Learning Algorithms with Self-Play: A New Approach to the Distributed Directory Problem

Pankaj Khanchandani
Cloud Technology
Adobe Systems, India
kpankaj@adobe.com

Oliver Richter, Lukas Rusch and Roger Wattenhofer
Department of Electrical Engineering and Information Technology
ETH Zurich, Switzerland
{richtero,ruschl,wattenhofer}@ethz.ch

***Abstract*—Many deep learning methods have been proposed recently to learn algorithms for combinatorial problems. However, most approaches focus on either supervised/imitation learning (the target algorithm is known) or single agent reinforcement learning (the input distribution is fixed). In some cases, however, the input distribution scales combinatorially as well and cannot easily be fully represented in a concise data set. In this paper, we propose a self-play approach to learn a *distributed directory protocol* to coordinate concurrent requests to a shared mobile resource among a network of nodes. The self-play is between two agents – a *request* agent, which finds worst case request inputs – and a *route* agent, which finds an algorithm that works well on the proposed worst case inputs (and consequently, works well on most queries). We show that our self-play approach is successful in learning an algorithm that works well across diverse request sequences. The empirical performance of the learnt algorithms is within the best known theoretical bounds, and sometimes significantly better than the best known upper bounds.**

I. Introduction

A basic building block of computer science are algorithms. Given a problem, can we find an algorithm with low cost? Typically, algorithm designers play an elaborate game with themselves: First they create an algorithm. Then they try to find an input where their algorithm performs poorly. If they find such an input, they improve their algorithm. This process is repeated, until they eventually have an algorithm that performs well with every input. We seek to automate this process by modeling this interplay as an asymmetric competitive two player game between two reinforcement learning (RL) agents. The *algorithm agent* tries to improve the quality of the algorithm, while the *input agent* tries to find an input where the algorithm performs poorly. While this RL approach cannot give formal guarantees, it provides a template for automated creation of algorithms with good empirical performance. Also, an algorithm designer may learn the structure of hard inputs from the input agent, and the techniques to deal with these inputs from the algorithm agent. In general, the two RL agents may help the algorithm designer to gain deeper insights.

In this paper, we demonstrate this general approach on the *distributed directory problem*, a fundamental problem in distributed computing. As the backbone of concurrent computation, distributed directories are vital for the efficient management of shared resources. Studying the problem on an abstract level can thereby have efficiency implications in applications as diverse as multi-processor design and satellite communication. While asymptotically optimal protocols exist for a few network topologies [1], many settings remain an open problem with no known best solution. We show that our approach performs on par with optimal protocols where such protocols exist and even empirically improves upon well known protocols by a large margin otherwise.

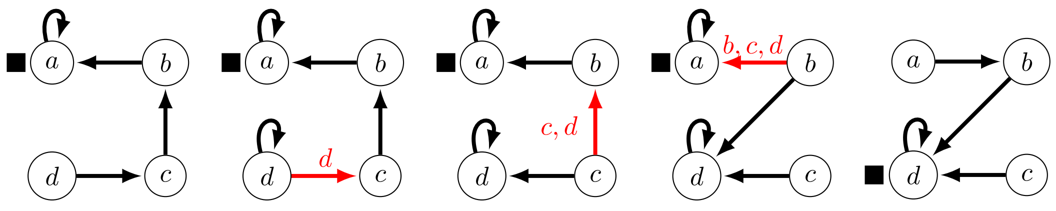
Further, we show that alternative learning approaches lead to sub-optimal protocols that can be exploited, while our self-play approach is robust against adversarial attacks.

II. Distributed Directory Problem

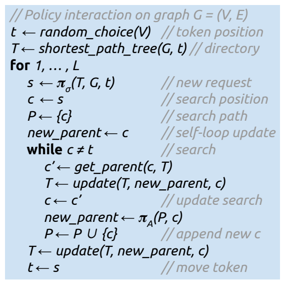
The distributed directory problem is a fundamental distributed computing problem. The problem asks to coordinate access to an exclusive resource that is shared among the nodes of a network. Formally, let $G = (V, E)$ be a network of nodes V that are connected by reliable communication links E. Initially, a single shared resource — called the *token* — is at some node $v \in V$. A simple solution to coordinate access to the token is to make v the "home" node and let every requesting node ask v for the token and return it to v after using it. However, this is very inefficient when a node that is "far from home" wants to repeatedly access the token. Hence, a better protocol is needed.

Arrow and Ivy are simple and classic protocols to solve the problem [2]–[4]. Their performance has been extensively studied, e.g., [5]–[8]. While distributed directory protocols beyond Arrow and Ivy do exist [9]–[11], Arrow and Ivy are simpler and practically more appealing. Recently, the Arvy family of distributed directory protocols has been proposed [1]. In every step, Arvy can choose from a number of options, including as special cases Arrow or Ivy. Due to its flexibility, in some families of networks Arvy will outperform all known protocols [1]. While the Arvy framework is generally applicable, the only known optimal Arvy protocol to date is limited to cyclic networks. In this paper we aim to search for efficient Arvy protocols on other network topologies.

Arrow, Ivy and Arvy solve the distributed directory problem by keeping the token mobile and hopefully closer to the (unknown) next request. Concretely, each node $v \in V$ holds a parent pointer $p(v)$ to either itself or another node. If the node points to itself, the node either holds the token or is already waiting for it. Otherwise, the parent pointers recursively point towards a node holding the token (or already



(a) Initial directory (b) Request from d (c) Message to b (d) Message to a (e) New directory



```
// Policy interaction on graph G = (V, E)
t ← random_choice(V)   // token position
T ← shortest_path_tree(G, t) // directory
for 1, ... , L
    s ← $\pi_\sigma$(T, G, t)          // new request
    c ← s                  // search position
    P ← {c}                // search path
    new_parent ← c         // self-loop update
    while c ≠ t            // search
        c' ← get_parent(c, T)
        T ← update(T, new_parent, c)
        c ← c'             // update search
        new_parent ← $\pi_A$(P, c)
        P ← P ∪ {c}        // append new c
    T ← update(T, new_parent, c)
    t ← s                  // move token
```

(f) Pseudo Code

Fig. 1: Example for Arvy. Only directory edges are shown, not the underlying graph. Red edges are messages in transit, labeled with all possible new parents. (a) The initial directory T. (b) Node d makes a request, and sends a find message to its parent c. (b),(c) Nodes c and b forward the request message along with the possible new parent options. Both choose node d as their new parent in this example. (d) Request arrived at node a, which chooses b as its new parent in this example. The token is moved and now the directory is a tree pointing towards the new root d. (f) Pseudo Code of the agent-environment interaction.

waiting for it). Whenever there are no active requests, the directory is a directed tree T rooted at the token node. A distributed directory protocol orders concurrent requests and enables each node to locate the token. Each node manages its parent locally. A node s requests the token by sending a message to its parent. The request is forwarded (following the parent pointers) on a path $s, ..., u, v, ..., t$ until it reaches a node t with a self-loop. Other than forwarding, the protocol must choose a new parent for each node v along the path. In Arrow [2], [3], this is the previous node u, effectively flipping the pointer. In Ivy [4], the new parent is s, the requesting node. In Arvy [1], the authors show that the new parent may be any node on the subpath $s, \dots, u$. Specifically, they prove that any Arvy choice ensures that every requesting node eventually receives the token, even if these requests are concurrent, and communication is asynchronous. Figure 1 (a)-(e) shows an example execution for a single request. Note that choosing a new parent is notoriously hard as future requests are unknown at the time of decision and every request changes the surrounding directory structure. Further, to be applicable in a distributed setup, the decision has to be based solely on locally aggregated information.

To evaluate a directory protocol we study its multiplicative overhead. We consider a sequential setting, where a new request is issued only after the previous request has finished. Note that this does not limit the learned protocols from operating in a setting with concurrent requests, but simplifies evaluating the performance and formulating our optimization objective. The cost of one request from node s is the cost of finding the token node t by sending messages along the unique path in the directory tree T. In case an edge (u, v) on this unique path is not part of the underlying graph G, the message is sent along the shortest path in G with cost $d_G(u, v)$. The cost of the request is then the distance $d_T(s, t)$ between s and t on the directory tree T, the summed cost of sending messages along the path. Any protocol, and therefore also any optimal protocol, costs at least the shortest distance $d_G(s, t)$ to find the token at t from s. Given a sequence of requests $\sigma = [v_1, v_2, \dots, v_L]$ with initial token location v_0 and directory T_0, the cost of a protocol A is defined as $cost_A(\sigma) = \sum_{i=0}^{L-1} d_{T_i}(v_{i+1}, v_i)$, where T_i is the updated tree after request v_i. The cost of any protocol is lower bounded by: $cost_A(\sigma) \geq \sum_{i=0}^{L-1} d_G(v_{i+1}, v_i) = cost_{opt}(\sigma)$. The *overhead ratio* for a specific request sequence σ is then defined as $\rho(\sigma, A) = \frac{cost_A(\sigma)}{cost_{opt}(\sigma)}$. Finally, the *competitive ratio* is the worst overhead over all possible request sequences: $\max_\sigma \rho(\sigma, A)$.

On cycles of n nodes, both Arrow and Ivy are $\Omega(n)$-competitive, that is, their worst performance scales linearly with the network size. In contrast, there exists an Arvy protocol that is 5-competitive on cycles [1]. We aim for similar protocols with constant competitive ratio for other topologies.

III. Learning the Protocol

We seek to show how learning can be used to find efficient Arvy protocols. We therefore formulate learning a competitive Arvy protocol as a multi-agent problem with two agents: The *algorithm* agent represents the core of the directory protocol and is hereafter referred to as *route agent* π_A. The *input* agent challenges the learned algorithm. In the Arvy setup this corresponds to generating the next request of the request sequence σ. We therefore refer to it as *request agent* π_σ.

The interaction between the agents is given in the pseudo code in Figure 1f and is roughly described as follows: We initialize the token location t and directory T such that the first request does not yield any overhead. We then query the request agent π_σ for a first request. This request is then iteratively solved by following the pointers in the directory. For all visited notes c, the route agent π_A is queried to choose a suitable new parent from the nodes on the search path P so far. Finally, when the token is found, we move the token to the requesting node and the request agent is queried for a new request. This interaction is repeated until a predefined number L of requests have been issued and resolved. Note that in contrast to classical two player games like chess, our agents do not take alternating turns. Rather, the request agent makes a move (a new request) followed by several moves of the route agent (choosing new parents for every node on the path). Only then the request agent is asked for a new request.

This formulation reflects the general Arvy framework. Specifically, it allows us to plug in a known, hardcoded protocol or a trainable policy for either the request and/or the route agent. As an example, it allows us to evaluate how Arrow — $\pi_A(P, c) :=$ *get_last*(P) — performs against random requests — $\pi_\sigma(T, (V, E), t) :=$ *random_choice*(V). However, it also allows us to train a request and/or route agent. That is, we can train a protocol optimized for a specific request sequence, e.g., if we know a-priori that the resource will be requested by the nodes in a specific order. Or we can train a request agent to find worst case sequences for known protocols. Putting it all together, our setup also allows to train both, a route and a request agent in a competitive, zero-sum self-play game — effectively generating a curriculum where the route agent gets more robust against all possible request sequences while the request agent gets better at exploiting the weaknesses of the route agent.

A. *Objective*

In our setup, learning a directory protocol and request sequence can be formulated as a multi-agent game. Here, each agent is represented by a stochastic policy π that maps its observations to a probability distribution over possible actions, where we consider other agents as part of the stochastic, partially observable environment. The action space of the request agent consists of all nodes that can start the next request. The action space of the route agent consists of all possible next parents - where to route a future request. The policies learn to maximize their cumulative rewards $R = \sum_{i=1}^{L} r_i$ with proximal policy optimization [12] in a self-play setup [13].

Consider as objective the competitive ratio $\max_\sigma \rho(\sigma, A)$ for some Arvy protocol A. In our environment we can evaluate the overhead ρ for the sampled sequence of L requests and give it as a reward to the request agent at the end of the episode. Training the agent with reinforcement learning should then give an incentive to maximize this overhead, effectively seeking out sequences that exploit the weaknesses of the countering protocol π_A. Note however that this overhead is tied to the sequence length L and does not reflect the competitive ratio $\max_\sigma \rho(\sigma, A)$, as a worst case sequence σ might be longer or shorter. As we consider a ratio, we do not expect it to be significantly different on longer sequences for a sufficiently large L. We therefore opt to optimize

$$\mathbb{E}_{\sigma \sim \pi_\sigma}\left[\max_{i \in [1..L]} \rho(\sigma_{:i}, A) | len(\sigma) = L\right]$$

as a proxy, where $\sigma_{:i}$ denotes the sub-sequence of the first i requests in σ. This formulation effectively moves the maximum into the expectation and removes the requirement to sample shorter sequences in order to optimize the overhead on them. We achieve this objective as follows: During an episode we track the overhead ratio $\rho^l(\sigma, A) = \max_{i \in [1,l]} \rho(\sigma_{:i}, A)$ to account for sequences stopping after l requests. We refer with r_l to the reward received after request l. We set $\rho^0(\sigma, A) = 0$ and define the reward for the request agent as $r_l^\sigma = \max(\rho(\sigma_{:l}, A) - \rho^{l-1}(\sigma, A), 0)$ For an episode with L requests the sum of all rewards for the request agent is then given by $\rho^L(\sigma, A) = \sum_{l=1}^{L} r_l^\sigma = \max_{i \in [1..L]} \rho(\sigma_{:i}, A)$ exactly the objective we seek to optimize. Note that this reward design also gives early feedback, rather than at the episode's end.

The route agent has the objective of minimizing $\rho^L(\sigma, A)$, so we set $r_l^A = -r_l^\sigma$ after every completed request. (While solving a request the agent receives no reward.) For both agents, this setup automatically gives 0 reward after the worst stopping point of a sequence.

B. *Architecture*

We allow the request agent π_σ to have access to the full distributed directory tree T, the underlying network structure G as well as the current token position t. In contrast, the route agent π_A only has access to the information from the nodes on the search path P so far as well as the current search location c. This restriction is done such that the learned protocol can be deployed in a distributed setting: We can expect the protocol to pass on information with the search request, however, the global state of the directory is hidden from the route agent. Note that the request agent does not need such a restriction, as we normally assume privileged information to find a lower bound of an algorithm. Note that our setup is different from most RL setups: The observation of the route agent (nodes on the search path P so far) as well as the action space (possible new parents for the current node – which are all nodes in P) grow as the search progresses. Similarly, we want our request agent architecture to be independent of the number of nodes n in the underlying graph G. We therefore opted to embed the inputs into per-node features and applied bidirectional LSTMs [14] over these features.

IV. Experiments

In all experiments, we set the number of requests to $L = 100$ and estimate the average return $\bar{\rho}$ over 50 episodes.

To see how learning compares to handwritten protocols, we compare in the following against several baseline protocols, given some predefined request sequences. As directory protocol baselines we take *Arrow* and *Ivy*. Further *Arvy Bridge* [1] is the Arvy policy achieving constant competitive ratio on cycles. Note that Arvy Bridge is not defined for other graphs, we can therefore only compare against it on cycles. Lastly, *Random* π_A gives a baseline which selects parents uniformly at random from P and represents an untrained route agent.

The baseline request sequences for evaluation are *Random* π_σ, *Roundrobin* and *Greedy*. *Random* π_σ selects the next request uniformly at random from all nodes other than the node with the token. *Roundrobin*, a simple request sequence on cycles, issues requests along the cycle, starting with a neighbor to the initial token. Note that this request sequence can be particularly bad for a distributed directory, as the directory tree cannot cover the full cycle. As a final strong baseline, the *Greedy* sequence selects the next request after a sequence σ is solved with protocol A as $\arg\max_{req} \rho([\sigma, req], A)$. Ties are

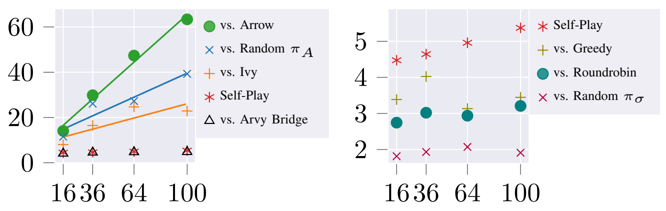


Fig. 2: **Left:** Trained request agents on cycles of varying size n (x-axis) against protocols Arrow, Ivy, Arvy and using self-play. We evaluate return $\bar{\rho}$ for each agent against their respective directory protocol. **Right:** Learned route agents on cycles of varying size, trained against the request sequences Random, Greedy, Roundrobin and using self-play. The return $\bar{\rho}$ for each agent against their respective request sequence is plotted. Note the scale difference in the y-axix.

broken first with the furthest node in the current directory and then randomly. This greedily maximizes the competitive ratio and is in many cases close to the worst-case sequence.

A. *Scaling for Cycles*

On cycles, both Arrow and Ivy have linear competitive ratio while the Arvy Bridge heuristic achieves a constant ratio of 5 [1]. First, we test if a learned request agent can attain these overheads. Then we train route agents to minimize the expected competitive ratio against the baseline request sequences to see their effectiveness in adapting to a given setup. In both cases, the performance is compared to a self-play approach. We test with different sizes n from 16 to 100 nodes in order to see scaling behavior.

Request sequences: Results for trained request agents are shown in Figure 2 (left). Shown is the return $\bar{\rho}$ of the trained agents against the different baseline directory protocols and the self-play approach, where the directory algorithm is learned simultaneously. Request agents trained against Arrow and Ivy scale linearly as desired. Also against a protocol that randomly chooses the next parents we find request sequences that yield an approximately linear increase in competitive ratio. Against Arvy Bridge a sequence is learned that, by inspection, attains the competitive ratio of 5 for n tending to infinity. The self-play request agent reaches an almost equivalent return, indicating the learned directory protocol is empirically as competitive as Arvy Bridge, which is optimal here. These results show that the learned request agents can exploit the weaknesses of known protocols while the route agent in the self-play approach efficiently fixes these weaknesses.

Directory Algorithms: Results for directory algorithms trained against baseline request sequences are shown in Figure 2 (right). For independent, random requests the competitive ratio of Arrow is bounded by 2 when the directory is the Minimum Communication Spanning tree on any graph [8]. Given uniform requests, the shortest-path initial directory matches. So we can expect a learned policy *vs. Random* π_σ to attain an expected competitive ratio below 2. The *Roundrobin* sequence achieves linear overhead for Ivy, but Arrow again has constant expected competitive ratio. In fact, given our directory initialization on a cycle of $n = 2m + 1$ nodes, the first m requests each cost 1 and the next is across the missing edge costing $2m$. The requests are always next to the token for an optimal cost of $m + 1$. So stopping the sequence after $m + 1$ requests already reaches overhead $\frac{3m}{m+1} \leq 3$. This is a lower bound for any Arvy protocol, as the possible directory changes during request $m + 1$ affect only future requests. Our learned protocol reaches this lower bound and achieves a competitive ratio against Roundrobin of approximately 3. The third, the *Greedy* baseline sequence is a worst-case sequence against Arrow and also attains a linear overhead against Ivy (not shown here). The route agent learns a protocol with overhead around 4 and sometimes as low as 3. In the latter case, the directory is set up such that only 3 neighboring nodes are active: 2 neighbors alternate requests, but are connected in the directory through the third node. The remaining nodes all have overhead less than 3 such that they are not requested by the greedy sequence. This result shows that our route agent can adapt to specific setups, exploiting the structure of requests. Lastly, the protocol trained with self-play is evaluated against the simultaneously trained request agent. It shows the highest $\bar{\rho}$, but optimizes the worst-case competitive ratio and thus should be robust to any possible sequence of requests.

TABLE I: Average return $\bar{\rho}$ testing robustness. Lower is better.

			Evaluation		
Training	Random	Roundr.	Greedy	Self-Play	Adversary
Random	2.0	7.3	8.3	6.5	9.8
Roundr.	1.9	**3.0**	5.3	5.7	5.8
Greedy	**1.8**	4.6	4.1	5.3	6.0
Self-Play	1.9	3.9	**4.0**	**4.7**	**4.9**

B. *Robustness*

The motivation for self-play is to be robust to any possible sequence of requests. However, the self-play setup on its own does not guarantee robustness [15]. We therefore additionally evaluate robustness to adversarial attacks as in [15], where an adversary has control over one agent in a multi-agent setup and learns a policy to exploit the other, already trained agent. In our setting a separate *Adversary* request agent is trained from scratch against an already trained route agent.

Table I shows results for agents trained on a cycle with $n = 36$ nodes. Route agents trained against a fixed sequence effectively minimize their objective, but evaluating a different sequence reveals a lack of generalization. Interestingly, both agents trained *vs. Random* π_σ and *vs. Roundrobin* do not learn Arrow even though it would be an optimal solution. This can be seen in that the adversary attains a lower overhead than the request agent *vs. Arrow* in Section IV-A. The self-play directory protocol shows good robustness as it performs well against the baseline request sequences and is robust to the adversary, suffering only a slight performance degradation.

C. *Treewidth Dependence*

The *treewidth* is a measure of how treelike a graph G is. Arrow is optimal on trees which all have treewidth 1. The Arvy

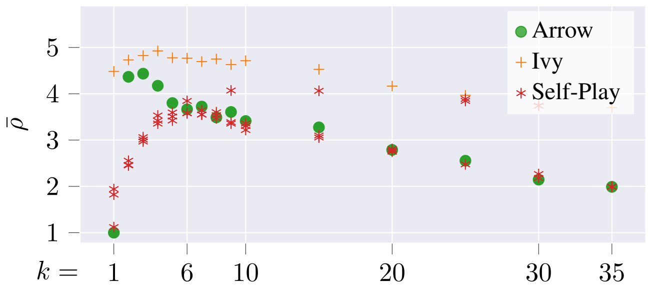


Fig. 3: Learning approach for random k-trees with 36 nodes for various treewidths k. Arrow, Ivy and learned directory protocol evaluated against *Greedy* requests. Lower is better.

Bridge heuristic has constant competitive ratio for cycles [1], which have treewidth 2. Note that Arrow and Ivy have linear overhead $\Omega(n)$ here as discussed earlier. Further, $m \times m$ grids have treewidth m, where Arrow and Ivy can be shown to be $\Omega(m)$ competitive. On fully connected graphs, i.e. graphs that are maximally non-treelike with treewidth $n-1$, Arrow again has a constant competitive ratio of 2. Given the existence of Arvy Bridge, we conjecture that there might be other Arvy protocols with constant competitive ratio on graphs of low tree width. To investigate this, we test self-play learning on random k-trees [16], a class of graphs with treewidth k. We test the self-play approach on graphs with $n = 36$ nodes, varying treewidth k from 1, which are regular trees, to 35, which gives a fully connected graph. At the beginning of each episode a new k-tree is sampled as underlying graph G. Therefore, a directory and request agent is learned which optimizes return $\bar{\rho}$ in expectation over the random k-tree graphs.

Figure 3 shows the corresponding results for 3 separate training runs. The learned protocol is evaluated against greedy requests and compared to Arrow and Ivy. For Arrow, we see a linear trend, improving with increasing k for this distribution of k-trees. Ivy has overall worse performance for all graphs here. The trained agent is able to improve on Arrow for treewidth 2 to 6 $(= \sqrt{n})$ with a significant improvement for lower treewidths, supporting our conjecture. For larger treewidths the learning approach cannot improve on Arrow. This indicates that Arrow might be the optimal solution here.

V. Conclusion

We propose to use self-play as a tool to guide the process of algorithm design. While many approaches have been proposed to learn algorithms, most require either an input-output dataset, which might not cover all cases, or synthetic execution traces, which limit the algorithmic flexibility. While approaches based on reinforcement learning overcome some of these limitations, they still can overfit to the input distribution. In constrast, the *input* agent in the self-play approach effectively seeks out inputs on which the current algorithm design performs poorly, thereby forcing the *algorithm* agent to come up with algorithms that generalize.

We demonstrate the approach by learning distributed directory protocols from the Arvy family — protocols which keep track of the location of a shared resource in a network of resource requesting nodes. Here, we model the *input* agent as a request agent π_σ that finds challenging request sequences for the *algorithm* agent, a route agent π_A that updates the distributed directory locally while solving a request. The interplay resembles algorithm prototyping in theoretical computer science: The request agent searches for a worst (high) competitive ratio of an Arvy protocol while the route agent tries to find the best (low) competitive Arvy protocol given the distribution of request sequences. Using a self-play approach, both can be trained simultaneously and we demonstrate that the learned Arvy protocols are robust to adversarial request sequences. Given that our approach can be applied to any connected graph, we learn novel Arvy algorithms that improve upon the Arrow and Ivy protocols, especially in graphs with low treewidth. We hope that our work serves as an inspiration for algorithm designers, specifically in areas where no tight performance bounds are known yet.

References

[1] P. Khanchandani and R. Wattenhofer, "The arvy distributed directory protocol," in *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*. ACM, 2019, pp. 225–235.

[2] K. Raymond, "A tree-based algorithm for distributed mutual exclusion," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 1, pp. 61–77, 1989.

[3] M. J. Demmer and M. P. Herlihy, "The arrow distributed directory protocol," in *International Symposium on Distributed Computing*. Springer, 1998, pp. 119–133.

[4] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, no. 4, pp. 321–359, 1989.

[5] D. Ginat, D. D. Sleator, and R. E. Tarjan, "A tight amortized bound for path reversal," *Information Processing Letters*, vol. 31, no. 1, pp. 3–5, 1989.

[6] F. Kuhn and R. Wattenhofer, "Dynamic analysis of the arrow distributed protocol," in *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, 2004, pp. 294–301.

[7] A. Ghodselahi and F. Kuhn, "Dynamic analysis of the arrow distributed directory protocol in general networks," *arXiv preprint arXiv:1705.07327*, 2017.

[8] D. Peleg and E. Reshef, "A variant of the arrow distributed directory with low average complexity," in *International Colloquium on Automata, Languages, and Programming*. Springer, 1999, pp. 615–624.

[9] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," *Distributed Computing*, vol. 20, no. 3, pp. 195–208, 2007.

[10] H. Attiya, V. Gramoli, and A. Milani, "A provably starvation-free distributed directory protocol," in *Symposium on Self-Stabilizing Systems*. Springer, 2010, pp. 405–419.

[11] G. Sharma and C. Busch, "Distributed transactional memory for general networks," *Distributed computing*, vol. 27, no. 5, pp. 329–362, 2014.

[12] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[13] B. Baker, I. Kanitscheider, T. M. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent autocurricula," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[15] A. Gleave, M. Dennis, N. Kant, C. Wild, S. Levine, and S. Russell, "Adversarial policies: Attacking deep reinforcement learning," *arXiv preprint arXiv:1905.10615*, 2019.

[16] C. Cooper and R. Uehara, "Scale free properties of random k-trees," *Mathematics in Computer Science*, vol. 3, no. 4, pp. 489–496, 2010.