Linear-Time Optimal Deadlock Detection for Efficient Scheduling in Multi-Track Railway Networks

Hastyn Doshi, Ayush Tripathi, Keshav Agarwal, Harshad Khadilkar, Shivaram Kalyanakrishnan

Department of Computer Science and Engineering, IIT Bombay
{200070025,harshadk,shivaram}@cse.iitb.ac.in, {ayush33143314,kshvgrwal}@gmail.com

Abstract
The railway scheduling problem requires the computation of an operable timetable that satisfies constraints involving railway infrastructure and resource occupancy times, while minimising average delay over a set of events. Since this problem is computationally hard, practical solutions typically roll out feasible (but suboptimal) schedules one step at a time, by choosing which train to move next in every step. The choices made by such algorithms are necessarily myopic, and incur the risk of driving the system to a deadlock. To escape deadlocks, the predominant approach is to stay away from states flagged as potentially unsafe by some fast-to-compute rule $R$. While many choices of $R$ guarantee deadlock avoidance, they are suboptimal in the sense of also flagging some safe states as unsafe. In this paper, we revisit the literature on process scheduling and describe a rule $R_0$ that is (i) necessary and sufficient for deadlock detection when the network has at least two tracks in each resource (station / track section), (ii) computable in linear time, and (iii) yields lower delays when combined with existing scheduling algorithms on both synthetic and real data sets from Indian Railways.

1 Introduction
Railway networks around the world form the backbones of national economies. However, due to the constraints imposed by movement on tracks, delays in railways have particularly large domino effects [Goverde, 2010]. In the US, the estimated cost of delays ranges from 200 USD to more than 1000 USD per train-hour [Schlake et al., 2011; Lovett et al., 2015]. Of the total delays in Britain in the 2000s, 40% was composed of primary delay (random events such as train or infrastructure faults), and 60% was secondary delay (caused by subsequent congestion) [Preston et al., 2009]. Khadilkar [2017a] observes that in India, 20% of passenger train services are delayed by at least 10 minutes, and that different scheduling strategies have a significant effect on operational efficiency. This fact motivates us to look at scheduling strategies in detail, with a particular focus on deadlocks, which affect both computation time and schedule efficiency.

The railway scheduling problem is a blocking version [Strotmann, 2007; Liu and Kozan, 2009] of the famous job shop scheduling problem (JSSP) [Manne, 1960]. The JSSP is a class of problems in which a number of jobs or processes (in this case, trains) need to be scheduled to pass through a pre-specified sequence of machines or resources (in this case, stations and inter-station track sections) in some “optimal” way. The blocking variant implies that once a job is loaded on a machine (train enters a track), it must be fully processed through that step before the unit (track) becomes available for the next job (train). Various versions of optimality exist in the literature, from makespan (time duration from start of the first job to the end of the last job) and average queueing time to average delay. Several JSSP variants have been shown NP-complete [Mascis and Pacciarelli, 2002].

With solving for optimality ruled out, common approaches for railway scheduling proceed by “rolling out” schedules over time [Khadilkar, 2018; Prasad et al., 2021]. Abstractly, such algorithms begin from an initial state in which trains are located in respective resources. Then, at each time step, a train is chosen from those eligible to move, and moved forward. This process is continued until (possibly) all the trains complete their journeys. Since the decision of which train to move at each step is made myopically, there is a possibility of reaching a deadlock state, from which no further progress is possible unless some train moves backward—which is expensive and induces large delays in practice. Figure 1 shows a network with three resources, with one free track in the middle resource. If a train heading right is moved into this free track, there is a deadlock. However, deadlock can be avoided by moving a train heading left into the free track.

Interestingly, detecting deadlocks in the general JSSP is also NP-complete [Araki et al., 1977; Cocco and Monasson, 2001]. It has consequently been accepted wisdom in the rail-

![Figure 1: Illustration of possible deadlock. The moves corresponding to red (solid) arrows lead to deadlock, while those corresponding to green (dashed) arrows admit a solution without deadlocks.](image-url)
way scheduling community [Törnquist and Persson, 2007; Pachl, 2012; Khadilkar, 2017b; Vujanic and Hill, 2022] that deadlock detection for railway scheduling is also NP-complete in all its forms. The motivation for our work is a result by Reveliotis et al. [1997] for a variant of JSSP called single-unit resource allocations systems (SURAS), showing that in many reasonable situations and for an arbitrary network topology, a necessary and sufficient condition for deadlock detection can be computed in polynomial time.

We make an explicit connection between the SURAS polynomial-time deadlock detection condition given by Reveliotis et al. [1997] and the railway scheduling problem. In the case where every resource (station and track section) has at least two tracks, we show that this condition can be evaluated for arbitrary network states with a linear complexity in the number of trains (slightly more efficiently than Reveliotis et al.). Thereafter, we demonstrate the significant benefit of implementing not just sufficient but necessary conditions for deadlock-free movement on real-world railway networks in India. These include single-track as well as multi-track lines, thus showcasing the wide applicability of the algorithm.

From an infrastructure perspective, laying two tracks has only modest additional cost compared to laying one track, because the land and utilities infrastructure already exists. Hence the only places where single tracks are typically laid are ones where the traffic is low (in which case sophisticated algorithms are not needed) or where the terrain is tough. Indeed it is apparent even from data sets used in the academic literature [Pappaterra et al., 2021; Prasad et al., 2021] that nodes in real-world railway lines usually contain more than one track. Even where single tracks exist (we have empirical results in Section 5), we can handle them so long as a set of feasible moves exist for moving trains to the nearest multi-track resource. At this point, we can drop empty single tracks from the analysis (since they effectively connect two resources, rather than act as independent resources) and the remaining analysis is valid. In the case of scheduling algorithms, we simply ensure that a train that moves into a single track must be moved on to a multi-track section before another train move is attempted.

After discussing related work in Section 2, we formalise the railway deadlock detection problem in Section 3. In Section 4, we present a novel and conceptually simple interpretation of the detection rule of Reveliotis et al. [1997] for multi-track networks. In Section 5, we empirically validate the utility of this rule in scheduling. We conclude in Section 6.

2 Related Work

The railway scheduling problem is that of establishing a feasible timetable for a set of ‘services’ between given origins and destinations [Törnquist and Persson, 2007]. Previous work [Cai and Goh, 1994; Liu and Kozen, 2009; Strotmann, 2007] shows that the Job Shop Scheduling Problem (JSSP) and railway scheduling are reducible to each other; their decision variants are both NP-complete. The train scheduling problem can be solved at leisure, combining exact and randomised search algorithms [Higgins et al., 1996; Törnquist and Persson, 2007]. The train rescheduling problem involves a disruption externally imposed on the timetable, from which one must quickly compute a set of recovery actions to return to the original timetable [Sinha et al., 2016]. Delays in this context can be computed relative to the corresponding event times in the original timetable. Deadlocks are also important in this context since disruptions to the timetable may require changes in the order of train moves, with uncertain implications for operational feasibility. In this paper, we consider both scheduling and rescheduling, with the understanding that computational complexity has more impact on rescheduling (disruption recovery) than on scheduling (timetabling).

Rescheduling and Deadlock Avoidance in Railways. While exact formulations of rescheduling as an optimisation problem are available [Higgins et al., 1996; Törnquist and Persson, 2007], they are not scalable. Practical approaches instead use heuristics [Higgins et al., 1997; Chen et al., 2015] or pretrained policies [Šemrov et al., 2016; Khadilkar, 2018; Prasad et al., 2021] to move trains forward through the network (in a manner analogous to checkers pieces) until they reach their destinations. The order of train movements, their timings, and track allocations vary by the algorithm. However, as illustrated in Figure 1, the risk in all such finite-lookahead methods is that of deadlock, or a situation where some or all trains are unable to move forward because of a circular dependence on each other [Pachl, 2012].

Pachl [2012] proposes four conditions which are necessary to create deadlocks. If a scheduling strategy ensures that at least one of these conditions is not met, it is sufficient to avoid deadlocks. Similarly, Mackenzie [2010] proposes sufficient conditions for avoiding deadlock, including the conservative path-to-destination approach. Khadilkar [2017b] proposes the critical-first approach for railway lines, which focuses on prioritising occupants of the most constrained resources in order to avoid bottlenecks. The condition in this case is to keep moving trains forward until at least one additional free track is available for other trains to pass. Vujanic and Hill [2022] make this more concrete by defining the notion of a safe state as one where all nodes (resources) have an unoccupied slot. If initialised from a compliant initial state, their procedure takes polynomial time for scheduling.

The basis for all these studies is that the NP-completeness of the general deadlock detection problem makes it hard to detect deadlocks in instances encountered in practice. Therefore, deadlock detection is typically performed by rules that provably detect deadlocks, but might also flag false positives. The novelty of our paper is in identifying the applicability of an optimal polynomial-time deadlock detection algorithm in JSSP to the railway context (the notion of optimality is formally defined below).

Deadlock Avoidance in JSSP. We shall first define the various terms used in this paper. The problem of evaluating an arbitrary state of the railway network, for the presence/absence of present/future deadlock, is deadlock detection. Any subsequent scheduling policies that reduce the probability of deadlock (but not eliminate it) are called deadlock avoiding policies, while scheduling policies that guaran-
the absence of deadlock are called deadlock free policies.

Similar to the railway scheduling case, it is well known that optimal deadlock detection in JSSP is also NP-complete. Araki et al. [1977] consider the question: “given a state $S$, is $S$ safe?” They reduce the 3-SAT problem [Cocco and Monasson, 2001] to optimal deadlock detection in JSSP, thereby proving the latter to be NP-complete. Fanti et al. [1997] derive necessary and sufficient conditions for deadlock in production systems with resource sharing, and then propose a ‘restriction policy’ that is tractable and provably correct (in the sense of sufficiency). Gold [1978] considered the question from a more practical perspective, examining under what restrictions on state $S$ one can efficiently detect deadlock. They derived some conditions under which deadlock detection can be solved in polynomial time.

**Optimal Deadlock Detection.** Previously published studies also consider two forms of optimality in the present context. The first definition of optimality implies the minimisation of delays in the schedule with respect to a reference timetable, or the minimisation of the makespan of the schedule if no reference timetable is available [Törnquist and Persson, 2007]. The second definition of optimality [Reveliotis et al., 1997] refers to the removal of unsafe transitions from the current state, with the objective of identifying the smallest (hence optimal) set of unsafe transitions that ensures the absence of present or future deadlocks. In this paper, by *optimal rule* we refer to the second definition: to a rule that characterises necessary and sufficient conditions of states or transitions to be safe, and hence can be used for deadlock-free scheduling.

Reveliotis et al. [1997] develop necessary and sufficient conditions for deadlock prevention in “single-unit sequential resource allocation systems” (SURAS). They show that deadlock detection in polynomial-time is possible in the special case where every resource in the system has a minimum capacity of 2 units. If the number of resources is $m$ and $C$ is the maximum capacity among these nodes, their detection condition has $O(m^2 C)$ complexity. The intuition behind this number is that a search algorithm makes $m$ passes through the set of $m$ resources, eliminating one eligible resource in each pass. Our observation is that this result applies to the case of arbitrary railway network topologies (branching and straight lines) as long as there are at least two tracks in each node (in railway terminology, at stations and inter-station track sections). Furthermore, (i) the result can actually be implemented in linear (and not quadratic) complexity, and (ii) we can handle single-resource nodes under reasonable assumptions, as explained in Section 1.

### 3 Deadlock Detection

In this section, we specify the problem of (optimal) deadlock detection. We begin from the broader context of railway scheduling, within which this problem arises.

#### 3.1 Railway Scheduling Problem

**Railway Infrastructure.** A railway network is made up of a number of *resources*, each containing some number of parallel tracks running from one end of the resource to the other. Tracks admit traffic in both directions. Stations (where trains may halt) as well as the inter-station track sections between them (where trains do not have scheduled halts) are modeled as resources. A resource connects to other resources through one of its ends. Typically, terminal resources have all their connections only from one end, but in general we could have cycles in the network topology. Figure 2 shows an illustrative railway network with branching and a cycle; the example in Figure 1 has a linear topology (often called a *line*).

**Desired Schedule.** The dynamic aspect of the scheduling problem arises from the movement of a set of trains through resources. The target is to meet a desired schedule $S_{\text{desired}}$ which may be represented as a set of $N$ events:

$$S_{\text{desired}} = \{ e[i], 1 \leq i \leq N \},$$

where $e[i] = (\text{train}[i], \text{start\_res}[i], \text{next\_res}[i], \text{time}[i])$.

Event $e[i]$ specifies that train $\text{train}[i]$ must be moved from resource $\text{start\_res}[i]$ to the adjoining resource $\text{next\_res}[i]$ at time $\text{time}[i]$. Now, it may not be possible to execute $S_{\text{desired}}$, due to constraints imposed by the railway infrastructure. For instance, if three events all mean to push trains into the same resource at the same time, but this resource only has two free tracks, then at least one of the events will have to be *delayed*. The goal of scheduling is to compute an operable schedule $S_{\text{operable}}$ that is feasible to execute, but at the expense of delaying a subset of events in $S_{\text{desired}}$. For each event $e[i]$, the operable schedule has a replacement $\overline{e}[i]$ with a new time $\text{time}[i] \geq \overline{\text{time}}[i]$:

$$S_{\text{operable}} = \{ \overline{e}[i], 1 \leq i \leq N \},$$

where

$$\overline{e}[i] = (\text{train}[i], \text{start\_res}[i], \text{next\_res}[i], \text{time}[i]).$$

Formally, the objective function to be minimised while computing $S_{\text{operable}}$ is the average departure delay

$$ADD = \frac{1}{N} \sum_{i=1}^{N} (\overline{\text{time}}[i] - \text{time}[i]). \quad (1)$$

Since the problem of computing an operable schedule that minimises ADD is NP-hard [Mascis and Pacchiarelli, 2002], one practical alternative is to roll out schedules over time, ensuring operability, while making greedy choices to reduce delays [Khadiilkar, 2018; Prasad et al., 2021].

#### 3.2 Roll-Out Algorithms

A roll-out algorithm executes the set of events $\{ e[i], 1 \leq i \leq N \}$ one by one. The algorithm begins with a counter $\tau$ set to the earliest event time, with state $s_{\tau}$ associating each train with its initial resource. An event $i$ is said to be executed (and inserted into $S_{\text{operable}}$) when the algorithm sets $\text{time}[i]$.

![Figure 2: Example of a network topology. Two trains are shown.](image-url)
At each counter value \( \tau \), the algorithm compiles the list of events that are \( \text{eligible} \): these are events \( e[i] \), \( 1 \leq i \leq N \), such that (i) \( \text{train}[i] \) is in \( \text{start}_{\text{res}}[i] \) in state \( s_\tau \); (ii) there is a free track in resource \( \text{next}_{\text{res}}[i] \) in \( s_\tau \); and (iii) \( \text{time}[i] \geq \tau \). If, indeed, there are eligible events, one of these events \( i \) is selected and executed by setting \( \text{time}[i] = \tau \). The updated event \( \pi[i] \) is moved into \( S_{\text{operable}} \), and \( e[i] \) is no longer eligible. As long as there are eligible events at \( \tau \), these are repeatedly executed, until there are no eligible events at \( \tau \); in this case \( \tau \) is incremented and the procedure continues. Since train journeys are a sequence of contiguous resources, any train can be in at most one eligible event at any time step. Hence, it is sometimes convenient to view the set of eligible events at \( \tau \) as the set of trains that are eligible to be moved at \( \tau \).

By construction, the set of events that have already been executed by a roll-out algorithm have no internal conflicts. Hence, if all \( N \) events in \( S_{\text{desired}} \) get executed, we are guaranteed an operable schedule \( S_{\text{operable}} \). However, the ADD of \( S_{\text{operable}} \) depends on the delays introduced while executing the events. The choice of which event among the eligible ones to execute at any step also has the long-term consequence of which events become eligible in subsequent time steps. By and large, roll-out algorithms make this choice greedily [Khodilkar, 2018; Prasad et al., 2021]. An unfortunate consequence is the possibility of a deadlock, wherein there remain events to execute, but these cannot become eligible at the current counter value \( \tau \) or anything larger.

### 3.3 Deadlock Detection Problem

Abstractly, the progress of a roll-out algorithm for generating a schedule can be viewed as a sequence of state transitions. The background data from the problem instance, which guide and constrain these transitions, are (1) the set of resources \( U \); (2) resource capacities encoded by \( C : U \to \mathbb{N} \); (3) the set of trains \( T \); and (4) the set of train journeys \( D = \{(t, u^1, u^2, \ldots, u^t), t \in T\} \). In \( D \), each journey \((t, u^1, u^2, \ldots, u^t)\) contains a train \( t \in T \) and the identities of some \( l_t \geq 1 \) resources through which \( t \) must pass in sequence. Exact event \textit{times} are not needed for deadlock detection. As motivated in Section 1, we make the following “multi-track” assumption while devising and analysing our algorithm, which is presented in Section 4.

**Assumption 1.** For all \( u \in U \), \( C(u) \geq 2 \).

However, the problem statement presented below does not depend on this assumption.

**States, Actions, Transitions.** Each state \( s \) in our system contains a subset of trains \( T_s^m \subseteq T \) that are yet to complete their journeys. In \( s \), each train \( t \in T_s^m \) is in some resource \( u \in U \). Hence, a state can be represented as a set of pairs \((t, u) \in T \times U \). Let \( S \) denote the set of all states. The desired terminal state \( s_T = s \) is the one in which all trains have reached their destinations. Destinations typically connect to “yards” with effectively infinite capacity, so trains do not occupy regular tracks at their destinations. Hence \( s_T = \emptyset \).

A useful quantity to associate with each state \( s \in S \) is its “potential” \( \phi(s) \), which we define to be the sum of the distances of the trains present in \( s \) to their respective termini.

Concretely, suppose \( s = \{(t_i, u_i), 1 \leq i \leq m\} \), where the remaining sequence of resources for train \( t_i \) to visit after departing \( u_i \) is \( u_i^1, u_i^2, \ldots, u_i^{l_i} \) for some \( l_i \geq 1 \). Then we have \( \phi(s) = \sum_{i=1}^{m} (1 + l_i) \). Observe that \( \phi(s_T) = 0 \).

The set of \textit{actions} available from state \( s \in S \) is denoted \( A(s) \). Each action \( a \in A(s) \) corresponds to moving some train from its current resource to the next one on its journey. Naturally, only moves corresponding to eligible events are present as actions in \( A(s) \).

When an action from \( A(s) \) is performed on state \( s \in S \), we denote the resulting state \( s + a \). Suppose action \( a \in A(s) \) moves train \( t \) in \( s \), where \( t \) is in resource \( u \), to its next resource \( u' \). If \( u' \) is the terminal resource for \( t \), then \( s + a = s \{ \{(t, u') \} \} \); otherwise \( s + a = (s \setminus \{(t, u)\}) \cup \{(t, u')\} \). Notice that when an action from \( A(s) \) is performed on \( s \), progress is achieved in the sense that \( \phi(s + a) = \phi(s) - 1 \). Since trains cannot move backwards, this progress cannot be undone. However, as we see next, an action may lead to a state from which further progress is not possible.

**Safe and Unsafe States.** By definition, the desirable terminal state \( s_T = \emptyset \) is a safe state; so also is every state for which there exists a sequence of actions to reach \( s_T \). We may write down recursively: for \( s \in S \),

\[
\text{SAFE}(s) \iff (s = s_T) \lor (\exists a \in A(s) : \text{SAFE}(s + a)) \text{.}
\]

This recursive definition gives rise to a straightforward procedure to compute \( \text{SAFE}(s) \), since any state \( s + a \) in the RHS has a potential value \( \phi(.) \) one unit lower than \( s \). However, there is branching by a factor of \( |A(s)| \), implying exponential complexity for a naive implementation. Our main observation, described in the next section, is that \( \text{SAFE}(s) \) can be computed in time that is only linear in the size of \( s \).

An unsafe state is a state that is not safe. A deadlocked state is a state containing trains, but on which no valid action can be performed. For \( s \in S \),

\[
\text{UNSAFE}(s) \iff \neg \text{SAFE}(s) ; \\
\text{DEADLOCK}(s) \iff (s \neq s_T) \land (A(s) = \emptyset) .
\]

Since our system contains a finite number of trains, and their journeys are also finite, it follows that \( \phi(.) \) has a finite upper bound. Since \( \phi(.) \) decreases by 1 unit after each action, the length of any action sequence starting from any initial state \( s_0 \) is also guaranteed to be finite. It follows that if \( s \) is unsafe, then any sequence of actions starting from \( s \) will lead to a deadlocked state, from which no further actions are available.

**Computational Problem.** We require a procedure that can efficiently identify whether a given state \( s \in S \) is safe or not. It is convenient to view this procedure as a rule or proposition \( R(s) \), which evaluates to a boolean value. \( R(s) \) may depend both on \( s \) and on the journey details of the trains in \( D \), but must be efficient to compute. Several “sufficient” rules \( R \) from the literature guarantee that \( R(s) \iff \text{SAFE}(s) \). We require a “necessary and sufficient” rule, also called an optimal rule, which satisfies \( R(s) \iff \text{SAFE}(s) \).

As described in Section 1, optimal rules are computationally hard on unrestricted problem instances. On the other hand, we show next that if Assumption 1 is satisfied, then an optimal rule can be implemented in only linear time.
4 Linear-Time Algorithm

The algorithm presented here is due to Reveliotis et al. [1997], who proposed it in the context of resource allocation and implemented it with quadratic complexity. We describe this algorithm from the perspective of deadlock detection in railway networks, using the vocabulary introduced in Section 3. We provide a concise proof of correctness based on a graph-theoretic model, and also show that a linear-time implementation is possible.

Next-Stop Graph. The main data structure involved in specifying $R_0$ is a directed graph constructed based on input state $s$. The construction also requires the resource capacities encoded by $C$ and the set of train journeys $D$. We denote this directed graph $G_s = (V, E, C)$, where $V$ is the set of vertices, $E$ the set of edges, and $C : V \rightarrow \{\text{red, black}\}$ a function that associates a colour with each vertex.

Recall that $T_s$ is the set of trains in $s$. A resource $u \in U$ is a vertex $v \in V$ in $G_s$ if and only if $u$ is the current resource for some train $t$ in $s$, or it is the next resource for some train $t$ in $s$ (as specified in $t$’s journey in $D$). The colour $C(v)$ of a vertex $v \in V$ is red if the corresponding resource has at least one free track (that is, the number of trains in this resource is strictly smaller than the capacity). Fully-occupied vertices are coloured black. Each train in state $s$ gives rise to an edge from the vertex of its current resource $u$ to the vertex of its next resource $u'$. Hence, each edge $e \in E$ corresponds to one or more trains in $s$. Notice that every vertex $v \in V$ must have at least one edge, either incoming or outgoing (but possibly one or more of each type).

Surprisingly, one does not need to access the extended journeys of trains in $s$ in order to construct $G_s$: one only needs the trains’ current and next resources. For this reason, we may refer to $G_s$ as the “next-stop graph” of $s$. Notice that the number of edges in $G_s$ is at most the number of trains in $s$: that is, $|E| \leq |T_s|$. Clearly $G_s$ does not exceed the size of $s$ or of the journey data $D$ beyond a constant factor, as both $s$ and $D$ use $\Omega(|T_s|)$ space. Even so, $G_s$ provides all the information required for optimal deadlock detection.

Optimal Rule. Our rule $R_0$ has an intuitive form.

Definition 2. For $s \in S$, $R_0(s)$ is the proposition: “In $G_s$, every black vertex has a directed path to some red vertex.”

We formally show that under the multi-track assumption, $R_0$ is an optimal deadlock detection rule.

Theorem 3. If the problem instance satisfies Assumption 1, then for $s \in S$, $R_0(s) \iff $ SAFE($s$).

Proof. Let $G_s = (V, E, C)$. We prove the theorem by induction on $\phi(s)$. As base case, consider arbitrary $s \in S$ for which $\phi(s) = 1$. If so, there is exactly one train $t$ in the network, in a resource that connects to $t$’s terminus. Clearly $s$ is safe since $t$ can be moved into its terminus (thus $s$ transitions into $s_{t\rightarrow}$). Also notice that in this case, $G_s$ comprises exactly two vertices $r_1, r_2 \in V$, with an outgoing edge from $r_1$ to $r_2$. Since each resource has at least two tracks, $r_1$ and $r_2$ must both be red, making $R_0(s)$ trivially true. In short, when $\phi(s) = 1$, $R_0(s)$ and SAFE($s$) are both true, and thereby equivalent.

Our induction hypothesis is that for some integer $m \geq 1$, $R_0(s) \iff $ SAFE($s$) for all $s \in S$ having $\phi(s) = m$. Now consider a state $s \in S$ for which $\phi(s) = m + 1$. We separately prove the two implications in the theorem.

1. Proof of $R_0(s) \implies $ SAFE($s$). Suppose that $R_0(s)$ is true: that is, in $G_s$, every black vertex has a directed path to some red vertex. We consider two complementary subcases.

1.1. Suppose $G_s$ contains some red vertex $r \in V$ with no outgoing edges (Figure 3a). As in our base case, $r$ must contain an incoming edge from some vertex $v \in V$. Notice that $r$ has two or more empty tracks. Hence, we can move a train $t$ from $v$ to $r$ to obtain a state $s'$ with $\phi(s') = m$. If there are any incoming edges into $v$ in $s$, or $v$ has trains other than $t$ in $s$, then $v$ would also be a vertex in $s'$, but now coloured red. Otherwise $v$ would not be a vertex in $G_{s'}$. Depending on $t$’s next stop from $s'$, $r$ could get a new edge to an existing vertex or a new red vertex in $G_{s'}$. Regardless, notice that if any black vertex had a path to a red vertex in $G_{s'}$, it would continue to have a path to a red vertex in $G_{s'}$. Hence, if $R_0(s)$ is true, then $R_0(s')$ is also true. By the induction hypothesis, $s'$ is safe, and hence $s$ is also safe.

1.2. The second subcase is that every red vertex in $G_s$ has an outgoing edge (Figure 3b); every black vertex in $G_s$ will anyway have at least one outgoing edge. In this case, we consider a subgraph $G'_s$ of $G_s$ (Figure 3c), which differs only in the set of edges. Indeed let $G' = (V, E', C)$ so that (i) each vertex $v \in V$ has exactly one outgoing edge in $E'$, and (ii) each black vertex in $G'$ has a directed path to some red vertex in $G'$. A natural way to construct $E'$ would be to first fix some outgoing edge to a red vertex from all black vertices having such an edge in $E$, then to fix an outgoing edge to one of those black vertices from all other black vertices having such an edge in $E$, and proceeding similarly until all black vertices have an outgoing edge. Thereafter each red vertex can be given an arbitrary outgoing edge from $E$.

By its definition, $G'$ cannot have a directed cycle with only black vertices (since that would imply that those vertices do
not have a directed path to some red vertex in \( G \), hence invalidating \( R_0(s) \)). Also, since it has an outgoing edge for each vertex, \( G' \) must contain a directed cycle. In summary, we infer that \( G' \) must contain a directed cycle with at least one red vertex. Indeed let such a cycle \( C \) contain the sequence of vertices \( r, v_1, v_2, \ldots, v_m \) for some \( m \geq 1 \), where \( r \) is a red vertex. We are indifferent to the colours of \( v_1, v_2, \ldots, v_m \).

Now consider the action of moving a train from \( v_m \) to \( r \), leading to next state \( s' \). The set of vertices in \( G_s' \) remains identical to that of \( s \), except that \( G_{s'} \) may get a new red vertex that is the next step for \( r \) in \( s' \). The set of edges in \( G_{s'} \) is also identical to that of \( G_s \), except for (i) the possible loss of the single edge from \( v_m \) to \( r \) in case \( t \) was the only train having that transition in \( s \), and (ii) the possible gain of a new outgoing edge from \( r \). Regardless, \( v_m \) is necessarily a red vertex in \( G_{s'} \), and moreover, any black vertex that had a directed path to \( r \) in \( G_s \) must have a path to \( v_m \) in \( G_{s'} \) (since \( r \) has a path through the sequence of vertices in \( C \) to \( v_m \)). Directed paths not involving \( C \) in \( G_s \) remain the same in \( G_{s'} \). Hence, we conclude that if \( R_0(s) \) is true, then \( R_0(s') \) is also true. Since \( \phi(s') = m \), we obtain from the induction hypothesis that \( s' \) is safe. Since we can go from \( s \) to \( s' \) by moving \( t \), we observe that \( s \) must also be safe.

2. **Proof of \( \text{SAFE}(s) \implies R_0(s) \)**: Suppose \( R_0(s) \) is not true: that is, there exists a black vertex \( b \in V \) with no directed path to any red vertex in \( V \) (Figure 3d). \( b \) is fully occupied in \( s \), and so it must have outgoing edges in \( G_s \). Since these edges do not lead to a red vertex in \( G_s \), we surmise that there exists a finite subset of black vertices \( B \subseteq V \) of size at least two such that for each vertex in \( B \), all outgoing edges lead to other vertices in \( B \). Since every vertex \( b' \in B \) is fully-occupied, no train can be moved out from or moved into any vertex in \( B \). If \( s \) is already in deadlock, by definition it is unsafe. On the other hand, after any possible move of any train outside of \( B \) in \( s \) to reach \( s' \), \( B \) remains a finite set of black vertices \( G_{s'} \), with no directed path to any red vertex. Since \( \phi(s') = m \), the induction hypothesis gives us that \( s' \) is unsafe, and hence \( s \) is unsafe.

Although \( R_0 \) is essentially a rephrasing of the rule given by Reveliotis et al. [1997] for SURAS, its interpretation in terms of a next-step graph \( G_s \) is novel. The algorithm given by Reveliotis et al. [1997] eliminates one node in each pass through the set of resources, hence takes time that is quadratic in the number of resources. On the other hand, it is easy to see that \( R_0(s) \) can be computed in time that is only linear in the number of edges in \( G_s \), which is generally much smaller than \( |U|^2 \). The pseudocode in Figure 4 is for an implementation of \( R_0 \) by a standard search algorithm [Russell and Norvig, 2022, see Chapter 3], taking \( O(|E|) \) operations for input \( G_s = (V, E, C) \).

5 Experimental Validation

We compare our proposed rule \( R_0 \) against other deadlock avoidance algorithms on real railway schedules as well as synthetically-generated ones. Data and code to reproduce the results reported in this section are available at: https://github.com/Hastyn/Linear-Time-Deadlock-Detection/.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Sn.</th>
<th>Tns.</th>
<th>Time Span</th>
<th>Events</th>
<th>Con. Sec.</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajmer</td>
<td>52</td>
<td>444</td>
<td>5.5 day</td>
<td>13129</td>
<td>51</td>
<td>0.016</td>
</tr>
<tr>
<td>Kanpur</td>
<td>27</td>
<td>190</td>
<td>1.4 day</td>
<td>3858</td>
<td>26</td>
<td>0.036</td>
</tr>
<tr>
<td>Konkan</td>
<td>59</td>
<td>85</td>
<td>2.2 day</td>
<td>2709</td>
<td>58</td>
<td>0.007</td>
</tr>
<tr>
<td>HYP-1</td>
<td>5</td>
<td>8</td>
<td>1.8 hrs</td>
<td>40</td>
<td>4</td>
<td>0.041</td>
</tr>
<tr>
<td>HYP-2</td>
<td>11</td>
<td>60</td>
<td>2.0 day</td>
<td>660</td>
<td>10</td>
<td>0.011</td>
</tr>
<tr>
<td>HYP-3</td>
<td>11</td>
<td>120</td>
<td>2.1 day</td>
<td>1320</td>
<td>10</td>
<td>0.021</td>
</tr>
<tr>
<td>HYP-4</td>
<td>4</td>
<td>350</td>
<td>0.8 hrs</td>
<td>1290</td>
<td>3</td>
<td>3.839</td>
</tr>
<tr>
<td>HYP-5</td>
<td>3</td>
<td>200</td>
<td>0.5 hrs</td>
<td>4766</td>
<td>2</td>
<td>31.773</td>
</tr>
<tr>
<td>HYP-6</td>
<td>6</td>
<td>6</td>
<td>0.8 hrs</td>
<td>23</td>
<td>5</td>
<td>0.044</td>
</tr>
<tr>
<td>HYP-7</td>
<td>26</td>
<td>500</td>
<td>3.0 hrs</td>
<td>4537</td>
<td>27</td>
<td>0.476</td>
</tr>
<tr>
<td>HYP-8</td>
<td>22</td>
<td>100</td>
<td>4.8 hrs</td>
<td>1046</td>
<td>21</td>
<td>0.084</td>
</tr>
</tbody>
</table>

Table 1: Data set description, giving number of stations (Sn.), trains (Tns.), span of the reference timetable, total number of departure events, number of connecting sections (Con. Sec.), and the traffic density in events per resource per minute. Note that the number of resources is the sum of stations and connecting sections.
very high traffic networks with only four stations but a large number of trains. HYP-6, HYP-7, and HYP-8 are branching networks with HYP-7 having high traffic. In Table 1, the density reported is the total number of events occurring per minute per resource in the network.

5.2 Comparison With Baselines

The goal of our experiments is to test the reduction in ADD (defined in (1)) as a result of dividing the action set in any given state optimally into safe and unsafe labels. Considering this to be a binary classification problem, we pick one baseline rule $R_g$ which allows false negatives (marking an actually unsafe state as safe) and one rule $R_c$ which allows false positives (marking an actually safe state as unsafe). We note that the rules $R_g$ and $R_c$ correspond to ‘greedy’ [Prasad et al., 2021] and ‘critical first’ [Khadikar, 2017b] in their original forms. Briefly, the greedy algorithm marks a train movement as safe if it has at least two feasible forward moves. Critical-first is a sufficient condition for deadlock free movement, which allows a train to move ahead as long as it only stops in a resource where at least one additional track is free. For every compounded set of primitive moves, this results in a state where every resource has at least one free track.

The critical first algorithm further provides a ranking order when multiple train movements are marked as safe, based on the number of presently free tracks in the resource the train is currently occupying (criticality of the node). We use this logic to rank train moves among the set marked safe by each rule. The resulting ADD for all problem instances and algorithms is summarised in Table 2. We emphasise that only the safe action masking (by using $R_0$, $R_g$, or $R_c$) differs among the algorithms, and the remaining scheduling/rescheduling policy is the same. In order to generate statistical results, we generate perturbed versions of each instance by moving the entire journey of each train in the reference timetable by an amount picked uniformly at random in $[-30, 30]$ minutes. Refer to Prasad et al. [2021] for details of the perturbations.

From Table 2, we first confirm that both $R_c$ (sufficient) and $R_0$ (necessary and sufficient) conditions result in deadlock-free schedules for all instances. Second, $R_0$ performs significantly better than $R_c$ in all instances, in terms of schedule efficiency. This demonstrates the advantage of employing an optimal deadlock detection condition. $R_g$ outperforms $R_0$ in three instances, all of which have low traffic density (see Table 1). However, the ADD for $R_0$ is competitive even in these instances. On the other hand, $R_g$ deadlocks in instances with high traffic density, and hence in general would not be a suitable choice for real-time rescheduling.

5.3 Policy Improvement

As a second experiment, we consider the effect of optimal deadlock detection on the efficiency of resulting schedules, by performing policy improvement using roll-outs [Tesauro and Galperin, 1996; Agarwal, 2022]. Under policy improvement, a base schedule is progressively improved by updating each action to one that minimises delay when followed by a roll-out policy. In Table 3, we start with the schedule produced by $R_0$ in Table 2 for all three algorithms (for a fair comparison). For every decision taken in the sequence, we roll out the individual trajectories for all alternative actions which are also marked as safe by the relevant rule. We choose the schedule with the least ADD out of these results, move to the next decision in the sequence, and repeat. The results in Table 3 show that the rollouts using $R_0$ (which provides the maximal set of feasible actions) are predominantly more effective than those using $R_g$ and $R_c$. In some cases, $R_c$ and $R_g$ are unable to improve on the baseline schedule given by $R_0$, while $R_0$ results in improvement over Table 2 in all instances.

6 Conclusion

In this paper, we show that in contrast to the accepted characterisation of railway scheduling in the literature, a polynomial-time deadlock detection method from the resource allocation literature applies to a large class of (re)scheduling problems. Our version of the implementation is in fact linear-time for arbitrary network topology, so long as each resource (station or connecting section) has at least two tracks. Further, we show that under a mild assumption (availability of a sequence of moves to bring all trains in single-track resources to a multi-track resource), we can also handle scheduling in the single-track scenario. Our empirical results show that using an optimal deadlock detection strategy significantly improves scheduling efficiency, in addition to providing feasibility guarantees. One important open question for the future is to evaluate the usefulness of optimal action masking while training data-driven scheduling policies.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Critical First ($R_c$)</th>
<th>Greedy ($R_g$)</th>
<th>$R_0$</th>
<th>Rollout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajmer</td>
<td>4.76 ± 0.07</td>
<td>4.19 ± 0.08</td>
<td>4.12 ± 0.09</td>
<td>3.43 ± 0.04</td>
</tr>
<tr>
<td>Kanpur</td>
<td>1.35 ± 0.07</td>
<td>3.57 ± 0.09</td>
<td>1.29 ± 0.05</td>
<td>3.79 ± 0.05</td>
</tr>
<tr>
<td>Konkan</td>
<td>60.33 ± 0.09</td>
<td>42.22 ± 0.59</td>
<td>42.60 ± 0.51</td>
<td>42.59 ± 0.51</td>
</tr>
<tr>
<td>HYP-1</td>
<td>19.30 ± 1.74</td>
<td>16.22 ± 1.29</td>
<td>16.49 ± 1.30</td>
<td>16.18 ± 1.28</td>
</tr>
<tr>
<td>HYP-2</td>
<td>6.33 ± 0.33</td>
<td>4.29 ± 0.21</td>
<td>4.31 ± 0.21</td>
<td>4.19 ± 0.18</td>
</tr>
<tr>
<td>HYP-3</td>
<td>7.02 ± 0.91</td>
<td>5.01 ± 0.16</td>
<td>0.83 ± 0.07</td>
<td>0.82 ± 0.07</td>
</tr>
<tr>
<td>HYP-4</td>
<td>177.73 ± 12.79</td>
<td>deadlock</td>
<td>1170.11 ± 2.46</td>
<td>1170.11 ± 2.46</td>
</tr>
<tr>
<td>HYP-5</td>
<td>654.9 ± 17.26</td>
<td>568.07 ± 2.71</td>
<td>524.23 ± 2.39</td>
<td>524.23 ± 2.39</td>
</tr>
<tr>
<td>HYP-6</td>
<td>12.07 ± 1.43</td>
<td>deadlock</td>
<td>6.42 ± 1.20</td>
<td>5.90 ± 1.01</td>
</tr>
<tr>
<td>HYP-7</td>
<td>1478.88 ± 22.11</td>
<td>deadlock</td>
<td>1228.30 ± 1.61</td>
<td>1228.30 ± 1.61</td>
</tr>
<tr>
<td>HYP-8</td>
<td>776.96 ± 27.74</td>
<td>215.74 ± 2.04</td>
<td>169.48 ± 1.90</td>
<td>169.48 ± 1.90</td>
</tr>
</tbody>
</table>

Table 2: ADD values in minutes with their standard error averaged over 10 perturbed versions of the reference timetables.

<table>
<thead>
<tr>
<th>Instance</th>
<th>$R_0$ Rollout</th>
<th>$R_c$ Rollout</th>
<th>$R_g$ Rollout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ajmer</td>
<td>3.43 ± 0.04</td>
<td>3.79 ± 0.05</td>
<td>3.43 ± 0.07</td>
</tr>
<tr>
<td>Kanpur</td>
<td>1.29 ± 0.05</td>
<td>1.13 ± 0.05</td>
<td>1.09 ± 0.05</td>
</tr>
<tr>
<td>Konkan</td>
<td>39.67 ± 0.57</td>
<td>42.59 ± 0.51</td>
<td>40.24 ± 0.61</td>
</tr>
<tr>
<td>HYP-1</td>
<td>15.75 ± 1.26</td>
<td>16.18 ± 1.28</td>
<td>15.74 ± 1.26</td>
</tr>
<tr>
<td>HYP-2</td>
<td>3.99 ± 0.24</td>
<td>4.19 ± 0.18</td>
<td>4.08 ± 0.20</td>
</tr>
<tr>
<td>HYP-3</td>
<td>0.83 ± 0.07</td>
<td>0.82 ± 0.07</td>
<td>0.74 ± 0.06</td>
</tr>
<tr>
<td>HYP-4</td>
<td>1170.11 ± 2.46</td>
<td>1170.11 ± 2.46</td>
<td>1166.26 ± 2.49</td>
</tr>
<tr>
<td>HYP-5</td>
<td>524.23 ± 2.39</td>
<td>524.23 ± 2.39</td>
<td>517.23 ± 2.09</td>
</tr>
<tr>
<td>HYP-6</td>
<td>6.35 ± 1.22</td>
<td>5.90 ± 1.01</td>
<td>4.20 ± 0.62</td>
</tr>
<tr>
<td>HYP-7</td>
<td>1215.73 ± 1.57</td>
<td>1228.31 ± 1.61</td>
<td>1225.99 ± 1.36</td>
</tr>
<tr>
<td>HYP-8</td>
<td>169.48 ± 1.90</td>
<td>169.48 ± 1.90</td>
<td>165.79 ± 2.42</td>
</tr>
</tbody>
</table>

Table 3: Policy improvement starting with the baseline schedule produced by $R_0$ in (the last column of) Table 2.
Acknowledgements
We thank Spyros Reveliotis for sharing full versions of relevant literature that was otherwise not available. We thank Abhiram Ranade from IIT Bombay for informative discussions, and Shripad Salsingkar from TCS for providing railway data.

References


