# Multiagent Coordination with Roombas: 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*—Developing artificial intelligence agents which can cooperate with each other is an important research problem. The *multiagent search problem* is a key example of a problem which requires agent coordination in order to achieve good results. In this paper, we apply reinforcement learning and neruoevolution techniques to teams of agents. We show how communication and coordination can be leveraged to improve the learning process and more effectively solve the task at hand. Here the problem of agent coordination is formulated in terms of multiagent search. In particular, we develop intelligence for autonomous vacuum cleaning robots (Roombas) as they attempt to explore a search space. Our experiments are conducted using the OpenNERO environment.**

## I. Introduction

Decentralized decision making has a long history, originated from the team theory [1]–[5], where the decisions made by team members need to contribute to the fulfillment of global objectives. Individuals 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 in order to achieve their goals. Researchers in the field of Distributed Artificial Intelligence (DAI) have been developing efficient mechanisms for coordinating the activities of multiple autonomous agents [6], [7]. Previous work on multiagent coordination include using sophisticated information to exchange protocols, investigating heuristics for negotiation, and developing formal models to describe the possibility for conflict and cooperation among agents.

Reinforcement learning has been widely used as an effective techniques for game playing (TD-Gammon [8] and NERO [9]) in either single-agent or multi-agent environment. An early study [10] shows that under the reinforcement learning framework, additional information from another agent is beneficial if it can be used efficiently. The benefits include a faster learning process and better performance for joint tasks. Based on this investigative study, we consider the application of reinforcement learning to the space of multiagent search problems.

Neuroevolution has been widely studied for its applications to game playing and control systems. Previous work has shown that neuroevolution can be used to coordinate agents. In particular, it has been used to develop cooperative search for both static and moving targets [11]–[13]. For our experiments, we use the OpenNERO platform [14], which includes an implementation of real-time *Neuroevolution of Augmenting Topologies* (NEAT) [9].

In this paper, we investigate various multiagent coordination techniques under the framework of reinforcement learning and neuroevolution, with the aim of developing improved distributed search techniques. Our contributions include: (i) setting up the Roomba environment for multiagent learning, (ii) implementing reinforcement learning mechanisms and correcting significant issues with the Roomba module's neuroevolution setup, (iii) designing sensors and other mechanisms specifically for multiagent communication and coordination, and (iv) comparing agents' performances through the various learning approaches and settings.

The remained part of this paper is organized as follows. In section II, a detailed presentation about the existing reinforcement learning and neuroevolution literature 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. Our experiments using *reinforcement learning* and *neuroevolution* based multiagent coordination are covered in section IV and the section V respectively. We summarize our conclusions and discuss some promising future works in the section VI.

## II. Technical Literature

This section summarizes previous works on the topic of multiagent planning and coordination from the perspectives of optimization, reinforcement learning, and neuroevolution respectively.

### A. An Optimization Perspective

The *Distributed Constraint Optimization Problem* (DCOP) has for a long time been a promising approach for modeling distributed reasoning tasks that arise in multiagent systems, where agents need to coordinate their value assignments to maximize the sum of the resulting constraint utilities. Within the DCOP framework, multiagent problems can be solved efficiently using fully distributed algorithms which are often based on existing centralized techniques. For instances, the OptAPO algorithm [15] performs partial centralization and serves as an effective approach to find solutions for the multiagent planning. In addition, the ADOPT method, introduced in [16], maintains distribution of the DCOP and achieves theoretical guarantees on its global solution. It also turns out that the degree of allowed centralization has significant impact on the amount of computation required by an agent [17]. Other solvers include an utility-propagation method that is amenable to large-scale multiagent constraint optimization problem [18] and an alternating direction method of multipliers (ADMM) based algorithm [19].

*Dynamic DCOP* (Dyn-DCOP) was proposed to model dynamically changing multiagent coordination problems. A Dyn-DCOP is a sequence of static DCOPs, each of which partially different from the DCOP preceding it. The formulation of Dyn-DCOP 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. Support-Based Distributed Optimization (SBDO) was proposed as a robust method to solve the Dyn-DCOP problems [20].

*Markovian Dynamic DCOP* (MD-DCOP) was recently proposed by the study [21]. Under MD-DCOP formulation, the DCOP in the next time step is a function of the value assignments in the current time step. The study also proposed two distributed reinforcement learning algorithms: the Distributed RVI Q-learning algorithm and the Distributed R-learning algorithm. This demonstrated that the MD-DCOP enables model-based reinforcement learning techniques to solve DCOPs.

## *B. A Markov Decision Process Perspective*

Another thread that motivates the reinforcement Learning approach sources from *Markov decision process* (MDP), which is the fundamental model of agents interacting with an environment. A *partially observable Markov decision process* (POMDP) is a generalization of an MDP in which an agent must base its decisions on *incomplete* information about the state of the environment. In one multiagent system where autonomous agents are enforced, agents have to make their decision in terms of their private perceptions and limited communication with other agents. Such *decentralized partially observable Markov decision process*, abbreviated as DEC-POMDP, is the modelling that suits most of multiagent systems. A series of academic works have been proposed to address practical problems of DEC-POMDP, including sampling [22] and initialization [23].

*Network distributed POMDP* (ND-POMDP) was introduced as a synthesis of distributed constraint optimization and partially observable Markov decision process [24]. On the one hand, POMDP capture the real-world uncertainty in multiagent domains, they fail to exploit such locality of interaction. On the other hand, DCOP captures the locality of interaction but fails to capture planning under uncertainty. One model-free, scalable learning approach, employing the special structure of ND-POMDP, finds globally optimal policy under the assumption of groupwise observability and outperforms the nearly-optimal non-communication policies computed offline [25].

*Factored Markov decision process* (Factored MDP) is a common model-based reinforcement learning technique 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) [26]. 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 [26]. 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 [27]. 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, the *accelerated gradient method* comes up with $\mathcal{O}(\epsilon^{-1})$ rate of convergence for solving the distributed multi-agent planning with Factored MDPs [28].

## *C. Neuroevolution for Multiagent Coordination*

Neuroevolution is a prominent technique for training neural networks, with many applications to game playing and agent control problems. The *predator-prey problem* is one such example to which neuroevolution has been successfully applied. In this problem, several predator agents attempt to capture a fleeing prey. Since the prey is faster than the predators, the agents must learn to cooperate in order to succeed.

Research into the predator prey problem by [11] found that learning is faster in a distributed model where each agent has it's own network, rather than using a single centralized network to coordinate agents' actions. A more surprising discovery was that communication can actually detract from agents' ability to coordinate. Without direct communication, agents were able to coordinate simply by observing each others' changes to the environment. This form of indirect communication is referred to as *stigmergy*, and agents proved to learn faster using stigmergy than direct communication.

In a follow up work, [12] compared communication and stigmergy in more complicated environments with multiple prey. They discovered that in more complicated environments communicating networks outperform noncomunicating ones. This indicates that while stigmergy is effective when changes to the enviroment are obvious, it becomes increasingly difficult as the enviroment becomes more complicated.

In the setting of game playing, work by [29] shows that multiagent cooperation be achieved through neuroevolution of homogeneous networks. Furthermore these networks can be trained such that agents adopt completely distinct strategies despite having identical networks.

Yet another method for multiagent training is *social learning*. Social learning allows an agent to leverage the knowledge of other networks to enhance the learning process. In many social learning techniques, agents select a teacher or model, from which they attempt to learn information or behavior [30]. This is called the *student-teacher model*, and the success of these methods depends highly on the mechanism by which teachers are selected.

Other social learning approaches do not use the student-teacher method. *Opportunistic Cooperative Learning* (OCL) is a technique designed for multiagent search problems which combines neural networks and social learning [31]. In OCL, nearby agents can learn from each other, with less successful agents adopting network weights from more successful teammates.

In another method, called *Egalitarian Social Learning* (ESL), agents also learn from each other without designated teachers [32]. In ESL, the population is partitioned into subcultures, with agents free to learn from any member of their subculture. This method helps improve diversity and avoid premature convergence.

## III. The Roomba Environment

We use OpenNERO as a simulation environment and artificial intelligence framework for our experiments. OpenNERO is an open-source platform for artificial intelligence research and education [14]. The public release of the OpenNERO contains an underdeveloped implementation of Roomba environment, which supports the communication and coordination of multiple robots. This module is designed for agent-based search problems, and, with some modification, it is well suited to our study of multiagent coordination.

### *A. Structure*

The Roomba environment is a virtual office space with crumbs distributed on the floor. In this virtual space, there are four classes of objects: agents, crumbs, walls, and obstacles. Roomba agents, shown as grey cylinders in Fig. 1, collect the static crumbs that are depicted as blue cubes. Agents have a full 360° movement; each action is denoted as a continuous radius value. The agents automatically pick up all crumbs within a certain radius. All movement is simultaneous, and agent actions are synchronized. Walls indicate boundaries of the environment, and there are other obstacles objects within the virtual environment such as tables and chairs. Collision detection for these obstacles can be enabled or disabled in order to vary the complexity of the experiment. For our purposes, only collisions with walls are enabled.

As shown in the Fig. 1, the environment can be configured to select which type of AI agents to load in as well the number of agents to employ. On each run of the simulation, only one type of AI agents may be loaded and this setting cannot be changed while the simulation is in progress. If one needs to switch to the other type of AI agents, the running agents have

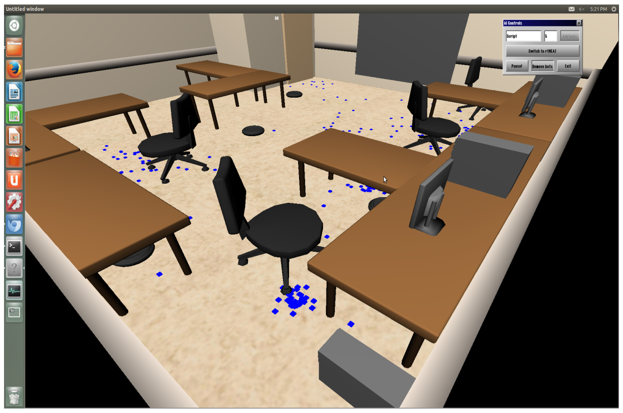

Fig. 1: **The Roomba Environment.** Roomba agents are shown as grey cylinders. The blue cubes represent crumbs to be collected. Tables and chairs provide obstacles – though in most experiments, collision detection is disabled.

to be removed and then the user is able to effectively load the desired AI agents.

The distribution of crumbs takes place in one of three different modes (MANUAL, RANDOM, and CLUSTER). 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. Note that in any of the three modes, crumb placement does not change when the environment is reset, so agents will train on the same crumb distribution for the lifetime of the environment.

The lifecycle of the Roomba environment takes place in *episodes*. By default, the configured length for one episode is 100 steps, and Roomba agents and pellets are reset to their initial status once one episode expires. Episodic experience and learned knowledge (such as accumulated Q values and network weights) are preserved across episodes. As there is no hard cap on the number of episodes in an experiment, training can continue as long as necessary. The experiments in this paper are of no standard length, but rather vary with the configuration being examined.

### *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.

### C. *Expected Multiagent Behaviors*

*Workload Balance.* This is the foremost coordinative behaviors the agents are supposed to reflect in the Roomba System. This is also known as competition avoidance. From the global perspective, if the workload can be evenly distributed to each agent, the team is more likely to yield the globally optimal performance. Nevertheless, given that all agents make their autonomous decisions in terms of local perceptions and limited amount of communicative information, it is still a challenging issue to find the joint policies that are globally optimal.

*Collisions Avoidance.* Collision-free vacuum cleaners, with no doubt, consume less time and thus achieving better working efficacy with less physical damages. The general ”collisions” refers to (i) collisions between Roombas agents, (ii) collision of agents with the decorations (i.e. desks, chairs, etc.), and (iii) bumping of agents towards walls. The scenarios (i) are formulated as an issue explicitly about the multiagent coordination, which is the focus of our research. The second and third class of collisions can be implicitly correlated to the multiagent behaviors, in a sense that if robots are able to share their geographic knowledge about the computer lab, these two types of collisions can be avoided in a large degree. Note that concerns for the (ii) category of collisions can be ruled out, since we disable the collisions of agents with the decorations for simplicity.

## IV. Reinforcement Learning Experiments

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

### A. *Experimental Settings*

The default settings are set for fair performance comparison between all experimented agents in this section. The detailed specifications are as follows:

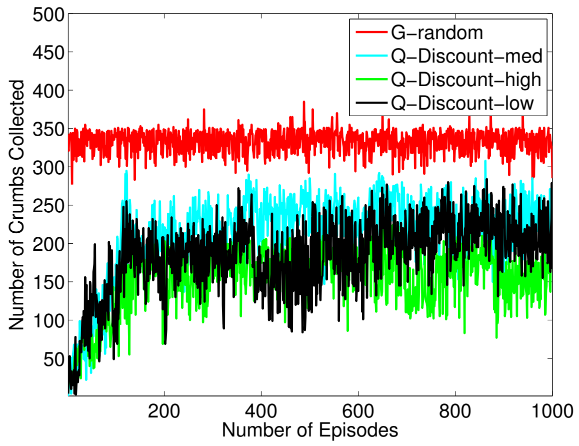

Fig. 2: **Tiled Tabular Q-learning Agents with various discounting factors.** (i) Red: G-random curve represents the greedy agent with 0.1 probability to make a random decision, which serve as the benchmark agents. (ii) (iii) (iv) the tilted tabular Q-learning with discounting factors $\gamma = 0.1, 0.5, 0.8$ for Q-Discount-low, Q-Discount-med, and Q-Discount-high respectively.

- Number of Agents: 5.
- Number of Pellets: 500.
- Number of Episodes Observed: 1000.
- Size of the Lab: (200, 200)
- Size of an Agent : (10, 10)
- Locations of Agents: fixed by RANDOM seeds
- Locations of Pellets: fixed by MANUAL specifications
- Collision with Decorations (desks, chairs): disabled

### B. *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 their final decisions are formed with some probability to reject such greedy behaviors and make random movements instead. In our experiments, the rejection probability is set to 0.1. The reason of incorporating such randomness is to make the greedy-random agents consistent with other Q-learning agents, all of which yields random explorative movements with 0.1 probability.

### C. *Tiled Tabular Q-learning Agents*

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

Q-learning is a model-free reinforcement learning technique and is utilized to find an optimal action-selection policy for any given MDP. It works by learning an action-value function that

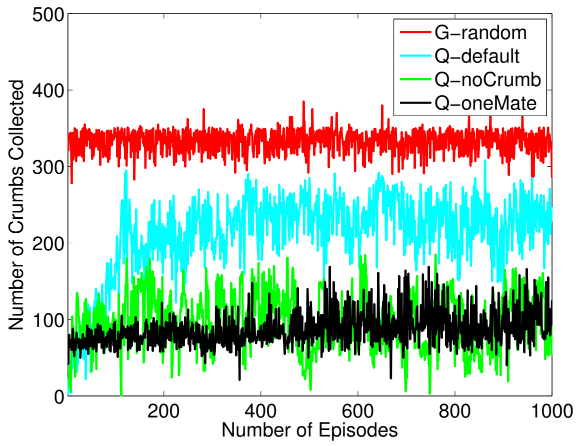

Fig. 3: **Tiled Tabular Q-learning Agents with various simple sensors.** (i) Red: G-random curve represents the greedy agent with $0.1$ probability to make a random decision, which serve as the benchmark agents. (ii) (iii) (iv) agents receive various collection of sensors for their decision making of 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.

ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. Out of simplicity, the implementation of Q-learning, in our experiments, simply uses tables to store data. On top of that, the tiling technique is employed to discretize all real-value sensors for less learning complexity.

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

The Fig. 2 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.

We also investigate the issue of how the collaborative communication benefits by using various collections of sensors. The sensors designed for the local perspective of each agent are as follows.

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

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

The learning outcomes derived from these simply designed 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, comparing the performances between the Q-default curve and the Q-noCrumb curve. The 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. Nevertheless, the relative position to the closest collaborated agent indeed hinder the Q-learning process during the observed episodes, such that the communicative agents (black curve) performs worse than the team that do not monitor collaborators' positions (light-blue curve).

### D. *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) [33]. In particular, we evolved these networks with real-time NEAT, which does not use generations – instead replacing agent's networks continuously throughout 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.

### A. *Setup*

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. This creates difficulties when attempting to compare the results of different experiments. One solution is to have a large population of networks of which only a small number are used by agents at a given time. Fig.4 shows that this approach considerably slows the learning process.

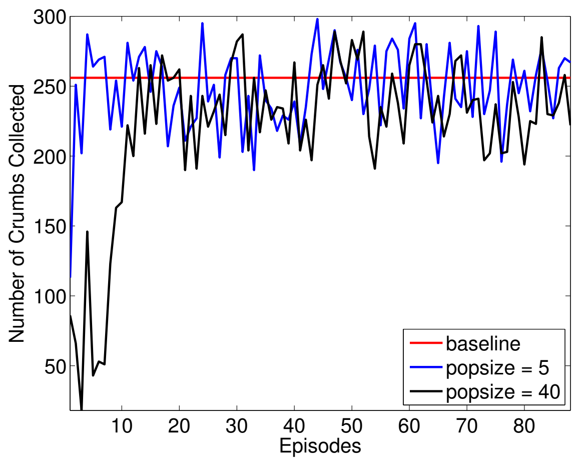

Fig. 4: **Effect of population size on neuroevolution.** Demonstrates the effect of choosing different network population sizes in an environment with five agents. Observe that having more networks than agents slows the learning process. Further note that this is using a simple one input / one output network; with a more complicated network, the large population shows only marginal results even after 1,000 episodes.

For complicated networks, this approach quickly becomes unfeasible.

One can avoid this issue by scaling the number of agents to match the network population size. These modification present an issue, however, as the Roombas often manage to collect all of the available crumbs before time expires. In order to prevent this, our methodology has been to reduce agents' size and speed such that 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.

## B. Fitness

Originally, an agent's fitness was based purely on collecting crumbs. Various other mechanisms were considered, including the ability to avoid obstacles and other agents. Experimental evidence showed the best results came from moderately penalizing collisions while heavily rewarding crumb collection. The fitness function for these experiments is:

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

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

Performance generally improved with increases to either $\phi$ or $\psi$. Interestingly, combining high values for both $\phi$ and $\psi$ resulted in worse performance than increasing either value individually, suggesting that the two behaviors interfere with each other in some way. Experimentally derived results show $\phi = 1, \psi = 0.05$ as good choices, which moderately incentively collision avoidance while primarily focusing on crumb collection.

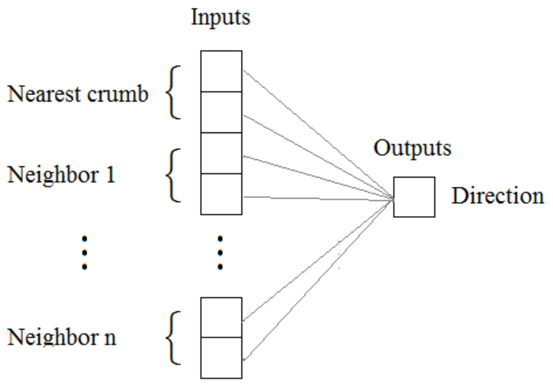

Fig. 5: **Communicating Roomba network.** Initial network to be evolved with rtNEAT. Sensors detect the polar offsets (angle and magnitude) of the $n$ nearest neighboring agents, as well as the nearest crumb. Output represents the (angular) direction in which the agent should move.

## C. Neuroevolution Baseline

In all of our experiments, we compare the results of neuroevolution against hardcoded search behavior as a baseline. This baseline consists of 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 evolved agents will learn to better cooperate in order to perform a better search of the target space.

For a starting point, the simplest possible network is trained to match the behavior and performance of the greedy algorithm. 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 needs only to map the input directly to the 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.

This result is both logical and significant. It demonstrates that agents learn to leverage the additional information of how far away a crumb is. An agent is more likely to capture nearby crumbs, improving its fitness. When the crumb is further away, however, it is more likely that another agent will reach that crumb first. Thus, the agent has better odds by moving to another area. One might wonder if agents could benefit from additional sensors detecting furhter away crumbs. In practice, however, there was improvement from simply adding sensors that pointed to further crumbs.

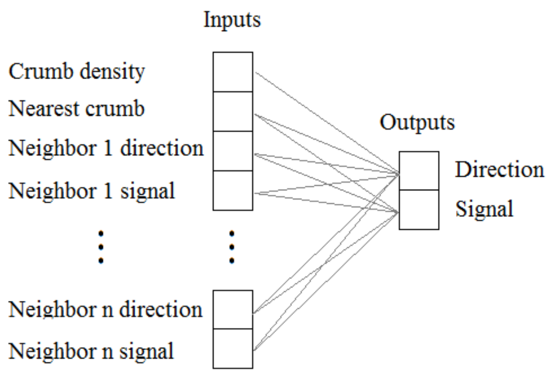

Fig. 6: **Communicating Roomba network with evolved signals.** The addition of a broadcast signal gives agents a more expressive means of communication. This network receives the direction of and signal being broadcast by each of the $n$ nearest neighboring agents. It also has stronger information about its local environment, receiving both the density of nearby crumbs as well the angle to the nearest crumb.

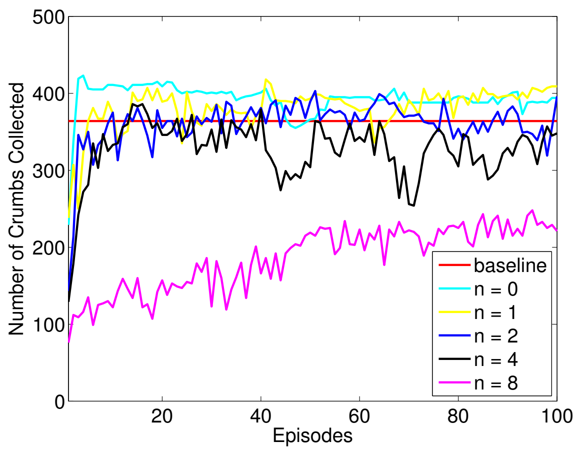

Fig. 7: **Communicating Roomba results.** Comparison of non-communicating agents ($n = 0$) with communicating ones. The more neighbors an agent communicates with, the slower it learns. Furthermore, none of the communicating agents exceed the performance of non-communicating ones, with more communication in fact resulting in worse performance.

## *D. Communication*

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. 5, 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. This section compares experimental results for different choices $n$, where $n = 0$ represents no communication, and each increment adds another neighbor to each agent's awareness.

As seen in Fig. 7, the more neighbors the agent communicates with, the longer it takes to learn. Fig. 7 shows that the $n = 1$ neighbor network catches up with the $n = 0$ network after about 20 episodes. The other networks, however do not reach this level even after several hundred episodes of training. Note that while the networks appear to cap out at 400 crumbs per episode, there are approximately 30-50 crumbs uncollected at this point. 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 [12] has showed that sharing fitness functions between agents can improve multi-agent learning. To explore this, we use a modification of the Roomba environment to share fitness amongst agents. Instead of being rewarded only when picking up a crumb, agents share the reward from their teammates success.

Fig. 8 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.

## *E. Evolved Communication*

Counterintuitive, these experiments have shown that communication is in fact 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. 6). 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 – larger radii rendered the sensor useless.

The goal is for agents to learn to communicate their short range observations to each other in order to improve their overall performance. Unfortunately, results were little better than with the simple communicating network (Fig. 9). Again, the more neighbors communicated with, the longer it took a network to learn. Ultimately, the higher $n$-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

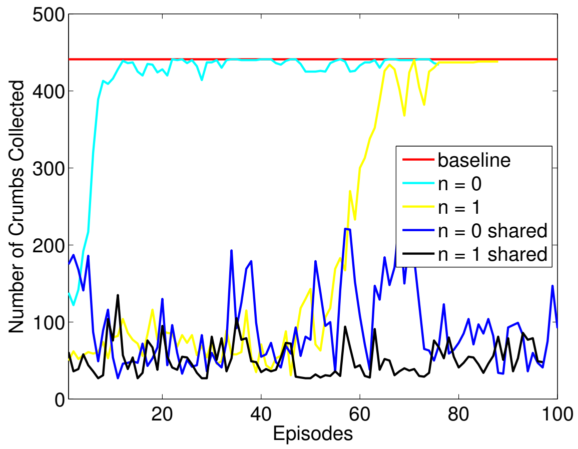

Fig. 8: **Comparison of individual vs shared fitness.** Shows the difference between individual and shared fitness functions. As before, the non-communicating network learns faster than the communicating one. We also observe that agents with shared fitness do not learn at all – or at least learn extremely slowly.

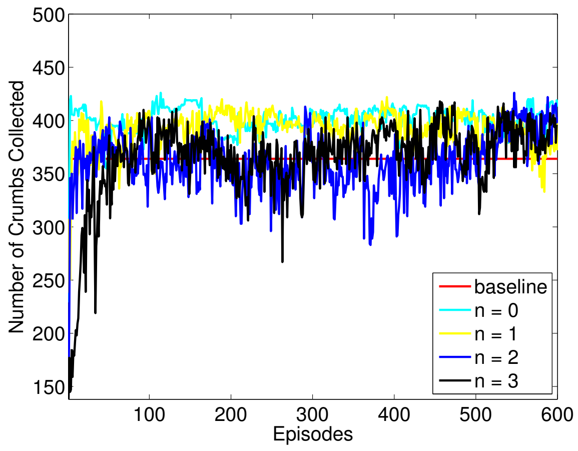

Fig. 9: **Comparison of evolved communication.** Over a long timespan, the higher $n$-value networks eventually reach, but do not exceed the performance of the non-communicating ($n = 0$) network. For any $n > 1$, this takes several hundred episodes.

consistently outperform the $n = 0$ network. Nether did this communication with evolved signal perform any better than the simple communication of neighbors' location.

## VI. Conclusions

**TODO:**

The conclusion goes here. In this paper, we investigate.

Using neuroevolution, we were able to successfully evolve a network which performs better than the hardcoded greedy algorithm. This demonstrates the power of neuroevolution and its ability to find good solutions to complicated optimization problems.

Perhaps a more surprising result, our experiments show that communication can in fact impede the learning process. Presenting a network with more information makes learning more difficult, and there is no guarantee that the network will be able to use that information productively. Effective communication seems especially difficult in a complicated environment such as this, where there is far too much state information for agents to try to digest and communicate effectively.

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] G. Tesauro, "Td-gammon, a self-teaching backgammon program, achieves master-level play," *Neural computation*, vol. 6, no. 2, pp. 215–219, 1994.

[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] 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.

[11] C. H. Yong and R. Miikkulainen, "Cooperative coevolution of multiagent systems," *University of Texas at Austin, Austin, TX*, 2001.

[12] P. Rajagopalan, A. Rawal, R. Miikkulainen, M. A. Wiseman, and K. E. Holekamp, "The role of reward structure, coordination mechanism and net return in the evolution of cooperation," in *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*. IEEE, 2011, pp. 258–265.

[13] A. Rawal, P. Rajagopalan, and R. Miikkulainen, "Constructing competitive and cooperative agent behavior using coevolution," in *IEEE Conference on Computational Intelligence and Games (CIG 2010)*, Copenhagen, Denmark, August 2010. [Online]. Available: http://www.cs.utexas.edu/users/ai-lab/?rawal:cig10

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

[15] R. Mailler and V. Lesser, "Solving distributed constraint optimization problems using cooperative mediation," in *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*. IEEE Computer Society, 2004, pp. 438–445.

[16] 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.

[17] J. Davin and P. J. Modi, "Impact of problem centralization in distributed constraint optimization algorithms," in *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*. ACM, 2005, pp. 1057–1063.

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

[19] 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.

[20] 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.

[21] 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.

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

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

[24] R. Nair, P. Varakantham, M. Tambe, and M. Yokoo, "Networked distributed pomdps: A synthesis of distributed constraint optimization and pomdps," in *AAAI*, vol. 5, 2005, pp. 133–139.

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

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

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

[28] 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.

[29] B. D. Bryant and R. Miikkulainen, "Neuroevolution for adaptive teams," in *Proceedings of the 2003 congress on evolutionary computation*, vol. 3, 2003, pp. 2194–2201.

[30] A. Acerbi and D. Parisi, "Cultural transmission between and within generations," *Journal of Artificial Societies and Social Simulation*, vol. 9, no. 1, p. 9, 2006. [Online]. Available: http://jasss.soc.surrey.ac.uk/9/1/9.html

[31] Y. Yang, M. Polycarpou, and A. Minai, "Opportunistically cooperative neural learning in mobile agents," in *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, vol. 3, 2002, pp. 2638–2643.

[32] R. Miikkulainen, E. Feasley, L. Johnson, I. Karpov, P. Rajagopalan, A. Rawal, and W. Tansey, "Multiagent learning through neuroevolution," in *Advances in Computational Intelligence*. Springer, 2012, pp. 24–46.

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