Humanized Computing and Communication
Vol. 2, No. 1 (2021) 46-67

DOI: 10.35708/HCC1869-148367

Extending the learning and communicating shout-ahead architecture with user-defined exception rules – a case study for traffic light controls

Christian Roatis and Jörg Denzinger

Department of Computer Science at University of Calgary, Calgary, Canada
{christian.roatis,denzinge}@ucalgary.ca

Received (12/15/2020)
Revised (02/05/2021)
Accepted (02/18/2021)

Abstract. We present an extension of the shout-ahead agent architecture that allows for adding human user-defined exception rules to the rules created by the hybrid learning approach for this architecture. The user-defined rules can be added after learning as reaction to weaknesses of the learned rules or learning can be performed with the user-defined rules already in place.
We applied the extended shout-ahead architecture and the associated learning to a new application area, cooperating controllers for the traffic lights of intersections. In our experimental evaluations, adding user-defined exception rules to the learned rules for several traffic flow instances increased the efficiency of the resulting controllers substantially compared to just using the learned rules. Performing learning with user-defined exception rules already in place decreased the learning time substantially for all flows, but had mixed results with respect to efficiency. We also evaluated user-defined exception rules for a variant of the architecture that is not using communication and saw similar effects as for the variant with communication. For the communicating version, both variants of adding user-defined exception rules create controllers that are much more flexible than what using the original shout-ahead architecture with its learning is able to create as indicated by experiments with variations of flows.

Keywords: learning behavior; shout-ahead architecture; traffic light controllers.

An earlier version of the paper was presented at the First IEEE International Conference on Humanized Computing and Communication with AI.

1 Introduction

Machine Learning (ML), especially using Neural Networks, has seen many successes in recent years and is now also used by businesses in their products or processes (see, for example, [13] or [11]). But these many experiences with Machine Learning have also revealed that many of the knowledge representations used by ML techniques are very difficult or even impossible to be understood by humans and therefore do not allow for corrections of the learned knowledge by human experts (neural networks, for example). There are knowledge representations that are easily understandable by humans, like rules (see, for example, [16]) or situation-action pairs (see [3]), but any changes a human makes to such a knowledge base requires still a very good understanding not only of the application area of the knowledge, but also the specifics of how a system using the knowledge does its reasoning and sometimes even how the ML technique used to create the knowledge works. And the more complex the system is, the greater is the need for a human wanting to improve it to have a deep understanding of all the areas mentioned above.

In this paper, we extend a rule-based agent architecture, the *shout-ahead architecture*, for communicating and cooperating agents that was developed to allow learning of the cooperative behavior of agents to also include rules provided by a human (expert). Shout-ahead uses two sets of weighted rules, one set to compute an intended action for the agent that it communicates to other agents and one set that also uses the intended actions communicated to it by the other agents to determine the action it really takes. Learning uses evolutionary learning to evolve these rule sets and reinforcement learning to update the rule weights (see [8]). Our extension adds to each of the two rule sets an additional rule set which takes precedent over the original sets, as a form of exception rules (see [14]). A human expert can provide such exception rules either before the learning takes place (and the learning process uses them already) or after learning is finished (to correct and/or enhance the learned rules).

We evaluated this extended shout-ahead architecture by instantiating it to learn control agents for the traffic lights of intersections. We used the SUMO simulator (see [2]) and different traffic flow scenarios for a given map of intersections. In our experiments, using the original shout-ahead was able to create better controllers than a reference controller from SUMO for several, but not all, flows. When adding to these learned controllers good exception rules, the resulting controllers were substantially better than the reference controller and the learned controllers without the exception rules. Having the exception rules already available during learning created better results for some flows, but not all of them. But for all flows it allowed for much faster learning than what the original architecture achieved. We have reported on this already in [12].

In this paper, we also evaluated a variant of our system where communication is disabled (having only the first rule set of shout-ahead and an exception rule set). Our experiments for this variant showed that the ability to shout-ahead intentions leads to much better results than not having this ability. But also for this variant we could observe that the use of exception rules creates better re-

sults than without using these rules. Experiments aimed at testing the flexibility of the created controllers (using shout-ahead) showed that both variants using exception rules are able to deal with variations of the flows much better and more consistently than what the original architecture was able to achieve.

This paper is organized as follows: After this introduction, in Section 2, we present the shout-ahead architecture and the approach for learning the rules for agents using it. In Section 3, we present the modifications to it that allow for additional user-defined rules. In Section 4, we present the instantiation of the modified architecture and learning to create behaviors for control agents for traffic lights. Then, in Section 5, we evaluate this instantiation for various traffic scenarios. Section 6 discusses related work and we conclude in Section 7 with remarks on future work.

2 The shout-ahead architecture

In this section, we will first present structure and decision making of agents using the shout-ahead architecture and then present the learning method that creates behaviors for such agents.

2.1 Shout-ahead-based agents

The shout-ahead architecture was developed as an agent architecture allowing for learning the behavior of an agent that communicates with other agents and takes these communications into account in its decision making (see [8]). As such, an agent $\mathcal{A}g$ acts in an environment $\mathcal{E}nv$ as part of a group of cooperating agents A and its actions are from a set Act. $\mathcal{A}g$ also has some internal data areas Dat in which it can store information and in the shout-ahead architecture an agent uses observations out of a set Obs to create conditions for rules. An element of Obs can be a predicate about $\mathcal{E}nv$ (as observable by $\mathcal{A}g$), about the current element of Dat or about other agents in A. The latter predicates can represent that an other agent intends to perform an action a (and naturally only one such predicate can be true for another agent at the current point in time, as described in [8]), but also that the other agent communicated having taken a particular action in some time interval in the past (this is a new possibility required for our new application, see Section 4). These predicates form the subset Obs_{int} in Obs.

There are two data areas in $\mathcal{A}g$ that contain sets of weighted rules, RS and RS_{int}. A weighted rule has the form

IF *cond* THEN a, w

where $cond = \{obs_1, ..., obs_m\}$ with $obs_i \in Obs$, $a \in Act$ and w the weight of the rule. A rule is applicable, if the current observable environment and the current values of the other data areas of an agent make all predicates in *cond* true. Rules in RS can only contain elements from $Obs \backslash Obs_{int}$ in their conditions, while rules in RS_{int} have only elements from some subset $Obs_{coop} \subseteq Obs$ with $Obs_{int} \subseteq Obs_{coop}$ in their conditions.

The decision function f_{Ag} of an agent uses RS and RS_{int} as follows: to compute $\mathcal{A}g$'s intended action it divides all rules in RS that are currently applicable into the sets $RS_{C_{max}}$ and $RS_{C_{rest}}$ such that for all rules $rl_j, rl_k \in RS_{C_{max}}$

$w_j = w_k$ and for all rules $rl_q \in RS_{C_{rest}}$ we have $w_j > w_q$, i.e. $RS_{C_{max}}$ contains the rules with maximal weight in RS_C (see [8]). We do this, so that an agent when learning is not just exploiting the best rule, but also explores other rules in its set. This is achieved by selecting the rule to apply probabilistically. The probability P of selecting a rule $rl \in RS$ with weight w is defined by

$$P(rl) = \begin{cases} (1-\epsilon) \times \frac{w}{w \times RS_{C_{max}}} & \text{if} \quad rl \in RS_{C_{max}} \\ \epsilon \times \frac{w}{\sum_{rl_j \in RS_{C_{rest}}} w_j} & \text{if} \quad rl \in RS_{C_{rest}} \end{cases}$$

The parameter ϵ, $0 \leq \epsilon \leq 1$, is used to determine the importance of doing exploration (the higher ϵ, the more exploration). Note that the probabilities of the rules in the two candidate sets each have to sum up to 1. If we have negative weights, then we first have to project the whole spectrum of rule weights into the interval $[\beta, 1]$ to achieve that (β is a parameter slightly bigger than 0 to avoid division by zero in P). This selection process leads us to a rule rl^{int} (with weight w^{int} and action a^{int}).

$\mathcal{A}g$ communicates the action of the rule selected above as its intention to all other agents in A and receives their intentions. It then uses the same process as above, but now applied to RS_{int}, to select a rule rl^{coop} (with weight w^{coop} and a^{coop}). As last step of the decision making process, $\mathcal{A}g$ has to select either rl^{int} or rl^{coop}. We again do this probabilistically (but also based on the weights). If $w^{coop} < w^{int}$, then $\mathcal{A}g$ performs a^{coop} with probability p_{coop}, and with probability $(1 - p_{coop})$ it performs a^{int}. If $w^{int} \leq w^{coop}$, then $\mathcal{A}g$ performs a^{coop}. Again, p_{coop} is a parameter, allowing us to determine how much an agent is influenced by the intentions of the other agents. In all of these steps, if no rules are applicable, then the action of the agent for that step is to do nothing.

2.2 Learning shout-ahead-based agents

The learning process for the shout-ahead architecture is a hybrid process using evolutionary learning of the rules and reinforcement learning in form of a modification of the Sarsa method (see [15]) for adjusting rule weights. Both kinds of learning are performed using so-called simulation runs that allow to evaluate the quality of the sets of rules for a group of agents and require some kind of simulation of the application area for the agents (if it is too dangerous or costly to perform the learning in the real world).

In contrast to many other evolutionary learning methods for cooperative behavior (like, for example, [3]), an individual for the evolutionary learning of behavior for shout-ahead agents is not a behavior for each element of A, but only for one particular type of agent. The individuals of a particular agent type are forming an agent pool (and the evolutionary learning evolves the agent pools). This means that in order to perform a simulation run, for each agent in the simulation (which we call a role) an individual from the appropriate agent pool needs to be chosen to form a so-called simulation team. During each simulation run, the individuals in the simulation team perform reinforcement learning and at the end of a simulation run for each individual agent a run fitness value is

computed. The fitness of an individual is then the average run fitness of it over all the simulation runs it took part in.

The reinforcement learning updates the weight w of a rule that is applied by the agent based on a reward r that is based on the quality of the situation that the agent finds itself in after applying the rule and, as usual in Sarsa, the weight w' of the rule that it enables to be applied in the new situation. Or, more formally,

$$w \leftarrow w + \alpha[r + \gamma w' - w]$$

where γ, $0 \leq \gamma \leq 1$, is the so-called discount rate and α, $0 \leq \alpha \leq 1$ is the learning factor. These parameters have to be chosen by the developer of the learning system. Sarsa was developed for the usual architecture of a situation-action matrix of Q-values. A rule is less fine-grained than a Q-value since it essentially covers several entries in a situation-action matrix, but [8] has shown that using the so-learned weights results in good behaviors of the agents (due to the fact that the entries in the situation-action matrix covered by one rule can overlap with the entries of another rule, so that the evolutionary process can create rules with overlaps that avoid bad behaviors).

The evolutionary learning of an agent pool follows the usual scheme of creating a new generation out of the old one by creating new individuals, in our case pairs of rule sets. Given two individuals (RS, RS_{int}) and (RS', RS'_{int}), a crossover creates the strategy $(RS^{new}, RS^{new}_{int})$ by selecting the $|RS|$ best rules (with regard to their weights) from $RS \cup RS'$ to create RS^{new} and the $|RS_{int}|$ best rules from $RS_{int} \cup RS'_{int}$ to create RS^{new}_{int} (see [8]). If any of the unions of rules initially contains duplicates, then the copy with the lower weight is mutated and the weight of this mutated rule is set to 0. The mutation of a rule IF *cond* THEN a results in changing *cond* to a $cond^{new}$ with a probability p^{cond} and a to an a^{new} with probability p^{act}. a^{new} is a random action from the set *Act* of $\mathcal{A}g$, while $cond^{new}$ is created by deleting a random set of elements from *cond* and adding another randomly chosen set of elements (see [8], again). Naturally, the new elements in $cond^{new}$ obey the requirements for the rule set the rule belongs to. Mutation is also used as stand-alone operator. To create the new generation, a certain number n_{off} of new individuals is created using crossover and mutation and then the n_{off} worst individuals (with regard to their fitness fit) of the old generation are deleted to make room for the new individuals.

The initial individuals in a pool are created randomly and the fitness fit of an individual is the average of the run fitnesses ($rfit$) the individual was part in. Each individual needs to be in a minimal number of simulation runs, but, in order to have enough runs for all individuals in all pools, it can be in more than this minimal number of runs. The fitness of an individual that survives into the next generation is taken over, so that good individuals get more and more chances to be evaluated (and also the weights of their rules are based on more and more simulation runs).

3 The extended shout-ahead architecture

In this section, we present an extension of the shout-ahead architecture that allows for user-defined exception rules. We will describe the new architecture and how the learning is still possible.

Extending the shout-ahead architecture to be able to use user-defined exception rules is not very difficult. In addition to the existing rule sets, we have two additional rule sets, RS^{exp} and RS^{exp}_{int}, for the user-defined exception rules. As the symbols suggest, the conditions of all rules in RS^{exp} contain only elements from $Obs \backslash Obs_{int}$ (like RS) and the conditions of rules in RS^{exp}_{int} only elements from Obs_{coop} (like RS_{int})[1].

The new decision function f^{exp}_{Ag} modifies f_{Ag} in the following manner: if there are rules in $RS^{exp} \cup RS^{exp}_{int}$ that are currently applicable, then the agent selects among those rules the rule with highest weight and sends its action as intended action to the other agents. If due to the received intended actions from the other agents now additional rules from RS^{exp}_{int} are applicable and we have a new rule with heighest weight, then the agent performs the action of this rule. Otherwise it performs the action of the originally selected rule. If there are more than one rule with highest weight in any of these steps then f^{exp}_{Ag} randomly selects among the rules with highest weight. If no rules from $RS^{exp} \cup RS^{exp}_{int}$ are applicable at any of the steps, then the agent uses f_{Ag} as described before. This modification allows a user to use the rule weights of the exception rules to resolve any conflicts between rules (which are difficult to avoid). Obviously, choosing the same weights for some of the exception rules should not be done (and choosing then a random highest weight rule is intended to remind the human user that he or she was not clear enough in the definition of the rules).

Given the above, it is not necessary to modify the hybrid learning method, since user-defined exception rules are obviously *not* learned (and the modified architecture allows for learning with user-defined rules in place or without them). And with regard to the reinforcement learning component, the use of rule weights in the user-defined exception rules as described above should naturally not be diffused by modifying them via reinforcement learning.

4 Learning traffic light controllers

In this section, we first describe the new application area for the shout-ahead architecture, traffic light controllers within the SUMO simulator. We then present the instantiation of the architecture followed by the instantiation of the hybrid learning method.

4.1 The SUMO traffic simulator

Learning the behavior of the controllers of the traffic lights of intersections is naturally not something that can be done in the real world (although the application of these controllers can naturally be moved into the real world). Therefore

[1] It is also possible to only use one set of exception rules (see the definition of f^{exp}_{Ag}), but we think that having the two rule sets suggests to a user to think about the "normal" behavior without communication and also what he or she wants to get out of communication.

we created our environments for the agents in SUMO (Simulation of Urban MObility, see [2]). SUMO is an open source, microscopic traffic simulator, developed for the purpose of aiding vehicular transportation related research like ours. It allows users to create and manipulate road networks, traffic flows, traffic light control algorithms and much more. Additionally, SUMO offers a plugin called TraCI (Traffic Control Interface), which uses a TCP based client/server architecture to provide runtime access to road traffic simulations, allowing for the value retrieval and behaviour manipulation of simulated objects on line. TraCI, and by extension SUMO, allows interfacing from Python, meaning complex systems, like a traffic signal controller agent architecture, can be built to run in the background with inputs from a SUMO simulation, manipulating simulated objects based on its outputs.

In addition to allowing us to create environments (in the form of maps and traffic flows), using existing vehicle controllers, and observations about these environments, SUMO also already provides two control algorithms for the traffic lights of an intersection. One asserts that all traffic lights work on a fixed cycle, with a default cycle time of 90 seconds. That is, all possible phases in an intersection are completed within the 90 second interval before beginning again. The cycle time can be adjusted by the user. The other control algorithm built into SUMO involves "Actuated Traffic Light" (ATL) (see [2]). This algorithm supports gap-based actuated traffic control, which allows for phase cycle lengths to adapt to dynamic traffic conditions. SUMO's ATL algorithm uses set phase cycles like the first algorithm, but if a continuous traffic stream is detected at the end of a "normal" green phase, for instance, that phase is prolonged until a sufficient time-gap between vehicles is detected. This is similar to the more modern systems used around the world (see [7]). Therefore we will compare our learned controllers to ATL and also use it as a base case for normalizations needed by our learners.

4.2 Instantiating the extended shout-ahead architecture

To instantiate our extended shout-ahead architecture to traffic light controllers, we need to define the set *Act* of actions and the set *Obs* of observations. From the perspective of a single traffic light, the choices are very limited, namely changing from red to green, from green to yellow, from yellow to red and not doing anything. But an intersection naturally contains more than one traffic light, so that the possible actions from the perspective of an agent that is a controller for the whole intersection are more complicated and dependent on the particular type of intersection. We describe the possible actions as the light colors for each traffic light in the intersection that the agent wants to have and that are safe, resp. acceptable as defined when the intersection was designed and build. This light combinations often also are called the phases of the intersection and Figure 1 and Figure 2 show two different such phases for a standard 4-arm intersection with an indication what traffic flows through the intersection are

allowed in the phase (green lines)[2]. To this set of phases we added as action to do nothing, which means that the current light colors stay. A controller performs the decision what to do every 5 seconds. As already mentioned, if no rule is applicable, the action that will be taken is to do nothing, except that for this instantiation we have that there is a build-in maximal length for green and yellow phases in SUMO (of 225 and 5 seconds, respectively) which forces the appropriate change action on the intersection when these time limits are reached.

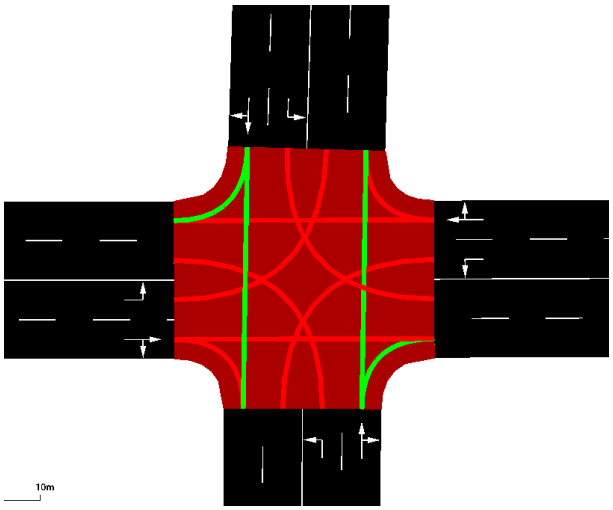

Fig. 1. One of the straight phases for an intersection

The set *Obs* is also rather complicated. We have predicates in *Obs* that purely deal with $\mathcal{E}nv$ and have chosen to make them sensor independent (i.e. regardless of the used concrete sensors, it should be possible to establish if such a predicate is fulfilled, using a more or less complicated computation). These predicates deal with the number of cars that are currently waiting at the intersection. At first glance, creating a predicate dealing with a number does not seem straightforward (given that a predicate is either true or false). Therefore we need to include particular numbers into the predicate and in order to not have too many of these predicates, we have to bin the numbers into intervals (and deciding on the bins obviously requires human input). So, instead of having a predicate NumCarsWaitStraight(e) (standing for number of cars waiting to proceed straight) in situation e, we have 8 predicates

NumCarsWaitStraight0(e),
NumCarsWaitStraight1-5(e),
NumCarsWaitStraight6-10(e),

[2] In our examples for rules in Section 5.2 we will use only as many light color indications as are necessary to identify a particular phase and the action to switch to it.

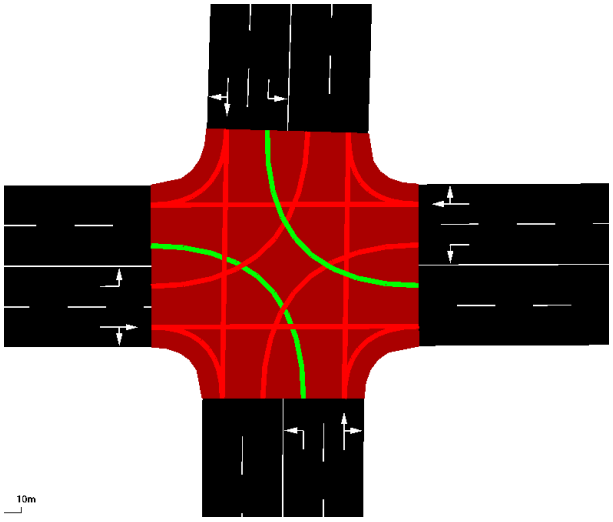

Fig. 2. One of the left turn phases for an intersection

NumCarsWaitStraight11-15(e),
NumCarsWaitStraight16-25(e),
NumCarsWaitStraight26-35(e),
NumCarsWaitStraight36-45(e), and
NumCarsWaitStraight46+(e).

Similarly, we have 7 predicates

NumCarsWaitLeft0(e),
NumCarsWaitLeft1-3(e),
NumCarsWaitLeft4-6(e),
NumCarsWaitLeft7-9(e),
NumCarsWaitLeft10-12(e),
NumCarsWaitLeft13-15(e), and
NumCarsWaitLeft15+(e)

representing the number of cars waiting at the intersection to turn left. We also have predicates concerned with the past of an intersection, which naturally means that information from the current value d of Dat is required. Again, we are interested in numbers, so that we do binning and create groups of predicates. These predicate groups are LongestWaitStraightINT(e,d), LongestWaitLeftINT(e,d) and TimeCurrentPhaseINT(e,d), where we replace INT with appropriate intervals (14 for each of them, one for 0 seconds, then between 0 and 15 seconds, for 15 to 30 seconds, 30 to 45 seconds, 45 to 60 seconds, 60 to 90 seconds, 90 to 120 seconds, 120 to 150 seconds, 150 to 180 seconds, 180 to 210 seconds, 210 to 240 seconds, 240 to 270 seconds, 270 to 300 seconds, and beyond 300 seconds).

We also have predicates that describe the state of the traffic lights of the intersection. Obviously, some of the traffic lights are connected, for example the lights for one direction of the traffic flow with the lights for the exact oppo-

site direction, and we therefore have only predicates for one of those connected lights. Each of these connected lights are assigned a direction (in our examples we used vertical Ver and horizontal Hor) and we naturally also had to split off the left turning cars, so that we have as predicates VerGreen(e), VerRed(e), VerYellow(e), VerLeftGreen(e), VerLeftRed(e), VerLeftYellow(e), HorGreen(e), HorRed(e), HorYellow(e), HorLeftGreen(e), HorLeftRed(e), and HorLeftYellow(e). As with the cars, we also have a predicate dealing with how long the intersection has been in the current phase, namely TimeInPhaseINT(e,d), with INT standing for one of 14 intervals (located between 0 and 300 seconds following the above scheme).

The predicates in *Obs* that constitute Obs_{int} not only deal with the currently received communications from the other agents (controllers of other intersections in the neighborhood of the intersection, where neighborhood has to be defined by the human experts) but also with previous communications (given that time needs to be spent for cars getting from one intersection to another). Similar to the above schemes, we have each of the predicates for each other agent. We have a predicate for each of the actions this agent is capable of, see above, and predicates for the time since the agent communicated the action, again a predicate for each "bin" representing a time interval (with 0 being the first interval and then the intervals naturally depending on the physical layout between the different intersections).

Given the connections between the predicates, we require that there is only one predicate from a connection group in a rule condition. So, for example, only one of the NumCarsWaitStraight predicates is allowed in a rule or only one of VerGreen, VerRed and VerYellow can be in a condition (since, obviously, in each of the groups only one of the predicates can be true at any given time).

4.3 Instantiating the hybrid learning method

To instantiate the hybrid learning method for shout-ahead agents to learn behaviors for intersection traffic light controllers we need to define the fitness function fit of an individual (via its run fitnesses $rfit$) in its agent pool *pool* and the reward r for updating the rule weights (which is a function evaluating the environmental state after the rule was applied compared to the state before the rule application). Both have to reflect the goals we have for intersections, with the main goal being to optimize the flow of the traffic through all intersections in the particular intersection layout. But this is a goal that only can be measured at the end of a simulation run, so that we also use other goals to inspire our instantiations.

For an individual *ind*, its fitness $fit(ind)$ is the average of all the run fitnesses $rfit(ind, run)$ it participated in. To compute $rfit(ind, run)$ for a particular *run*, we used some "milestone" goals about the end result of a run with different weights to guide the evolutionary process towards good overall results. Additionally, in order to create individuals that use their rules (and not rely on the do nothing default if no rules are applicable) and to make use of the computations leading to rewards for the reinforcement learning part of the architecture,

we have as part of $rfit$ a penalty computation $fitpen(ind, run)$ that adds up results after each action taken by the agent in the simulation run.

Formally, we have:
$rfit(ind, run) =$

$$\begin{cases} simT(run) - sumoRT & if\ simT(run) < sumoRT \\ AVT(ind) & elif\ AVT(ind) = bestAVT(pool) \\ AVT(ind) * 10 & elif\ AVT(ind) < 1.1 * bestAVT(pool) \\ AVT(ind) * 20 & elif\ AVT(ind) < 1.2 * bestAVT(pool) \\ AVT(ind) * 30 & elif\ AVT(ind) < 1.3 * bestAVT(pool) \\ AVT(ind) * 40 & else \end{cases}$$
$$+ fitpen(ind, run)$$

Here, $simT(run)$ denotes the run time of the simulation run (which is set up to service a particular flow, i.e. getting a given number of vehicles from given start points to given end points) and sumoRT is the run time of a simulation run for the same flow using SUMO's Actuated Traffic Light algorithm in the controllers. This component obviously produces a negative number, in contrast to the other components and focusses the evolutionary algorithm on individuals that are better than a base system (for which we chose the best system we had available: SUMO's ATL), once individuals are created that allow using this component. Note that the values created by this component are substantially smaller than what the other components produce, so that getting an individual for which this component is used has passed an important milestone in our learning. The other components use the aggregated wait times of the vehicles (AVT) the individual produced at its intersection over the simulation and relates it to the aggregated vehicle wait times the currently best individual in the individual's agent pool produced (bestAVT). The different cases are there to separate the individuals in a pool a little bit more than would happen if we just use AVT.

The run fitness also includes the fitpen-component, which aggregates possible penalties after each action taken by the individual in the simulation. Formally,

$$fitpen(ind, run) = \sum_{i=1}^{maxaction(run)} (nrpen(ind, sit(i)) * w_{nr} + wspen(ind, sit(i), sit(i+1)) * w_{ws}),$$

where $nrpen(ind, sit(i)) = 1$, if the action taken by the agent in the situation $sit(i)$ was not due to applying a rule in int and 0, else, and $wspen(ind, sit(i), sit(i+1)) = 1$, if the action taken in $sit(i)$ leads to a worse situation $sit(i+1)$ at the intersection than $sit(i)$, and 0, else. A situation is worse, if the reinforcement learner was using a negative reward (in other words a penalty) at that point of the simulation run. w_{nr} and w_{ws} are system parameters that in our experiments were set to 30 and 10, respectively.

The reward r for the reinforcement learning part of our hybrid model, which obviously is a function comparing the situation sit before the action was taken and the situation sit' after, is based on three components. The first component, $throughput(sit, sit')$ is the ratio of the number of cars that got through the intersection between the two situations to the total number of cars at the intersection (in sit). The second component $waitTimeR(sit, sit')$ is the ratio of the sum of the waiting times the cars that got through had at the intersection to the sum of all waiting times of all cars at the intersection in sit. The third component $QueueDiff(sit, sit')$ is the difference between the number of cars waiting at the intersection in sit and sit'. To get r, the components are combined as

$$\begin{aligned} r(sit, sit') = & throughput(sit, sit') \times w_{thru} + \\ & waitTimeR(sit, sit') \times w_{wait} + \\ & QueueDiff(sit, sit') \times w_{qdiff} \end{aligned}$$

Here, w_{thru}, w_{wait}, and w_{qdiff} are, as usual, weight parameters of the system allowing to define the relative importance of the components by an expert.

5 Evaluation

In this section we present our experiments with our instantiation of the shout-ahead and extended shout-ahead architectures (with communicated intended actions and without communication) to traffic light controllers using SUMO. We first present the setup of the experiments and then present and comment on the results we achieved.

5.1 Setup

For our experiments, we chose a not too complicated map containing 3 different types of intersections (see Figure 3). The intersection to the left is the standard 4-arm intersection with left-turn lanes and two additional lanes in each direction (see Figure 4). The intersection at the bottom to the right is a T-intersection with left-turn lanes and additionally 2 lanes in each direction (see Figure 5). Finally, the intersection at the top and right is a more unusual intersection, a so-called 4-arm incoming intersection as depicted in Figure 6.

We used this map with 3 traffic flows: a small flow of 75 vehicles with randomly created start positions and goal positions that start their travel at a randomly created time in the simulation, which could represent some nighttime traffic, a medium flow of 225 vehicles, again with random start positions and times and end positions, which could represent non rush hour traffic and finally a large flow of 425 vehicles, generated similarly to the others, which could be a rush hour traffic flow. Each vehicle uses the default control that SUMO offers for vehicles which does both path planning for the vehicle and the actual driving decisions obeying all traffic laws and laws of physics.

To evaluate the importance of communicating intended actions we created a non-communicating variant of the shout-ahead architecture (which then obviously is not shouting ahead anymore) by setting the number of rules allowed in

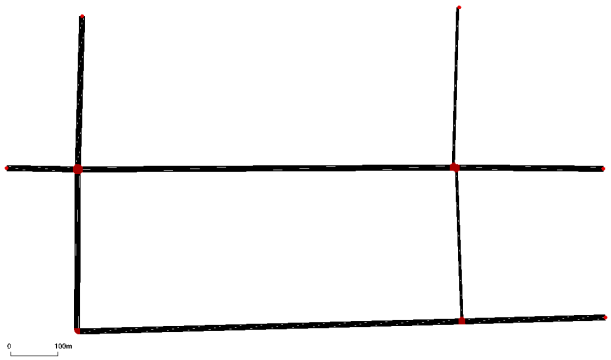

Fig. 3. Map of our evaluation network in SUMO.

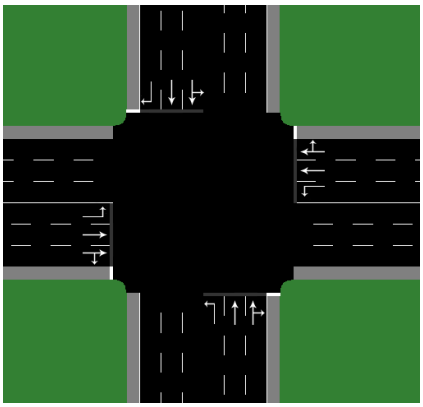

Fig. 4. Example of a standard 4-arm Intersection in SUMO.

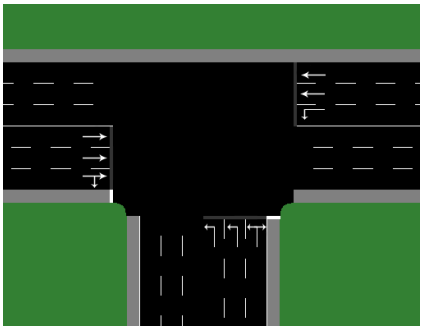

Fig. 5. Example of a T-Intersection in SUMO.

RS_{int} to 0, which means that always a rule from RS is chosen (using the decision function presented in Section 2.1). In the extended version of shout-ahead the number of exception rules allowed in RS^{exp}_{int} is also 0, which means that a human expert is not allowed to use any predicates from Obs_{int} in his/her rules.

We used the following parameter settings in all experiments. For the agent architecture itself, we have $\epsilon = 0.5$, $\beta = 0.001$, and $p_{coop} = 0.5$.

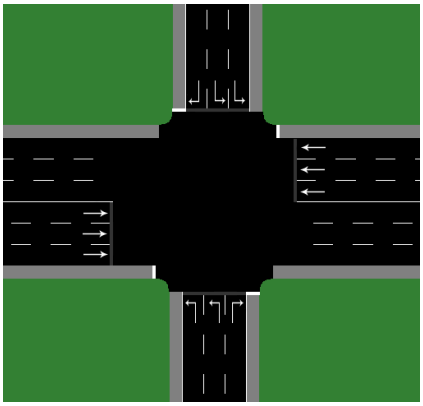

Fig. 6. Example of a 4-arm Incoming Intersection in SUMO.

For the evolutionary learner, we have 3 agent pools (one for each type of intersection) with 30 individuals in each. An individual had 10 elements in each of RS and RS_{int} and each rule could have up to 3 observations in its condition. In a generation, an individual participated in at least 3 simulation runs. $p^{cond} = 0.16667$ and $p^{act} = 0.083335$. The number of new individuals created n_{off} was 21.

For the reinforcement part of the learner, we used $\gamma = 0.5$ and $\alpha = 0.5$. The parameters for the reward function were set as follows: $w_{thru} = 1$, $w_{wait} = 1$, and $w_{qdiff} = -0.05$.

All experiments were performed on a Windows-based ASUS ROG Strix desktop, with an Intel Core i7 processor running at 3.6GHz, and 8.00 GB of RAM.

5.2 Results

In this subsection, we look at the results of various experiments we performed to evaluate if the shout-ahead architecture with its hybrid learning approach can be used for learning controllers for the traffic lights of intersections and if our extension to the shout-ahead architecture offers improvement possibilities over the old architecture. And we also conducted experiments with the shout-ahead without communication variant to see how important the ability of communicating intended actions is for the success of the learning.

Experiments with the full extended shout-ahead architecture Our first experimental series compares the extended shout-ahead architecture with the original shout-ahead architecture and the SUMO ATL reference architecture for the 3 different traffic flows through our test map. As the first and second data rows in Table 1 show, for the medium and the large vehicle flows the hybrid learning for the shout-ahead architecture creates better teams of controllers (we used the team consisting of the best individual in each agent pool) than the ATL controller, in case of the large flow a substantially better controller (needing only two-thirds of the time of the ATL controller to deal with the flow). But for the small traffic flow, the learner for the shout-ahead architecture was not able to

learn a team of controllers that was able to finish the simulation within the time we allotted for it (10000 seconds).

Table 1. Quality comparison SUMO ATL with the best team for the different versions of shout-ahead (times in seconds)

Controller version	Flow (vehicles)		
	75	225	425
SUMO ATL	1265	1690	2837
shout-ahead	-	1485	1939
shout-ahead + user-def. rules 1 after	836	2847	4822
shout-ahead + user-def. rules 2 after	718	1179	1346
shout-ahead + user-defined rules 2	453	1164	2131

Obviously, this is exactly the situation the extended shout-ahead architecture is aimed for. When looking at the simulation run for this small flow, we observed that vehicles effectively got stranded in the run because the learner did not come up with rules that changed the lights quickly enough (so, no overruling of SUMO's build-in light period lengths). This lead to creating our first rule set (number 1 in Table 1) consisting of rules in RS^{exp} for each agent that achieve a shorter restriction of the length of a signal phase of 113 seconds for the green phase, 3 seconds for the yellow phase, and 116 seconds for the red phase. As the third data row in Table 1 shows, this results in not only finding a team of controllers that is able to finish the small flow, it does so much faster than SUMO's ATL. But as the other columns in this row show, the rules substantially slow down the medium and large flows. This is not so surprising, since obviously larger flows should be allowed longer period times, which, due to aiming at user-defined rules being exception rules and therefore having priority, is with the set 1 not possible anymore.

With the focus on exceptions, we took a second look at the original simulation run for the small flow and realized that what really was not working well were situations when there were cars waiting at an intersection for one direction and no cars were driving in another direction. This allowed for a smaller rule set (number 2 in Table 1) with a more precise rule for each direction, namely switching to the green phase of the direction whenever vehicles wait for that direction while all other directions are not having vehicles at all. As the fourth data row in Table 1 shows, this resulted in also creating a successful result for the small flow (even better than with user-defined rule set 1) and also better results for the other two flows. It should be pointed out that evaluating the usefulness of the two user-defined rule sets only required one simulation run for each of them (per flow), which, as Table 2 shows, is substantially faster than trying to improve the learning.

The next experiments used the rule set 2 already when performing learning. As the last row in Table 1 shows, the resulting team of controllers was better than

Table 2. Learning run-time comparison shout-ahead only vs shout-ahead with user-defined rules

Controller version	Flow (vehicles)		
	75	225	425
shout-ahead	37.39 h	4.53 d	6.95 d
shout-ahead with user-defined rules	5.3 h	16.32 h	1.63 d

ATL, better than just using learning for the shout-ahead architecture without user-defined rules but only better than adding the rules to the learned controllers after learning for the 75 vehicle and the 225 vehicle flows (and for the 225 vehicle flow just barely better). The resulting team for the large flow was quite worse than just adding the rules afterwards and even worse a little bit than what the original shout-ahead architecture produced. It seems that adding the rules from the beginning biased the learning in an unfortunate way. But if we look at Table 2, for all flows the time needed for the learning improved substantially, since now the individuals in the early generations of all pools were not totally useless and the evolutionary process more focussed.

To provide an idea what kind of rules were learned by the system, in the following we will take a look at some of the rules for the team learned without the user-defined rules for the 225 vehicles flow. As for the user-defined rules, we present the rules using natural language descriptions for better readability.

For the standard 4-arm intersection controller, there were 2 rules in RS with a weight greater than 0 and all 10 rules in RS_{int} had weights greater than 0 (after learning a rule with weight 0 was never used). The rule in RS with the highest weight had the lights change to the phase where the left turn lights for the vertical flow through the intersection are yellow, if there are no vehicles waiting to proceed straight through the intersection. The other rule put the intersection into the phase where the lights for the straight horizontal flow are green, if the longest wait time of vehicles going straight was between 30 and 45 seconds. The top rule (with regard to weight) in RS_{int} had the intersection going into the phase where the lights for the straight horizontal flow are green if the incoming 4-arm intersection intended to turn into its phase where its left turn lights are green and another high-weight rule had the intersection turning into the phase where the horizontal left turn lights are green, if within the last 5 seconds the incoming 4-arm intersection intended to turn into its phase where the horizontal straight flow got a green light.

For the T-intersection controller, there was only one rule in RS with a weight greater than 0 and 8 rules in RS_{int} . The rule in RS has the intersection providing green lights for the traffic going straight through it from the east to the west and also for the left turn traffic coming from the east, if the longest wait time for turning left is between 15 and 30 seconds. The top rule in RS_{int} tells the controller to switch to the phase where the horizontal straight traffic flow gets the yellow light, if the standard 4-arm intersection intended to turn its horizontal straight traffic flow to green. The next best rule switches the intersection lights

for going from west to east to yellow, both for the straight flow and the left turn flow, if the time since the last communication was between 5 and 10 seconds.

For the incoming 4-arm intersection controller, there were no rules in RS with a weight greater than 0, but all 10 rules in RS_{int} had positive weights. The strongest rule (with regard to weight) told the controller to switch to the phase that had green for the straight horizontal traffic flow, if the standard 4-arm intersection intended to switch its straight horizontal traffic flow to green. The second strongest rule had the intersection switch to the light phase that had the left-turn light green for the cars going from south to north (coming from the bottom in Figure 6), if the T-intersection communicated that it intended to switch to the phase where the light for the west to east straight flow is yellow (which also means that the left turn light for that flow turns to yellow). Compared to rules of other intersections, the weights for the rules for this intersection were higher, indicating that these rules really were very successful in having the traffic moving.

For all intersection controllers there were several rules that had as condition a particular time interval for the last communication and as action to do nothing. This makes sure that the actions of previous rules leading to the current phase of the intersection were not undone before the traffic from the other intersections has reached this intersection. It should also be mentioned that there were rules with more than one condition, although not with the highest weights.

Table 3. Quality comparison of the best teams for the different versions of shout-ahead without communicating intended actions (times in seconds)

Controller version	Flow (vehicles)		
	75	225	425
shout-ahead no comm.	-	9776	9653
shout-ahead no comm. + user-def. rules 2 after	4648	7950	4978
shout-ahead no comm. + user-def. rules 2	4617	6132	6779

Experiments with non-communicating shout-ahead architecture variants The next series of experiments targeted the importance of communication for the shout-ahead architecture and was motivated by the fact that we had so many rules in the RS_{int} sets of agents resulting from the previous experiments. Table 3 presents the results of the teams consisting of the best learned controller for each of the different intersections. Similar to the results using the full shout-ahead architecture, the learner was not able to produce controllers that were successful for the small flow, if no user-defined exception rules were used. The results for the two other flows are substantially worse than for SUMO ATL and consequently also much worse than what the learner was able to achieve for the full shout-ahead architecture. Even when adding the exception rule set 2 to the learned rule sets, the results are worse than SUMO ATL and naturally much worse than the corresponding results for the full shout-ahead architecture. But,

as for the full shout-ahead architecture, the addition of the exception rules produces a team of controllers able to solve the small flow and we see improvements in handling the other two flows, too.

Table 4. Learning run-time comparison shout-ahead no communication only vs shout-ahead no communication with user-defined rules

Controller version	Flow (vehicles)		
	75	225	425
shout-ahead no comm.	3.28 d	7.83 d	14.63 d
shout-ahead no comm. with user-defined rules	1.78 d	4.51 d	12.52 d

Using the exception rule set 2 already during learning shows similarities to the results of the learning for the full shout-ahead architecture with communication of intends, again. For the small and medium flow examples, using the exception rules already for learning produces better results, but for the large flow example just adding the exception rules after learning is better (quite a bit, in fact). With regard to the run-time of the learning, Table 4 is giving us a very similar picture to Table 2, just with much longer run-times. Without using the exception rules, the learning took several times longer, with the length increase for the large flow being a factor of 10. With using the exception rules, we have, again, shorter learning times and due to that the slowing down factor compared to the version with communication is smaller, but still substantial.

The results so far have shown that the ability to communicate intentions of controllers to other controllers allows for learning better control strategies in a faster time. The right user-defined exception rules can improve the quality of already learned strategies quite a bit and when already used during the learning the run-times for learning are reduced.

Experiments with varied flows All the previous experiments evaluated the learned agents on exactly the flows that the learner used to create the agents. Naturally, in the real world this is rather unrealistic. There will be at least some variations when vehicles will start their journeys, although a lot will travel from the same start and end point every day (at least on most work days). So, additional experiments are needed that evaluate the flexibility of the learned controllers and controller teams.

Our experiments with this regard created for each of the original flows variants by randomly selecting half of the vehicles and modifying their start times by adding or subtracting a random number between 0 and 10 seconds while making sure that the overall last start time for the group of vehicles did not change. For each of the 3 flows we created 10 such variants and measured how long it took for the teams from Table 1 (without the one for rule set 1) to run each variant through the simulation (we did not bother with performing experiments without communicating intentions, due to the inferior quality of the produced controllers). The results are presented in Table 5. As the first 3 data rows show,

Table 5. Comparison original shout-ahead learning vs shout-ahead with user defined rules added after learning vs shout-ahead learning already with user defined rules on 10 variations of the original flows regarding average time, minimum time and maximum time (times in seconds)

Controller version		Flow (vehicles)		
		75	225	425
shout-ahead	avg.	2487.2	3853.2	5243.4
user-def. rules 2 after	avg.	553.3	1421	2253.8
user-defined rules 2	avg.	549.2	1397.2	2250.1
shout-ahead	min.	1560	2322	4019
user-def. rules 2 after	min.	481	1186	2117
user-defined rules 2	min.	477	1250	2121
shout-ahead	max.	-	5710	7245
user-def. rules 2 after	max.	968	1702	2707
user-defined rules 2	max.	992	1728	2602

on average the team created by learning with the original shout-ahead architecture is quite a bit off from what this team produces for the flow it learned from, although the average for the 75 vehicle flow (which we computed only out of the experiments that finished) indicates that this team was better for some variants than for the original flow. The averages for the two variants with the user-defined rule set 2 are very similar, indicating that having these exception rules either added after the learning or already learning with them has a very positive influence and creates teams of comparable flexibility.

If we look at the second group of data rows in Table 5 which reports on the best results achieved for any variation we can see, again, that having user-defined exception rules provides more flexibility than what the original shout-ahead architecture with learning accomplishes. In fact, the difference to the averages for the teams with user-defined exception rules is not too big, in contrast to what just learning for the original shout-ahead architecture achieves. Especially for the small flow we have variants that are solved nearly as good or even better than the original flow. The medium and larger flows require, due to the way the variants are constructed more flexibility which shows for the controller version that just adds the exception rules after learning, whereas the team that was learned with exception rules already in place solves one variant faster than the original.

Finally, if we look at the third group of data rows that presents the maximum simulation run times that we observed among the variants we can see for the small flow variants that they are rather far away from the minimum run time (and, fortunately, also from the average run time indicating that we have very few such runs). For both variants of using exception rules the differences between these variants are small, but the difference to the original shout-ahead is rather large. With respect to indicating flexibility, the differences between maximum and minimum are not large, indicating that the created teams are rather flexible.

In summary, using the extended shout-ahead architecture with user-defined exception rules results in better results than what just using the learning for the original shout-ahead architecture achieves. If the exception rules are already used in the learning the run time for the learner is substantially reduced. And using user-defined exception rules also results in more flexible controllers than what just learning for shout-ahead produces. From the perspective of the agent architecture, the ability to use communicated intended actions is very important, since without the ability to use this information the resulting controllers are much worse and the time needed to learn them is also substantially longer.

6 Related work

Due to the importance of efficient traffic flow in cities, naturally there have been quite a number of scientific publications dealing with the problem. And there have also been several works that employ machine learning methods to the problem. Most of them use purely reinforcement learning. In fact, [9] highlighted that many then current solutions to the TSC problem were centralized, and argued that this centralized nature makes them infeasible for large scale adaptive traffic signal control due to the high dimension of the joint action space and then proposed a reinforcement solution. Other purely reinforcement solutions have been proposed, for example, in [5] or [1]. The use of evolutionary algorithms, either alone or in combination with other learning methods, has not been reported in the literature, so far.

There are two works using reinforcement learning that highlight communication between the controller agents, namely [6] and [4]. [6] utilizes an already existing algorithm, the max-plus algorithm, to achieve coordination between the controllers. Essentially, this algorithm is an iterative process where the agents exchange information about their local rewards for different actions given actions of the other agents. Theoretically this results into a convergence of the local reward functions towards the optimal global one. In contrast to this communication during learning, our approach aims at using communication directly in the decision making of a controller to provide more information. The authors of [4] defined their own coordination algorithm that is based on learning models of the other controllers which then can be used to predict what these controllers will do in the various situations and a controller will then use the models of the neighboring controllers to determine its best action. Again, the communication is aimed at the learning phase and not at the application phase of the system.

Finally, [10] also highlights communication and evaluates what information is useful to share and what the consequences of such cooperation are. Intentions are not part of the discussed information types. None of these works allow for the use of user-defined exception rules or use other learning approaches than reinforcement learning.

7 Conclusion and future work

We presented an extension to the shout-ahead architecture that allows for user-defined exception rules that can be added to already learned agents or already

used when learning the agents and we also presented a new application area for shout-ahead based agents and the hybrid learning method for them, namely cooperating controllers for the traffic lights of intersections. Our experimental evaluations showed that already the learning for the original shout-ahead architecture can produce better results than a well established control algorithm for some traffic flows, but just making use of the ability to improve learned results with exception rules can substantially improve the created controllers, although what exception rules are used naturally has an influence on this. Using exception rules already when performing the learning can improve the created controllers, but not always. It does result in much faster learning times, though. We also showed that the shout-ahead part of the shout-ahead architecture, i.e. the sending of intended actions to other controllers, is essential for good results and acceptable learning times. Both variants of the usage of user-defined exception rules for the complete shout-ahead architecture result in controllers that are rather flexible, still performing well even if the encountered traffic flows are variants of the flow used for learning.

There are a number of future research directions around the shout-ahead architecture, its hybrid learning method and the presented extension allowing for user-defined exception rules. In addition to exploring other application areas and exploring other maps, we will look into leveraging the faster learning when exception rules are already used when learning to increase flexibility by evaluating agents using several maps or groups of flows to create the fitness measure. We also intend to integrate handling of various emergency vehicles (with priority) into our controllers.

References

1. N.M.M. Abdoos and A.L.C. Bazzan: Traffic light control in non-stationary environments based on multi agent q-learning, Proc. ITSC 2011, Washington, 2011, pp. 1580–1585.
2. J.E.M. Behrisch, L. Bieker, D. Krajzewicz: Sumo-simulation of urban mobility: an overview, Proc. SIMUL 2011, Barcelona, 2011, pp. 55–60.
3. J. Denzinger and M. Fuchs: Experiments in Learning Prototypical Situations for Variants of the Pursuit Game, Proc. ICMAS'96, Kyoto, 1996, pp. 48–55.
4. S. El-Tantawy, B. Abdulhai, H. Abdelgawad: Multiagent Reinforcement Learning for Integrated Network of Adaptive Traffic Signal Controllers (MARLIN-ATSC): Methodology and Large-Scale Application on Downtown Toronto, IEEE Trans. Intelligent Transportation Systems, vol. 14, no. 3, pp. 1140–1150.
5. J.L.P. Gregoire, C. Desjardins, B. Chaib-draa: Urban traffic control based on learning agents, Proc. ITSC 2007, Seattle, 2009, pp. 916–921.
6. B.B.L. Kuyer, S. Whiteson N. Vlassis: Multiagent Reinforcement Learning for Urban Traffic Control Using Coordination Graphs, In: Daelemans W., Goethals B., Morik K. (eds.) Machine Learning and Knowledge Discovery in Databases, ECML PKDD 2008, pp. 656–671.
7. National Academies of Sciences, Engineering, and Medicine: Signal Timing Manual - Second Edition, The National Academies Press, Washington, DC, 2015.
8. S. Paskaradevan, J. Denzinger, D. Wehr: Learning cooperative behavior for the shout-ahead architecture, WIAS Vol. 12(3), IOS Press, 2014, pp. 309–324.

9. D. de Oliveria, A.L.C. Bazzan, B.C. da Silva, E.W. Basso, L. Nunes, R. Rosseti, E. de Olivera, R. da Silva, L. Lamb: Reinforcement Learning-based Control of Traffic Lights in Non-stationary Environments: A Case Study in a Microscopic Simulator, Proc. EUMAS'06, Lisbon, 2006.
10. D. de Oliveira and A.L.C. Bazzan: Multiagent Learning on Traffic Lights Control: Effects of Using Shared Information, in Ana Bazzan, Franziska Klügl (eds.): Multi-Agent Systems for Traffic and Transportation Engineering, IGI Global, 2009, pp. 307–321.
11. K. Rao, H. Sak, R. Prabhavalkar: Exploring architectures, data and units for streaming end-to-end speech recognition with RNN-transducer, Proc. ASRU 2017, Okinawa, 2017, pp. 193–199.
12. C. Roatis and J. Denzinger: Extending the learning shout-ahead architecture with user-defined exception rules – a case study for traffic light controls, Proc. HCCAI 2020, Irvine, 2020, pp. 9–16.
13. D. Silver, A. Huang, C.J. Maddison, A. Guez, A. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis: Mastering the game of Go with deep neural networks and tree search, Nature 529, 2016, pp. 484–489.
14. J-P. Steghöfer, J. Denzinger, H. Kasinger, B. Bauer: Improving the Efficiency of Self-Organizing Emergent Systems by an Advisor, Proc. EASe 2010, Oxford, 2010, pp. 63–72.
15. R.S. Sutton and A.G. Barto: Reinforcement Learning: An Introduction, The MIT Press, 1998.
16. C. Zhang and S. Zhang: Association rule mining: models and algorithms, Springer-Verlag, 2002.