes in practice. Note however, that `Polyspace` converges to a recursive form of the asymptotically faster algorithm `Lawler` as α approaches 0, which suggests that faster runtimes than Lauerbach's bound are possible for small α. Howerver, the drawback of this recursive form, compared to the iterative form discussed in section 3.2, is that duplicate k-colorable subsets are not eliminated. On the complete undirected graph K_n, this leads to a worst case of $\mathcal{O}(n!)$ branches, and therefore $\mathcal{O}^*(n!)$ runtime, as each of its n vertices is a maximal independent set and therefore every permutation of vertices is checked. If the number of duplicates is small for graphs encountered in practice, then we expect smaller values of α to provide a runtime behavior closer to that of `Lawler` than to Lauerbach's bound on the asymptotic runtime of `Polyspace`.

Thus, `Polyspace` has variants based on the choice of α, as we wish to test its runtime for different values of α, and based on its MIS°-implementation.

Algorithm 3: `Polyspace`. A polynomial-space divide-and-conquer algorithm for MixedColoring by Lauerbach

Input: Graph G
Output: Chromatic number χ of G

```
PolyspaceColoring(G):
    if V = ∅ then
        return 0
    foreach S ⊂ V with |S| ≤ ⌈nα − 1⌉ and d⁻_G(S) = 0 do
        x₁ = PolyspaceColoring(G − S)
        foreach I ∈ MIS°(G − S) do
            T = V \ (S ∪ I)
            x₂ = PolyspaceColoring(G[T])
            x = min(x, x₁ + 1 + x₂)
    return x
```

3.4 `ILP` (Gurobi)

Integer Linear Programming (ILP) is a mathematical optimization method in which a problem is encoded as a set of linear constraints. A linear objective $c^T x$ is optimized with respect to some linear constraints $Ax \leq b$ and some bound constraints $l \leq x \leq u$. Any number of decision variables may be constrained to take only integer or binary values.

ILP has been studied for decades [CL25], making solvers very fast and efficient. We choose Gurobi as our ILP solver due to its exceptional performance in comparison to other solvers, demonstrated by Luppold, Oehlert, and Falk [LOF18], and because it provides a C++ API.

To solve MixedColoring using an ILP, the problem must be described in terms of variables and constraints. The colors of vertices are represented in two different ways, once with binary variables and once with integer variables. We denote these variants as `ILP-BIN` and `ILP-INT` respectively. We compare performance between these implementations experimentally. Based on previous work [LM07] we expect that the binary implementation performs better.

The representation of edges and arcs through constraints also differs between the implementations. Because χ represents a quantity and must be minimized, it must be an integer variable in both implementations. We must model colors up to $\chi_{\max}$, so we use the set of colors $C = \{1, \ldots, \chi_{\max}\}$. The next two sections detail how `ILP` is implemented to solve MixedColoring.

3.4.1 Binary Variables

To model vertex colors with binary variables we introduce a variable for every combination of vertices and colors such that, for every vertex $u \in V$ and every color $i \in C$, the variable $c_{u,i}$ is 1 if u has the color i and is 0 otherwise.

$$\forall u \in V, \forall i \in C \colon c_{u,i} = \begin{cases} 1 & \text{if vertex } u \text{ has color } i \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

This definition allows the solver to assign multiple colors or no color to a vertex, so a constraint is required to ensure that for each vertex u exactly one $c_{u,i}$ is assigned the value 1. We do this by requiring their sum to equal 1.

$$\forall u \in V \colon \sum_{i \in C} c_{u,i} = 1 \tag{3.3}$$

The chromatic number χ must equal the largest used color. In this definition, that is the largest $i \in C$ where any $c_{u,i}$ is 1. We implement this by requiring $\chi \geq i \cdot c_{u,i}$ for every $u \in V$ and every $i \in C$. Obviously χ may not be greater than the highest used color, but no constraint is required for this because the optimization target is minimizing χ, so the solver will never set it higher than required.

$$\forall u \in V, \forall i \in C \colon c_{u,i} \cdot i \leq \chi \tag{3.4}$$

If there is an edge $\{u, v\} \in E$, the vertices u and v may not have the same color. This means that for every color $i \in C$ the variables $c_{u,i}$ and $c_{v,i}$ cannot both equal 1, which we impose by requiring their sum to be less than or equal to 1.

$$\forall \{u, v\} \in E, \forall i \in C \colon c_{u,i} + c_{v,i} \leq 1 \tag{3.5}$$

If there is an arc $(u, v) \in A$, the color of vertex u must be less than the color of vertex v. This means that for every color $i \in C$, if $c_{v,i}$ is 1, the variables $c_{u,j}$ must be 0 for all $j \in \{i, \ldots, \chi_{\max}\}$. We impose this by requiring the sum of $c_{v,i}$ and all $c_{u,j}$ to be less than or equal to 1.

$$\forall (u, v) \in A, \forall i \in C \colon c_{v,i} + \sum_{j=i}^{\chi_{\max}} c_{u,j} \leq 1 \tag{3.6}$$

3.4.2 Integer Variables

To model vertex colors with integer variables we introduce a variable for every vertex such that, for every vertex $u \in V$, the variable c_u is equal to the color of u.

$$\forall u \in V \colon c_u = \text{color of } u \tag{3.7}$$

We constrain the variables to only take on valid colors.

$$\forall u \in V \colon 1 \leq c_u \leq \chi_{\max} \tag{3.8}$$

The chromatic number χ must equal the largest used color. In this definition, that is the largest c_u. We implement this by requiring $\chi \geq c_u$ for every $u \in V$. Obviously χ may not be greater than the highest used color, but no constraint is required for this because the optimization target is minimizing χ, so the solver will never set it higher than required.

$$\forall u \in V \colon c_u \leq \chi \tag{3.9}$$

If there is an edge $\{u, v\} \in E$, the vertices u and v may not have the same color. This means that the variables c_u and c_v cannot be equal, which is a non-linear constraint. We first rewrite this inequality by requiring some minimal absolute difference ϵ, which is 1 when working with integers.

$$\forall \{u, v\} \in E \colon |c_u - c_v| \geq \epsilon$$

We then use a Big M constraint, in which a large positive constant M is conditionally applied to a constraint depending on a binary variable y, to split the absolute value into two equations. They represent the case where the difference of c_u minus c_v is positive and y is 1, and the case where the difference is negative and y is 0, respectively.

$$\forall \{u, v\} \in E : c_u - c_v \geq \quad \epsilon - M(1 - y) \tag{3.10}$$

$$\forall \{u, v\} \in E : c_u - c_v \leq -\epsilon + My \tag{3.11}$$

The constant M must be greater than the maximal possible difference of c_u and c_v, so we set M equal to $\chi_{\max}$. Note that with a sufficiently large M:

- if y is 1, Eq 3.11 is always fulfilled and Eq 3.10 is fulfilled if $c_u > c_v$ and
- if y is 0, Eq 3.10 is always fulfilled and Eq 3.11 is fulfilled if $c_u < c_v$.

Therefore these two equations enforce inequality of the integer variables c_u and c_v.

If there is an arc $(u, v) \in A$, the color of vertex u must be less than the color of vertex v, so $c_u < c_v$.

$$\forall (u, v) \in A \colon c_u + 1 \leq c_v \tag{3.12}$$

3.5 SAT (Z3 Solver)

The Boolean Satisfiability problem asks whether or not there exists an interpretation that satisfies a given Boolean formula, i.e. whether some assignment of values to the formula's

variables can make the formula evaluate to true. It has a long history of research [FBHS23], and solvers have become very fast and efficient. We use Z3 Solver as our SAT implementation in view of its excellent performance shown by Roselli, Bengtsson, and Akesson [RBA18], and because it provides a C++ API. Z3 Solver is a Satisfiability Modulo Theories (SMT) [DMB11] solver, which is an extension of SAT, but we only use its SAT capabilities to implement MixedColoring.

Because SAT only returns a Boolean answer for a given formula, satisfiable or unsatisfiable, MixedColoring cannot be directly encoded as SAT. We instead encode whether or not a given graph can be k-colored and test different k until we find χ, the minimal k with which the graph can be colored. We first encode k-colorability as a Boolean formula in section 3.5.1 and then explore different search algorithms to find χ quickly in section 3.5.2. These different ways of searching are the variants of `SAT`.

3.5.1 k-Colorability

Many SAT solvers internally reshape a given formula into Conjunctive Normal Form (CNF), i.e. a conjunction of disjunctions of literals, before solving, and the structure of this problem lends itself to CNF. Therefore we directly encode k-colorability as a Boolean formula in CNF by adding disjunctions to the formula at each step. We must model colors up to k, so we use the set of colors $C = \{1, \dots, k\}$

To model vertex colors with Boolean variables we introduce a variable for every combination of vertices and colors such that, for every vertex $u \in V$ and every color $i \in C$, the variable $c_{u,i}$ is true if u has the color i and is false otherwise.

$$\forall u \in V, \forall i \in C \colon c_{u,i} = \begin{cases} true & \text{if vertex } u \text{ has color } i \\ false & \text{otherwise} \end{cases} \tag{3.13}$$

This definition allows the solver to assign multiple colors or no color to a vertex, so a constraint is required to ensure that for each vertex u exactly one $c_{u,i}$ is assigned the value true. We enforce at least one color to be assigned by adding the disjunction of all colors to the CNF.

$$\forall u \in V \colon \bigvee_{i \in C} c_{u,i} \tag{3.14}$$

To enforce that at most one color is assigned to each vertex u, for every pair of different colors $i \in C, j \in C, i \neq j$ we add the disjunction of the negations of $c_{u,i}$ and $c_{u,j}$ to the CNF.

$$\forall u \in V, \forall i \in C, \forall j \in C, i \neq j \colon \neg c_{u,i} \vee \neg c_{u,j} \tag{3.15}$$

If there is an edge $\{u, v\} \in E$, the vertices u and v may not have the same color. This means that for every color $i \in C$ the variables $c_{u,i}$ and $c_{v,i}$ cannot both equal true, which we impose by adding the disjunction of their negations to the CNF.

$$\forall \{u, v\} \in E, \forall i \in C \colon \neg c_{u,i} \vee \neg c_{v,i} \tag{3.16}$$

If there is an arc $(u, v) \in A$, the color of vertex u must be less than the color of vertex v. This means that for every color $i \in C$, if $c_{v,i}$ is true, the variables $c_{u,j}$ must be false

for all $j \in \{i, \ldots, k\}$. We impose this by adding, for every pair of variables which cannot be simultaneously true, the disjunction of the negation of those variables.

$$\forall (u, v) \in A, \forall i \in C, \forall j \in \{i, \ldots, k\} \colon \neg c_{v,i} \vee \neg c_{u,j} \tag{3.17}$$

3.5.2 Search Algorithms

Known upper and lower bounds $\chi_{\max}, \chi_{\min}$ on the chromatic number χ also bound the range of ks which must be tested for colorability to $K = \{\chi_{\min}, \ldots, \chi_{\max}\}$. To find χ, we must find a pair k_0, k_1 such that k_1 is $k_0 + 1$ and the graph is k_1-colorable but not k_0-colorable, at which point we know that χ is k_1. We can search through this range in different ways to optimize for different factors. We denote these variants of `SAT` as `SAT-` followed by a shorthand of the search type.

SAT solvers are known [BHM21, Chapter 15.1] to have longer runtimes on unsatisfiable formulas than on satisfiable formulas, so it is of interest to minimize the amount of unsatisfiable formulas that the solver must test. A naive approach to achieve this is a descending linear search `SAT-DESC`, which is guaranteed to test only satisfiable formulas until the first unsatisfiable formula at k_0 is found. However, this maximizes the size of tested formulas.

Because the k-colorability formula's size increases quadratically with k, it is also of interest to test only the smallest necessary ks to minimize SAT input size, and consequently its runtime. We can achieve this with a naive ascending linear search `SAT-ASC`, which only tests ks until the first satisfiable formula at k_1 is found, guaranteeing that no k larger than necessary is tested. However, this maximizes the number of unsatisfiable formulas we test.

Linear searches minimize one undesirable property of formulas but maximize the other. A different approach is to minimize the overall number of formulas tested regardless of their size or satisfiability. A binary search `SAT-BIN`, in which the remaining range of ks to test is halved with each test, only tests $\mathcal{O}(\log |K|)$ formulas. We therefore expect that its worst-case runtime is much better than that of a linear search.

Because it is possible that one undesirable property has a larger impact than the other, we additionally combine directional and binary search into exponential search `SAT-EXPASC`, `SAT-EXPDESC`, starting from $\chi_{\min}$ and $\chi_{\max}$ respectively. Exponential search traverses an ordered search space by starting at one end and stepping towards the other end, doubling its step size with every step. Once a step passes the desired value, the section of search space between the previous and current step is searched by binary search. Exponential search retains the logarithmic behavior of binary search, but tests more k on the starting side of the range and potentially fewer k on the opposite side. If the impact of unsatisfiability on runtime is notably different from that of large formulas, we expect that one exponential search performs better than binary search, while the other performs worse.

As a reference for the performance of the other search types we also introduce the search type `SAT-ALL`, which simply tests every $k \in K$.

4 Experimentation and Analysis

In this section, we describe the experiments we perform to benchmark the runtime behavior of `Lawler`, `Polyspace`, `SAT`, and `ILP` and the graph instances used for experimentation. We obtain graphs from random generation and from a job-scheduling-problem dataset. For benchmarking we first examine each algorithm individually and determine how its variants, and modifications of the input graph, affect its runtime in order to obtain the optimal configuration for minimizing the algorithm's runtime on a given graph. While it's possible that multiple configurations of an algorithm could be optimal on different types of graphs, our experiments show that this is not the case for our algorithms. With optimal algorithm configurations in hand, we then compare runtimes between algorithms to determine which algorithm is the fastest, depending on the properties of a given input graph.

As there is a large variance of runtimes of an algorithm depending on the input graph, even within the same graph size, and it is infeasible to wait multiple hours for a single graph to be colored, we must abort an algorithm if it takes longer than some timeout threshold. Timeout thresholds are specified for each experiment. Additionally, if graphs are tested in order of ascending size, it may not make sense to continue evaluating an algorithm if it times out on a majority of recent graphs, especially when evaluating multiple algorithms or variants of the same algorithm in parallel. We therefore, in most experiments, stop evaluation of an individual algorithm or variant if it timed out more often than some specified timeout percentage on some specified amount of most recently tested graphs.

`Lawler` and `Polyspace` are implemented directly in C++. `ILP` and `SAT` are implemented through their respective C++ APIs. Apart from the Z3 [DMB08] and Gurobi [Gur26] libraries, only the standard library is used. The project is compiled using CMake and the Visual Studio Toolchain, and managed using the CLion IDE. Data from benchmarks is written to a csv file for further use. Plotting of data is done with Python and Matplotlib.

The C++ code used to evaluate algorithms for this thesis, the data obtained from evaluation, the python code used to generate plots, as well as any other related files, are available on our university's gitlab in the repository `https://gitlab2.informatik.uni-wuerzburg.de/s372212/mixed-coloring`. Files from this repository are referenced in this thesis by their path and name relative to the repository's root directory. We provide a table that associates every plot in this thesis with the file that contains the underlying data in Appendix A.

All computation is performed on an AMD Ryzen 7 3700X 8-Core processor with 32 GB of DDR4 RAM. The processor's base speed is 3.60 GHz and its boost speed 4.40 GHz. Although some algorithms support parallel execution, we restrict all algo-

Algorithm 4: An algorithm that randomly generates an undirected tree.

Input: tree size n
Output: a random undirected tree

```
1 RandomTree(n):
2     V = {1, ..., n}
3     Let f_R be a random bijective mapping of V to {1, ..., n}.
4     Construct the set E by randomly picking, for every u ∈ V with f_R(u) ≠ n,
        an edge {u, v} such that f_R(u) < f_R(v).
5     T = (V, E)
6     return T
```

rithms to sequential execution to allow meaningful comparison. Note that experiments were performed over a range of temperatures and at varying utilization of the processor. Some small fluctuations visible in some plots are likely a result of this.

4.1 Graph Instances for Experimentation

To evaluate our algorithms we require a large number of graph instances. We are interested in controlling properties of these graphs, such as the size, as well as the number of edges and arcs, to ensure that sufficient data points exist for all combinations of properties that we study. Additionally, because we study real-world runtime, some or all graph instances should resemble real world problems. We achieve these goals by randomly generating a large number of graphs instances, which is fast and easy, and supplementing them with a smaller selection of instances with real-world relevance that are harder to find. This gives us a large and evenly distributed set of instances to analyze for behavior and trends, as well as a small set of instances to verify that these behaviors hold in non-random graphs.

We considered, but discarded, a third idea for obtaining mixed graphs. It may be of interest to a reader who requires a large number of mixed graphs with known chromatic number and is available in Appendix D.

4.1.1 Randomly Generated Graphs

A large number of graph instances are required to evaluate the runtime behavior of our algorithms. We use Algorithm 6 to randomly generate graphs of a given size n with given numbers of edges n_e and arcs n_a. The algorithm guarantees that graphs are simple, connected and don't have directed cycles. Any graph which has these properties can be generated. Note that, as a connected graph of size n must have at least $n-1$ connections and a simple graph of size n can have at most $\frac{n(n-1)}{2}$ connections, the algorithm's arguments must satisfy

$$n - 1 \leq n_e + n_a \leq \frac{n(n-1)}{2} \tag{4.1}$$

Algorithm 5: An algorithm that obtains a mixed graph based on the structure of a given undirected graph by turning a given number of its edges into arcs.

Input: undirected graph G', number of edges to orient n_a
Output: a mixed graph G without directed cycles obtained by orienting n_a edges of G'

`OrientEdges`(G', n_a)`:`
  Construct the set $\bar{E}$ by randomly picking, without duplicates, n_a elements from $E(G')$.
  Let f_R be a random bijective mapping of $V(G')$ to $\{1, \dots, n\}$.
  $A = \left\{ \begin{array}{ll} (u,v) & \text{if } f_R(u) < f_R(v) \\ (v,u) & \text{otherwise} \end{array} \mid \{u,v\} \in \bar{E} \right\}$
  $G = (V(G'), E(G') \setminus \bar{E}, A)$
  return G

Algo 6 begins by generating a random undirected tree T of size n, that is, a connected undirected graph with n vertices and $n-1$ edges, using Algo 4 as a subroutine.

Algo 4 generates a random tree by first creating its vertices, labeled 1 to n. Then it creates some random bijective mapping f_R of the vertices to the first n natural numbers to obtain an order of vertices that is separate from the order of their labels, so that the structure of the resulting tree does not depend on vertex labels. Then, for every vertex $u \in V$ except the one with $f_R(u) = n$, it randomly picks a vertex v with $f_R(u) < f_R(v)$ and adds the edge $\{u, v\}$ to the set of edges of the tree. This ensures that each vertex is connected to the rest of the tree using only $n-1$ edges. Finally the tree is returned.

Next, Algo 6 determines the number of additional edges $c = n_e + n_a - |V(T)|$ required to meet the desired total of $n_e + n_a$ connections. It constructs the set of additional edges E' by randomly drawing c elements from the set $(\binom{V(T)}{2} \setminus E(T))$ of all possible edges that T does not have. Using this set, it then constructs the undirected graph G' from the vertices $V(T)$ and the edges $(E(T) \cup E')$. This graph has the structure and desired number of connections of the final graph, but is undirected.

To obtain a mixed graph from this undirected graph, the subroutine Algo 5 is used, which changes n_a of G's edges into arcs by orienting them. It again creates some random bijective mapping f_R of the vertices to the first n natural numbers to obtain an order of vertices that is separate from the order of their labels, so that the orientation of the arcs does not depend on vertex labels. Next, it constructs the set $\bar{E}$ of edges to orient by randomly picking n_a elements from E'. Each edge in $\bar{E}$ is turned into an arc (u, v) by orienting it such that $f_R(u) < f_R(v)$ and added to the set of arcs A, i.e. f_R is the topological order of A. Finally, the subroutine constructs the mixed graph G from the vertices $V(G')$, edges $(E(G') \setminus \bar{E})$, and arcs A and returns it. Algo 6 also returns this graph.

Proposition 4.1. *Graphs generated by Algo 6 are simple, connected, and don't have directed cycles.*

Algorithm 6: An algorithm that randomly generates a mixed graph that is simple, connected, has no directed circuits, and has a specified size, number of edges, and number of arcs.

Input: graph size n, number of edges n_e, number of arcs n_a
Output: a random graph that is simple, connected, has no directed cycles, and has the properties specified by Input

```
RandomGraph(n, n_e, n_a):
    T = RandomTree(n)
    c = n_e + n_a − |V(T)| // number of missing connections
    Construct the set E' by randomly picking, without duplicates, c elements
      from (binom(V(T), 2) \ E(T)).
    G' = (V(T), E(T) ∪ E')
    G = OrientEdges(G', n_a)
    return G
```

Proof. **Trees** We first prove that an undirected graph T returned by Algo 4 is an undirected tree by showing that it is connected, meaning there is a walk between each pair of vertices, and that it has at most $n-1$ edges. Note that at least $n-1$ edges is implied by connectedness.

Number of Edges The edges E of T are constructed by adding an edge for every vertex $u \in V$ except one. Because $|V| = n$, there can be at most $n-1$ elements in E.

Connectedness We label as w the vertex $w \in V(T)$ with $f_R(w) = n$. Obviously there is a path from w to w.

Suppose, for some static i, that for all $v \in V(T)$ with $f_R(v) > i$ there is a path from v to w. By construction, for any vertex $u \in V(T)$ with $f_R(u) = i$, there exists an edge to some vertex $v \in V(T)$ with $f_R(v) = j$ and $i < j$. Thus a path from u to w exists, obtained by prepending u to the path from v to w.

Therefore, for all $u \in V(T)$, there exists a path from u to w. Given any pair of vertices $u, v \in V(T)$, we can then find a walk between them by first following the path $\langle u, \ldots, w\rangle$ and then following the reverse of the path $\langle v, \ldots, w\rangle$. Therefore T is connected.

This is sufficient for showing that a graph is a tree, so every graph returned by Algo 4 is a tree.

Undirected Graph Next we prove that the intermediate undirected graph G' in Algo 6 is simple and connected. By definition, the tree T is simple. We construct the edges of G' from the edges of T and from edges that are picked without duplicates from a set that does not contain any edges of T, therefore G' does not contain duplicate edges and is simple. Note that $\binom{V(T)}{2}$ contains all possible edges, so there are enough edges to choose from without duplicates. Additionally, because

T is connected and G' contains the edges of T, G' is connected.

Mixed Graph Lastly we prove that the final mixed graph G returned by Algo 5 is simple, connected, and has no directed cycles. Because the undirected input graph G' is simple and connected, and the algorithm only replaces connections with other connections, the output graph G must also be simple and connected.

Every arc of G is constructed such that its tail u has a lower index in the order defined by f_R than its head v. This means that any directed path can only start at a vertex u with lower index and end at a vertex v with higher index. Because f_R is bijective and u, v have different indices, u and v must be different vertices. Therefore, the start and end vertex of a directed path in G cannot be the same, so it cannot be closed, meaning a directed cycle cannot exist in G.

Therefore every graph generated by Algo 6 is simple, connected, and has no directed cycles.

□

Proposition 4.2. *Algo 6 can generate every graph that is simple, connected, and doesn't have directed cycles.*

Proof. **Every Tree** Algo. 4 can generate every undirected tree. To demonstrate that a given undirected tree $\bar{T}$ can be constructed, we show that there is a mapping f_R such that for every $u \in V(\bar{T})$ with $f_R(u) \neq n$ there is a $v \in V(\bar{T})$ with $f_R(u) < f_R(v)$ where the edge $\{u, v\}$ is in $E(\bar{T})$. We construct the mapping as follows.

First, pick a vertex $u \in V(\bar{T})$ and set $f_R(u) = n$. Create the set $S = \{u\}$. Then, for $i = (n-1), \ldots, 1$:

> Pick a pair of vertices $u \in (V(\bar{T}) \backslash S)$ and $v \in S$ such that $\{u, v\} \in E(\bar{T})$. Such a pair must exist because otherwise $\bar{T}$ would be disconnected into $S, (V(\bar{T}) \setminus S)$ as there would be no edge between them. Set $f_R(u) = i$. By construction of S in descending order of f_R, the selected v must have $f_R(v) > i$. Insert u into S.

As shown during construction of f_R, there is a v with the desired properties for every u, so the algorithm can generate any given tree $\bar{T}$.

Every Undirected Graph Algo. 6 can, as its intermediate step G', generate every simple, connected, undirected graph. This is easily verifiable by recalling that every connected graph contains at least one spanning tree. This spanning tree can be generated by Algo. 4, and the remaining edges can be picked in line 4.

Every Mixed Graph Algo. 6 can generate every simple, connected, mixed graph without directed cycles. Given such a graph $\bar{G}$, the underlying undirected graph $\bar{G}_U = (V(\bar{G}), E(\bar{G}) \cup E')$ with $E' = \{\{u, v\} \mid (u, v) \in A(\bar{G})\}$ can be generated as an intermediate step, as shown above. Naturally, E' must be chosen as the set of $\bar{G}_U$'s edges to orient by Algo. 5. Additionally, as edges are oriented in order

of f_R, f_R must be set to the topologically sorted order of $V(\bar{G})$. Sorting $V(\bar{G})$ topologically is possible because it has no directed cycles. In this way, any such $\bar{G}$ can be generated by Algo. 6.

□

4.1.2 Job Scheduling Instances

Randomly generated graphs may not necessarily reflect graphs as they appear in real world applications. The jobshop1 [MV] dataset contains 82 job scheduling problem (JSP) instances, compiled by Mattfeld and Vaessens from various related papers. We discard the durations of job steps to convert these instances to unit-time JSP as Kouider et al. [KHOO15] did.

A unit-time JSP instance consists of a number of machines $\{m_1, \ldots, m_M\}$ and jobs $\{j_1, \ldots, j_J\}$ where each job j_i has a number of steps $\{s_{i,1}, \ldots, s_{i,S_i}\}$. Each step has an assigned machine which it must run on, and steps of a job must be performed in order. The solution of a unit-time JSP is the minimum number of unit-time steps it takes to complete all jobs. This problem can be modeled as MixedColoring: Each step $s_{i,j}$ is a vertex. Each pair $s_{i,j}, s_{k,l}$ of steps which run on the same machine is an edge $\{s_{i,j}, s_{k,l}\}$. Each pair $s_{i,j}, s_{i,j+1}$ of consecutive steps of the same job is an arc $(s_{i,j}, s_{i,j+1})$. The chromatic number of the resulting graph is the minimum unit-time it takes to complete all jobs.

Many of our experiments are performed on graphs of ascending size, while the instances of the Jobshop1 dataset only have a few different sizes with most of them at or above size 100. To obtain comparable results on random graphs and Jobshop1 instances, we create Jobshop1 instances of ascending size from each of them as follows. We first pick all Jobshop1 instances of size 10x10. These are: abz5-6, ft10, la16-20, orb01-10. Then, for each of them, we alternate between removing the first job and removing a machine to create instances of size 10x10, 9x10, 9x9, 8x9, ..., 3x3, 2x3, 2x2.

Recall that a job is modeled as a sequence of steps connected by arcs in order of their execution. To remove a job from a graph created from a Jobshop1 instance, we first pick a vertex with indegree 0, the first step of a job. We then find the longest directed path starting at this vertex. This path contains exactly the vertices that model one job. We remove all these vertices, and connections that they are incident to, from the graph.

Recall that a machine is modeled by adding edges between all vertices that represent steps that must run on this machine. To remove a machine from a graph created from a Jobshop1 instance, we pick a vertex v. The closed neighborhood $N[v]$ then contains exactly the vertices that model steps that must run on the same machine. We remove all these vertices, and connections that they are incident to, from the graph.

4.1.3 Benchmark Sets

Distribution of Connections When generating multiple random graphs of the same size, instead of leaving everything up to chance, we want to choose the number of edges and arcs of each graph such that the experiments we perform on the graphs are repre-

sentative of as many graphs of that size as possible. Our approach to achieving this is as follows (See Fig. 4.1 for reference).

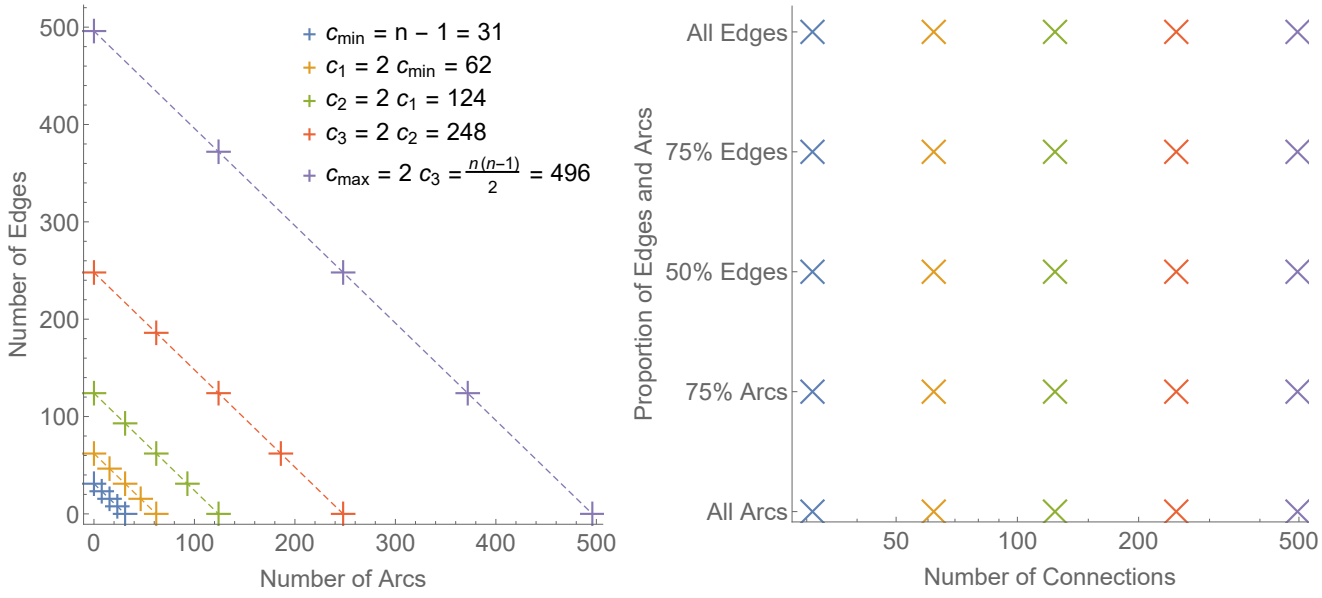


Fig. 4.1: A distribution of 25 graphs of size n=32 into 5 groups with 5 members per group such that members of a group have the same number of connections and groups are logarithmically distributed from the minimal to the maximal number of connections. The legend applies to both graphics. Left: Visualization of the distribution by number of edges and arcs. Right: Demonstration of how we place such distributions of graphs in a plot.

We first divide the instances into m groups with l members by the number of their connections, starting with the smallest possible number of connections $c_{\min} = n - 1$, with intermediate values $c_1, \ldots, c_{m-2}$ spaced logarithmically by a factor f such that the final group has the largest possible number of connections $c_{\max} = \frac{n(n-1)}{2}$.

$$f^{m-1} c_{\min} = f^{m-2} c_1 = \cdots = f c_{m-2} = c_{\max} \tag{4.2}$$

As $c_{\min}$ is linear in the number of vertices while $c_{\max}$ is quadratic, a linear distribution between them would neglect values closer to $c_{\min}$ for larger graphs. A logarithmic distribution provides a smooth transition between the different scalings, which we have also observed in practice with plots of small preliminary tests.

Within each group of graphs we distribute the number of edges and arcs linearly between the maximum number of edges with zero arcs and the maximum number of arcs with zero edges. Linear distribution suffices here, as the numbers of edges and arcs

scale symmetrically. Although groups with a larger number of connections have a larger number of possible graphs, we consider each group equally interesting and therefore generate the same number of graphs for each group.

Description of Benchmark Sets The first benchmark set we create is the one we use to compare configurations of algorithms. For this initial comparison we are most interested in the runtime depending on graph size, so we generate a number of graphs of each size in ascending order. We choose 25 graphs per size, distributed into $m = 5$ groups based on the number of connections with $l = 5$ graphs per group. This provides a good trade-off of fast evaluation time while still covering the possible numbers of edges and arcs well. There is no formal limit to the size of graphs in this benchmark set, though in practice it is naturally limited to some size that is sufficient for all experiments that we do. We denote this benchmark set as **A25**.

Secondly, we create a similar ascending benchmark set with $m = 10$ groups and $l = 10$ graphs per group. We use it for the final comparison between the optimal configurations of each algorithm, accepting a longer evaluation time to obtain a better resolution for this important experiment. We denote this benchmark set as **A100**.

Lastly, to study the effect of edges and arcs on the runtime of algorithms, we create a benchmark set of 2500 graphs of equal size $n = 20$, distributed into $m = 50$ groups with $l = 50$ graphs per group. This eliminates effects where two graphs could have the same number of edges and arcs, but different runtimes due to having a different size. We denote this benchmark set as **E2500S20**. We create the same benchmark set with graph size $n = 50$, denoted **E2500S50**, graph size $n = 100$, denoted **E2500S100**, and graph size $n = 150$, denoted **E2500S150**.

In summary, we use the following benchmark sets:

- **A25**: Graphs of **A**scending size with **25** graphs per size. The number of connections are distributed into 5 groups, each with 5 members.
- **A100**: Graphs of **A**scending size with **100** graphs per size. The number of connections are distributed into 10 groups, each with 10 members.
- **E2500S20**: **2500** graphs of **E**qual **S**ize $n = \mathbf{20}$. The number of connections are distributed into 50 groups, each with 50 members.
- **E2500S50**: **2500** graphs of **E**qual **S**ize $n = \mathbf{50}$. The number of connections are distributed into 50 groups, each with 50 members.
- **E2500S100**: **2500** graphs of **E**qual **S**ize $n = \mathbf{100}$. The number of connections are distributed into 50 groups, each with 50 members.
- **E2500S150**: **2500** graphs of **E**qual **S**ize $n = \mathbf{150}$. The number of connections are distributed into 50 groups, each with 50 members.
- **JOBSHOP1A**: The graphs obtained by converting **jobshop1** first from JSP to unit-time JSP, then to MixedColoring, and then converting each instance of size 10x10 to multiple instances of **A**scending size as per Section 4.1.2.

For reproducibility, we describe in Appendix B how exactly our benchmark sets are created in our implementation.

4.1.4 Plotting Choices

In this section we provide a brief overview of our choices when displaying experimental results, and the reasoning behind them, so as to not repeat the same information for every plot.

Much of the data we obtain is noisy and spread across a large range of values. For this reason, we often display the median of values to show a clear trend. We use the median instead of the mean because the median is unaffected by the ceiling effect caused by algorithm timeouts. Naturally, this requires us to interpret data points that timed out as having a longer runtime than data points that did not time out. When plotting the median alongside all values, we generally show the median at full opacity and the other values at significantly reduced opacity.

When we plot the relative runtime of algorithms, this reasoning no longer holds because data points where one or both algorithms timed out do not necessarily have a longer relative runtime than data points without timeouts. Depending on the purpose of the plot we may calculate the relative median in two ways.

For plots where we intend to show whether or not a variant provides an improvement over some other variant without much consideration for accurately representing the course of runtimes, we calculate the relative median by first calculating the relative runtime of all data points and then taking the median of these relative runtimes, that is, we take the *median of quotients*. We discard data points where one or both algorithms timed out. This accurately displays whether data points of one algorithm are generally above or below data points of another, but can distort the plot's course compared to data without timeouts.

For plots where the intent is to show the course of one runtime relative to the other, we calculate the relative median by dividing the median of one runtime by the median of the other runtime, that is, we take the *quotient of medians*. This accurately displays how the medians of algorithm runtimes behave relative to each other, but can lead to situations where the median is not in the middle of the other values.

We generally use the median of quotients, and mention explicitly when we use the quotient of medians.

4.2 Comparison between Algorithm Configurations

The primary goal of this section is to find the fastest configuration of each algorithm. We are additionally able to confirm or reject some assumptions made in Chapter 3. Note that analysis is qualitative, as it is clear from plots which variant performs better without statistical methods.

We begin experimentation by analyzing the runtime of each algorithm's configurations. From this, we obtain an optimal configuration for minimizing the runtime of each algorithm on a given graph. We perform each experiment in this section

- on the benchmark set **A25** with a timeout threshold of 10 seconds, stopping evaluation of a variant if it times out on more than 70% of the 25 most recently colored graphs, and
- on the benchmark set **JOBSHOP1A** with a timeout threshold of 10 seconds, but without stopping variant evaluation as this benchmark set is too small to warrant it.

We use **A25** to find trends across a large variety of graphs, and **JOBSHOP1A** to test if these trends hold in real-world applications. Note that, as job scheduling instances are only representative of a small subset of graphs, not all trends will be present in the results of **JOBSHOP1A** experiments. If we find a trend that is present in **A25**, but not in **JOBSHOP1A**, we consider this trend valid so long as it is not directly contradicted by **JOBSHOP1A**. Note that, as all trends we find in this section are either confirmed or not contradicted, we provide the **JOBSHOP1A** plots in appendix C and briefly reference them in the relevant section.

We obtain the following optimal configurations for each algorithm in this section:

`Lawler`: `Lawler-sMISmod` with transitive reduction of the input graph.

`Polyspace`: `Polyspace-sMISmod` with $\alpha = 0.001$ and transitive reduction of the input graph.

`ILP`: `ILP-INT` with transitive reduction of the input graph.

`SAT`: `SAT-EXPASC` with transitive reduction of the input graph.

4.2.1 `Lawler`

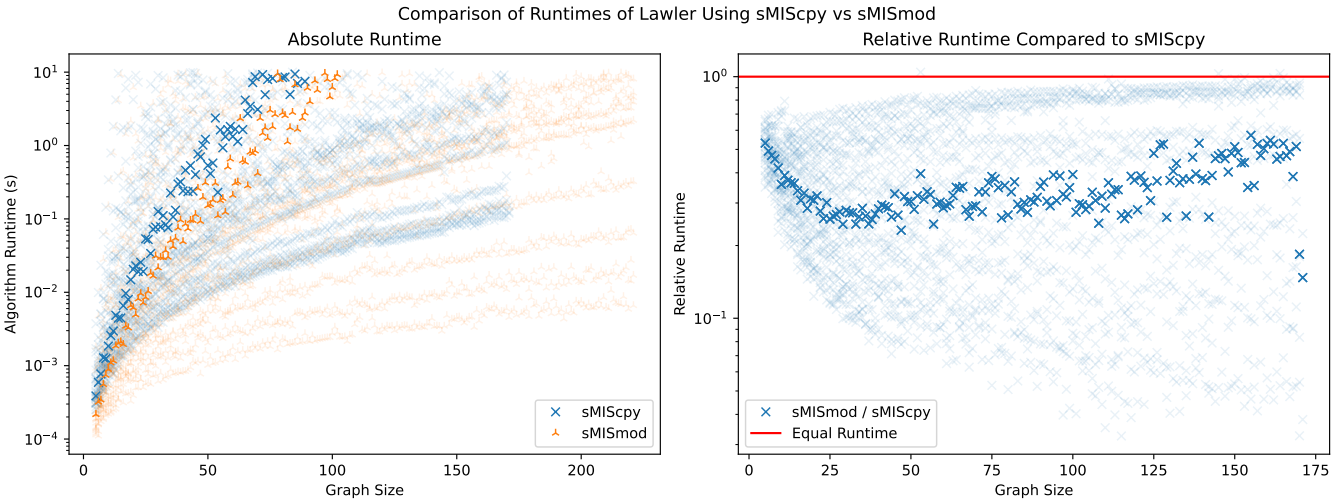


Fig. 4.2: The runtimes of `Lawler` with `sMISmod` versus `sMIScpy` on the benchmark set **A25**. Execution has a timeout of 10s per graph per configuration. Evaluation of a configuration is stopped if it times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity.

MIS°-Variants The only variants of `Lawler` are different implementations of MIS°. We show in Fig. 4.2 the runtime of `Lawler` with `sMIScpy` as a subroutine compared to `sMISmod` as a subroutine. It's clear that `sMISmod` performs better, having a faster runtime on almost every single graph. There is, however, a lot of variance in how much benefit it provides, ranging from a two- to ten-fold speedup of the median runtime. We show in Fig. C.1 the same runtime on the benchmark set **JOBSHOP1A**. On this benchmark set, `sMISmod` also performs better, confirming that the trend holds in a real-world application. We select `Lawler-sMISmod` as the optimal variant.

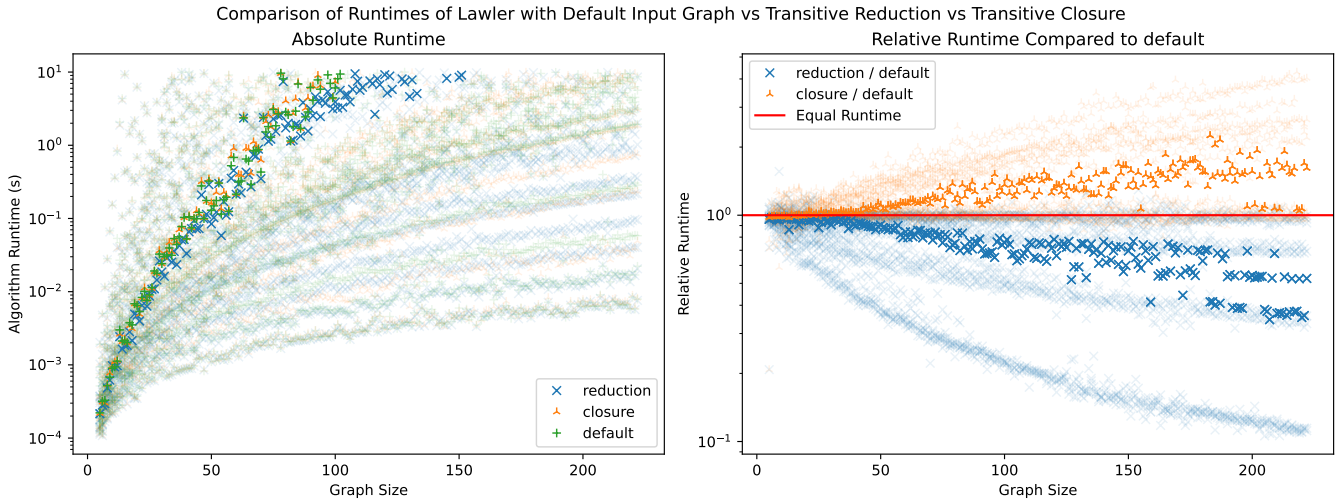


Fig. 4.3: The runtimes of `Lawler` when run on the original graph and its transitive variants on the benchmark set **A25**. Execution has a timeout of 10s per graph per configuration. Evaluation of a configuration is stopped if it times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity.

Transitive Variants We additionally test the effect of coloring the transitive variants the input graph compared to the input graph itself. We show in Fig. 4.3 the runtime of `Lawler` on the original input graph, the graph's transitive reduction, and its transitive closure. While the effect is less pronounced, with only an up to two-fold speedup of the median runtime, it's still clear that coloring the transitive reduction of an input graph instead of the original graph is beneficial for minimizing `Lawler`'s runtime. This trend is neither confirmed nor contradicted on **JOBSHOP1A** (see Fig. C.2). We select the transitive reduction as the optimal graph variant for `Lawler`.

4.2.2 `Polyspace`

`Polyspace` has variants based on the choice of α and MIS°-implementation.

Alpha-Variants We show in Fig. 4.4 the runtime of `Polyspace` with values of α ranging from 0.1 to 0.9. We observe that the runtime decreases as α decreases, suggesting that

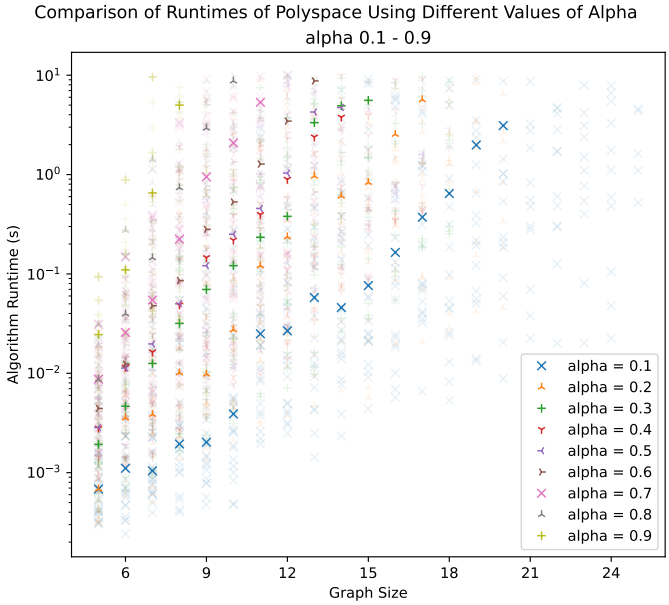


Fig. 4.4: The runtimes of `Polyspace` with `sMISmod` and a large range of values of α on the benchmark set **A25**. Execution has a timeout of 10s per graph per configuration. Evaluation of a configuration is stopped if it times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity.

α should be as small as possible for the shortest runtime. This motivates a further experiment with α values between 0 and 0.1 to confirm that this trend continues. Recall that α must be greater than 0 and `Polyspace` partitions a graph into $\langle S, I, T\rangle$ with $|S| < n\alpha$. Note that, if $n\alpha \leq 1$, then $|S| = 0$. As such, while we cannot set α to 0, we can choose values where $|S|$ is 0.

We show in Fig. 4.5 the results of this second experiment, with variant runtimes relative to the variant with $\alpha = 0.01$. This way, we can clearly observe the thresholds where $|S|$ switches from 0 to 1 in this plot, with variants experiencing a five- to ten-fold runtime increase when crossing it. Up to $n = 10$, nearly all markers are stacked at equal runtime, as all displayed αs behave the same up to this point. At $n = 11$, the markers of $\alpha = 0.1$ (red) detach from that stack and rise to a longer runtime. Similar behavior can be observed for all variants

This confirms that α should be set to a value that guarantees $|S| = 0$ for all graph sizes. In practice we use $\alpha = 0.001$ for this, but any sufficiently small α would behave the same. This trend is confirmed on **JOBSHOP1A** (see Figs. C.3, C.4).

For comparison, we show in Fig. 4.6 only the runtimes on complete undirected graphs in **A25**. Lauerbach's choice of $\alpha \approx 0.563964$ for minimizing asymptotic runtime shows good practical performance on these graphs. As we discussed in Section 3.3, this is because complete undirected graphs are the worst case for small values of α and lead to a runtime that is factorial in graph size. However, even with Lauerbach's choice of α, `Polyspace` already times out on graphs of size 9, while our other algorithms perform

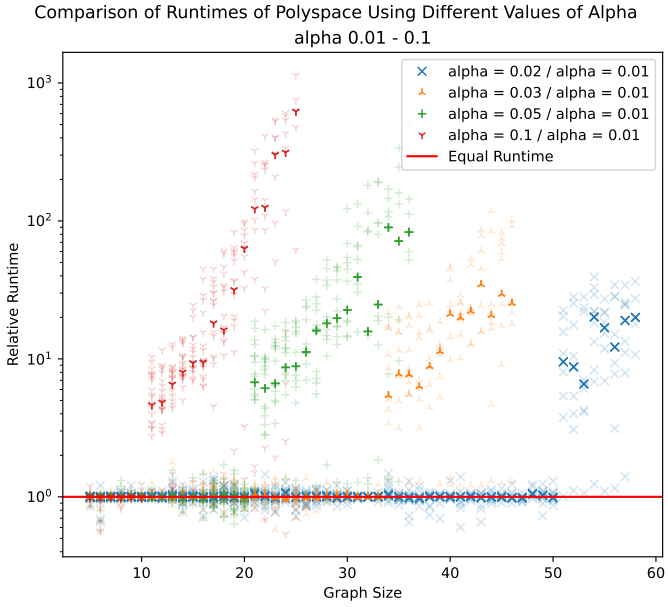


Fig. 4.5: The runtimes of `Polyspace` with `sMISmod` and values of α close to 0 on the benchmark set **A25**. Execution has a timeout of 10s per graph per configuration. Evaluation of a configuration is stopped if it times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity.

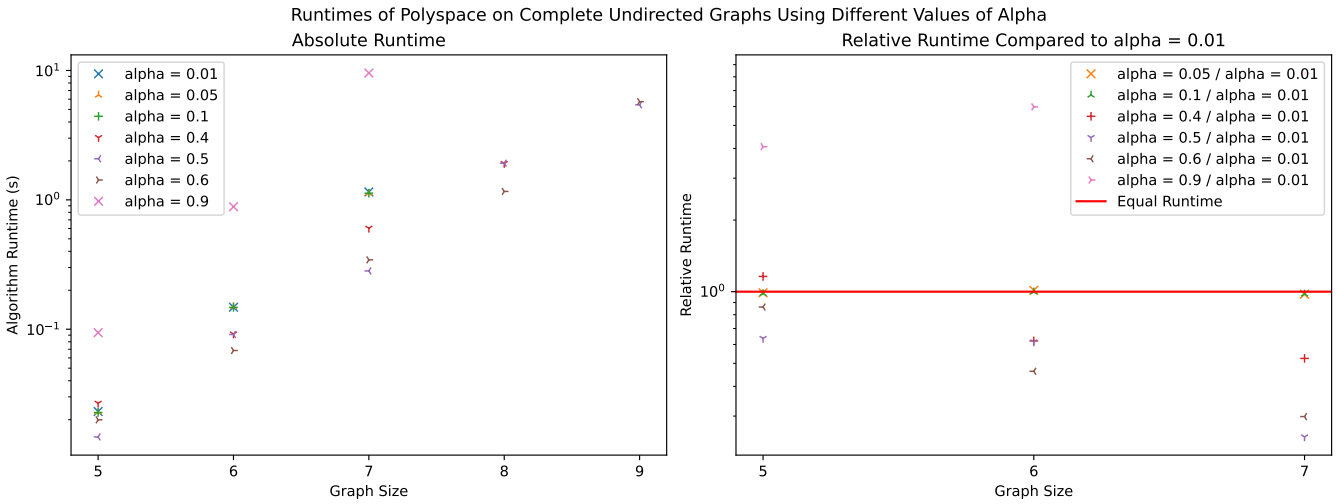


Fig. 4.6: The runtimes of `Polyspace` with `sMISmod` on the complete undirected graphs in the benchmark set **A25** with different values of α. Execution has a timeout of 10s per graph per configuration.

better on these graphs, so it seems unnecessary to introduce a special case for `Polyspace` on complete undirected graphs.

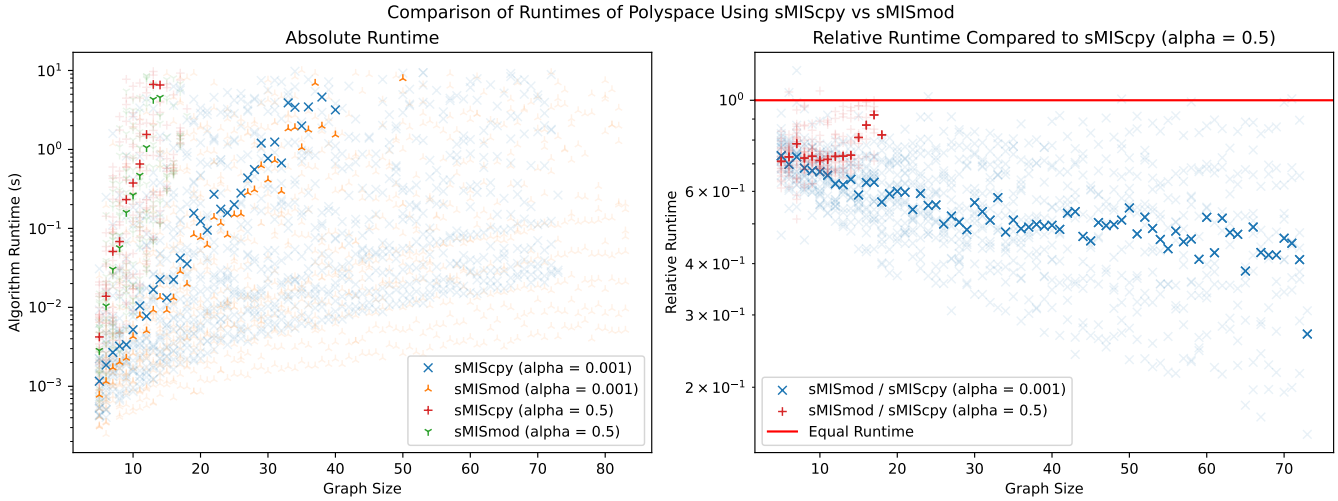


Fig. 4.7: The runtimes of `Polyspace` with `sMISmod` versus `sMISpy` and α taking the values 0.001 and 0.5 on the benchmark set **A25**. Execution has a timeout of 10s per graph per variant. Evaluation of a variant is stopped if it times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity.

MIS°-Variants We show in Fig. 4.7 the runtime of `Polyspace` with `sMISpy` as a subroutine compared to `sMISmod` as a subroutine. While the benefit of `sMISmod` is smaller than for `Lawler`, providing only a two-fold decrease in runtime, it is clearly the faster subroutine. This trend is confirmed on **JOBSHOP1A** (see Fig. C.5). We select `Polyspace-sMISmod` with $\alpha = 0.001$ as the optimal variant.

Transitive Variants We additionally test the effect of coloring the transitive variants of the input graph compared to the input graph itself. We show in Fig. 4.8 the runtime of `Polyspace` on the original input graph, the graph's transitive reduction, and its transitive closure. While the effect of coloring transitive variants on the runtime of `Polyspace` is minimal, there is still a definite pattern of the transitive reduction being slightly faster than the original graph and the transitive closure. This trend is neither confirmed nor contradicted on **JOBSHOP1A** (see Fig. C.6). We select the transitive reduction as the optimal graph variant for `Polyspace`.

4.2.3 `ILP`

Variable-Type-Variants The only variants of `ILP` are the implementations with binary variables `ILP-BIN` and with integer variables `ILP-INT`. We show in Fig. 4.9 the runtime of `ILP-BIN` compared to `ILP-INT`. It's clear that `ILP-INT` performs better, having a faster runtime on almost every single graph, and with up to a hundred-fold decrease in median runtime compared to `ILP-BIN`. This trend is confirmed on **JOBSHOP1A** (see Fig. C.7). We select `ILP-INT` as the optimal variant.

Our assumption based on previous work, that a binary formulation would perform

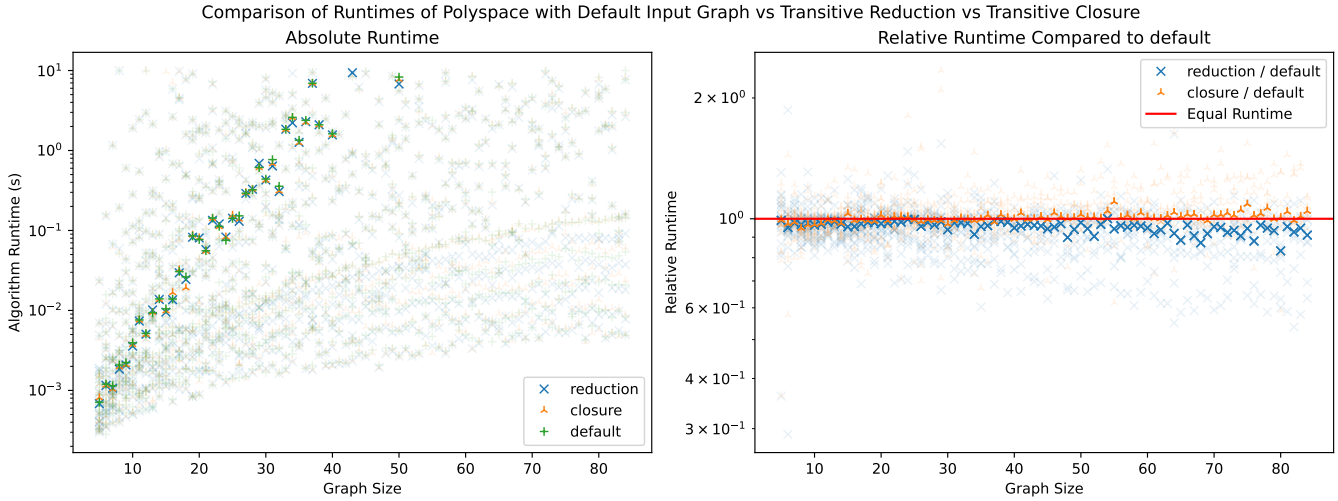


Fig. 4.8: The runtimes of `Polyspace` when run on the original graph and its transitive variants on the benchmark set **A25**. Execution has a timeout of 10s per graph per variant. Evaluation of a variant is stopped if it times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity.

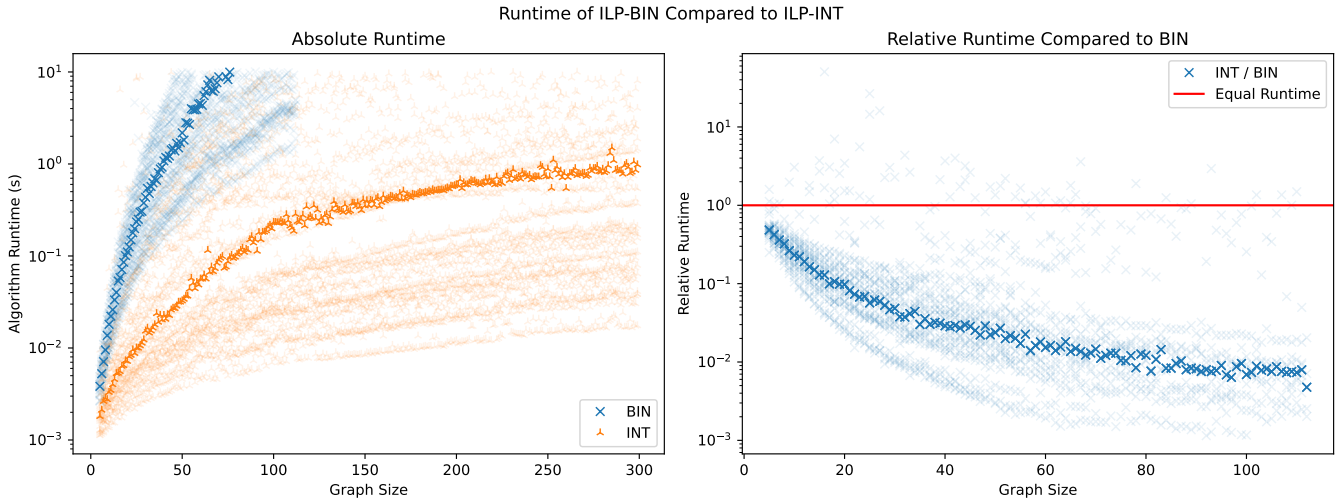


Fig. 4.9: The runtimes of `ILP-BIN` versus `ILP-INT` on the benchmark set **A25**. Execution has a timeout of 10s per graph per configuration. Evaluation of a configuration is stopped if it times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity. Model size of median runtime data points is shown as a line.

better, was incorrect. For further discussion of the dependence of `ILP`'s runtime on the ILP formulation, we show in Fig. 4.10 the median runtimes of Fig. 4.9, and we additionally show the model size of the `ILP` variants for each such data point. We calculate the model size as the sum of the numbers of variables and constraints in the

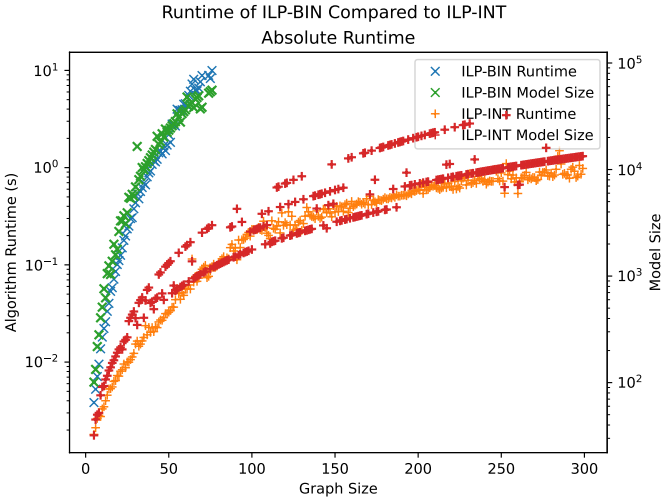


Fig. 4.10: The median runtimes of `ILP-BIN` versus `ILP-INT` shown in Fig. 4.9. For each median data point, the model size of `ILP-BIN` and `ILP-INT` is also shown.

model. For `ILP-BIN`, this size is

$$\underbrace{n^2+1}_{\text{Variables}}+\underbrace{n+n^2+2n_en+n_an}_{\text{Constraints}}, \tag{4.3}$$

and for `ILP-INT` it is

$$\underbrace{n+1+2n_e}_{\text{Variables}}+\underbrace{n+4n_e+n_a}_{\text{Constraints}}, \tag{4.4}$$

where n is the graph size, n_e the number of edges, and n_a the number of arcs.

We observe a strong correlation between runtime and model size that even holds between the variants, i.e. if `ILP-BIN` has similar model size on one graph as `ILP-INT` on another graph, then they have similar runtimes on these graphs as well. This suggests that if a ILP model would be notably larger in binary formulation than in integer formulation, it is preferable to use an integer formulation. However, the results could also indicate that our binary formulation is poorly designed. Other binary formulations would have to be tested to verify this, which goes beyond the scope of this thesis.

Transitive Variants We additionally test the effect of coloring the transitive variants the input graph compared to the input graph itself. We show in Fig. 4.11 the runtime of `ILP-INT` on the original input graph, the graph's transitive reduction, and its transitive closure. The original graph and transitive closure have similar runtimes across the full range of graph sizes, while the transitive reduction shows a small but consistent improvement to median runtime and significant improvements on some graphs. This trend is neither confirmed nor contradicted on **JOBSHOP1A** (see Fig. C.8). We select the transitive reduction as the optimal graph variant for `ILP-INT`.

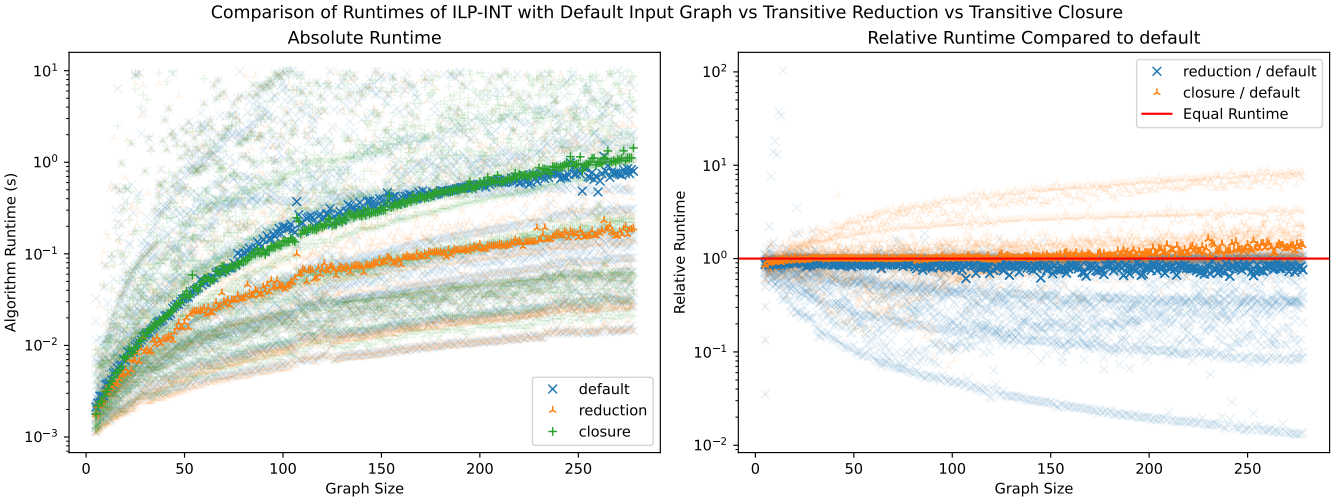


Fig. 4.11: The runtimes of `ILP-INT` when run on the original graph and its transitive variants on the benchmark set **A25**. Execution has a timeout of 10s per graph per variant. Evaluation of a variant is stopped if it times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity.

4.2.4 `SAT`

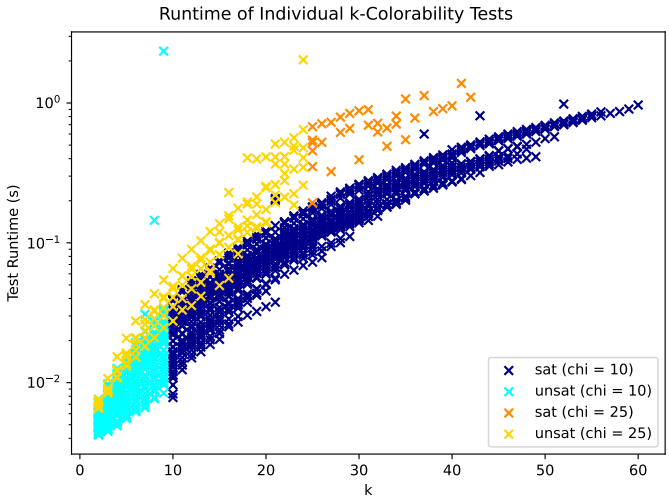


Fig. 4.12: The runtimes of `SAT-ALL`'s k-colorability tests on the benchmark set **A25**, limited to graphs with $\chi = 10$ and $\chi = 25$, color coded by satisfiability of their formula. Execution times out if `SAT-ALL` runtime exceeds 20s on a graph. Evaluation is stopped if `SAT-ALL` times out on more than 70% of the 25 most recent graphs.

Search-Variants The only variants of `SAT` are its different search types `SAT-ASC`, `SAT-DESC`, `SAT-BIN`, `SAT-EXPASC`, `SAT-EXPDESC`. To obtain the runtimes of these search types, instead of running each variant separately, we test k-colorability for every k. We then

simulate the runtime each variant would have had by summing up the runtimes of the k-colorability tests it would have performed. For this reason, the timeout for this experiment depends only on the runtime of `SAT-ALL`, and the timeout is set to 20 seconds instead of 10 seconds to compensate for `SAT-ALL` having a longer runtime than the real search types of `SAT`.

We begin analysis by testing how strong the effects discussed in Section 3.5.2 are. The first effect is an increase in the runtime of k-colorability with larger k, and the second effect is a longer runtime of unsatisfiable formulas compared to satisfiable formulas. To visualize these effects, we directly show the runtimes of k-colorability tests in Fig. 4.12. We group the graphs by their chromatic number, so that within each group there is a distinct cutoff where formulas switch from unsatisfiable to satisfiable, allowing us to see the second effect clearly. We only show two groups, as more groups clutter the plot and reduce visibility. As expected, we observe that runtime increases sharply with larger k. However, we do not observe any reduction in runtime at the boundary of unsatisfiability to satisfiability. This suggests that the second effect either does not exist for our formulation, or is significantly weaker than the effect of incrementing k by 1. Due to the large difference between these two effects, we expect that ascending search types perform well and that descending search types perform poorly.

This leads us to Fig. 4.13 (top left, top right), where we show the runtime of each search type, with `SAT-ALL` as a worst-case reference. Note that, despite comparing the courses of runtimes in this plot, we calculate the relative median as the median of quotients. This is because, by nature of the experiment, either all variants time out or no variant times out, so the distorting effect of using the median of quotients is negligible.

`SAT-DESC` runtimes are almost as long as those of `SAT-ALL` across all graphs, while `SAT-ASC` has lower runtime on most graphs by a factor of 10 to 1000 depending on graph size, which is expected due to the much longer k-colorability runtime of larger k. However, it is worth noting that `SAT-ASC` has a worst-case performance equal to `SAT-ALL`. This is because for graphs with a chromatic number equal to their size, `SAT-ASC` has to test all possible k to find χ.

As expected, `SAT-BIN` has a much better worst-case runtime than linear searches due to testing fewer k overall. However, its best-case runtime is more than an order of magnitude higher than the best-case runtime of `SAT-ASC` and `SAT-EXPASC`. This is because `SAT-BIN` always starts in the middle of the range of ks, so it cannot take full advantage of the low runtime of testing small ks on graphs with a small χ. This effect is even more detrimental for `SAT-EXPDESC` because it starts at the largest k. In contrast, `SAT-EXPASC` can take full advantage of the low runtimes of small ks for small χ, while also retaining the binary-search advantage of testing only logarithmically many ks. This gives it a best-case runtime close to `SAT-ASC` and a worst-case runtime close to `SAT-BIN`.

Additionally, as shown by Fig. 4.13 (bottom left), `SAT-EXPASC` is at worst half as fast as other search types, and at best 10 to 100 times faster. These trends are confirmed on **JOBSHOP1A** (see Fig. C.9). We therefore select `SAT-EXPASC` as the optimal variant.

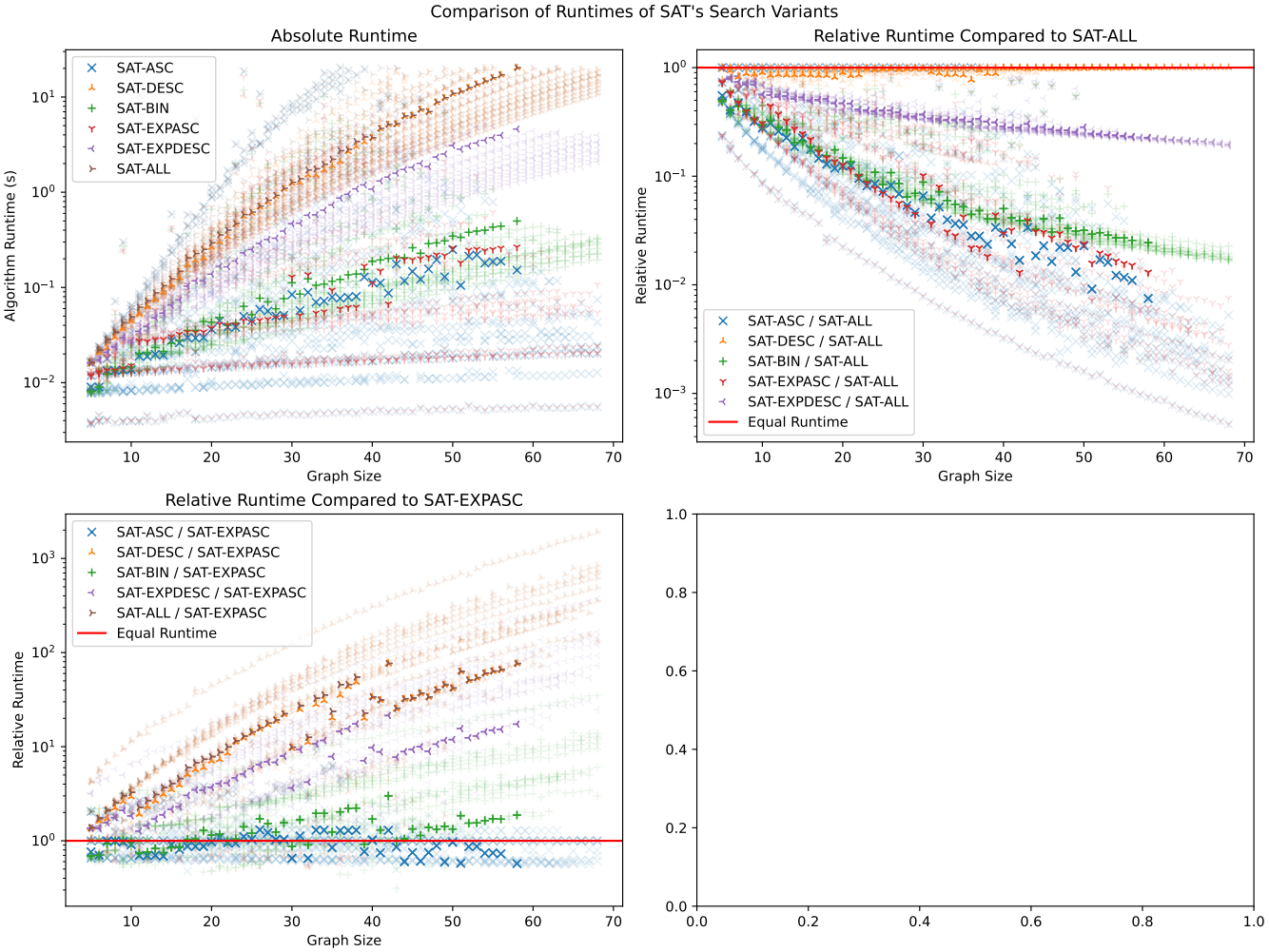


Fig. 4.13: The runtimes of `SAT` with the search variants `SAT-ASC`, `SAT-DESC`, `SAT-BIN`, `SAT-EXPASC`, `SAT-EXPDESC`, `SAT-ALL` on the benchmark set **A25**. Execution times out if `SAT-ALL` runtime exceeds 20s on a graph. Evaluation is stopped if `SAT-ALL` times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity.

Transitive Variants We additionally test the effect of coloring the transitive variants of the input graph compared to the input graph itself. We show in Fig. 4.14 the runtime of `SAT-EXPASC` on the original input graph, the graph's transitive reduction, and its transitive closure. Coloring the transitive closure results in a consistent increase in runtime compared to the original graph. Coloring the transitive reduction has a negligible effect on the median runtime, but results in a significant decrease in runtime on many graphs, with very few outliers where it results in an increase in runtime. This trend is neither confirmed nor contradicted on **JOBSHOP1A** (see Fig. C.10). We select the transitive reduction as the optimal graph variant for `SAT-EXPASC`.

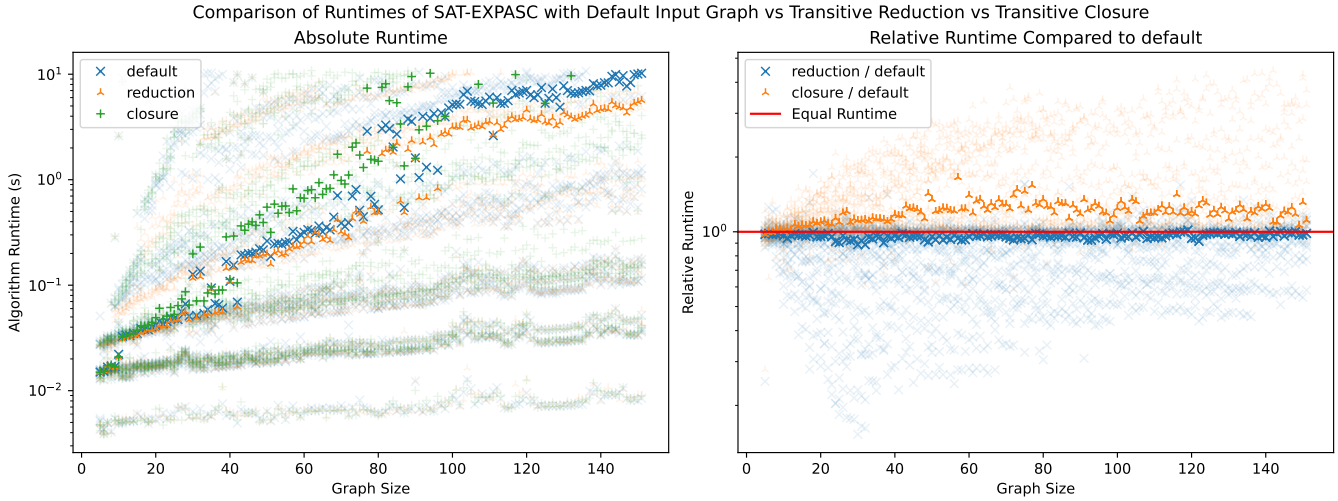


Fig. 4.14: The runtime of `SAT-EXPASC` on the benchmark set **A25**. Execution has a timeout of 10s per graph per variant. Evaluation of a variant is stopped if it times out on more than 70% of the 25 most recent graphs. The median is at full opacity, other values at reduced opacity.

4.3 Comparison between Algorithms

Having found optimal configurations of each algorithm in the previous section, we now compare performance between them. We begin by evaluating all algorithms on benchmark set **A100** to compare their runtime behavior depending on their size. We find that analysis purely by size is not sufficient, as some algorithms show significantly different behavior on **JOBSHOP1A**. This motivates analysis of algorithm runtime depending on graph properties other than size, which we perform using the benchmark sets **E2500S20** through **E2500S150**.

In this section, when we refer to an algorithm, we mean the algorithm's optimal configuration, e.g. `Polyspace` refers to `Polyspace-sMISmod` with $\alpha = 0.001$, run on the transitive reduction of the input graph.

4.3.1 Comparison by Graph Size

We compare the performance of algorithms by graph size on the benchmark set **A100**, accepting a longer evaluation time to obtain a better resolution for this important experiment. We additionally do not cut off algorithm evaluations based on their number of timeouts so that trends are visible as clearly as possible. The results of this are presented in Fig. 4.15. We calculate the median of relative runtimes as the quotient of medians here to accurately display the course of runtimes.

It is clear that, in most cases, `ILP` has a faster runtime than the other algorithms. It seems natural to use `ILP` as a default choice, and to conduct further experiments from a perspective of searching for cases in which other algorithms are preferable to `ILP`. Notably, for small graph sizes (up to 25), `Lawler` is faster than `ILP`. While runtime is

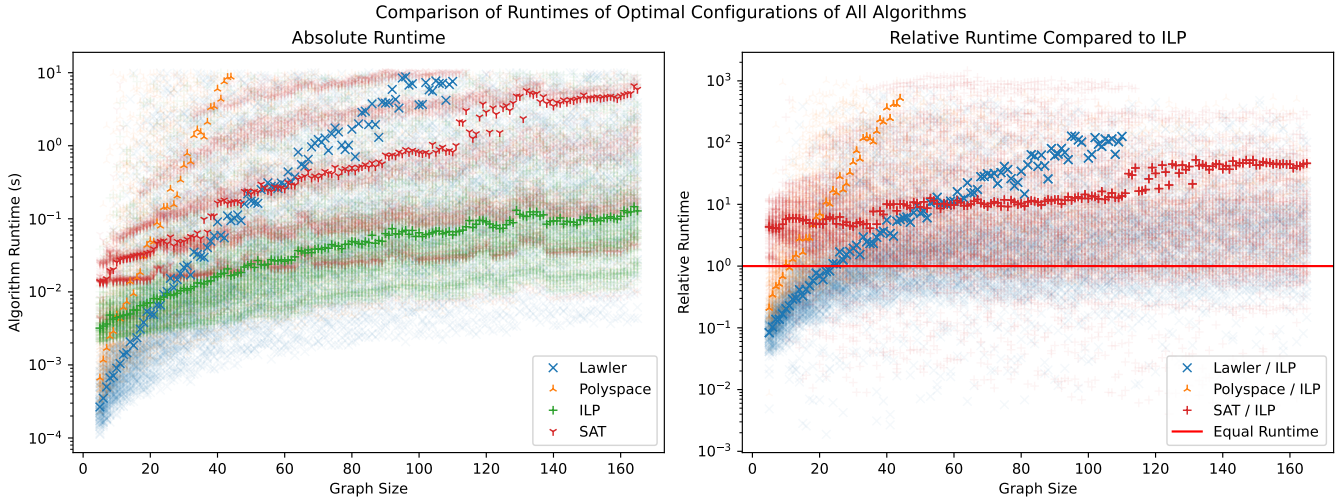


Fig. 4.15: The runtime of the optimal configurations of `Lawler`, `Polyspace`, `ILP`, and `SAT` on the benchmark set **A100**. Evaluation of all algorithms continues up to graph size 165. The median is at full opacity, other values at reduced opacity.

typically a concern for large graphs, in a scenario where many small graph instances must be colored it would be preferable to use `Lawler` over `ILP`.

Additionally it is worth noting that, while `ILP` is faster on most graphs, there is a substantial number of graphs across all sizes where `Lawler` or `SAT` are up to five times faster, with a more sparse band of graphs showing as much as a hundredfold improvement in runtime compared to `ILP`.

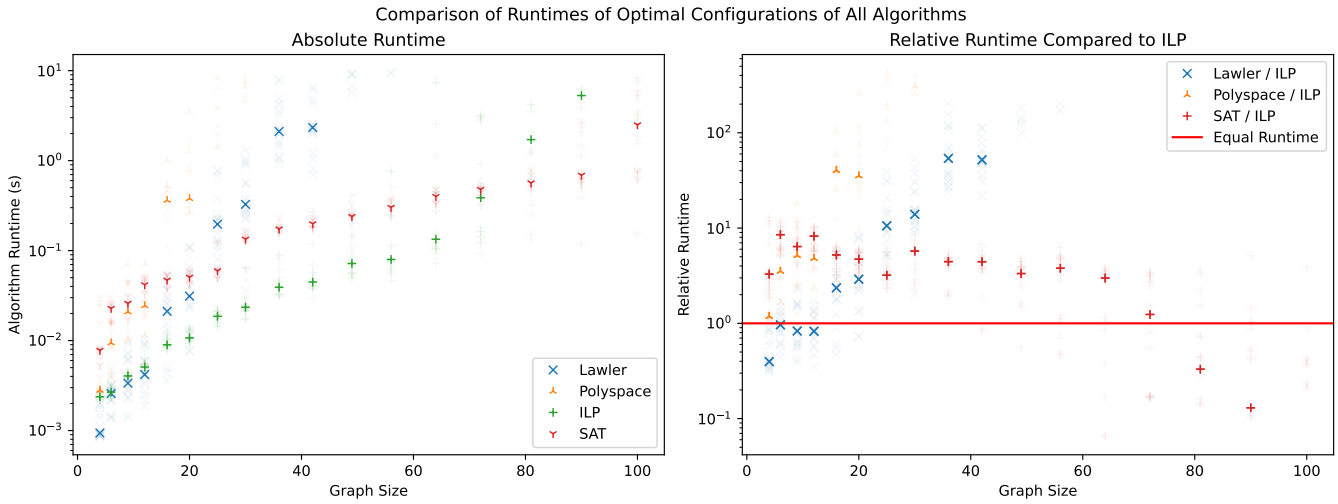


Fig. 4.16: The runtime of the optimal configurations of `Lawler`, `Polyspace`, `ILP`, and `SAT` on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per variant. The median is at full opacity, other values at reduced opacity.

We show in Fig. 4.16 the results of the same experiment performed on the dataset **JOBSHOP1A**. The algorithms `Lawler`, `Polyspace`, and `ILP` show similar relative behavior as on **A100**, though with increased absolute runtime. Notably, `Lawler` becomes slower than `ILP` at a much smaller graph size compared to Fig. 4.15. However, the behavior of `SAT` is drastically different. Rather than continuously increasing in runtime compared to `ILP` as on **A100**, `SAT` overtakes `ILP` in performance as graph size increases, achieving a ten-fold reduction in runtime compared to `ILP` at size 100. This makes it clear that algorithms can have significant differences in performance depending on graph properties other than the size, and that we cannot choose any one fastest algorithm for all graphs.

Many graph properties, such as the chromatic number, correlate with graph size. This motivates the next sections, where we analyze large numbers of graphs of the same size to find trends in runtime that are independent of graph size.

4.3.2 Comparison by Connections

	Size n:			
Algorithm	20	50	100	150
`Lawler`	$< 5\%$	30%	50%	55%
`Polyspace`	15%	55%	75%	80%
`ILP`	$< 5\%$	10%	15%	20%
`SAT`	$< 5\%$	5%	20%	40%

Tab. 4.1: The expected number of timeouts with a timeout of 10 seconds of each algorithm on several graph sizes, based on the evaluation results of Section 4.3.1.

We compare the performance of algorithms on graphs on the benchmark sets **E2500S20**, **E2500S50**, **E2500S100**, **E2500S150**. As there is no specific order in which we expect algorithm runtime to increase on graphs of equal size, we do not cut off algorithm evaluations based on their number of timeouts. Based on the previous section's experiment, we expect the numbers of timeouts on these benchmark sets to be as shown in Tab. 4.1. Due to the high number of expected timeouts at large sizes, we use a higher timeout threshold of 20 seconds for all evaluations in this section. The results of these evaluations are presented in Figs. 4.17, 4.18, 4.19. The results for **E2500S150** are available in Fig. C.11 in Appendix C, as the results are very similar to those for **E2500S100**.

`Polyspace` is not included in these figures because, as expected, it times out a lot and performs worse than the other algorithms. For these reasons we exclude it from further analysis. Its runtime on **E2500S20** and **E2500S50** is shown in Fig. C.12, and it has not been run on **E2500S100** and **E2500S150**. Additionally, the reader may find Figs. C.13 through C.16, which show the runtime of `ILP` relative to the runtimes of `Lawler` and `SAT`, useful as a complementary perspective to the figures presented in this section.

We observe the following patterns:

- `Lawler`'s runtime increases significantly (spanning four orders of magnitude) as the proportion of edges increases. This effect is present regardless of the graph size

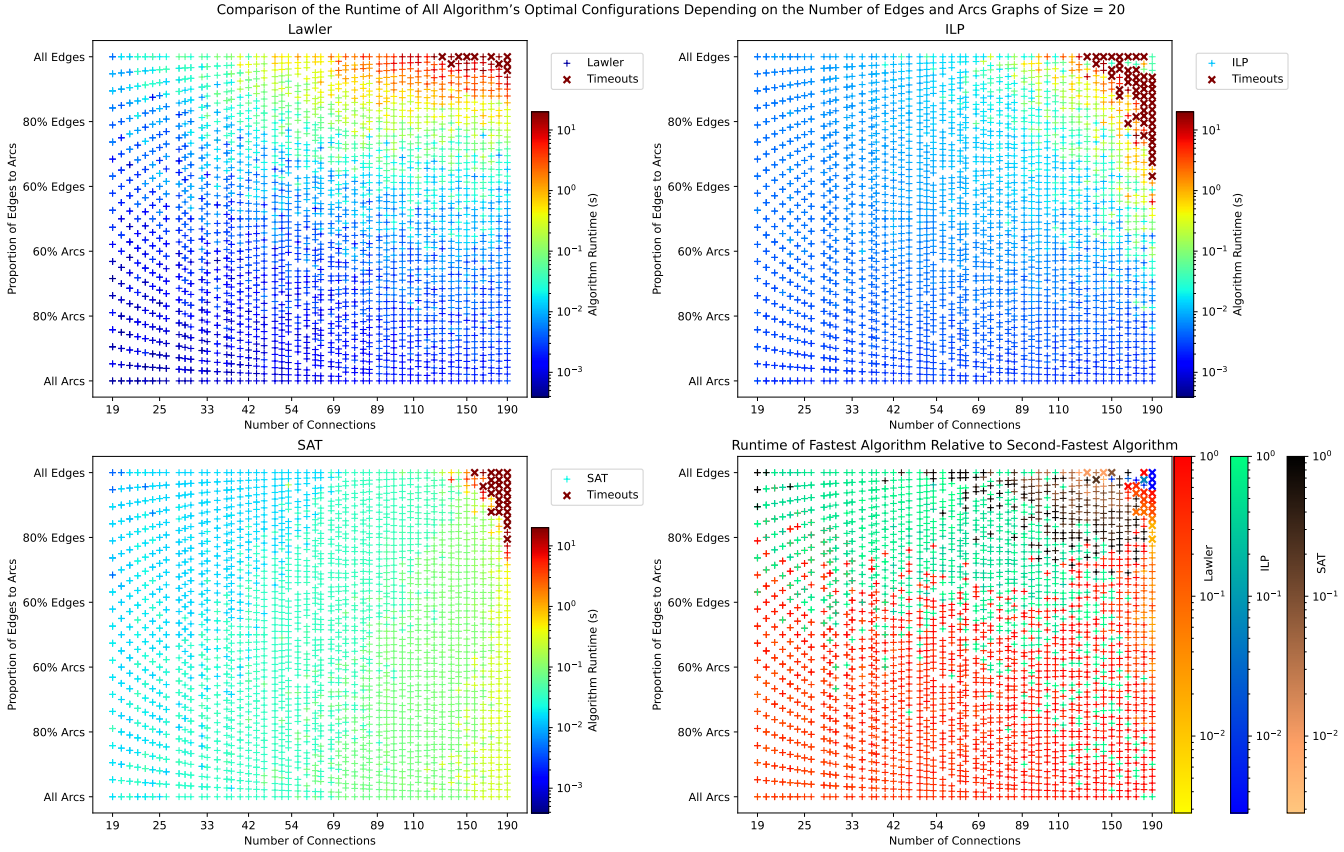


Fig. 4.17: The runtimes of `Lawler`, `ILP`, and `SAT` on the benchmark set **E2500S20**. Top, Bottom Left: Absolute runtime of algorithms depending on edges and arcs. Bottom Right: Relative runtime of the fastest algorithm on each graph relative to the second-fastest. If both other algorithms timed out, the relative runtime is calculated relative to the timeout threshold of 20s. Such timeout-markers are shown as a bold x instead of a +.

and the number of connections. For small graphs, it is most pronounced at the maximal number of connections, while on large graphs it is most pronounced at the minimal number of connections. For a large proportion of arcs ($> 80\%$), `Lawler` consistently performs well, though at a larger total number of connections `ILP` pulls ahead. This behavior is explained by `Lawler`'s subroutine MIS° being limited to vertices with indegree 0. On a graph with many arcs, each execution of MIS° only considers a small subgraph, leading to more executions, but significantly reducing the amount of branching within each execution. As the number of branches is exponential in the size of subgraphs, while the number of executions is inversely proportional to the size of subgraphs, this effect lowers runtime significantly for large proportions of arcs.

- `Lawler` has, for the same proportion of edges to arcs, similar performance re-

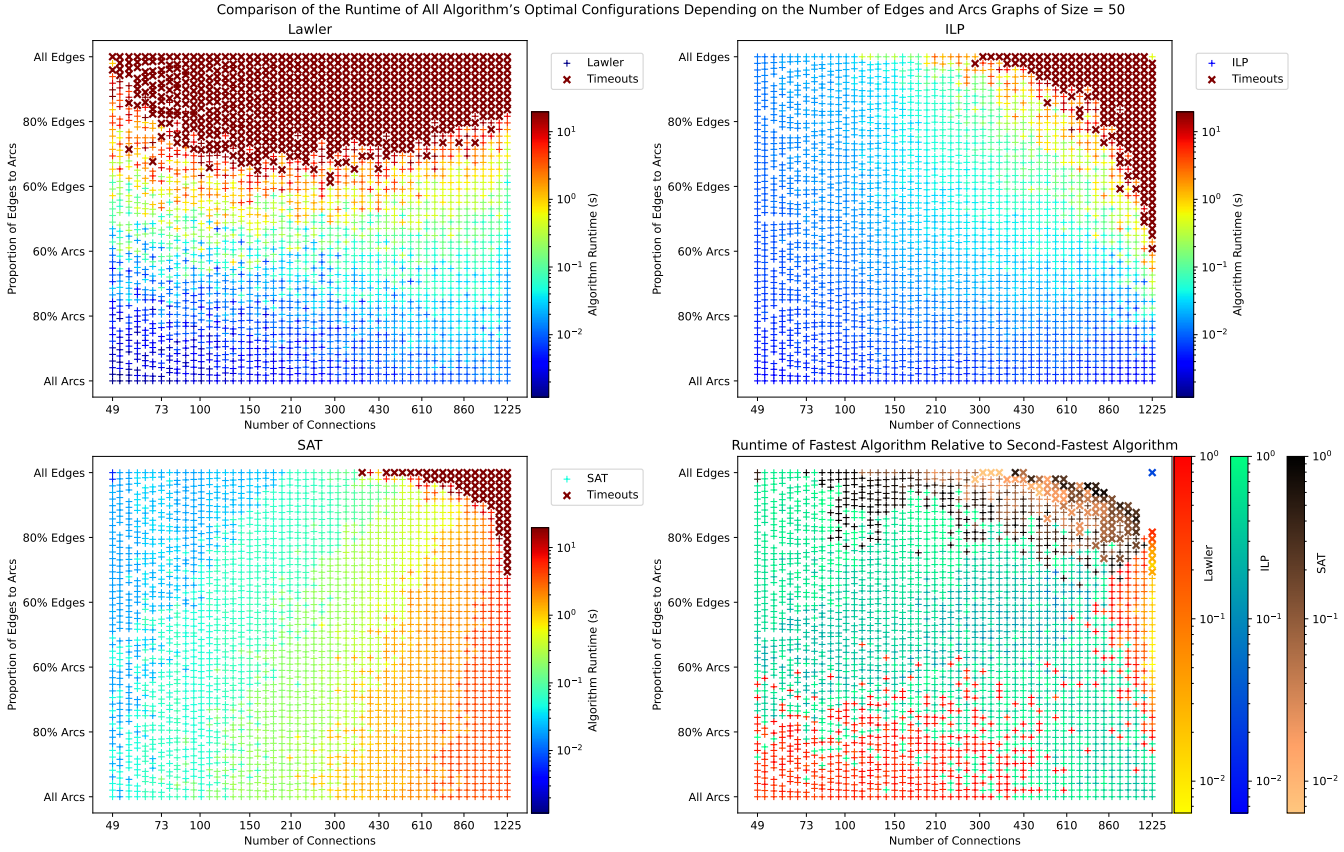


Fig. 4.18: The runtimes of `Lawler`, `ILP`, and `SAT` on the benchmark set **E2500S50**. Top, Bottom Left: Absolute runtime of algorithms depending on edges and arcs. Bottom Right: Relative runtime of the fastest algorithm on each graph relative to the second-fastest. If both other algorithms timed out, the relative runtime is calculated relative to the timeout threshold of 20s. Such timeout-markers are shown as a bold x instead of a +.

gardless of the total number of connections. In contrast, on graphs with a large proportion of edges, both `ILP`'s and `SAT`'s performance decreases as the number of total connections increases, though the effect is stronger for `ILP`. We conclude that this is an effect of applying a general solver to MixedColoring, as general solvers do not necessarily have optimizations specific to graphs.

For example, two colorings may have the same number of colors but a different assignment of colors to vertices. `Lawler` categorically eliminates all such duplicate colorings as a consequence of representing partial colorings as sets of vertices without color-data, while ILP solvers are known [MDZ06] to struggle with them. Most duplicate colorings result from edges, as arcs limit the solution space much more than edges.

Note that `ILP` recognizes specifically the complete undirected graph much faster

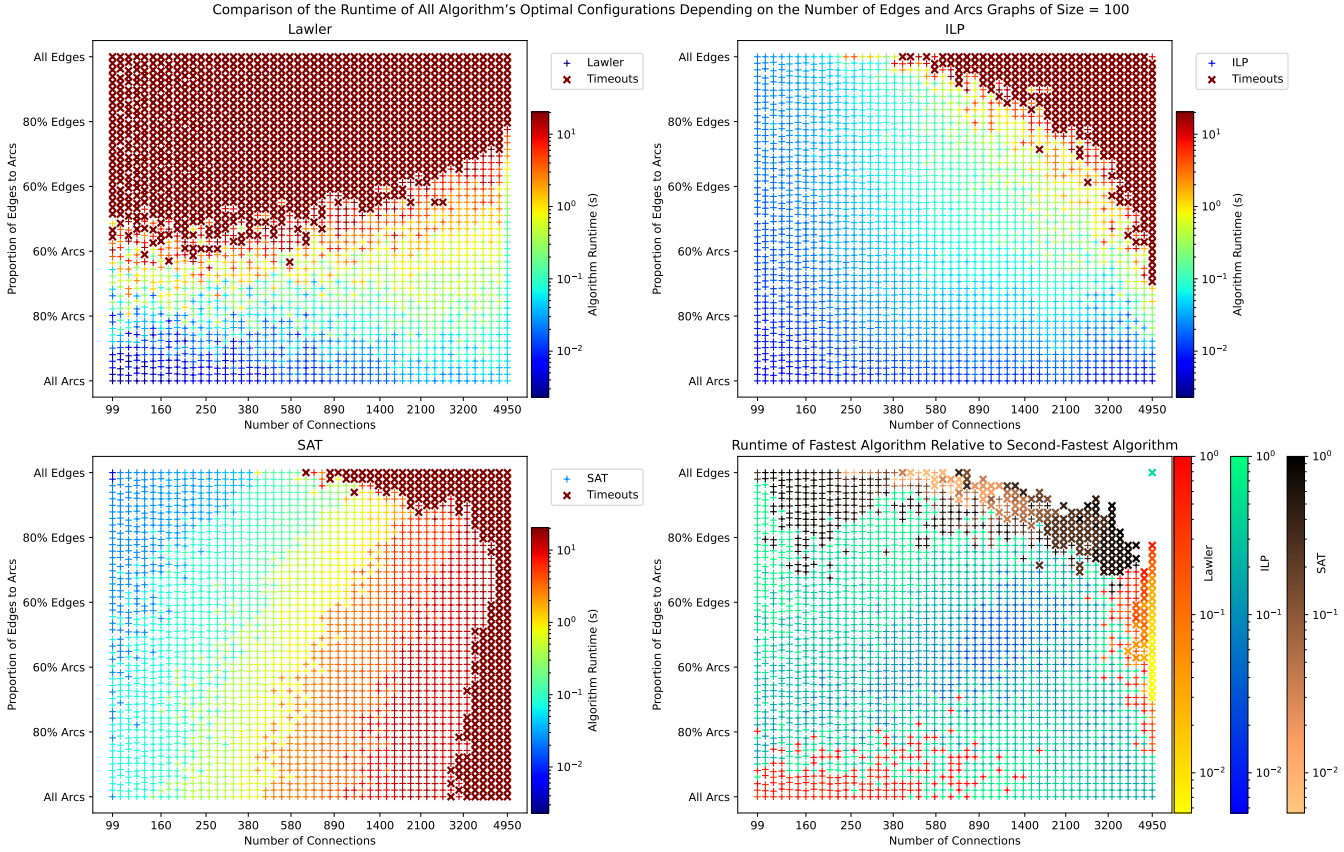


Fig. 4.19: The runtimes of `Lawler`, `ILP`, and `SAT` on the benchmark set **E2500S100**. Top, Bottom Left: Absolute runtime of algorithms depending on edges and arcs. Bottom Right: Relative runtime of the fastest algorithm on each graph relative to the second-fastest. If both other algorithms timed out, the relative runtime is calculated relative to the timeout threshold of 20s. Such timeout-markers are shown as a bold x instead of a +.

than other algorithms, so it is likely that Gurobi contains an optimization aimed at constraints that behave like cliques in a graph. On small graphs, it additionally recognizes graphs that are close to the complete undirected graph, though only those that have at least some arcs.

- `SAT`, unlike `ILP`, performs poorly with a large number of arcs, which becomes more evident on larger graph sizes. This behavior is explained by the formulation of arcs in `SAT`, which is, for each arc, quadratic in the number of potential colors and therefore quadratic in graph size. Because the maximal number of arcs in a graph is quadratic in graph size as well, the number of `SAT` constraints required by arcs is quartic in graph size for graphs with a large proportion of arcs and many connections.

  However, on graphs that do not have close to the maximum number of total con-

nections, SAT is the only algorithm that performs better on graphs that have a larger proportion of edges. This is the reason for the contradictory result obtained in the previous section on **JOBSHOP1A**. Instances of size 100 of this benchmark set have 90 arcs and 450 edges, placing them in the region of Fig. 4.19 where SAT has the best performance compared to Lawler and ILP.

While some trends are consistent across all sizes, there are notable differences between the runtime behavior on graphs of size 20 compared to the other tested sizes. We therefore summarize the results for graphs of size 20 separately from the other results:

Small Graphs For a majority of the graphs of size 20, Lawler is the fastest algorithm. This is consistent with results from the previous section, where the median of Lawler's runtime is below that of all other algorithm's medians at size 20, and is likely a result of the model set-up time of ILP and SAT. However, Lawler is slower on graphs with many edges, which is more pronounced the more total connections a graph has. Above a proportion of 60% edges, we observe ILP outperforming Lawler. On graphs that have more than 60% edges and more than 50% of total possible connections, SAT outperforms both ILP and Lawler. Lastly, if fewer than 10% of a graphs connections differ from those of the complete undirected graph, ILP is the fastest algorithm.

Medium and Large Graphs On graphs of size 50 and larger, ILP is the fastest algorithm in most cases. However, there are graphs on which ILP performs poorly while other algorithms perform well. SAT generally performs equally or better than ILP for graphs with more than 80% of their connections being edges, with negligible effect at a low number of connections and up to a hundred-fold speedup at a large number of connections. On graphs that have close to the maximal number of connections, with more than 20% of them being edges, Lawler significantly outperforms ILP. Additionally Lawler has a negligible but consistent performance advantage on graphs with few connection of which at least 80% are arcs, though we expect this effect to disappear on sufficiently large graphs as it lessens with increasing graph size.

4.3.3 Comparison by Chromatic Number

We reuse the previous section's evaluation results to analyze the behavior of Lawler, ILP, and SAT depending on the graph's chromatic number. Note that the chromatic number of a graph is correlated with the number of connections, so this analysis is not independent of the previous section's analysis. To demonstrate this dependency, we show in Fig. 4.20 the chromatic number of the graphs in the benchmark sets **E2500S20**, **E2500S50**, **E2500S100**, and **E2500S150**. If all algorithms timed out on a graph, then the chromatic number is unknown. Note that, since there are unknown chromatic numbers, we do not know how many timeouts happened for each chromatic number. This prevents us from using the median to eliminate the ceiling effect caused by timeouts, so we do not have a way of visualizing the behavior of runtime depending on chromatic

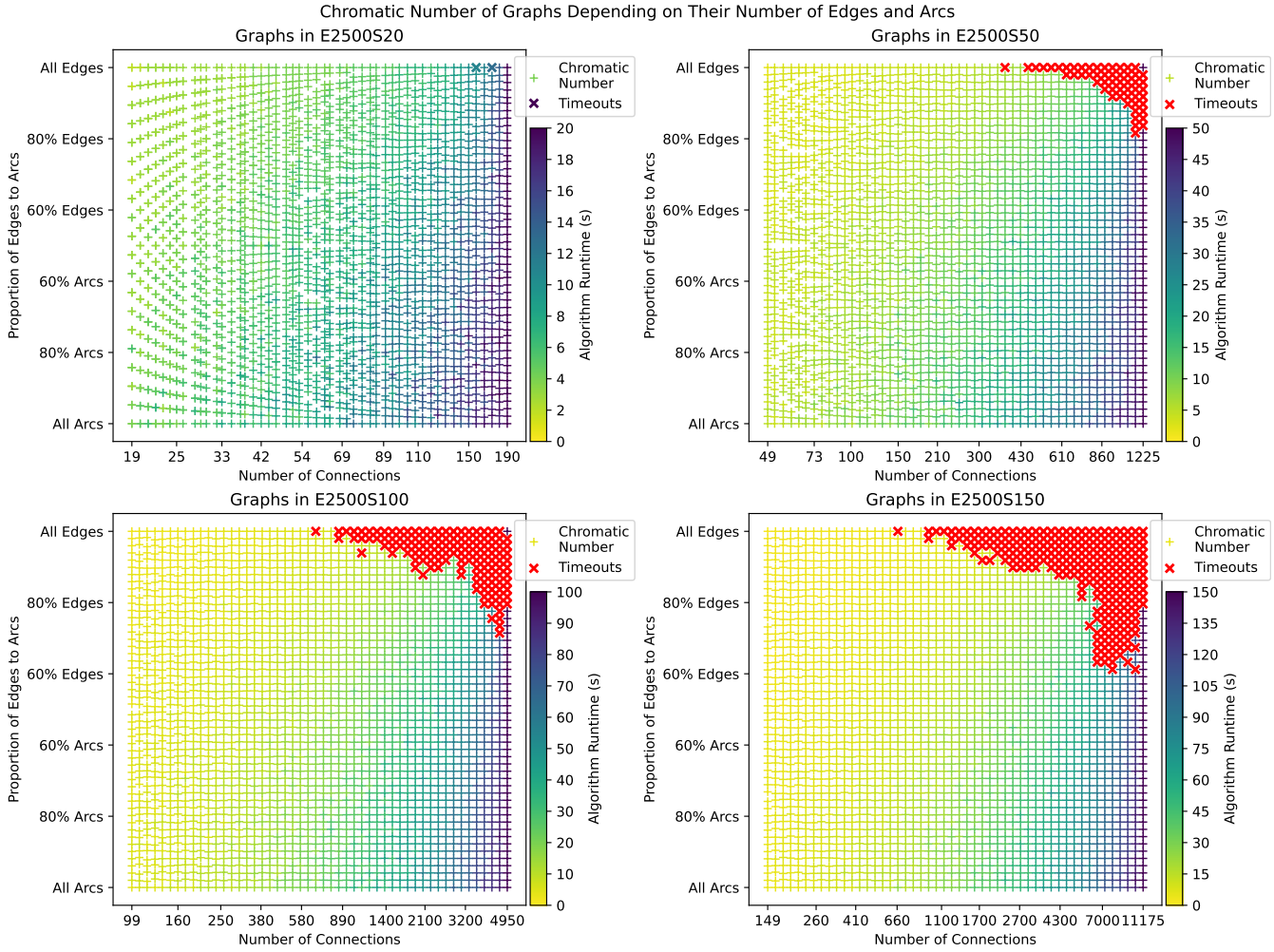


Fig. 4.20: The chromatic number of the graphs in the benchmark sets **E2500S20**, **E2500S50**, **E2500S100**, and **E2500S150**, depending on their number of edges and arcs. X-markers indicate that all algorithms timed out. Red timeouts have an unknown chromatic number. For non-red timeouts, we retroactively calculated the chromatic number.

number that is fully representative of reality. For size 20, we retroactively calculate the missing chromatic numbers, meaning we can display the median here. We are not able to do so for larger sizes due to time constraints[1].

However, we can leverage the correlation between connections and the chromatic number to justify analysis on plots without ceiling-effect correction. We can check that our results are, through Fig. 4.20, consistent with results from the previous section. Additionally, although strong correlation between two properties of test samples would

[1]For size 50, `SAT-EXPASC` did not terminate within 10 hours on the fifth missing graph (out of 55). We aborted evaluation at this point.

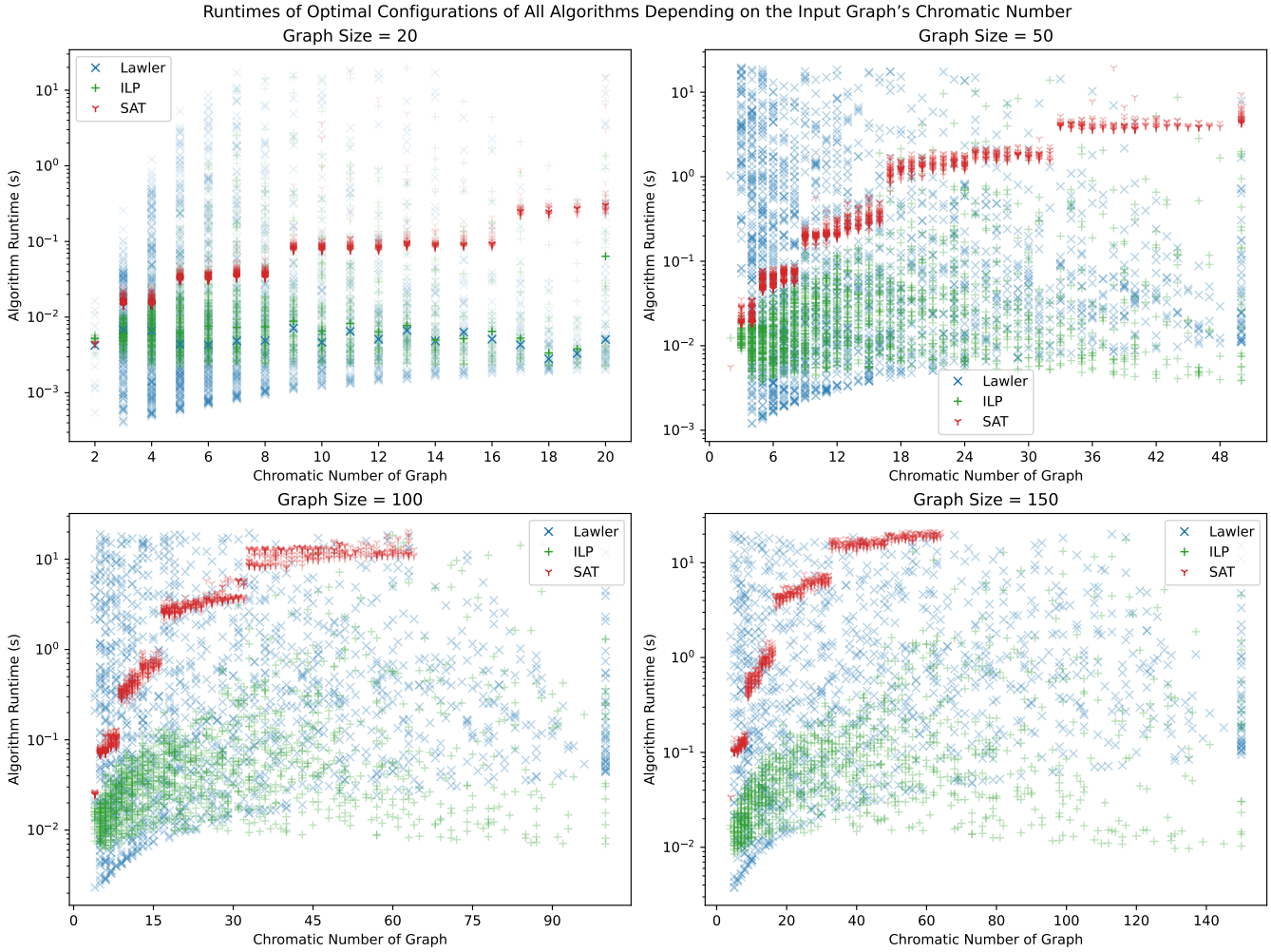


Fig. 4.21: The runtimes of `Lawler`, `ILP`, and `SAT` on the benchmark sets **E2500S20**, **E2500S50**, **E2500S100**, and **E2500S150**, grouped by the graph's chromatic number with a timeout threshold of 20s. For graph size 20, the median is shown at full opacity while other data points are shown at low opacity.

usually render independent analysis of those properties obsolete, this is not the case here. Firstly, most graphs encountered in real-world scenarios are likely to share this correlation, unless specifically constructed not to. Secondly, although the properties are correlated, phrasing runtime behavior in terms of chromatic number may be simpler, and can therefore lead to more intuitive understanding and to a more compact heuristic in Section 4.3.4. For these reasons, we now analyze the behavior of algorithm runtime depending on the chromatic number of a graph.

We show in Fig. 4.21 the runtime of `Lawler`, `ILP`, and `SAT` on the benchmark sets **E2500S20**, **E2500S50**, **E2500S100**, and **E2500S150**, grouped by the graph's chromatic number. We observe that the runtime of `SAT` increases significantly for larger

chromatic numbers. For large graphs, SAT has timeouts on all graphs with a chromatic number above approximately 65. This is consistent with the previous section's observation, where the runtime of SAT increases with a larger number of connections. Additionally, the course of the runtime is sectioned into plateaus, ranges of chromatic numbers where SAT has similar runtime, that are inter-spaced by sharp jumps. This is a result of SAT-EXPASC using exponential search. The plateaus are ranges between two steps of the initial exponential search. If the chromatic number of a graph exceeds this range, then exponential search takes another step, which doubles the range that the subsequent binary search must cover and increases the SAT formula size of each test. These plateaus can also be observed in the previous section's Figs. 4.17-4.19, which show distinct jumps in the runtime of SAT whereas Lawler and ILP transition more smoothly between runtimes. Note that this does not mean that these jumps are the cause of SAT's poor performance at high chromatic numbers. SAT-EXPASC has, at worst, double the runtime of other variants of SAT (see Fig. 4.13), so while other variants may have different runtime behavior, they would not perform much better than SAT-EXPASC, even at high chromatic numbers.

Lawler does not show any difference in median runtime depending on the chromatic number, but shows significant variance in runtime. Additionally, the minimum runtime slightly increases for higher chromatic numbers. This is consistent with the previous section's results, where Lawler's runtime did not vary much depending on the number of connections but varied significantly within the same number of connections, depending on the proportion of edges to arcs. Additionally, at a high number of connections the minimum runtime increases slightly.

ILP has low variance in runtime at low chromatic numbers and high variance in runtime at large chromatic numbers. This is consistent with the previous section's results, where ILP consistently performed well on graphs with a low number of connections, but for a high number of connections performance ranged across four orders of magnitude.

The reader may additionally note, that the density of data points falls off towards larger chromatic numbers and then increases for the maximal chromatic number. This is a consequence of the distribution of the number of connections in graphs (see Section 4.1.3). In the group of graphs with the maximum number of connections, every graph has a chromatic number equal to its size. In contrast, a group of graphs with fewer connections is distributed across a range of chromatic numbers, and additionally, there are fewer groups at high numbers of connections compared to low numbers of connections due to logarithmic distribution.

4.3.4 Heuristic for Choosing an Algorithm

In this section we summarize the results of comparing algorithm runtimes into a compact heuristic for choosing a MixedColoring algorithm depending on the graph. Note that this heuristic neglects some regions of the above plots where choosing a different algorithm would provide a consistent but negligible decrease in runtime. We deliberately choose to neglect these cases to avoid bloating the heuristic with rules that provide little benefit. The reader is encouraged to compile a more fitting heuristic for their application

from our results, if necessary.

On general mixed graphs, if no properties of the graphs are considered, `ILP` has the best median performance and should be the default choice. On small graphs (up to size 25), `Lawler` should be the default choice. Rules in our heuristic are listed in order of increasing importance, i.e. if two rules apply, the latter takes precedence.

Heuristic for Small Graphs (Up to Size 25) Choose `Lawler` by default. If the graph

- has more edges than arcs, and at most 50% of possible connections: choose `ILP`.
- has more edges than arcs, and more than 50% of possible connections: choose `SAT`.
- has almost 100% edges and almost 100% of possible connections: choose `ILP`.

Heuristic for Large Graphs (Above Size 25) Choose `ILP` by default. If the graph

- has more than 80% edges: choose `SAT`, unless there is some lower bound on the chromatic number that restricts it to more than half the graph's size.
- has more than 20% edges and more than 90% connections: choose `Lawler`
- has almost 100% edges and almost 100% of possible connections: choose `ILP`.

5 Conclusion and Outlook

In this thesis, we have experimentally analyzed the runtime of four MixedColoring algorithms and their different configurations, and created a heuristic for choosing the fastest algorithm based on a graph's properties. In the following, we take a look at open problems and directions for future research.

Other MixedColoring Algorithms Lauerbach mentions in his thesis the possibility of generalizing algorithms [Epp01, Bys04, BHK09] that are asymptotically faster than `Lawler` to MixedColoring. If such generalizations are possible, it would be interesting to see how the generalized algorithms perform experimentally in comparison to `Lawler`.

Additionally, other variants of our existing algorithms could be tested. For example, while we implemented `ILP-BIN` as an assignment model, it would be interesting how models based on partial ordering [JM17] or set covering [HLS09] would perform.

Bounds on the Chromatic Number In the scope of this thesis, we were not able to analyze the effect of bounds on χ on the runtime of algorithms. It would be interesting to see how much known bounds affect the runtime of algorithms. It would additionally be possible to preemptively determine χ and simulate any number of arbitrarily good bounds based on this information. This would make it possible to predict the performance impact of any bounds discovered in the future, or bounds that the author is unaware of.

The effects that bounds on χ would have on each of our algorithms are as follows:

`Lawler` Neither bound would make a difference for `Lawler`, as partial k-colorings are constructed incrementally in ascending order of k, starting at $k = 0$, and stopping at $k = \chi$.

`Polyspace` An upper bound on χ would allow branches of `Polyspace` to be cut when the number of colors exceeds the bound.

`ILP` An upper bound on χ would limit the range of numbers that variables in the integer-formulation of `ILP` can take, which limits the search space. A lower bound would not affect the formulation, but the information that a minimum of colors is necessary may help the solver.

`SAT` Bounds on χ would limit the k for which k-colorability is tested. Note that low k are very fast to test, so a lower bound would likely have little effect, even on ascending search types. An upper bound would improve the performance of descending search types, and, if the bound is good enough, would limit the step effect we observe in Fig. 4.21.

For these reasons we predict that a sufficiently good upper bound would result in notable performance improvements for `Polyspace`, `ILP`, and `SAT`, while a lower bound would have minimal to no effect on all algorithms.

Memory Usage The primary benefit of `Polyspace` is not in its runtime, but in taking only polynomial space, compared to the exponential space requirement of our other algorithms, so it is not surprising that `Polyspace` performed poorly in comparison. While, at cursory observation, there were no notable differences in memory usage during evaluation of different algorithms, it would be interesting to rigorously analyze the memory usage of our algorithms. It is very likely that, depending on hardware limitations, there may be graphs that cannot be solved by other algorithms due to memory constraints, but can be solved in reasonable time by `Polyspace`.

Other Graph Implementations We implemented mixed graphs using inbound, outbound, and undirected adjacency lists. It would be interesting to see if the runtimes of our algorithms are notably different with a different underlying graph implementation, such as an adjacency matrix or a single adjacency list with directionality metadata.

Bibliography

[BB04] Nicolas Barnier and Pascal Brisset: Graph Coloring for Air Traffic Flow Management. Annals of Operations Research, 130(1–4):163–178, August 2004, 10.1023/B:ANOR.0000032574.01332.98.

[BCGM12] Gunnar Brinkmann, Kris Coolsaet, Jan Goedgebeur, and Hadrien Melot: House of Graphs: A database of interesting graphs. 2012, 10.48550/ARXIV.1204.3549. Version Number: 2.

[BHK09] Andreas Björklund, Thore Husfeldt, and Mikko Koivisto: Set Partitioning via Inclusion-Exclusion. SIAM Journal on Computing, 39(2):546–563, January 2009, 10.1137/070683933.

[BHM21] Armin Biere, Marijn Heule, and Hans van Maaren (editors): Handbook of satisfiability. Number 336 in Frontiers in Artificial Intelligence and Applications. IOS Press, second edition, 2021.

[Bys04] Jesper Makholm Byskov: Enumerating maximal independent sets with applications to graph colouring. Operations Research Letters, 32(6):547–556, November 2004, 10.1016/j.orl.2004.03.002.

[CL25] François Clautiaux and Ivana Ljubić: Last fifty years of integer linear programming: A focus on recent practical advances. European Journal of Operational Research, 324(3):707–731, August 2025, 10.1016/j.ejor.2024.11.018.

[CNS21] Tristan Cazenave, Benjamin Negrevergne, and Florian Sikora: Monte Carlo Graph Coloring. In Tristan Cazenave, Olivier Teytaud, and Mark H. M. Winands (editors): Monte Carlo Search, pages 100–115, Cham, 2021. 10.48550/arXiv.2504.03277.

[DERT15] Marc Demange, Tınaz Ekim, Bernard Ries, and Cerasela Tanasescu: On some applications of the selective graph coloring problem. European Journal of Operational Research, 240(2):307–314, January 2015, 10.1016/j.ejor.2014.05.011.

[DMB08] Leonardo De Moura and Nikolaj Bjørner: Z3: An Efficient SMT Solver. In Tools and Algorithms for the Construction and Analysis of Systems, volume 4963, pages 337–340. 2008, 10.1007/978-3-540-78800-3_24. Series Title: Lecture Notes in Computer Science.

[DMB11] Leonardo De Moura and Nikolaj Bjørner: Satisfiability modulo theories: introduction and applications. Communications of the ACM, 54(9):69–77, September 2011, 10.1145/1995376.1995394.

[DWELM99] D. De Werra, Ch. Eisenbeis, S. Lelait, and B. Marmol: On a graph-theoretical model for cyclic register allocation. Discrete Applied Mathematics, 93(2-3):191–203, July 1999, 10.1016/S0166-218X(99)00105-5.

[Epp01] David Eppstein: Small Maximal Independent Sets and Faster Exact Graph Coloring. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Frank Dehne, Jörg Rüdiger Sack, and Roberto Tamassia (editors): Algorithms and Data Structures, volume 2125, pages 462–470. 2001, 10.48550/arXiv.cs/0011009.

[ERT79] Paul Erdös, Arthur L. Rubin, and Herbert Taylor: Choosability in graphs. Volume 26, pages 125–157, Congressus Numerantium, 1979.

[FBHS23] Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider: The Silent (R)evolution of SAT. Communications of the ACM, 66(6):64–72, June 2023, 10.1145/3560469.

[FK10] Fedor V. Fomin and Dieter Kratsch: Exact Exponential Algorithms. Texts in Theoretical Computer Science. An EATCS Series. 2010, 10.1007/978-3-642-16533-7.

[GC26] Stefano Gualandi and Marco Chiarandini: Graph coloring benchmarks., 2026. `https://sites.google.com/site/graphcoloring`, visited on 2026-03-16.

[GMR+23] Grzegorz Gutowski, Florian Mittelstädt, Ignaz Rutter, Joachim Spoerhase, Alexander Wolff, and Johannes Zink: Coloring Mixed and Directional Interval Graphs. Volume 13764, pages 418–431. 2023, 10.1007/978-3-031-22203-0_30.

[Gur26] Gurobi Optimization, LLC: Gurobi Optimizer Reference Manual, 2026. `https://www.gurobi.com`.

[HKD97] Pierre Hansen, Julio Kuplinsky, and Dominique De Werra: Mixed graph colorings. Mathematical Methods of Operations Research, 45(1):145–160, February 1997, 10.1007/BF01194253.

[HLS09] P. Hansen, M. Labbé, and D. Schindl: Set covering and packing formulations of graph coloring: Algorithms and first polyhedral results. Discrete Optimization, 6(2):135–147, May 2009, 10.1016/j.disopt.2008.10.004.

[JM17] Adalat Jabrayilov and Petra Mutzel: New Integer Linear Programming Models for the Vertex Coloring Problem, 2017. 10.48550/ARXIV.1706.10191. Version Number: 2.

[Kar72] Richard M. Karp: Reducibility among Combinatorial Problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger (editors): Complexity of Computer Computations, pages 85–103. Springer US, 1972, 10.1007/978-3-540-68279-0_8.

[KH21] Ahmed Kouider and Hacène Ait Haddadène: A bi-objective branch-and-bound algorithm for the unit-time job shop scheduling : A mixed graph coloring approach. Computers & Operations Research, 132:105319, August 2021, 10.1016/j.cor.2021.105319.

[KHOO15] Ahmed Kouider, Hacene Ait Haddadène, Samia Ourari, and Ammar Oulamara: Mixed integer linear programs and tabu search approach to solve mixed graph coloring for unit-time job shop scheduling. In 2015 IEEE International Conference on Automation Science and Engineering (CASE), pages 1177–1181. IEEE, August 2015, 10.1109/CoASE.2015.7294257.

[Lau25] Antonio Lauerbach: Coloring Mixed Graphs. Master's thesis, Julius-Maximilians-Universität, Würzburg, September 2025. `https://www1.pub.informatik.uni-wuerzburg.de/pub/theses/2025-lauerbach-masterarbeit.pdf`.

[Law76] Eugene L. Lawler: A note on the complexity of the chromatic number problem. Information Processing Letters, 5(3):66–67, August 1976, 10.1016/0020-0190(76)90065-X.

[LM07] Jon Lee and François Margot: On a Binary-Encoded ILP Coloring Formulation. INFORMS Journal on Computing, 19(3):406–415, August 2007, 10.1287/ijoc.1060.0178.

[LOF18] Arno Luppold, Dominic Oehlert, and Heiko Falk: Evaluating the performance of solvers for integer-linear programming. Technical report, TUHH Universitätsbibliothek, 2018, 10.15480/882.1839.

[MDZ06] Isabel Méndez-Díaz and Paula Zabala: A Branch-and-Cut algorithm for graph coloring. Discrete Applied Mathematics, 154(5):826–847, 2006, 10.1016/j.dam.2005.05.022.

[MS10] Vinod Maan and Dhiraj Sangwan: A Genetic Implementation of Graph coloring for Cellular network. International Journal of Electronics Engineering, 2:page 183–187, June 2010.

[MT10] Enrico Malaguti and Paolo Toth: A survey on vertex coloring problems. International Transactions in Operational Research, 17(1):1–34, January 2010, 10.1111/j.1475-3995.2009.00696.x.

[MV] Dirk C. Mattfeld and Rob J.M. Vaessens: jobshop1. `http://www.brunel.ac.uk/depts/ma/research/jeb/orlib/files/jobshop1.txt`, visited on 2005-01-22.

[RBA18] Sabino Francesco Roselli, Kristofer Bengtsson, and Knut Akesson: SMT Solvers for Job-Shop Scheduling Problems: Models Comparison and Performance Evaluation. In 2018 IEEE 14th International Conference on Automation Science and Engineering (CASE), pages 547–552, Munich, Germany, August 2018. IEEE, 10.1109/COASE.2018.8560344.

[SDA00] Yuri N. Sotskov, Alexandre B. Dolgu, and Georgi V. Andreev: Unit-Time Job-Shop Scheduling by Algorithm for Mixed Graph Coloring (Invited paper). IFAC Proceedings Volumes, 33(17):681–686, July 2000, 10.1016/S1474-6670(17)39485-5.

[SDW01] Yuri Sotskov, Alexandre Dolgui, and Frank Werner: Mixed Graph Coloring for Unit-Time Job-Shop Scheduling. International Journal of Mathematical Algorithms, 2:289–323, January 2001.

[Sot20] Yuri N. Sotskov: Mixed Graph Colorings: A Historical Review. Mathematics, 8(3):385, March 2020, 10.3390/math8030385.

[ST76] Yuri N. Sotskov and Vjacheslav Tanaev: Chromatic polynomial of a mixed graph. Vestsi Akademii Navuk BSSR, Ser. Fiz. Mat. Navuk, 6:20–23, 1976.

[WCP+20] Yiyuan Wang, Shaowei Cai, Shiwei Pan, Ximing Li, and Monghao Yin: Reduction and Local Search for Weighted Graph Coloring Problem. Proceedings of the AAAI Conference on Artificial Intelligence, 34(03):2433–2441, April 2020, 10.1609/aaai.v34i03.5624.

A Appendix: Plot-Data Association Table

Plot	Data File(s)
Fig. 4.2	lawler-MIS-comparison-random_logarithmically_distributed_26-02-28T092244.csv
Fig. 4.3	lawler-transitive-random_logarithmically_distributed_26-03-09T092030.csv
Fig. 4.4	polyspace-alpha-comparison-random_logarithmically_distributed_26-03-01T101547.csv
Fig. 4.5	polyspace-alpha-comparison-random_logarithmically_distributed_26-03-01T101547.csv
Fig. 4.6	polyspace-alpha-comparison-random_logarithmically_distributed_26-03-01T101547.csv
Fig. 4.7	polyspace-mis-comparison-random_logarithmically_distributed_26-03-02T151926.csv
Fig. 4.8	polyspace-transitive-random_logarithmically_distributed_26-03-08T110155.csv
Fig. 4.9	MILP-vartypes-random_logarithmically_distributed_26-02-25T192729.csv
Fig. 4.10	MILP-vartypes-random_logarithmically_distributed_26-02-25T192729.csv
Fig. 4.11	MILP-transitive-random_logarithmically_distributed_26-03-07T104719.csv
Fig. 4.12	SAT-allK-random_logarithmically_distributed_26-03-04T135011.csv
Fig. 4.13	SAT-allK-random_logarithmically_distributed_26-03-04T135011.csv
Fig. 4.14	SAT-transitive-random_logarithmically_distributed_26-03-06T090006.csv
Fig. 4.15	optimal-configurations-ascending-random_logarithmically_distributed_26-03-11T103138.csv
Fig. 4.16	optimal-configurations-ascending-jobshop1_26-03-20T221950.csv
Fig. 4.17	optimal-configurations-equalsize20-random_logarithmically_distributed_26-03-19T110535.csv
Fig. 4.18	optimal-configurations-equalsize50-random_logarithmically_distributed_26-03-19T140900.csv
Fig. 4.19	optimal3-configurations-equalsize100-random_logarithmically_distributed_26-03-15T083809.csv optimal-configurations-equalsize20-random_logarithmically_distributed_26-03-19T110535.csv optimal-configurations-equalsize50-random_logarithmically_distributed_26-03-19T140900.csv
Fig. 4.20	optimal3-configurations-equalsize100-random_logarithmically_distributed_26-03-15T083809.csv optimal3-configurations-equalsize150-random_logarithmically_distributed_26-03-18T140348.csv missing_chi_graphs_equal2500_s20_26-03-24T105607.csv
Fig. 4.21	optimal-configurations-equalsize20-random_logarithmically_distributed_26-03-19T110535.csv optimal-configurations-equalsize50-random_logarithmically_distributed_26-03-19T140900.csv optimal3-configurations-equalsize100-random_logarithmically_distributed_26-03-15T083809.csv optimal3-configurations-equalsize150-random_logarithmically_distributed_26-03-18T140348.csv
Fig. C.1	lawler-MIS-comparison-jobshop1_26-03-21T003632.csv
Fig. C.2	lawler-transitive-comparison-jobshop1_26-03-21T012641.csv
Fig. C.3	polyspace-alpha-comparison-comparison-jobshop1_26-03-21T023445.csv
Fig. C.4	polyspace-alpha-comparison-comparison-jobshop1_26-03-21T023445.csv
Fig. C.5	polyspace-mis-comparison-comparison-jobshop1_26-03-21T151937.csv
Fig. C.6	polyspace-transitive-comparison-jobshop1_26-03-21T180220.csv
Fig. C.7	MILP-vartypes-jobshop1_26-03-21T193805.csv
Fig. C.8	MILP-transitive-jobshop1_26-03-21T200134.csv
Fig. C.9	SAT-allK-jobshop1_26-03-21T201906.csv
Fig. C.10	SAT-transitive-jobshop1_26-03-21T204209.csv
Fig. C.11	optimal3-configurations-equalsize150-random_logarithmically_distributed_26-03-18T140348.csv
Fig. C.12	optimal-configurations-equalsize20-random_logarithmically_distributed_26-03-19T110535.csv optimal-configurations-equalsize50-random_logarithmically_distributed_26-03-19T140900.csv
Fig. C.13	optimal-configurations-equalsize20-random_logarithmically_distributed_26-03-19T110535.csv
Fig. C.14	optimal-configurations-equalsize50-random_logarithmically_distributed_26-03-19T140900.csv
Fig. C.15	optimal3-configurations-equalsize100-random_logarithmically_distributed_26-03-15T083809.csv
Fig. C.16	optimal3-configurations-equalsize150-random_logarithmically_distributed_26-03-18T140348.csv

Tab. A.1: A table that associates every plot in this thesis with the file containing the underlying data of the plot. These files are located in the folder `evaluation_output` in our gitlab repository.

B Appendix: Details of Benchmark Set Creation

Benchmark Set	Function Call
A25	graph_list_random_evenly_distributed(n_{min} 5, n_{max} 300, groups: 5, members: 5, seed: 2092426581)
A100	graph_list_random_evenly_distributed(n_{min} 5, n_{max} 300, groups: 10, members: 10, seed: 3854542806)
E2500S20	graph_list_random_evenly_distributed(n_{min} 20, n_{max} 20, groups: 50, members: 50, seed: 672354896)
E2500S50	graph_list_random_evenly_distributed(n_{min} 50, n_{max} 50, groups: 50, members: 50, seed: 1125016478)
E2500S100	graph_list_random_evenly_distributed(n_{min} 100, n_{max} 100, groups: 50, members: 50, seed: 1012347857)
E2500S150	graph_list_random_evenly_distributed(n_{min} 150, n_{max} 150, groups: 50, members: 50, seed: 2356789678)
JOBSHOP1A	graph_list_jobshop_ascending(file: "jobshop_benchmark_instances/jobshop1.txt")

Tab. B.1: A table that associates all benchmark sets used throughout this thesis with the function and arguments that we use to create it in our implementation. All such functions are located in the file `evaluation.cpp` in our gitlab repository. We pass a seed to the random number generator to ensure that the same graphs are generated every time.

C Appendix: Auxiliary Plots

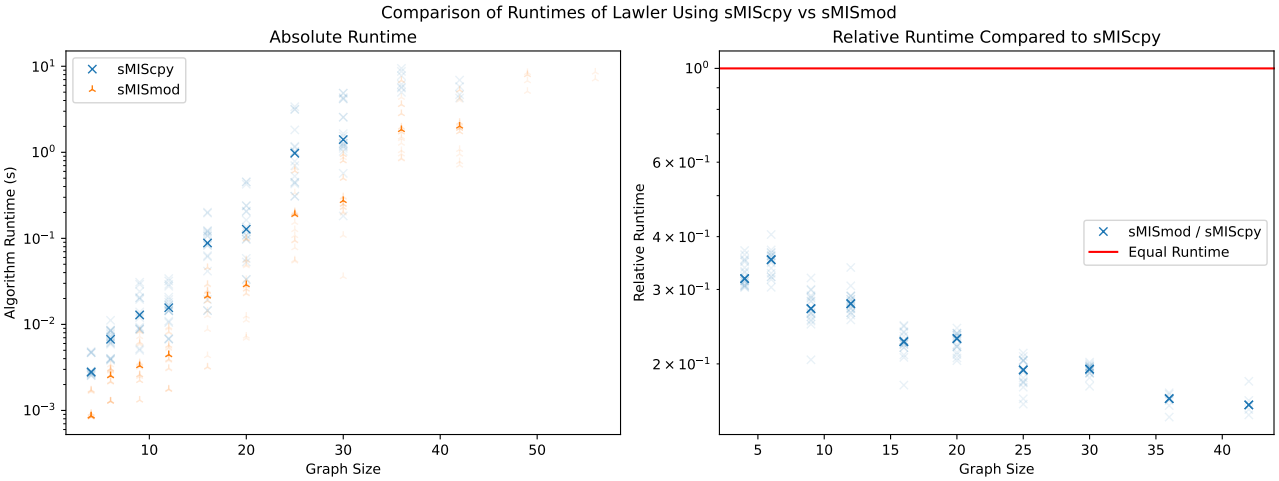


Fig. C.1: The runtimes of `Lawler` with `sMISmod` versus `sMIScpy` on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per configuration. The median is at full opacity, other values at reduced opacity.

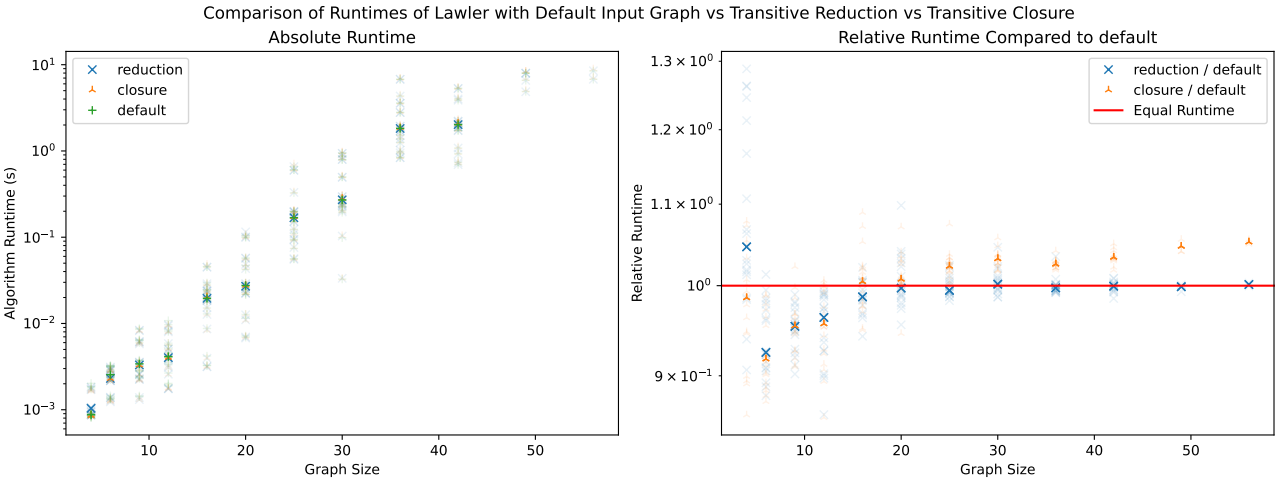


Fig. C.2: The runtimes of `Lawler` when run on the original graph and its transitive variants on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per configuration. The median is at full opacity, other values at reduced opacity.

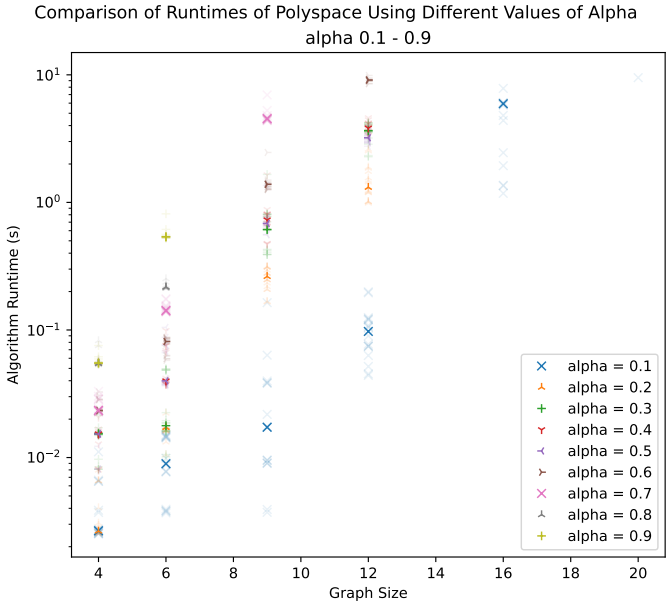


Fig. C.3: The runtimes of `Polyspace` with `sMISmod` and a large range of values of α on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per configuration. The median is at full opacity, other values at reduced opacity.

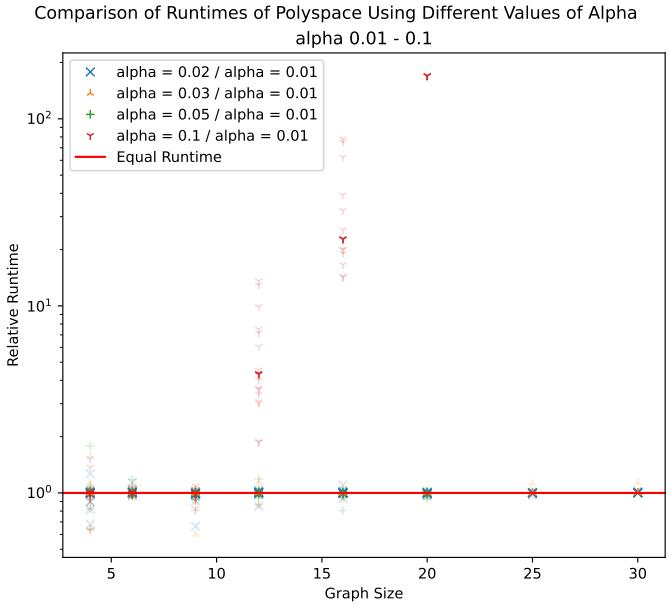


Fig. C.4: The runtimes of `Polyspace` with `sMISmod` and values of α close to 0 on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per configuration. The median is at full opacity, other values at reduced opacity.

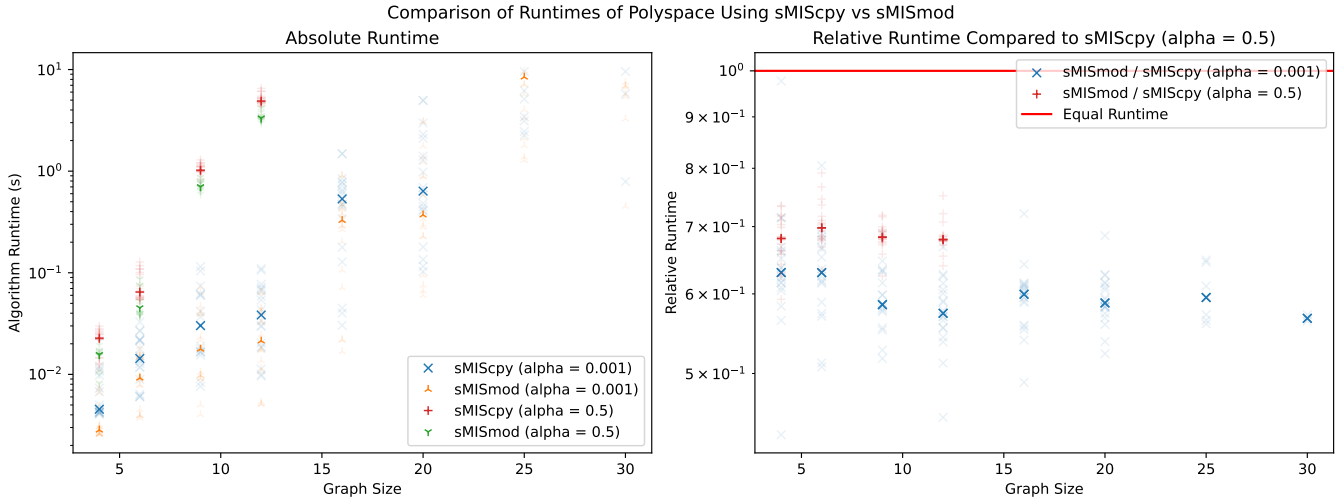


Fig. C.5: The runtimes of `Polyspace` with `sMISmod` versus `sMIScpy` and α taking the values 0.001 and 0.5 on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per variant. The median is at full opacity, other values at reduced opacity.

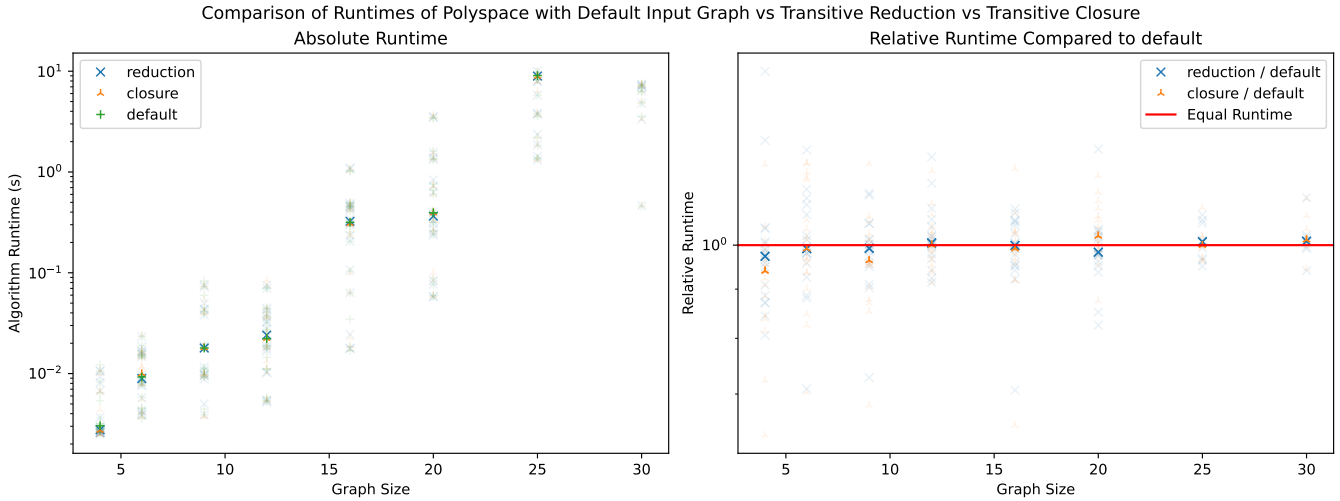


Fig. C.6: The runtimes of `Polyspace` when run on the original graph and its transitive variants on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per variant. The median is at full opacity, other values at reduced opacity.

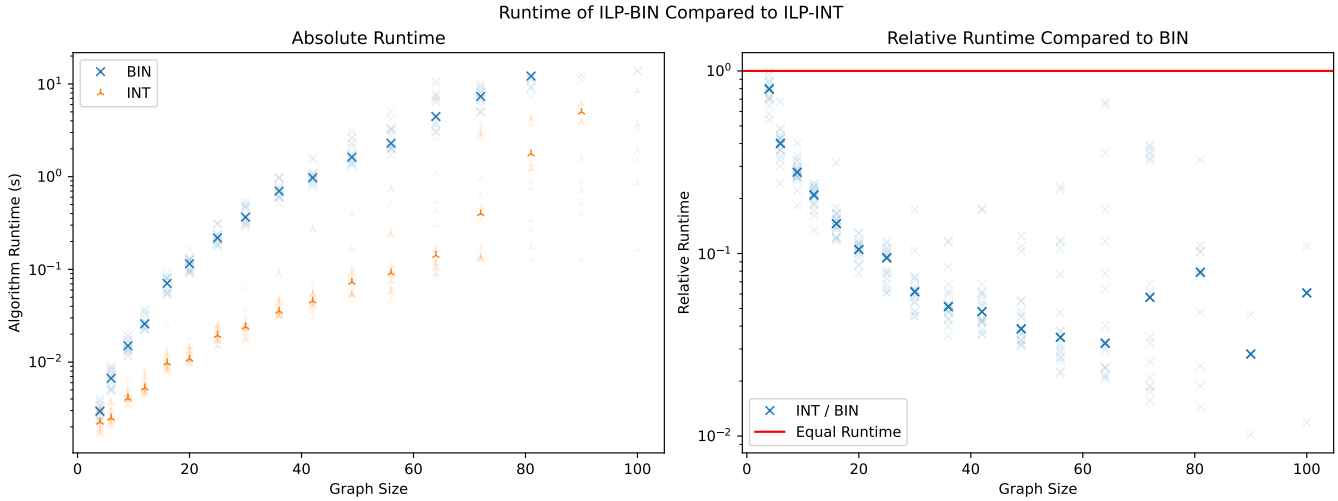


Fig. C.7: The runtimes of `ILP-BIN` versus `ILP-INT` on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per configuration. The median is at full opacity, other values at reduced opacity.

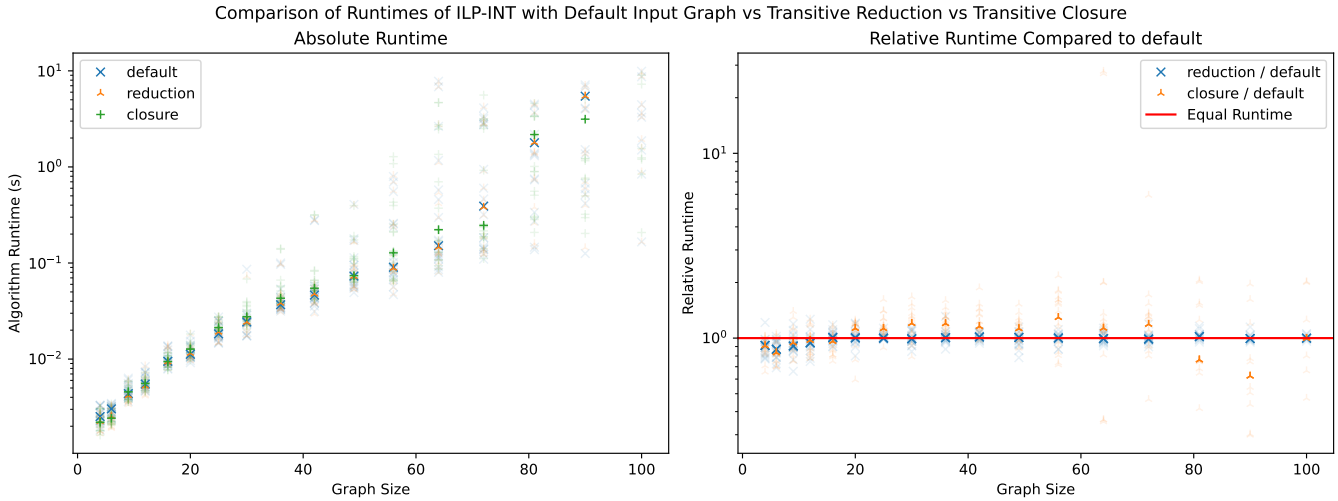


Fig. C.8: The runtimes of `ILP-INT` when run on the original graph and its transitive variants on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per variant. The median is at full opacity, other values at reduced opacity.

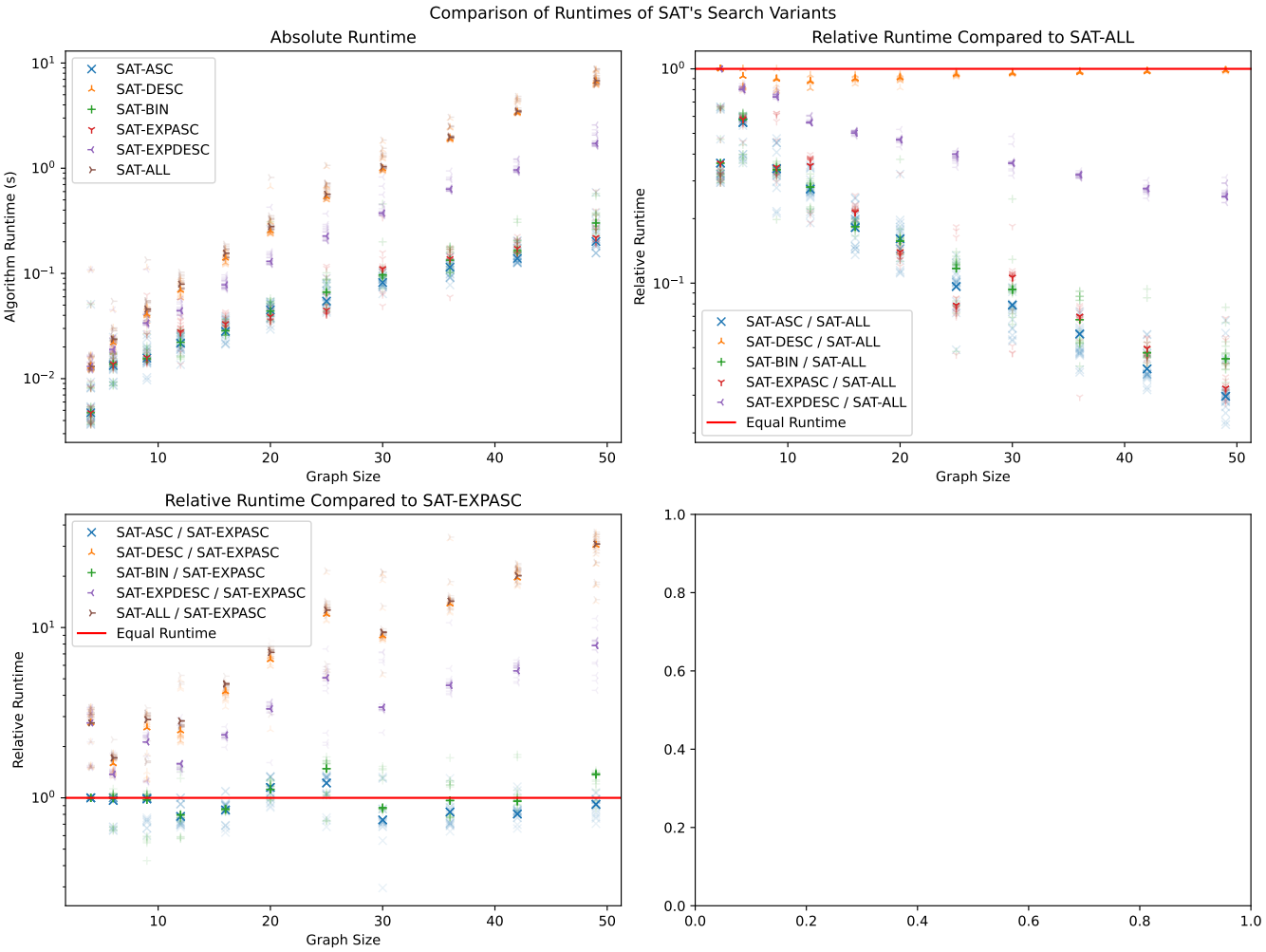


Fig. C.9: The runtimes of `SAT` with the search variants `SAT-ASC`, `SAT-DESC`, `SAT-BIN`, `SAT-EXPASC`, `SAT-EXPDESC`, `SAT-ALL` on the benchmark set **JOBSHOP1A**. Execution times out if `SAT-ALL` runtime exceeds 20s on a graph. The median is at full opacity, other values at reduced opacity.

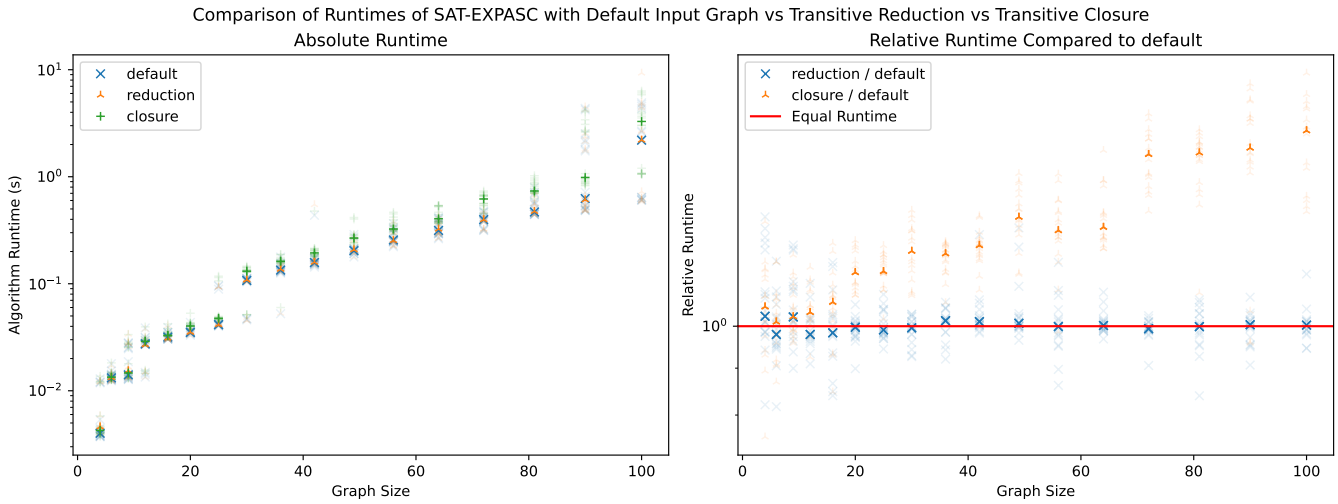


Fig. C.10: The runtime of `SAT-EXPASC` on the benchmark set **JOBSHOP1A**. Execution has a timeout of 10s per graph per variant. The median is at full opacity, other values at reduced opacity.

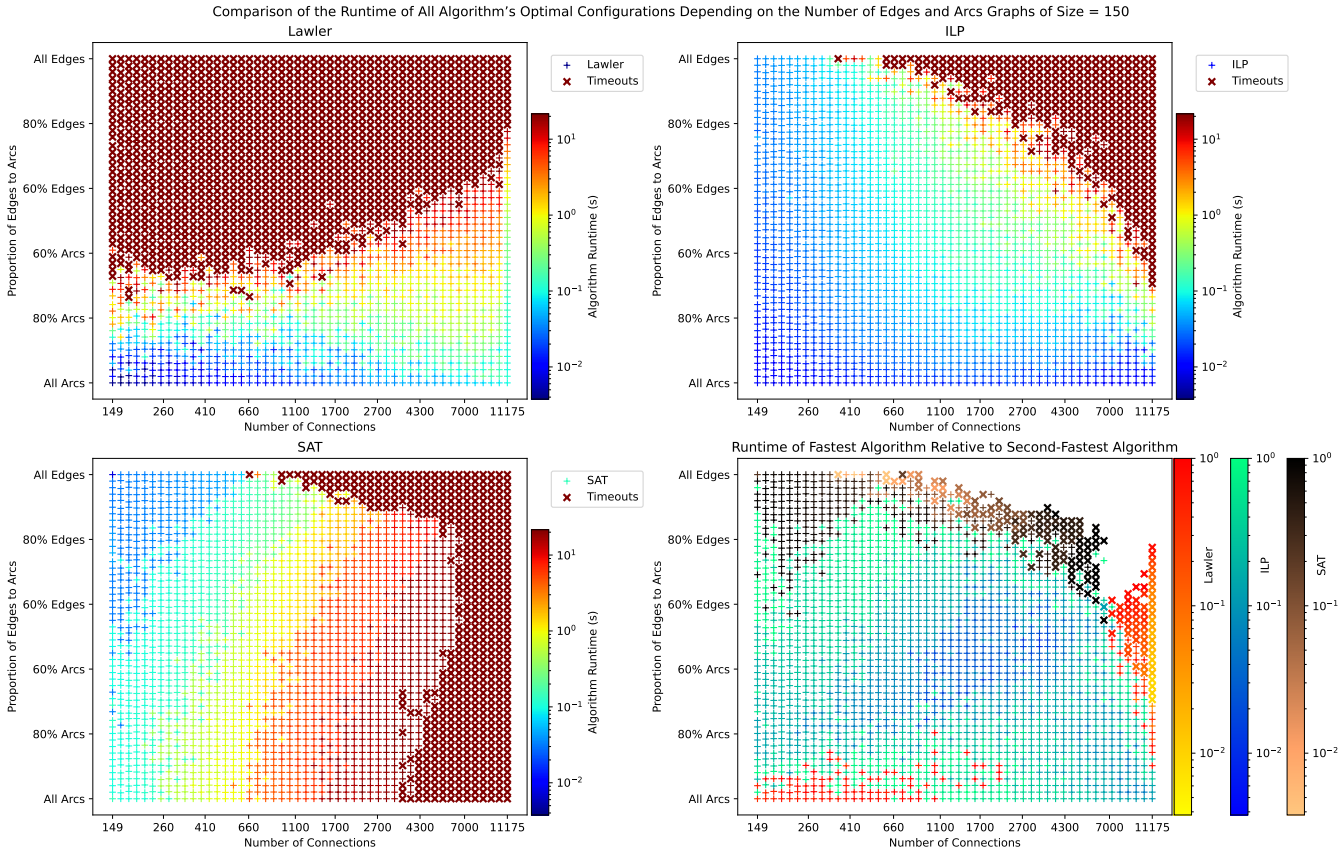


Fig. C.11: The runtimes of `Lawler`, `ILP`, and `SAT` on the benchmark set **E2500S150**. Top, Bottom Left: Absolute runtime of algorithms depending on edges and arcs. Bottom Right: Relative runtime of the fastest algorithm on each graph relative to the second-fastest. If both other algorithms timed out, the relative runtime is calculated relative to the timeout threshold of 20s. Such timeout-markers are shown as a bold x instead of a +.

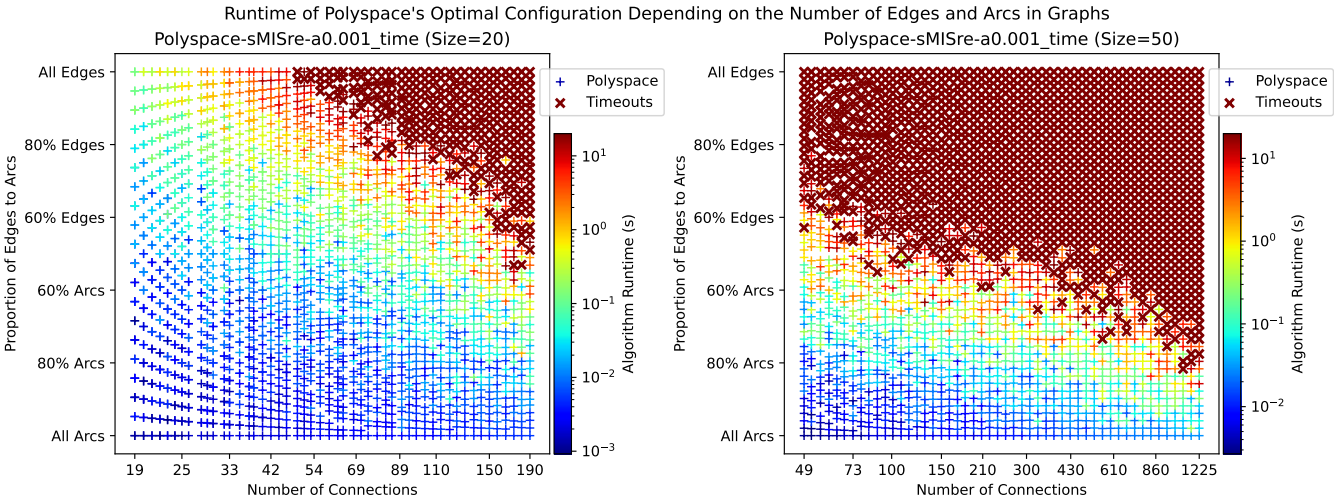


Fig. C.12: The runtime of the optimal configuration of `Polyspace` on the benchmark sets **E2500S20** and **E2500S50**.

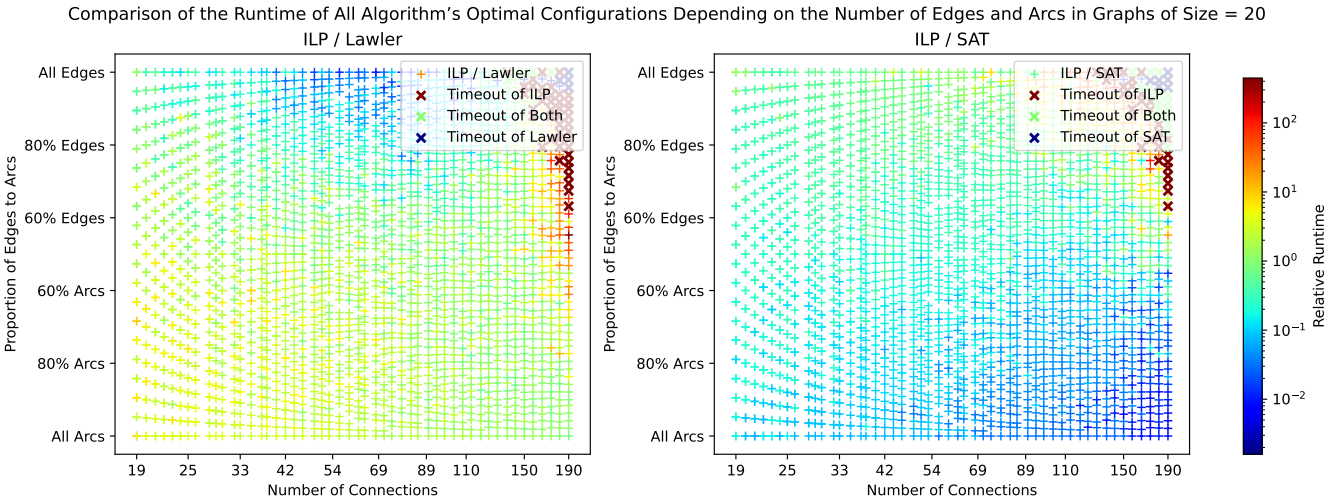


Fig. C.13: The relative runtime of the optimal configurations of `Lawler` and `SAT` compared to `ILP` on the benchmark set **E2500S20**.

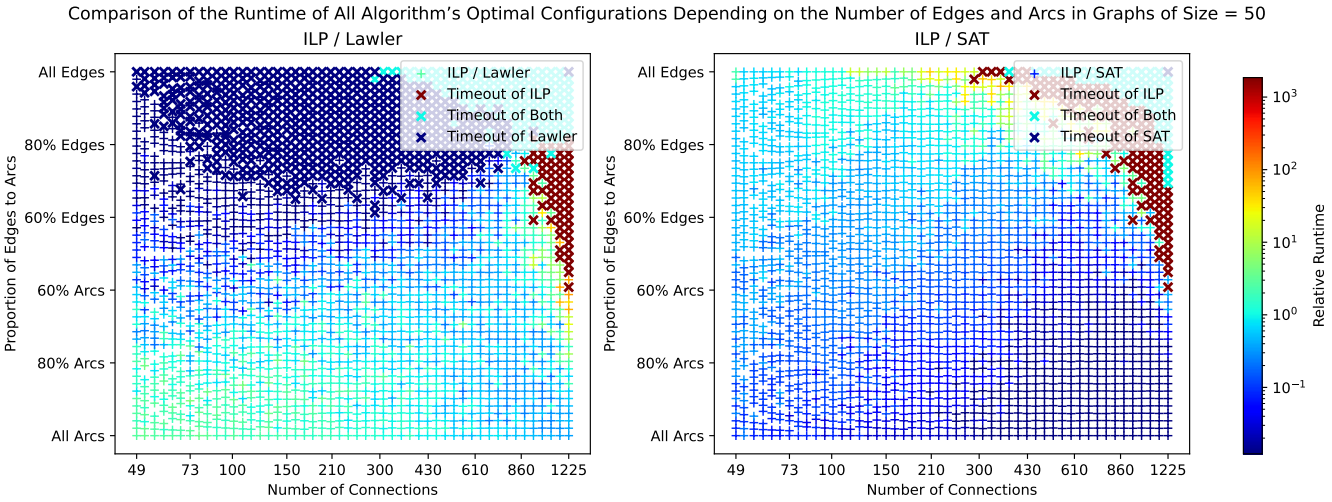


Fig. C.14: The runtime of the optimal configurations of `Lawler` and `SAT` compared to `ILP` on the benchmark set **E2500S50**.

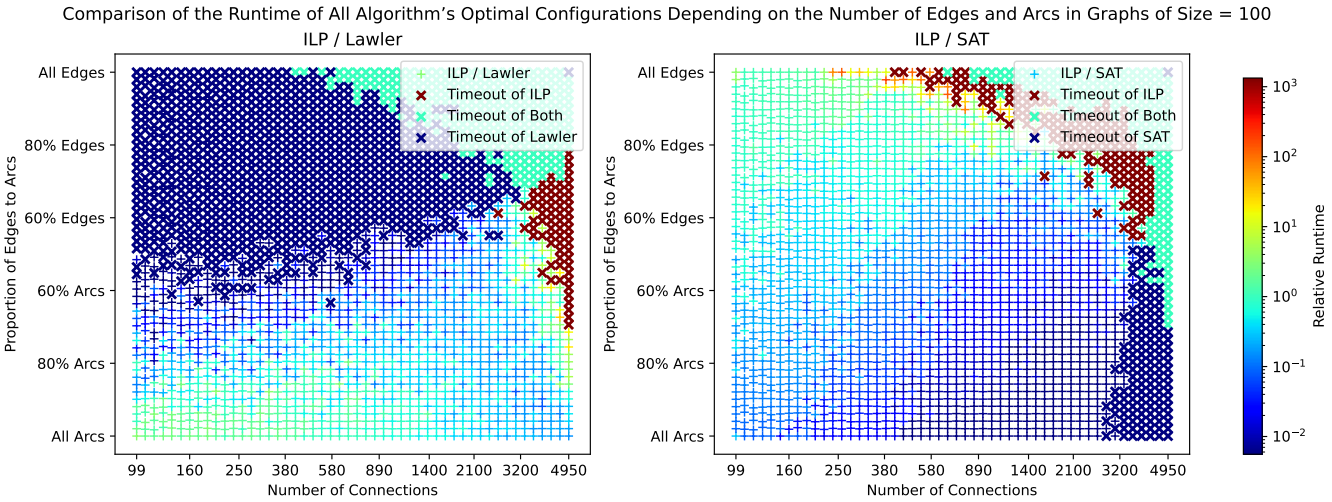


Fig. C.15: The runtime of the optimal configurations of `Lawler` and `SAT` compared to `ILP` on the benchmark set **E2500S100**.

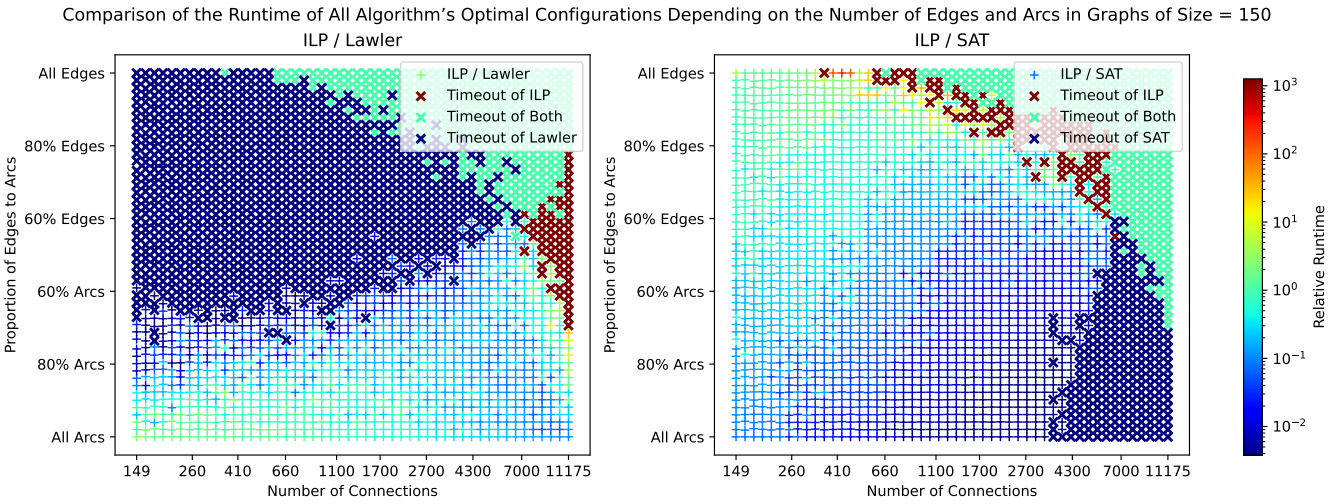


Fig. C.16: The runtime of the optimal configurations of `Lawler` and `SAT` compared to `ILP` on the benchmark set **E2500S100**.

D Appendix: House of Graphs

This section describes a method for obtaining mixed graphs that we considered, but discarded. A reader who has a use for a large number of mixed graphs with known chromatic numbers may be interested in it.

House of Graphs [BCGM12] is "a database of interesting graphs", all of which are undirected. The undirected graphs that are hosted on it are added by people who consider them to be, in some way, of interest. House of Graphs provides a large number of properties of these undirected graphs, including the chromatic number, if it is calculable in reasonable time.

To obtain a mixed graph from undirected graphs, we considered the following method. Given a set of undirected graphs $\{G_0, \ldots, G_n\}$, for every pair G_i, G_{i+1}, we add an arc (u, v) from every vertex $u \in G_i$ to every vertex $v \in G_{i+1}$ to obtain a resulting mixed graph G. This method ensures that G has a chromatic number equal to the sum of the chromatic numbers of $\{G_0, \ldots, G_n\}$.

One reason for considering this method was, that mixed graphs with known chromatic numbers would allow us to verify that our algorithms are implemented correctly. However, as we have four different MixedColoring algorithms, this was also be easily done by comparing the output of all algorithms across many graphs. The probability that all four algorithms are incorrectly implemented in exactly the same way is practically nonexistent.

Another reason for considering the method was, that the graphs would be more representative of graphs encountered in the real world due to their components being considered "interesting" by humans. However, it is unlikely that a person would find a graph interesting simply on the grounds that its components were randomly chosen from a database of graphs that are interesting to different people and chained together with large number of arcs. Additionally, these chained graphs would cover only a narrow subset of all possible graphs, in contrast to the wide spread of graphs we can obtain through random generation. For these reasons, we decided against using this source of graphs.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Die benutzte Literatur sowie sonstige Hilfsquellen sind vollständig angegeben. Wörtlich oder dem Sinne nach dem Schrifttum oder dem Internet entnommene Stellen sind unter Angabe der Quelle kenntlich gemacht.

Weitere Personen waren an der geistigen Leistung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich nicht die Hilfe eines Ghostwriters oder einer Ghostwriting-Agentur in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar Geld oder geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Arbeit stehen.

Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit ist vollständig. Mir ist bewusst, dass nachträgliche Ergänzungen ausgeschlossen sind.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung zur Versicherung der selbstständigen Leistungserbringung rechtliche Folgen haben kann.

Würzburg, den 26. März 2026

.....

Maximilian Schramm