Chapter 1

Introduction and background

1.1 Introduction

Given the ongoing advances in self-driving vehicles and wireless communication, it is very natural to study how these new technologies can be applied to enable network-wide traffic coordination. Some of the potential benefits of coordinating the motion of groups of automated vehicles are increased network throughput, reduced energy consumption and better guarantees on safety in terms of avoiding dangerous situations.

Coordination of automated vehicles with communication has been studied at various levels of organization [2]. A good example of a local coordination methods is platooning of vehicles, where the aim is to lower energy consumption by reducing aerodynamic resistance. It has been shown that platooning can also result in a more efficient use of intersections. On a larger scale, methods like dynamic route optimization have been proposed to reduce travel delay for all vehicles in the network. The coordination problem has very many aspects that could be modeled and analyzed. For example, one may think of heterogenous vehicles—in terms of dynamics or priority—different models of centralized/decentralized communication between vehicles or with the infrastructure, under different guarantees on reliability; complex road topology, curved lanes, merging lanes.

However, as we will see, even the most basic models already present fundamental challenges in ensuring safety and efficiency. Therefore, we will only consider the two most essential elements in this thesis, being vehicle dynamics and the constraints that are required to model the allowed paths vehicles may take in order to avoid collisions with the infrastructure and other vehicles. To keep things simple, we assume that all vehicles are automated and share the same dynamics. Each vehicle follows a fixed route through the network and is centrally controlled through acceleration inputs under the assumption of perfect communication. With these assumptions, the traffic coordination task can be modeled a trajectory optimization problem. To illustrate the kind of model that we will study, we present a concrete minimal example of a single intersection with two vehicles.

Intersection model. Consider two vehicles, modeled as rigid bodies of width W and length L, each driving on its own straight lane of width W and infinite length. Suppose that

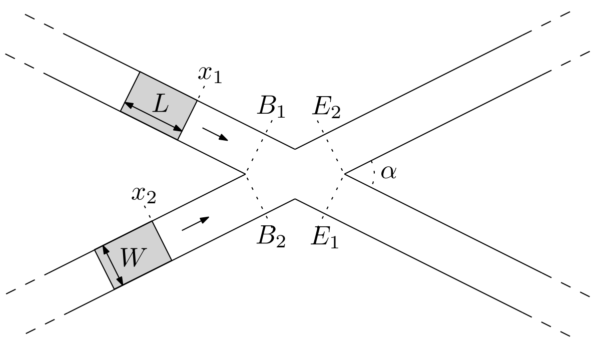

Figure 1.1: Example of two lanes intersection intersecting at some angle α. The indicated positions B_i and E_i for each vehicle i are such that if the front bumper position x_i satisfies either $x_i \leq B_i$ or $x_i - L \geq E_i$, then the intersection area is guaranteed to be completely free.

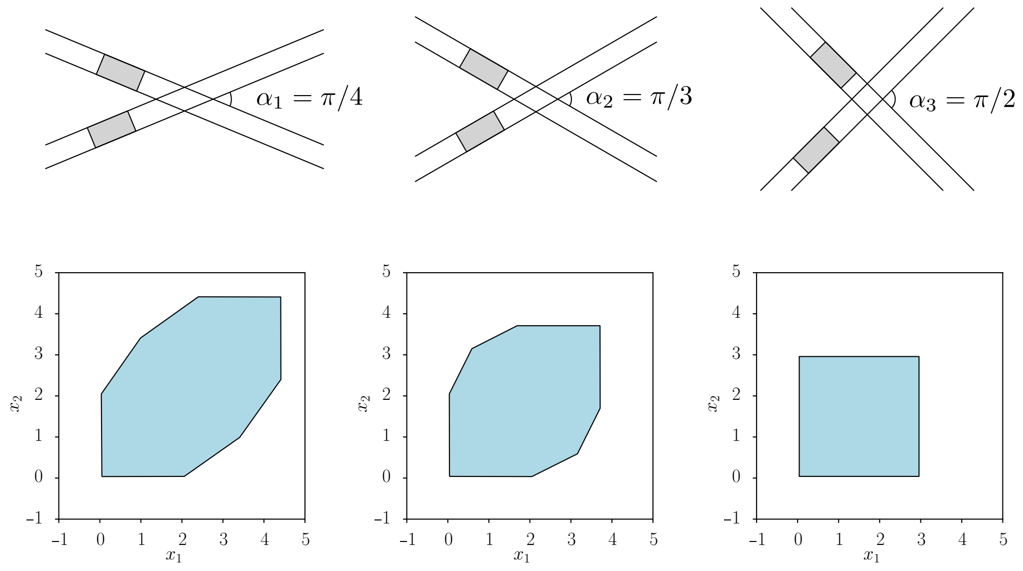

Figure 1.2: The first row shows three instances of the intersection model with for three different angles α_k and fixed vehicle dimenions $W = 1$ and $L = 2$. Below each model, we plotted the region in configuration space space corresponding to collisions, marked in blue. We briefly explain our method to compute these plots in Appendix A.

the lanes intersect at some angle α, as shown in Figure 1.1. To uniquely define the position of each vehicle $i \in \{1, 2\}$ on its lane, we use $x_i \in \mathbb{R}$ to denote the position of its front bumper. Hence, $x_i - L$ is the position of the rear bumper.

We might try to characterize the set $\mathcal{X} \subset \mathbb{R}^2$ of all feasible configurations (x_1, x_2) for which the two vehicles do not intersect. Fix the positions $x_i = B_i$ and $x_i = E_i$, as shown in Figure 1.1. Whenever we have $x_i \leq B_i$ or $x_i - L \geq E_i$, it is clear that vehicle i does not occupy the intersection at all, so the other vehicle j is free to take any position $x_j \in \mathbb{R}$. Therefore, whenever $x_1 \notin (B_1, E_1)$ and $x_2 \notin (B_2, E_2)$, we know for sure that (x_1, x_2) is a feasible configuration. Hence, we readily obtain some subset of feasible configurations

$$\mathbb{R}^2 \setminus ((B_1, E_1) \times (B_2, E_2)) \subset \mathcal{X}. \tag{1.1}$$

In general, the set of feasible configurations is a little larger, as illustrated by the three examples in Figure 1.2. In case of the third example, when the intersections make a right angle, it can be shown that there is equality in (1.1), which makes subsequent analysis easier.

We now discuss the dynamics of the vehicles, so let $x_i(t)$ denote the position of vehicle i at time t. Let $\dot{x}_i$ and $\ddot{x}_i$ denote the speed and acceleration of vehicle i, respectively, and consider the following bounds

$$\dot{x}_i(t) \in [0, 1], \qquad \ddot{x}_i(t) \in [-\bar{\omega}, \omega], \tag{1.2}$$

for some positive $\bar{\omega}, \omega > 0$. Assume the initial vehicle state $(x_i(0), \dot{x}_i(0)) = (x_i^0, \dot{x}_i^0)$ is fixed.

Let $J(x_i)$ be some functional that measures how desirable trajectory $x_i(t)$ is. Given some final simulation time $T > 0$, we consider the problem of maximizing $J(x_1) + J(x_2)$ such that $(x_1(t), x_2(t)) \in \mathcal{X}$ and the bounds (1.2) hold for all times $t \in [0, T]$. It is easy to see that this optimization problem is non-convex, because the space of feasible configurations $\mathcal{X}$ is non-convex. An interpretation of this non-convexity is that we must decide which of the vehicles crosses the intersection first.

Crossing time scheduling. As the example shows, we need to take a discrete decision regarding the crossing order of vehicles at intersections. This remains true for any generalization of the model to multiple lanes and intersections. However, after fixing these ordering decisions, the remaining problem is often much easier to solve. This observation motivates the decomposition of the trajectory optimization problem into two parts. The upper-level problem determines the times at which vehicles cross the intersections on their routes, to which we will refer as *crossing times*. Once these are fixed, we solve a set of lower-level problems to find the corresponding vehicle trajectories satisfying the crossing times.

Without additional assumptions, the upper-level problem is still as difficult as before, because the feasibility of a crossing time schedule may depend on the feasibility of the lower-level trajectory optimization problems in a non-trivial way. We will provide assumptions under which this coupling becomes particularly simple. Specifically, we show that feasibility of the lower-level problems can be states as a system of linear inequalities in terms of the crossing times and we will see that they also have simple interpretations. This allows us to formulate the upper-level problem as a mixed-integer linear problem that looks very similar to the classical job shop scheduling problem.

Learning to schedule. Because the lower-level problems can in general be solved efficiently, the original trajectory optimization problem is essentially reduced to a scheduling problem. This enables us to explore some applictions of recent machine learning techniques for such problems. Specifically, the solution of a scheduling problem can be interpreted as a sequence of decisions. Instead of manually trying to develop good heuristics and algorithms, we try to learn what optimal solutions are, by treating it as a learning task on sequences.

1.1.1 Related work

We briefly survey some releated work that addressed coordination of autonomous vehicles in a similar setting. A good example of an early centralized approach is the "Autonomous Intersection Management" (AIM) paper [14], which is based on a reservation scheme. The conflict zone is modeled as a grid of cells. Vehicles that want to cross the intersection send a request to the central controller to occupy the cells containing its trajectory for a certain amount of time. The central controller then decides to grant or deny these requests based on previous granted requests, in order to facilitate collision-free trajectories. If a request is denied, the vehicle slows down and attempts to obtain a new reservation after some timeout.

Direct transcription. Optimal control problems can be approached in an end-to-end fashion by *direct transcription* to an equivalent mixed-integer optimization problem, which can be solved using off-the-shelf solvers (e.g., SCIP [15] or Gurobi [5]). Such methods can be used to compute optimal trajectories up to any precision, by choosing a fine enough time discretization. However, it is exactly this time discretization that causes prohibitive growth of the number of variables with respect to the size of the network and the number of vehicles, so this method is only useful for relatively small problem instances. Therefore, approximation schemes have been studied in previous works [16, 17, 18], which we will review next.

Decomposition methods. The approximation method in [16] is based on a bilevel decomposition and considers a quadratic objective involving velocity as a proxy for energy. The first stage optimizes a schedule of vehicle crossing times. It uses approximations of each vehicle's contribution to the total objective as a function of its crossing time. Next, for each vehicle, the second stage computes an optimal trajectory satisfying the crossing time schedule, by solving a quadratic program. This approach has been shown to reduce running times significantly. Unfortunately, the study is limited to a single intersection and it is assumed that each lane approaching the intersection contains exactly one vehicle. The paper [17] proposes a trajectory optimization scheme for a single intersection, also based on the bilevel decomposition. The lower-level problem is employed to maximize the speed at which vehicles enter the intersection. Both parts of the problem are solved in an alternating fashion, each time updating the constraints of the other part based on the current solution.

The optimization scheme in [18] deals explicitly with the complexity of the crossing order decisions by defining groups of consecutive vehicles on the same lane. The first step is to group vehicles into these so-called “bubbles”. All vehicles in a bubble are required to cross the intersection together, while maintaining feasibility with respect to safe trajectories. Next, crossing times are assigned to bubbles while avoiding collisions. Based on this schedule, a local vehicular control method [19] is used that guarantees safety to reach the assigned crossing times.

1.1.2 Contributions and outline

This thesis is centered around the following two main contributions:

(i). Decomposition. Our first contribution is to show that, under certain conditions, our joint trajectory optimization problem for vehicles in a network of intersections decomposes into an upper-level crossing time scheduling problem and a set of lower-level trajectory optimization problems. We show that feasibility of the upper-level scheduling problem is completely characterized in terms of a system of linear inequalities involving the crossing times. This allows us to first solve the scheduling problem and then generate trajectories for it once we have the optimal crossing time schedule.

(ii). Learning to schedule. Our second contribution is an illustration of how machine learning techniques can be applied to solve scheduling problems in practice. Many practical instances contain structure that classic solutions techniques try to exploit, e.g., by defining smart heuristics based on human intuition and experience. We aim to automated this manual endeavor by formulating parameteric sequence models to capture the conditional probability of optimal solutions, given a problem instance. As has been noted before, we confirm that the order of evaluation during inference matters a lot for the final solution quality.

Outline. The rest of this chapter discusses some preliminaries: we briefly discuss the job shop scheduling problem in, because our crossing time scheduling problem may be seen as an extension; we provide a brief overview of how machine learning methods can be applied to solve combinatorial optimization problems, with a focus on job shop scheduling. In Chapter 2, we consider a simple model of a single intersection, like the example above. After discussing the decomposition method, we present some classical solutions methods to solve the crossing time scheduling problem. We explain how the problem can be treated as a learning problem in Chapter 3. In order to generalize to a network of intersections, we need to precisely study the feasibility of trajectories in lanes of finite length, which is done in Chapter 4. The resulting scheduling problem is then subjected to a learning algorithm in Chapter 5. We provide some general discussion and pointers for further research in Chapter 6.

1.2 Job shop scheduling

The job shop model provides a mathematical framework to study systems where a given set of—possibly distinct—facilities must be shared among a number of heterogenous tasks over time. We begin by providing a fairly general definition of this model and then present a small example for a specific problem. Next, we introduce the disjunctive graph, which is a standard auxiliary representation of both problem instances and solutions. Finally, we briefly discuss simple heuristics and illustrate how job shop problems can be approached within the mixed-integer programming framework. For a comprehensive textbook treatment of job shop scheduling, we refer the reader to Chapter 7 of [3].

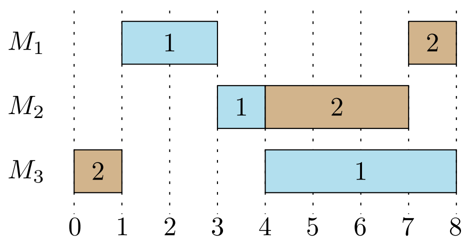

Figure 1.3: Example of an optimal schedule for Example 1, shown as a Gantt chart. Each row M_i corresponds to machine i and each block numbered j on this row represents the operation (i, j). The dashed lines indicate unit time steps. Note that machine 2 is kept idle, while operation $(2, 2)$ could have already been scheduled at time 1. Furthermore, for this particular instance, it can be checked that this is the unique optimal schedule.

General definition. Originally motivated by production planning problems, the job shop model is phrased in terms of a set of n jobs that require to be processed on a set of m machines. Each machine can process at most one job at the same time. We use the pair of indices (i, j) to identify the operation that machine i performs on job j, which takes a fixed amount of time $p(i, j)$. Each job j visits all machines[1] following a predetermined machine sequence, which may be different among jobs. Let $\mathcal{N}$ denote the set of all operations, then the general Job Shop Scheduling Problem (JSSP) is to determine a schedule $y = \{y(i, j) : (i, j) \in \mathcal{N}\}$ of starting times such that some objective function $J(y)$ is minimized. Variants of this basic problem can be obtained by specifying a concrete objective function and by introducing additional constraints, which we will both illustrate in the following example.

Example 1. Let s_j and e_j denote the first and last machine that job j vists, respectively. For each job j, we define a so-called release date $r(j)$ by requiring that $y(s_j, j) \geq r(j)$. As objective function, we consider the so-called makespan $J(y) := \max_j y(e_j, j) + p(e_j, j)$. The resulting problem is known as $Jm|r_j|C_{\max}$ in the commonly used three-field classification notation [4], see also Chapter 2 of [3]. Now consider a specific problem instance with $m = 3$ machines and $n = 2$ jobs. We specify the order in which jobs visit machines by providing the corresponding ordering of operations, which we choose to be $(1, 1) \to (2, 1) \to (3, 1)$ and $(3, 2) \to (2, 2) \to (1, 2)$. Using matrix notation $r(j) \equiv r_j$ and $p(i, j) \equiv p_{ij}$, the release dates and processing times are given by

$$r = \begin{pmatrix} 1 & 0 \end{pmatrix}, \qquad p = \begin{pmatrix} 2 & 1 \\ 1 & 3 \\ 4 & 1 \end{pmatrix}.$$

For this problem, Figure 1.3 shows an optimal schedule y^* with makespan $J(y^*) = 8$.

Disjunctive graph. A commonly used representation of job shop problems is through their disjunctive graph, which is a directed graph with vertices $\mathcal{N}$ corresponding to the operations and two types of arcs. The conjunctive arcs $\mathcal{C}$ are used to encode the predetermined machine sequence of each job. Each such arc $(i, j) \to (k, j)$ encodes that job j should first be processed on machine i before it is processed on machine k. When two distinct jobs j_1 and j_2 both require processing on the same machine i, we say that they are conflicting. The disjunctive arcs $\mathcal{D}$ are used to encode the possible choices of resolving such conflicts, by deciding which of j_1 or j_2 visits i first. More specifically, let j_1 and j_2 be conflicting on some machine i, then the nodes (i, j_1) and (i, j_2) are connected by two arcs in opposite directions.

[1]When some job j requires only processing on a proper subset of the machines, observe that we can simply assume that $p(i, j) = 0$ for each machine i that is not involved.

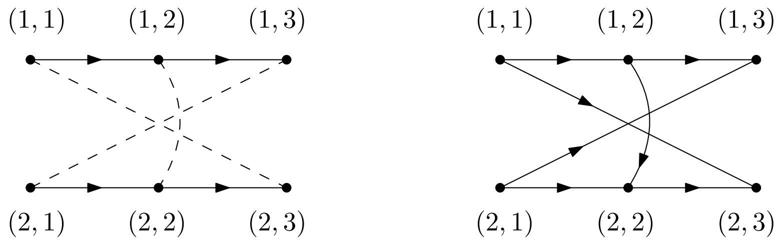

Figure 1.4: Illustration of disjunctive graphs for Example 1. Horizontal arrows represent conjunctive arcs. We used dashed lines to for the pairs of disjunctive arcs as dashed lines. The left graph corresponds to an empty selection $\mathcal{O} = \varnothing$ while the right graph shows the selection $\mathcal{O}$ that corresponds to the optimal schedule of Figure 1.3.

The disjunctive graph can also be used to encode (partial) solutions as follows. It can be shown that each feasible solution corresponds to a selection $\mathcal{O}$ of exactly one disjunctive arc from each pair such that the induced graph $(\mathcal{N}, \mathcal{C} \cup \mathcal{O})$ is acyclic [3]. More precisely, consider two conflicting operations (i, j_1) and (i, j_2), then $\mathcal{O}$ contains either $(i, j_1) \to (i, j_2)$ or $(i, j_1) \to (i, j_2)$. To illustrate this, the empty and complete disjunctive graphs for the instance in Example 1 are shown in Figure 1.4.

Solution methods. Most job shop problems are very hard to solve. For example, the class of problems $Jm|r_j|C_{\max}$ considered in Example 1 is known to be NP-hard [4], even without release dates, which is denoted $Jm||C_{\max}$. As a consequence, much effort has gone into developing good heurstics. A type of heuristic that is often considered is to apply a so-called *dispatching rule* in order to build a schedule in a step-by-step fashion. At each step, the rule chooses some job from all jobs with remaining unscheduled operations and schedules this next operation at the earliest time possible, given the current schedule.

A more principled way of solving job shop problems relies on the mathematical programming framework. We illustrate this for the problem $Jm|r_j|C_{\max}$ of Example 1. Using the notation of the disjunctive graph, the problem can be concisely stated as

$$\begin{array}{lll}
\min\limits_{y} & J(y) & \\
\text{such that} & y(s_j, j) \leq r(j) & \text{for each job } j, \\
& y(i,j) + p(i,j) \leq y(r,k) & \text{for each conjunction } (i,j) \to (r,k) \in \mathcal{C}, \\
& \left.\begin{array}{c} y(i,j) + p(i,j) \leq y(i,k) \\ \text{or (not both)} \\ y(i,k) + p(i,k) \leq y(i,j) \end{array}\right\} & \text{for each disjunction } (i,j) \leftrightarrow (i,k) \in \mathcal{D}, \\
& y(i,j) \in \mathbb{R} & \text{for each operation } (i,j).
\end{array}$$

Note that this is almost an mixed-integer linear program (MILP). Let $M > 0$ be some sufficiently large number and introduce a binary decision variable $b_{(i,j)\leftrightarrow(i,k)} \in \{0, 1\}$ for each pair of disjunctive arcs, then the pair of disjunctive constraint can be rewritten to

$$\begin{aligned}
y(i,j) + p(i,j) &\leq y(i,k) + M b_{(i,j)\leftrightarrow(i,k)}, \\
y(i,k) + p(i,k) &\leq y(i,j) + M(1 - b_{(i,j)\leftrightarrow(i,k)}),
\end{aligned}$$

which is generally refered to as the *big-M method.* The resulting MILP can be solved by any off-the-shelf solver, e.g., we used the commerical Gurobi Optimizer software [5] for this thesis.

1.3 Reinforcement learning

For machine learning problems where data-collection is restricted in some way, the supervised learning paradigm, i.e., learning from labeled examples, is sometimes no longer appropriate or feasible. Very generally, the reinforcement learning paradigm can viewed as a generalization of supervised learning in which the data collection and selection process is not fixed anymore. The classical perspective is that of an *agent* that tries to maximize some cumulative *rewward* signal when interacting in some *environment*, which is formalized by the Markov Decision Process (MDP) model. We refer the reader to [6] for a friendly textbook introduction to RL from this perspective.

Problem definition. Consider finite sets of states $\mathcal{S}$ and actions $\mathcal{A}$. Given some current state s, the agent sends some action a to the environment, upon which it responds by providing some scalar reward signal r and transitions to state s', which happens with probablity $p(s', r|s, a)$. By fixing a policy π, which is a function $\pi(a|s)$ that gives the probablity of the agent choosing action a in state s, we obtain the induced *state Markov chain* with transition probabilities

$$\Pr(s \to s') = \sum_a \sum_r \pi(a|s) p(s', r|s, a).$$

Given some initial state distribution $h(s)$, we sample $S_0 \sim h(s)$ and use $S_0, S_1, S_2, \dots$ to denote some sample trajectory. Moreover, we can also consider a more fine-grained Markov chain by considering the sequence of states, actions and rewards

$$S_0, A_1, R_1, S_1, A_2, R_2, S_2, \dots,$$

in which which the state Markov chain is naturally embedded. Such a sample trajectory is also refered to as an *episode*. Let the corresponding *return* at step t be defined as

$$G_t = \sum_{k=t+1}^{\infty} R_k.$$

By marking a subset of states as being *final states*, we can consider finite episodes

$$S_0, A_1, R_1, S_1, A_2, R_2, S_2, \dots S_N,$$

by using the convention that final states return zero reward and transition to themselves almost surely. For finite episodes, the goal is to find a policy π that maximizes the expected return $\mathbb{E}[G_0]$.

Solution methods. Most classical methods to find such an optimal policy π can be categorized as either being value-based or policy-based. Value-based can be generally understood as producing some estimate $v(s)$ for the expected return $\mathbb{E}[G_0|S_0 = s]$. The optimal policy is then parameterized in terms of these estimates $v(s)$. In contrast, policy-based methods use a more direct parameterization of the policy space and often rely on some kind of gradient-based optimization. Specifically, let π_θ be some policy with parameters θ, then we aim to apply the gradient descent updating

$$\theta \leftarrow \theta - \alpha \nabla \mathbb{E}[G_0]$$

where α is refered to as the learning rate. However, in almost all interesting situations, it is infeasible to compute the gradient directly.

1.4 Neural combinatorial optimization

This section introduces the idea of applying a Machine Learning (ML) perspective on Combinatorial Optimization (CO) problems, which has gained a lot of attention recently. One of the key ideas in this line of research is to treat problem instances as data points and to use machine learning methods to approximately map them to corresponding optimal solutions [7].

Algorithm execution as MDPs. It is very natural to see the sequential decision-making process of any optimization algorithm in terms of the MDP framework, where the environment corresponds to the internal state of the algorithm. From this perspective, two main learning regimes can be distinguished. Methods like those based on the branch-and-bound framework are often computationally too expensive for practical purposes, so *learning to imitate* the decisions taken in these exact algorithms might provide us with fast approximations. In this approach, the ML model's performance is measured in terms of how similar the produced decisions are to the demonstrations provided by the expert. On the other hand, some problems do not even allow efficient exact methods, so it is interesting to study solution methods that *learn from experience*. An interesting feature of this direction is that it enables the algorithm to implicitly learn to exploit the hidden structure of the problems we want to solve.

Because neural networks are commonly used as encoder in these ML models for CO, we will refer to this new field as *Neural Combinatorial Optimization* (NCO). A wide range of classical combinatorial optimization problems has already been considered in this framework, so we briefly discuss the taxonomy used in the survey [8]. One distinguishing feature is whether existing off-the-shelf solvers are used or not. On the one hand, *principal* methods are based on a parameterized algorithm that is tuned to directly map instances to solutions, while *joint* methods integrate with existing off-the-shelf solvers in some way (see the survey [9] on integration with the branch-and-bound framework). An illustrative example of the latter category are the use of ML models for the branching heuristic or the selection of cutting planes in branch-and-cut algorithms [10]. The class of principal methods can be further divided into *construction* heuristics, which produce complete solutions by repeatedly extending partial solutions, and *improvement* heuristics, which aim at iteratively improving the current solution with some tunable search procedure.

Constraint satisfaction. A major challenge in NCO is constraint satisfaction. For example, solutions produced by neural construction policies need to satisfy the constraints of the original combinatorial problem. To this end, neural network components have been designed whose outputs satisfy some specific type of constraint, for example being a permutation of the input [11]. Constraints can also be enforced by the factorization of the mapping into repeated application of some policy. For example, in methods for TSP, a policy is defined that repeatedly selects the next node to visit. The constraint that nodes may only be visited once can be easily enforced by ignoring the visited nodes and taking the argmax among the model's probabilities for unvisited nodes.

Instead of enforcing constraints by developing some tailored model architecture, like construction and improvement heuristics, general methodologies have recently been explored for the problem of constraint satisfaction in neural networks. For example, the DC3 framework [12] employs two differentiable processes, completion and correction, to solve any violations of equality or inequality constraints, respectively. The more recent HardNet framework [13] uses a closed-form projection to map to feasible solutions under affine constraints and relies on a differentiable convex optimization solver (e.g., OptNet [**?**]) when general convex constraints are considered.

1.4.1 Neural job shop scheduling

Various NCO methods have already been studied for JSSP with makespan objective, of which we now highlight some works that illustrate some of the above classes of methods. A lot of the policies used in these works rely on some graph neural network architecture, which is why the survey [20] provides an overview based on this distinguishing feature.

Dispatching rules. A very natural approach to model JSSP in terms of an MDP is taken in [21], where a dispatching heuristic is defined in an environment based on discrete scheduling time steps. Every available job corresponds to a valid action and there is a so-called No-Op action to skip to the next time step. States are encoded by some manually designed features. They consider the makespan objective by proposing a dense reward based on how much idle time is introduced compared to the processing time of the job that is dispatched. In some situation, some action can be proved to be always optimal ("non-final prioritization"), in which case the policy is forced to take this action. Additionally, the authors design some rules for when the No-Op action is not allowed in order to prevent unnecessary idling of machines. The proposed method is evaluated on the widely used Taillard [22] and Demirkol [23] benchmarks, for which performance is compared to static dispatching rules and a constraint programming (CP) solver, which is considered cutting-edge.

From a scheduling theory perspective [3], it can be shown that optimal schedules are completely characterized by the order of operations for regular objectives (non-decreasing functions of the completion times). The start times are computed from this order by a so-called *placement rule*, so considering discrete time steps introduces unnecessary model redundancy.

The seminal "Learning to Dispatch" (L2D) paper [24] proposes a construction heuristic for JSSP with makespan objective. Their method is based on a dispatching policy that is parameterized in terms of a graph neural network encoding of the disjunctive graph belonging to a partial solution. Again, each action corresponds to choosing for which job the next operation is dispatched. The rewards are based on how much the lower bound on the makespan changes between consecutive states. They use a Graph Isomorphism Network (GIN) architecture to parameterize both an actor and critic, which are trained using the Proximal Policy Optimization (PPO) algorithm. Using the Taillard and Demirkol benchmarks, they show that their model is able to generalize well to larger instances. As we already alluded to above, this way of modeling the environment is better suited to JSSP with regular objectives, because it does not explicitly determine starting times. They use a dispatching mechanism based on finding the earliest starting time of a job, even before already scheduled jobs, see their Figure 2. By doing this, they introduce symmetry in the environment: after operations O_{11}, O_{21}, O_{31} have been scheduled, both action sequences O_{22}, O_{32} and O_{32}, O_{22} lead to exactly the same state S_5 shown in their Figure 2. In this particular example, this means that it is impossible to have $O_{11} \to O_{22} \to O_{32}$. In general, it is not clear whether the resulting restricted policy is still sufficiently powerful, in the sense that an optimal operation order can always be constructed.

Guided local search. Recently, the authors of L2D investigated an improvement heuristic for JSSP [25] with makespan objective. This method is based on selecting a solution within the well-known N_5 neighborhood, which has been used in previous local search heuristics. It is still not clear whether their resulting policy is complete, in the sense that any operation order can be achieved by a sequence of neighborhood moves. The reward is defined in terms of how much the solution improves relative to the best solution seen so far (the "incumbent" solution). The policy is parameterized using a GIN architecture designed to capture the topological ordering of operations encoded in the disjunctive graph of solutions. They propose a custom n-step variant of the REINFORCE algorithm in order to deal with the sparse reward signal and long trajectories. To compute the starting times based on the operation order, they propose a dynamic programming algorithm, in terms of a message-passing scheme, as a more efficient alternative to the classical recursive critical path method. Our proposal for efficiently updating the current starting time lower bounds in partial solutions can also be understood as a similar message-passing scheme, but where only some messages are necessary.

Joint method. An example of a joint method is given in [26], where the environment is stated in terms of a Constraint Programming (CP) formulation. This allows the method to be trained using demonstration from an off-the-shelf CP solver.

Chapter 2

Isolated intersection scheduling

Chapter 3

Learning for the isolated intersection

3.1 Reinforcement learning

For some fixed policy π and initial state distribution h, we consider the underlying *induced Markov chain* over states. Because we are working with finite episodes, the induced state process is a Markov chain with absorbing states. We want to analyze how often states are visited on average, over multiple episodes. To better understand what *on average* means here, imagine that we link together separate episodes to create a regular Markov chain without absorbing states, in the following way: from each final state, we introduce state transitions to the initial states according to distribution h, see also Figure 3.1. Furthermore, we will write $S_t^{(i)}$ to denote the state at step t of episode i.

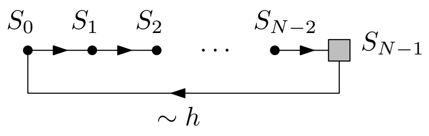

Figure 3.1: Illustration of the induced Markov chain when dealing with finite episodes. The next state after the final state, indicated as the grey rectangle, is sampled according to initial state distribution h.

Stationary distribution for finite episodes. Consider an absorbing Markov chain with transition matrix

$$P_{xy} = \sum_a \pi(a|x)p(y|x,a).$$

There are t transient states and r absorbing states, so P can be written as

$$P = \begin{pmatrix} Q & R \\ \mathbf{0} & I_r \end{pmatrix},$$

where Q is a t-by-t matrix, R is a nonzero t-by-r matrix, I_r is the r-by-r identify matrix and $\mathbf{0}$ is the zero matrix. Observe that $(Q^k)_{xs}$ is the probability of reaching state s in k steps without being absorbed, starting from state x. Hence, the expected number of visits to state s without being absorbed, starting from state x, is given by

$$\eta(s|x) := \sum_{k=0}^{\infty} (Q^k)_{xs}.$$

Writing this in matrix form $N_{xs} = \eta(s|x)$, we can use the following property of this so-called Neumann series, to obtain

$$N = \sum_{k=0}^{\infty} Q^k = (I_t - Q)^{-1}.$$

Now we can derive two equivalent equations

$$N = (I_t - Q)^{-1} \iff \begin{cases} N(I_t - Q) = I_t \iff N = I_t + NQ, & \text{or} \\ (I_t - Q)N = I_t \iff N = I_t + QN. \end{cases}$$

Expanding the first equation in terms of matrix entries $N_{xs} = \eta(s|x)$ gives

$$\begin{aligned}\eta(s|x) &= \mathbb{1}\{x = s\} + \sum_y \eta(y|x) Q_{ys} \\ &= \mathbb{1}\{x = s\} + \sum_y \eta(y|x) \sum_a \pi(a|y) p(y|x, a)\end{aligned}$$

and similarly, the second equation gives

$$\begin{aligned}\eta(s|x) &= \mathbb{1}\{x = s\} + \sum_y Q_{xy} \eta(s|y) \\ &= \mathbb{1}\{x = s\} + \sum_a \pi(a|x) \sum_y p(y|x, a) \eta(s|y)\end{aligned}$$

Now since the initial state is chosen according to distribution h, the expected number of visits $\eta(s)$ to state s in some episode is given by

$$\eta(s) = \sum_x h(x) \eta(s|x),$$

or written in matrix form $\eta = hN$, where η and h are row vectors. Therefore, we can also work with the equations

$$\begin{cases} hN = h + hNQ, & \text{or} \\ hN = h + hQN, \end{cases}$$

which are generally called *balance equations*. By writing the first variant as $\eta = h + \eta Q$ and expanding the matrix multiplication, we obtain

$$\eta(s) = h(s) + \sum_y \eta(y) \sum_a \pi(a|y) p(s|y, a).$$

Through appropriate normalization of the expected number of visits, we obtain the average fraction of time spent in state s, given by

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}.$$

Monte Carlo sampling. Suppose we have some function $f : \mathcal{S} \to \mathbb{R}$ over states and we are interested in estimating $\mathbb{E}_{S_t^{(i)} \sim \mu}[f(S_t^{(i)})]$. We can just take random samples of $S_t^{(i)}$, by sampling initial state $S_0^{(i)} \sim h$ and then *rolling out* π to obtain

$$\tau^{(i)} = (S_0^{(i)}, A_0^{(i)}, R_1^{(i)}, S_1^{(i)}, A_1^{(i)}, R_2^{(i)}, S_2^{(i)}, \dots, S_{N^{(i)}-1}^{(i)}) \sim \pi(\tau^{(i)}|S_0^{(i)}),$$

where $N^{(i)}$ denotes the total number of states visited in this episode. Given M such episode samples, we compute the estimate as

$$\mathbb{E}_{S_t^{(i)} \sim \mu}[f(S_t^{(i)})] \approx \left(\sum_{i=1}^{M} \sum_{t=0}^{N^{(i)}-1} f(S_t^{(i)}) \right) / \left(\sum_{i=1}^{M} N^{(i)} \right).$$

Observe that the analysis of the induced Markov chain can be extended to explicitly include actions and rewards as part of the state and derive the stationary distribution of the resulting Markov chain. However, we do not need this distribution explicitly in practice, because we can again use episode samples $\tau^{(i)}$. To keep notation concise, we will from now on denote this type of expectation as $\mathbb{E}_{\tau \sim h,\pi}[f(\tau)]$ and omit episode superscripts. Using this new notation, note that the average episode length is given by

$$\mathbb{E}_{h,\pi}[N] = \sum_{s'} \eta(s').$$

3.1.1 Policy gradient estimation

Let $v_{\pi_\theta} = \mathbb{E}_{h,\pi_\theta}[G_0]$ denote the expected episodic reward under policy π, where G_t is called the reward-to-go at step t, which is defined as

$$G_t := \sum_{k=t+1}^{\infty} R_k.$$

The main idea of policy gradient methods is to update the policy parameters θ in the direction that increases the expected episodic reward the most. This means that the policy parameters are updated as

$$\theta_{k+1} = \theta_k + \alpha \nabla v_{\pi_\theta},$$

where α is the learning rate and the gradient is with respect to θ. Instead of trying to derive or compute the gradient exactly, we often use some statistical estimate based on sampled episode. The basic policy gradient algorithm is to repeat the three steps

```
1. sample $M$ episodes $\tau^{(1)}, \ldots, \tau^{(M)}$ following $\pi_\theta$,
2. compute gradient estimate $\widehat{\nabla v_{\pi_\theta}}(\tau^{(1)}, \ldots, \tau^{(M)})$,
3. update $\theta \leftarrow \theta + \alpha \widehat{\nabla v_{\pi_\theta}}$.
```

REINFORCE estimator. We will now present the fundamental policy gradient theorem, which essentially provides a function f such that

$$\nabla v_{\pi_\theta} = \mathbb{E}_{\tau \sim h, \pi_\theta}[f(\tau)],$$

which allows us to estimate the policy gradient using episode samples. To align with the notation of [6], we write $\Pr(x \to s, k, \pi) := (Q^k)_{xs}$, for the probability of reaching state s in k steps under policy π, starting from state some x, so that the expected number of visits can also be written as

$$\eta(s) = \sum_x h(x) \sum_{k=0}^{\infty} \Pr(x \to s, k, \pi)$$

As proven in the chapter on policy gradient methods in [6], the gradient of the value function for a fixed initial state s_0 with respect to the parameters is given by

$$\nabla v_\pi(s_0) = \sum_s \sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \sum_a q_\pi(s,a) \nabla \pi(a|s). \tag{3.1}$$

When choosing the initial state s_0 according to some distribution $h(s_0)$, we verify that the final result is still the same as in [6]:

$$\begin{aligned}
\nabla v_\pi :=& \nabla \mathbb{E}_{s_0 \sim h}[v_\pi(s_0)] && \text{(3.2a)}\\
=& \sum_{s_0} h(s_0) \sum_s \sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \sum_a q_\pi(s,a) \nabla \pi(a|s) && \text{(3.2b)}\\
=& \sum_s \eta(s) \sum_a q_\pi(s,a) \nabla \pi(a|s) && \text{(3.2c)}\\
=& \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s) && \text{(3.2d)}\\
\propto& \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s), && \text{(3.2e)}
\end{aligned}$$

where the constant of proportionality is just the average episode length. Because we do not know μ or q_π explicitly, we would like to estimate ∇v_π based on samples. If we sample episodes according to h and π as explained above, we encounter states according to μ, so we have

$$\nabla v_\pi \propto \mathbb{E}_{h,\pi}\left[\sum_a q_\pi(S_t,a)\nabla\pi(a|S_t)\right] \tag{3.3a}$$

$$= \mathbb{E}_{h,\pi}\left[\sum_a \pi(a|S_t) q_\pi(S_t,a)\frac{\nabla\pi(a|S_t)}{\pi(a|S_t)}\right] \tag{3.3b}$$

$$= \mathbb{E}_{h,\pi}\left[q_\pi(S_t,A_t)\frac{\nabla\pi(A_t|S_t)}{\pi(A_t|S_t)}\right] \tag{3.3c}$$

$$= \mathbb{E}_{h,\pi}\left[G_t\nabla\log\pi(A_t|S_t)\right]. \tag{3.3d}$$

Baseline. Let $b(s)$ be some function of the state s only, then we have for any $s \in \mathcal{S}$

$$\sum_a b(s)\nabla\pi(a|s) = b(s)\nabla\sum_a \pi(a|s) = b(s)\nabla 1 = 0. \tag{3.4}$$

This yields the so-called REINFORCE estimate with *baseline*

$$\nabla v_\pi \propto \sum_s \mu(s)\sum_a (q_\pi(s,a)+b(s))\nabla\pi(a|s) \tag{3.5a}$$

$$= \mathbb{E}_{h,\pi}\left[\big(q_\pi(S_t,A_t)+b(S_t)\big)\nabla\log\pi(A_t|S_t)\right] \tag{3.5b}$$

$$= \mathbb{E}_{h,\pi}\left[\big(G_t+b(S_t)\big)\nabla\log\pi(A_t|S_t)\right]. \tag{3.5c}$$

Although estimates (3.3d) and (3.5c) are both equivalent in terms of their expected value, they may differ in higher moments, which is why an appropriate choice of b can make a lot of difference in how well the policy gradient algorithm converges to an optimal policy. As a specific baseline, consider the expected cumulative sum of rewards up to step the current step t, defined as

$$b(s) = \mathbb{E}_{h,\pi}\left[\sum_{k=1}^{t} R_k \middle| S_t = s\right], \tag{3.6}$$

then observe that

$$q_\pi(s,a)+b(s) = \mathbb{E}_{h,\pi}\left[\sum_{k=t+1}^{\infty} R_k \middle| S_t=s, A_t=a\right] + \mathbb{E}_{h,\pi}\left[\sum_{k=1}^{t} R_k \middle| S_t=s\right] \tag{3.7a}$$

$$= \mathbb{E}_{h,\pi}\left[\sum_{k=1}^{\infty} R_k \middle| S_t=s, A_t=a\right] \tag{3.7b}$$

$$= \mathbb{E}_{h,\pi}[G_0|S_t=s, A_t=a], \tag{3.7c}$$

which is just the expected total episodic reward. Now define function f to be

$$f(s,a) := (q_\pi(s,a)+b(s))\nabla\log\pi(a|s) = \mathbb{E}_{h,\pi}\left[G_0|S_t=s,A_t=a\right]\nabla\log\pi(a|s) \tag{3.8a}$$

$$= \mathbb{E}_{h,\pi}\left[G_0\nabla\log\pi(a|s)|S_t=s,A_t=a\right], \tag{3.8b}$$

then applying the law of total expectation yields

$$\nabla v_\pi \propto \mathbb{E}_{h,\pi}[f(S_t,A_t)] = \mathbb{E}_{h,\pi}\left[G_0\nabla\log\pi(A_t|S_t)\right]. \tag{3.9}$$

Chapter 4

Network scheduling

Chapter 5

Learning for network scheduling

Chapter 6

Conclusion and discussion

Appendix A

Feasible configurations for single intersection model

We provide a possible characterization of the configuration space of the two lanes that intersect at some arbitrary angle, as shown in Figure 1.1. Assume that $\alpha < \pi/2$ is the acute angle between the two intersections. We skip a thorough derivation of the following expressions, but we note that it is based on the type of the distances illustrated in Figure A.1. Roughly speaking, we encode the part of the intersection that vehicle $i = 1$ occupies in terms of x_2 coordinates, by defining the following upper and lower limit positions

$$U_1(x_1) := \begin{cases} -\infty & \text{if } x_1 \leq B_1 \text{ or } x_1 - L \geq E_1, \\ B_2 + (x_1 - B_1)/\cos(\alpha) & \text{if } x_1 \in (B_1, B_1 + c], \\ E_2 + (x_1 - E_1) \cdot \cos(\alpha) & \text{if } x_1 \in [B_1 + c, E_1), \\ E_2 & \text{if } x_1 \geq E_1 \text{ and } x_1 - L < E_1, \end{cases} \tag{A.1}$$

$$L_1(x_1) := \begin{cases} B_2 & \text{if } x_1 - L \leq B_1 \text{ and } x_1 > B_1, \\ B_2 + (x_1 - L - B_1)/\cos(\alpha) & \text{if } x_1 - L \in (B_1, E_1 - c], \\ E_2 + (x_1 - L - E_1) \cdot \cos(\alpha) & \text{if } x_1 - L \in [E_1 - c, E_1), \\ \infty & \text{if } x_1 - L \geq E_1 \text{ or } x_1 \leq B_1. \end{cases} \tag{A.2}$$

With this definition, x_2 must satisfy either $x_2 \leq L_1(x_1)$ or $x_2 - L \geq U_1(x_1)$. By symmetry of the system, we define $U_2(x_2)$ and $L_2(x_2)$ by swapping all indices in the above two expressions and obtain the condition that x_1 must satisfy either $x_1 \leq L_2(x_2)$ or $x_1 - L \geq U_2(x_2)$. These two pairs of equations completely characterize the configuration space, which can now be written as

$$\mathcal{X} = \{(x_1, x_2) \in \mathbb{R}^2 : x_1 \notin (L_2(x_2), U_2(x_2)) \text{ and } x_2 \notin (L_1(x_1), U_1(x_1))\} \tag{A.3}$$

In case the lanes intersect at a right angle $\alpha = \pi/2$, the situation is much simpler and the two limiting positions are simply given by

$$(L_1, U_1) = \begin{cases} (B_2, E_2) & \text{if } (x_1 - L, x_1) \cap (B_1, E_1) \neq \varnothing, \\ (\infty, -\infty) & \text{otherwise,} \end{cases} \tag{A.4}$$

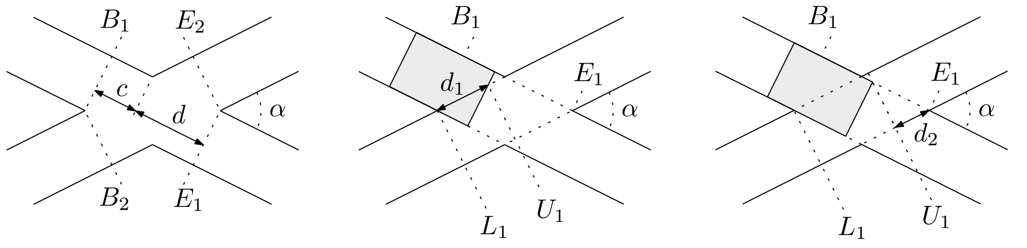

Figure A.1: Sketches to derive the configuration space of the intersection lanes. Using some basic trigonometry, the distances in the first figure can be shown to be $c = W/\tan(\alpha)$ and $d = W/\sin(\alpha)$. Furthermore, observe that we have $(x_1 - B_1)/d_1 = \cos(\alpha)$ for $x_1 \in (B_1, B_1 + c]$ as shown in the left figures and $d_2/(E_1 - x_1) = \cos(\alpha)$ for $x_1 \in [B_1 + c, E_1)$ as shown in the right figure. These two types of distances can be used to derive the full characterization.

such that the configuration space is simply given by

$$\mathcal{X} = \mathbb{R}^2 \setminus ((B_1, E_1) \times (B_2, E_2)). \tag{A.5}$$

Bibliography

[1] Daniel J. Fagnant and Kara Kockelman. Preparing a nation for autonomous vehicles: Opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice*, 77:167–181, July 2015.

[2] Stefano Mariani, Giacomo Cabri, and Franco Zambonelli. Coordination of Autonomous Vehicles: Taxonomy and Survey. *ACM Computing Surveys*, 54(1):1–33, January 2022.

[3] Michael L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer International Publishing, Cham, 2016.

[4] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. In P. L. Hammer, E. L. Johnson, and B. H. Korte, editors, *Annals of Discrete Mathematics*, volume 5 of *Discrete Optimization II*, pages 287–326. Elsevier, January 1979.

[5] Gurobi Optimization, LLC. Gurobi optimizer reference manual, 2024.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[7] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine Learning for Combinatorial Optimization: A Methodological Tour d'Horizon, March 2020.

[8] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement Learning for Combinatorial Optimization: A Survey, December 2020.

[9] Andrea Lodi and Giulia Zarpellon. On learning and branching: A survey. *TOP*, 25(2):207–236, July 2017.

[10] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement Learning for Integer Programming: Learning to Cut, July 2020.

[11] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer Networks, January 2017.

[12] Priya L. Donti, David Rolnick, and J. Zico Kolter. DC3: A learning method for optimization with hard constraints, April 2021.

[13] Youngjae Min, Anoopkumar Sonar, and Navid Azizan. Hard-Constrained Neural Networks with Universal Approximation Guarantees, October 2024.

[14] K. Dresner and P. Stone. A Multiagent Approach to Autonomous Intersection Management. *Journal of Artificial Intelligence Research*, 31:591–656, March 2008.

[15] Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Lixing Xu. The SCIP optimization suite 9.0. Technical Report, Optimization Online, February 2024.

[16] Robert Hult, Gabriel R. Campos, Paolo Falcone, and Henk Wymeersch. An approximate solution to the optimal coordination problem for autonomous vehicles at intersections. In *2015 American Control Conference (ACC)*, pages 763–768, Chicago, IL, USA, July 2015. IEEE.

[17] Weiming Zhao, Ronghui Liu, and Dong Ngoduy. A bilevel programming model for autonomous intersection control and trajectory planning. *Transportmetrica A: Transport Science*, 17(1):34–58, January 2021.

[18] Pavankumar Tallapragada and Jorge Cortés. Hierarchical-distributed optimized coordination of intersection traffic, January 2017.

[19] Pavankumar Tallapragada and Jorge Cortes. Distributed control of vehicle strings under finite-time and safety specifications, July 2017.

[20] Igor G. Smit, Jianan Zhou, Robbert Reijnen, Yaoxin Wu, Jian Chen, Cong Zhang, Zaharah Bukhsh, Wim Nuijten, and Yingqian Zhang. Graph Neural Networks for Job Shop Scheduling Problems: A Survey, 2024.

[21] Pierre Tassel, Martin Gebser, and Konstantin Schekotihin. A Reinforcement Learning Environment For Job-Shop Scheduling, April 2021.

[22] Éric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285, 1993.

[23] Ebru Demirkol, Sanjay Mehta, and Reha Uzsoy. Benchmarks for shop scheduling problems. *European Journal of Operational Research*, 109(1):137–141, 1998.

[24] Cong Zhang, Wen Song, Zhiguang Cao, Jie Zhang, Puay Siew Tan, and Chi Xu. Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning, October 2020.

[25] Cong Zhang, Zhiguang Cao, Wen Song, Yaoxin Wu, and Jie Zhang. Deep Reinforcement Learning Guided Improvement Heuristic for Job Shop Scheduling, February 2024.

[26] Pierre Tassel, Martin Gebser, and Konstantin Schekotihin. An End-to-End Reinforcement Learning Approach for Job-Shop Scheduling Problems Based on Constraint Programming, June 2023.