Multiagent Coordination in the Roomba: Using Reinforcement Learning and Neuroevolution

Jimmy Xin Lin
Department of Computer Science
the University of Texas at Austin
Austin, TX 78712
jimmylin@cs.utexas.edu

Barry Feigenbaum
Department of Computer Science
the University of Texas at Austin
Austin, TX 78712
baf@cs.utexas.edu

***Abstract*—This paper presents our research about the reinforcement learning approach and the neuroevolution approach, by which the crumb collection task can be more effectively and efficiently solved with communication and coordination between multiple agents under the simulated Roombas environment. The preliminary literature part gives a brief overview about how existing works fulfill the coordination under general multiagent environments. The initial setup experimentation shows our works about the learned agents that simulate greedy strategies. The key things ... are shown in the following multiagent experiments. It is observed from our experiments that .**

I. Introduction

In the past years, Roomba Vacuum has gained its popularity in the industry of domestic services. Most of existing studies about Roomba (iRobots) is to qualitatively investigate its utility in the home as a single autonomous domestic service provider. In the contrast, our interests focus on the working efficacy of Roomba agents under a decentralized system. **TODO: improve this section**

The decentralized decision making has a long history, originated from the team thoery [1]–[5], where the decisions made by team members need to contribute to the fulfillment of global objectives. However, the individual members have only partial information about the entire system, i.e. limited knowledge of common goals and global states. This motivates the need for coordination because agents have to share resources and expertise required to achieve their goals. Researchers in the field of Distributed Artificial Intelligence (DAI) have been developing efficient mechanisms to coordinate the activities of multiple autonomous agents [6], [7]. Specifically, previous works for the multiagent coordination include using sophisticated information to exchange protocols, investigating heuristics for negotiation, and developing formal models of possibilities of conflict and cooperation among agent interests.

Reinforcement learning has been widely used as an effective techniques for autonomous game playing (Atari, Pacman, and Angry Bird) for either single-agent or multi-agent environment. An early study [8] shows that under the reinforcement learning framework, additional sensation from another agent is benefial if it can be used effiently. The benefits lie in the faster learning process at the cost of communication and the better performance for joint tasks. Based on this investigative study, we consider to explore the Roomba multiagent system through reinforcement learning approaches.

TODO: make the intro here.. Neuroevolution.

[9] Real time neuroevolution for NERO game.

In this paper, we investigate various multiagent coordination techniques under the framework of reinforcemnet learning and neuroevolution could improve the working efficacy of Roomba in the task of cleaning the floor. Our contributions include: (i) set up the raw Roomba System. (ii) implemented reinforcement learning mechanism and fixed up the default neuroevolution mechanism. (iii) design sensors and their representations for multiagent communication and coordination. (iv) compare performances of the agents learned through various approaches and settings. **TODO: update and check the contributions**

The remained part of this paper is organized as follows. In section II, a detailed presentation about the existing reinforcement learning treatments and the neuroevolution treatments for multiagent systems. Section III describes in detail the virtual Roomba environment, as well as the practical issues encountered, and the multiagent behaviors we expected to observe in our experiments. The experiments that demonstrate our attempts and achievements on *the reinforcement learning based* and *the neuroevolution based* multiagent coordination are articulated in the section IV and the section V respectively. We summarize our conclusions and discuss some of promising future works in the section VI.

II. Technical Literatures

This section summarizes previous work on multiagent coordination using reinforcement learning and neuroevolution techniques.

A. Multiagent Planning as Optimization

The Distributed Constraint Optimization Problem (DCOP) is a promising approach for modeling distributed reasoning tasks that arise in multiagent systems. **Within the DCOP framework...** Agents need to coordinate their value assignments to maximize the sum of the resulting constraint utilities. An utility-propagation method was proposed to address the large-scale multiagent constraint optimization problem [10].

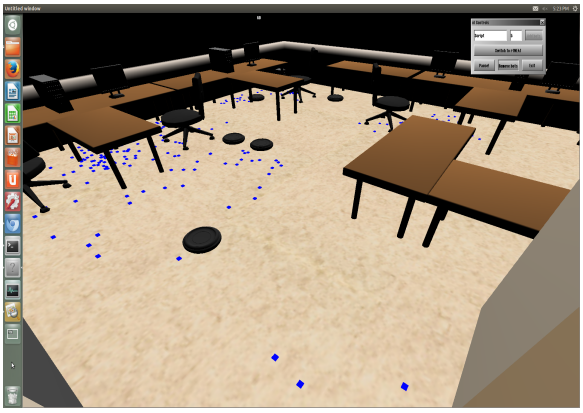

Fig. 1: A exemplar Roomba robot in the real world.

One recent study [11] employed alternating direction method of multipliers (ADMM) to derive solutions for the DCOP.

Dynamic Distributed Constraint Optimization Problem (Dynamic DCOP) was proposed to model dynamically changing multiagent coordination problems. A dynamic DCOP is a sequence of static DCOPs, each of which partially different from the DCOP preceding it. The formulation of Dynamic DCOPs allows us to model problems that can not be assumed to be static, with particular advantages when the problems change so frequently that by the time a DCOP solver has found a solution it is already obsolete. A series of asynchronous algorithms have been proposed to approximately solve the Dynamic DCOP. For example, the ADOPT method provides an available solver with theoretical guarantees on the global solution [12]. Also, Support-Based Distributed Optimization (SBDO) serves as a robust alternative for solving the Dynamic DCOPs problems [13].

B. Reinforcement Learning Based Coordination

Factored Markov Decision Process (Factored MDP) is a common model-based reinforcement learning formulation for cooperative multiagent dynamic systems. This framework simply views the entire multiagent system as a single, large MDP, which can be represented in a factored way using a dynamic Bayesian network (DBN) [14]. It implies that the action space of the resulting MDP is the exponential union of action space of the entire set of agents. One effective approach is to employ factored linear value functions as an approximation to the joint value function, which can be solved simply by a single linear program [14]. A large collection of algorithms are developed based on the factored MDP model. These algorithms have a parameterized, structured representation of a policy or value function, by which agents coordinate both their action selection activities and their parameter updates and then determine a jointly optimal action without explicitly considering every possible action in their exponentially large joint action space [15]. However, the factored MDP may not be applicable to the real application due to its large search space and associated complexity. To address this problem, an accelerated gradient method comes up with $\mathcal{O}(\epsilon^{-1})$ rate of convergence for solving the distributed multi-agent planning with Factored MDPs [16].

[17] This paper presents a model-free, scalable learning approach that synthesizes multi-agent reinforcement learning (MARL) and distributed constraint optimization (DCOP). By exploiting structured interaction in ND-POMDPs, our approach distributes the learning of the joint policy and employs DCOP techniques to coordinate distributed learning to ensure the global learning performance. Our approach can learn a globally optimal policy for ND-POMDPs with a property called groupwise observability. Experimental results show that, with communication during learning and execution, our approach significantly outperforms the nearly-optimal non-communication policies computed offline.

[18] Existing work typically assumes that the prob- lem in each time step is decoupled from the problems in other time steps, which might not hold in some applications. Therefore, in this paper, we make the following contributions: (i) We introduce a new model, called Markovian Dynamic DCOPs (MD-DCOPs), where the DCOP in the next time step is a function of the value assignments in the current time step; (ii) We introduce two distributed reinforcement learning algorithms, the Distributed RVI Q-learning algorithm and the Distributed R-learning algorithm, that balance exploration and exploitation to solve MD-DCOPs in an online manner; and (iii) We empirically evaluate them against an existing multi-arm bandit DCOP algorithm on dynamic DCOPs.

[19]

[20]

[21]

[22] Complex problems involving multiple agents exhibit varying degrees of cooperation. The levels of cooperation might reflect both differences in information as well as differences in goals. In this research, we develop a general mathematical model for distributed, semi-cooperative planning and suggest a solution strategy which involves decomposing the system into subproblems, each of which is specified at a certain period in time and controlled by an agent. The agents communicate marginal values of resources to each other, possibly with distortion. We design experiments to demonstrate the benefits of communication between the agents and show that, with communication, the solution quality approaches that of the ideal situation where the entire problem is controlled by a single agent.

[23]

C. Neuroevolution for Multiagent Coordination

Barry: Add literatures of Neuroevolution here...
ATA: barbarian

III. The Roomba Environment

To the best of our knowledge, the OpenNERO platform is the best choice for cheap software simulations. The OpenNERO is an open-source platform for artificial intelligence research and education [24]. The public release of the OpenNERO contains an underdeveloped implementation of Roomba

environment, which supports the communication and coordination of multiple robots.

A. Structures

The Roomba environment is a virtual computer lab with crumbs distributed on the floor. In this virtual lab, there are four classes of objects: agents, crumbs, walls, decorations. Vacuum cleaner agents, shown as grey cylinders in Fig. 2, are supposed to collect the static crumbs that are labelled as blue cubes. The agents will be rewarded if they move to a place where there are some crumbs. Walls are also set up as the boundaries of the computer lab, such that agents are not allowed to move beyond the walls and no pellets can be placed outside the walls. Other decorative objects within the virtual environment include tables, chairs, and computers. For simplicity at the moment, these decorations only serve as physically transparent decorations, which means they do not block agents' movements.

As shown in the Fig. 2, the small window floating on the top right is the controller that masters the type of scripted AI agents to load in, the number of agents, and particular commands that impact the progress of the Roombas simulation (Pause/Resume, Add/Remove Robots, Exit). On each run of the simulation, only one particular type of AI agents are allowed to be loaded. Note also that the type of AI agents employed cannot be switched to the other one during the intermediate process of crumb collection task. If one needs to switch to the other type of AI agents, the running agents have to be removed and then the user is able to effectively load the desired AI agents.

In the Roombas environment, the movements of vacuum cleaner agents are not constrained by four directions (left, right, forward, backward). Instead, agents are able to move towards all directions and each moving action is denoted as a continuous radius value. Besides, all agents move in a synchronous way. Agents are allowed to make their movements of the following step if and only if all agents have completed their movements in the previous step.

Three different modes (MANUAL, RANDOM, and CLUSTER) are employed to specify the placing position of each individual crumb. In the MANUAL mode, one pellet is deterministically placed at a user-specified position. In the RANDOM mode, the placement of one pellet is totally randomized; the environment will throw a rejection and repeat the random generation, if an invalid position is yielded. The last one is the CLUSTER mode, where the position of pellets are partially randomized. In this mode, environment samples the position of one pellet from a gaussian distribution whose centers and spreads at x and y coordinates are specified. In this paper, the specifications for the placement of crumbs are generated from the starting CLUSTER-mode experiment and remained invariant by employing MANUAL mode in all experiments afterwards.

The learning process designed in the Roomba environment is as follows. The physically dynamic component, consisting of robots and pellets, will be reset to their initial status once one episode expires. Except the spatial positions, all episodic experience and learned knowledge (i.e. the accumulated Q values from previous episodes) are allowed to preserve through two adjacent episodes. Note that the default configuration for one episode is 100 steps. That is, for every 100 steps, all agents will be placed back to their birth places no matter how many pellets they have collected and all crumbs are set at their specified locations, regardless of whether these crumbs exist at the end of last episode.

Fig. 2: Overall Picture of the Roomba Environment

B. Practical Issues

What agents are allowed to perceive from their local views is the practical issue that matters the most. This is important because both of the design of sensors and their representations have significant impact on the learning outcome of a team. The default implementation of Roomba environment allows each agent to sense all crumbs on the floor about their positions, existing status, and even rewards. In this sense, agents are able to perceive limitless information from the computer lab. In addition, the spatial information of other collaborated robots is available for each individual agent, as well as the user-defined working status of these teammates (e.g. a history of previous positions and movements). Although the Roomba environment provides a large design space for sensors, it is a practical treatment to start from a small number of simple sensors. For example, a combination of the bumping status, the position of its own, and the location of closest crumb suffice for an agent to learn a greedy strategy, as illustrated in the initial setup experimentation.

The reward design is another big issue for configuring the Roomba environment. By default, the only reward being set up for agents is when they successfully collect some pellets on the floor. In order to facilitate the learning of agents, an penalty for being alive is supposed to be incorporated to the reward system. That is, agents should receive some negative rewards, typically a very small quantity, for each step they move. Similarly, penalties can also be granted to the collisions between roombas, bumping of agents towards the world boundaries, and repetitive movements around an area.

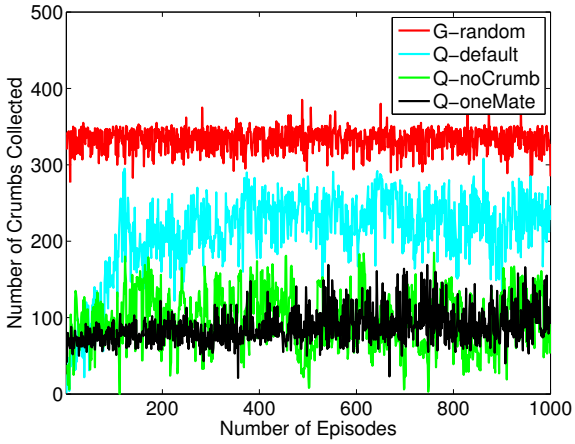

Fig. 3: Q-learning with various simple sensors. (i) Red: G-random curve represents the greedy agent with 0.1 probability to make a random decision, which serve as the benchmark agents. (ii) (iii) (iv) agents receive various collection of sensors for their decision making of each step. Q-noCrumb, Q-default, and Q-oneMate represent the learning performances of agents that receive S_{nc}, S_d, and S_{om} respectively.

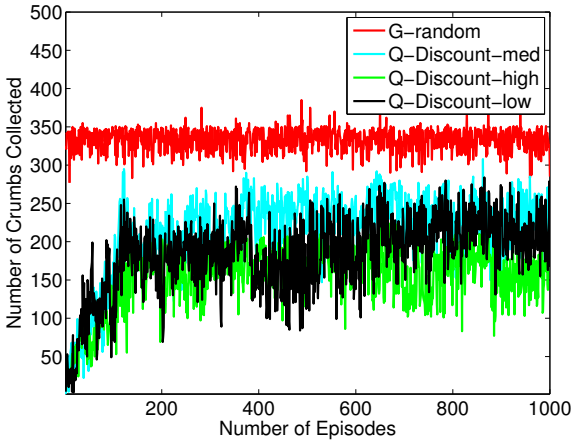

Fig. 4: Q-learning with various discounting factors. (i) Red: G-random curve represents the greedy agent with 0.1 probability to make a random decision, which serve as the benchmark agents. (ii) (iii) (iv) the tilted tabular Q-learning with discounting factors $\gamma = 0.1, 0.5, 0.8$ for Q-Discount-low, Q-Discount-med, and Q-Discount-high respectively.

C. Expected Multiagent Behaviors

Workload balance is the foremost coordinative behaviors the agents should reflect in the Roomba System. This is also known as competition avoidance. If the workload can be evenly distributed to each agent, in terms of local perceptions and limited amount of communicative information, the team is more likely to yield the globally optimal performance.

Another Collision avoidance.

IV. Reinforcement Learning Experiments

This section presents our experimental investigation in the thread of reinforcement learning based agents.

A. Greedy-Random Agents

Greedy-random Agents serve as the benchmark for our reinforcement learning based agents. Agents with greedy strategy simply approaches to the direction where the closest pellet to it is there. The greedy-random agents basically follow such greedy strategy but they make their final decisions with some probability to make random movements. We incorporate such randomness because we need to .

B. Tiled Tabular Q-learning Agents

The motives to use Tiled Tabular Q-learning include (i) verifying that we have set up the environment correctly for general reinforcement learning techniques. (ii) illustrating the benchmark (reinforcement learning) intelligence for the crumb collection problem.

Before diving into the investigation of multiagent coordination, we take preliminary experiments to investigate the impact of various collections of simple sensors and the effects of various discounting factors on the outcome of the tiled tabular Q-learning.

TODO: explain some other constans (Number of agents, pellets,)

The sensors designed for the local perspective of each agent are as follows.

$$S_{nc} = \begin{pmatrix} self.position.x \\ self.position.y \end{pmatrix} S_d = \begin{pmatrix} self.position.x \\ self.position.y \\ closestCrumb.x \\ closestCrumb.y \end{pmatrix}$$

$$S_{om} = \begin{pmatrix} self.position.x \\ self.position.y \\ closestCrumb.x \\ closestCrumb.y \\ DistToClosestMate.x \\ DistToClosestMate.y \end{pmatrix}$$

Note that we discretize all real-value sensors by using the tiling techniques mentioned in the previous section.

The learning outcomes derived from these sensors are compared in the Fig. 3. Observations tell us that the positional information of the closest crumb tremendously contributes to the quality of agents' learned policies. In addition to that, the relative position to the closest collaborated agent indeed hinder the Q-learning process, such that the communicative agents (black curve) performs worse than the team that do not monitor collaborators' positions (light-blue curve) during the observed episodes. One exciting discovery behind the figure is the narrow performance gap between the team of tiled tabular Q-learning agents and that of greedy-random agents. It turns out that the collective works made by naive Q-learning agents even outperform the team with greedy strategies.

The Fig. 4 shows the effects that various discounting factors will bring to the outcome of tilted tabular Q-learning. It turns out that the best learning outcome came from the discounting factor $\gamma = 0.5$ under the given setting. On top of that, this chart also indicates the negative effects of too large or too small discounting factors, by which worse learning outcome arises. Nevertheless, it can be concluded that the team formed by tabular Q-learning agents cannot never outperform the team whose members employ the greedy strategy, regardless of how the discounting factor is set.

C. High-level Reinforcement Learning

After the preliminary experiments, it is natural to investigate the problem of how to incorporate effective coordinations for this particular multiagent system.

V. Neuroevolution Experiments

In the following experiments, agents were controlled by neural networks trained using NeuroEvolution of Augmenting Topologies (NEAT) [25]. In particular, we evolved these networks with real-time NEAT, which does not use generations - instead replacing agent's networks without halting the simulation. However, the search space is eventually depleted of crumbs and must be reset. Simulations are therefore divided into *episodes* which last a fixed interval, and each episode begins by distributing agents and crumbs around the environment. Agents' networks are continually swapped out throughout an episode according to the rtNEAT algorithm.

Experiments using the original configuration of the Roomba module yielded poor results. Agents performance did not increase over time. Even after hundred of episodes, no evidence of learning occurred. This section discusses the steps necessary to enable even basic training of the Roomba agent. These results serve as a yardstick for later experiments on multiagent coordination.

In order to facilitate learning, a larger population size is necessary. However, anything more than a handful of Roombas quickly picks up all of the crumbs, even when moving in random directions. To combat this, Roombas are reduced to a third of their original radius. Despite this, they often manage to collect all of the available crumbs before time expires, making it difficult to compare the results of different experiments. In this situation, our methodology has been to lower the cap on agents' speed until they are unable to collect every crumb before an episode ends. Additionally, the total time taken to collect all crumbs was compared as a secondary performance metric.

A. Fitness

Originally, an agent's fitness was based purely on collecting crumbs. Various other mechanisms were considered, including ability to obstacles and avoid other agents. Experimental evidence showed the best results came from making the fitness gain from crumbs very high, or by moderately penalizing fitness for colliding with walls. The end fitness function is simply:

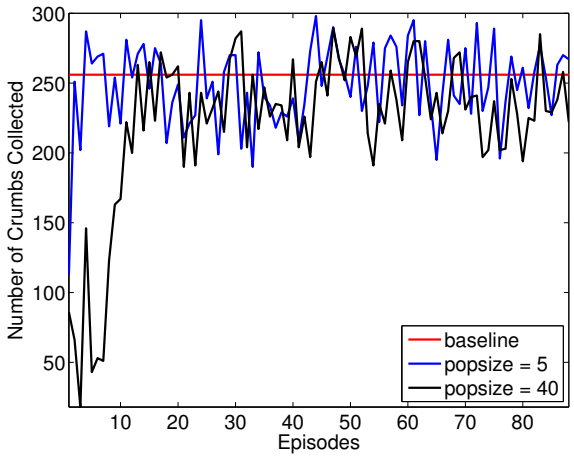

Fig. 5: Effect of population size on learning

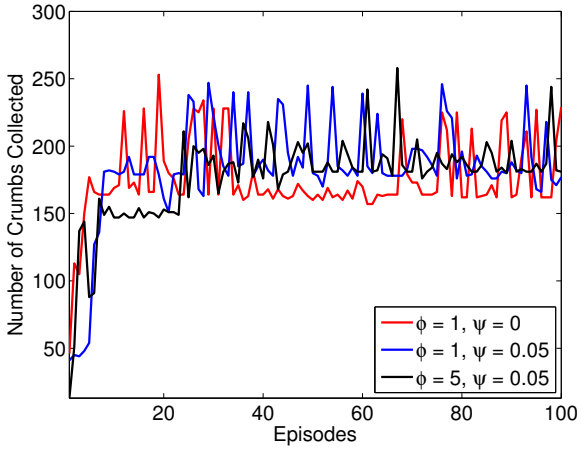

Fig. 6: Comparison of fitness functions

$$fitness = \phi * n_{crumbs} - \psi * n_{collisions}$$

Where n_{curmbs} is the number of crumbs collected and $n_{collisions}$ the number of wall collisions over the course of an episode.

Interestingly, using these two mechanisms together resulted in worse performance than either on their own, as if simultaneously increased reward and penalty negated each others effects. Experimentally derived results show $\phi = 1, \psi = 0.05$ as good choices (Fig. 6).

B. Multiagent Baseline

In all of our experiments, we compare the results of neuroevolution against hardcoded search behavior as a baseline. We use a simple greedy algorithm - each agent always moves directly towards the nearest crumb. Obviously, this is not an optimal algorithm, and we hope to show that neroevolution can produce a superior result. In particular, the greedy algorithm's agents do not coordinate their efforts. We hope that our

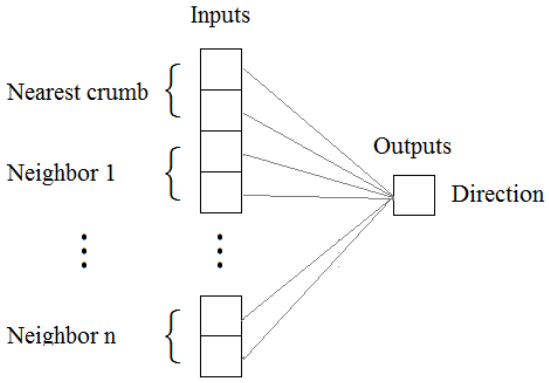

Fig. 7: Communicating network.

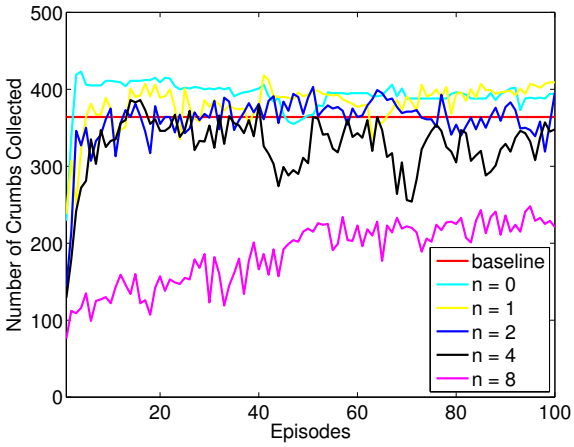

Fig. 8: Communicating network results

evolved agents will learn to better cooperate in order to achieve superior results.

As a starting point, we try to match the behavior of the greedy algorithm using the simplest possible network. A single sensor detects the nearest crumb. The network receives the angular direction to that crumb as input, and outputs the angular direction the roomba should turn to face. In order to emulate the behavior of the greedy algorithm, the network must map simply the input unmodified to the single output. Unsurprisingly, the network learns this behavior very quickly.

In order to improve over the greedy algorithm, it suffices to add just one more sensor. This sensor detects the distance to the nearest crumb. Equipped with both an angle and distance to the crumb, the network now learns behavior superior to the greedy algorithm. (Fig **??**)

This makes sense, as agents can now leverage the information of how far away a crumb is. When a crumb is close by, it is likely that an agent will be able to capture it, improving the agent's fitness. When the crumb is further away, however, it is more likely that another agent will reach that crumb first.

We have establishes that training with rtNEAT can outperform the hardcoded greedy algorithm. However, agents are still entirely independent, not directly sharing information. An argument can be made that they are indirectly coordinating through the position of crumbs in the environment, but one wonders if better results can be achieved through direct communication.

The network design, shown in Fig. 7, is straightforward. Inputs consist of the offset of the nearest crumb as well as that the nearest n agents. Offsets are polar, consisting of an angular direction and a scalar magnitude. As before, the single output represents the angular direction the agent will turn to face.

We compare results for different choices n.

As seen in Fig. 8, the more neighbors the agent communicates with, the longer it takes to learn. After these networks were left to train for 500 generation, only the network with $n = 1$ managed to equal the performance of the $n = 0$ network. Fig. 8 shows that the one neighbor network catches up after about 20 episodes. Note that what both of these networks cap out at aproximately 400 crumbs per episode, there are still crumbs remaining at this time. (The total number of crumbs is about 435.) So while there is still room for improvement over the $n = 0$ network, none of the communicating networks achieve better results. In fact the networks with higher n-values do *worse* than the non-communicating network, plateauing at significantly lower crumb counts.

Previous research by ??? and ??? has showed that sharing fitness functions between agents can improve multi-agent learning. In some cases, it is necessary to see any results [cite ???].

To explore this, we use a modification of the Roomba environment to share fitness between agents. Instead of being rewarded only when picking up a crumb, agents share the reward when one of their teammates finds a crumb.

Fig ??? show the results for a non-communicating network ($n = 0$) and a network which observes one neighbor ($n = 1$). In both cases, the shared fitness networks perform abysmally, showing no learning whatsoever.

Counterintuitive, or experiments have shown that communication is detrimental to the learning process. However, conveying the location of fellow agents is a very simple form of communication. It is possible that with different signals and messages, the agents could learn to use communication to their advantage.

One way to test this is to have the network evolve its own communication (Fig. 9). Each network is given an additional output - a signal to broadcast. As input, the agent senses the direction and strength of signal being broadcast from it's n nearest neighbors.

In order for communication to be valuable, agents must have information to share. Each agent is equipped with a sensor which detects how many crumbs are within a small radius. A radius of 1/8 of the length of the environment was chosen through experimentation - a larger radius rendered the sensor useless.

The hope was that agents would learn to communicate their short range observations to each other in order to improve their

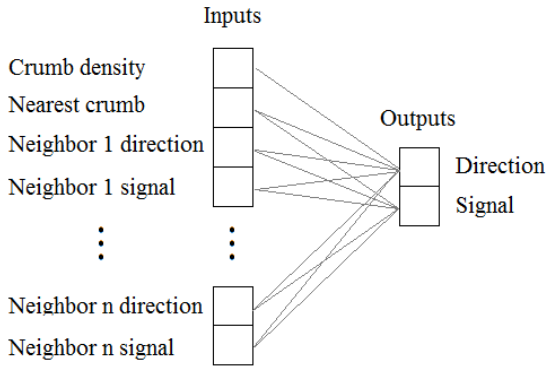

Fig. 9: Communicating network with evolved signals.

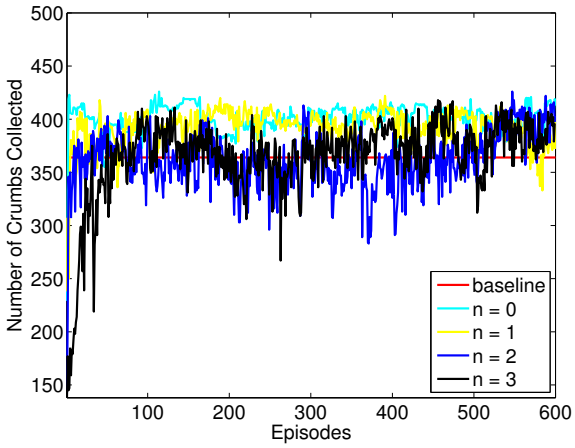

Fig. 10: Comparison of evolved communication at different n-values.

overall performance. Unfortunately, results were little better than with the simple communicating network (Fig. 10). Again, the more neighbors communicated with, the longer it took a network to learn. Ultimately, the heigher n-value networks did reach the same performance as the 0-neighbor network, but not until over 500 episodes. Whereas the 0-neighbor network peeked within 10 episodes, and the 1-neighbor in about 25. At no point did any of the communicating networks consistently outperform the $n = 0$ network.

VI. Conclusions

The conclusion goes here. In this paper, we investigate. Promising future Works go here.

References

[1] J. Marschak, "Elements for a theory of teams," *Management Science*, vol. 1, no. 2, pp. 127–137, 1955.

[2] R. Radner, "Team decision problems," *The Annals of Mathematical Statistics*, pp. 857–881, 1962.

[3] ——, "The application of linear programming to team decision problems," *Management Science*, vol. 5, no. 2, pp. 143–150, 1959.

[4] Y.-C. Ho *et al.*, "Team decision theory and information structures in optimal control problems–part i," *Automatic Control, IEEE Transactions on*, vol. 17, no. 1, pp. 15–22, 1972.

[5] J. N. Tsitsiklis and M. Athans, "On the complexity of decentralized decision making and detection problems," *Automatic Control, IEEE Transactions on*, vol. 30, no. 5, pp. 440–446, 1985.

[6] G. Weiss, *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

[7] M. N. Huhns, *Distributed artificial intelligence*. Elsevier, 2012, vol. 1.

[8] M. Tan, "Multi-agent reinforcement learning: Independent vs. cooperative agents," in *Proceedings of the tenth international conference on machine learning*, vol. 337. Amherst, MA, 1993.

[9] K. O. Stanley, B. D. Bryant, and R. Miikkulainen, "Real-time neuroevolution in the nero video game," *Evolutionary Computation, IEEE Transactions on*, vol. 9, no. 6, pp. 653–668, 2005.

[10] A. Petcu and B. Faltings, "A scalable method for multiagent constraint optimization," Tech. Rep., 2005.

[11] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[12] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo, "Adopt: Asynchronous distributed constraint optimization with quality guarantees," *Artificial Intelligence*, vol. 161, no. 1, pp. 149–180, 2005.

[13] G. Billiau, C. F. Chang, and A. Ghose, "Sbdo: A new robust approach to dynamic distributed constraint optimisation," in *Principles and Practice of Multi-Agent Systems*. Springer, 2012, pp. 11–26.

[14] C. Guestrin, D. Koller, and R. Parr, "Multiagent planning with factored mdps." in *NIPS*, vol. 1, 2001, pp. 1523–1530.

[15] C. Guestrin, M. Lagoudakis, and R. Parr, "Coordinated reinforcement learning," in *ICML*, vol. 2, 2002, pp. 227–234.

[16] S. A. Hong and G. Gordon, "An accelerated gradient method for distributed multi-agent planning with factored mdps." in *4th NIPS Workshop on Optimization for Machine Learning*, 2011.

[17] C. Zhang and V. R. Lesser, "Coordinated multi-agent reinforcement learning in networked distributed pomdps." in *AAAI*, 2011.

[18] D. T. Nguyen, W. Yeoh, H. C. Lau, S. Zilberstein, and C. Zhang, "Decentralized multi-agent reinforcement learning in average-reward dynamic dcops," in *Proceedings of the 2014 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2014, pp. 1341–1342.

[19] C. Zhang and V. Lesser, "Coordinating multi-agent reinforcement learning with limited communication," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 1101–1108.

[20] B. Banerjee, J. Lyle, L. Kraemer, and R. Yellamraju, "Sample bounded distributed reinforcement learning for decentralized pomdps." in *AAAI*, 2012.

[21] L. Kraemer and B. Banerjee, "Informed initial policies for learning in dec-pomdps." in *AAAI*, 2012.

[22] A. Boukhtouta, J. Berger, W. B. Powell, and A. George, "An adaptive-learning framework for semi-cooperative multi-agent coordination," in *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2011 IEEE Symposium on*. IEEE, 2011, pp. 324–331.

[23] S. Sen, M. Sekaran, J. Hale *et al.*, "Learning to coordinate without sharing information," in *AAAI*, 1994, pp. 426–431.

[24] I. Karpov, J. Sheblak, and R. Miikkulainen, "Opennero: A game platform for ai research and education." in *AIIDE*, 2008.

[25] K. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.