A Multi-Agent Team Formation Framework for Classroom Instruction *

Adam Anthony, Marie desJardins, and Steve Martin
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
(410) 455-8894
aanthon2@cs.umbc.edu

Abstract

A challenge in AI education is the creation of interesting term projects. An ideal term project is fun, engaging, and allows students to focus on the interesting parts of a problem, rather than spending several weeks designing infrastructure or coming up to speed on a complex simulation framework. In this paper, we describe our efforts in designing a multi-agent team formation environment using the Repast toolkit, which we used successfully for a course project in an upper-level class on intelligent agents. The infrastructure that we developed resulted in a project that was both enjoyable and pedagogically effective, and that could be completed in only a few weeks. We begin with a description of the initial design and deployment of the project, followed by a discussion of problems we encountered, and the changes we have made for future classroom deployments. We conclude with a description of some open issues with our project and how we will address them.

Introduction

This paper describes our experience in developing and using a multi-agent team formation testbed for a course project in UMBC's CMSC 477/677 (*Agent Architectures and Multi-Agent Systems*), an advanced AI class with both undergraduate and graduate students. The goals of the class are to teach students about intelligent agent models (cognitive models, planning, learning) and interactions in multi-agent systems (game theory, trust and reputation models, team/coalition formation and coordination, distributed decision making).

In order to give students hands-on experience with designing competing agents in a multi-agent system, we developed the Team Formation Environment (TFE). The TFE is a testbed that permits students to design individual agents for a networked team formation problem, provides a simulation environment that allows the agents to compete against each other by performing tasks together, and includes a set of visualization tools so that students can watch the competition unfold in real time.

The assignment was to design a policy that a single agent could use to decide whether or not to join a team working on a task, taking into account factors such as the complexity of the task, the skills and states of neighboring agents, and the previous history of the agent and its neighbors. The project culminated in a competition in which the students' agents had to cooperate in order to complete tasks.

Our design allowed students to design their agents to be as complex or as simple as they wished; the resulting agent designs ranged from very simple heuristics such as "always-accept" to complex learning-based strategies. Because the TFE is built within Repast (Repast 2003), an open-source agent simulation framework, there are many opportunities to expand and improve upon the existing environment. We describe the team formation model used and its implementation, discuss its deployment in the classroom, and summarize some features that we developed in response to reactions from students.

A Simple Multi-Agent Team Formation Model

In multi-agent team formation, individual intelligent agents must form groups to accomplish tasks. A *networked multi-agent team formation* problem has three major components: a set of agents, a social network that connects the agents, and a task generator (Gaston 2005).

The agent community in the TFE consists of N heterogeneous agents. Each agent a_i has one skill s_i, drawn randomly from a collection of k skills: $s_i \in S = \{1, \ldots, k\}$. The skill set is intended as a simplification of the idea that agents in a real environment will have different sensors, mobility, tools, and knowledge. At any given time, each agent is in one of three states: {`UNCOMMITTED, COMMITTED, ACTIVE`}.

TFE's *task generator* produces tasks at a fixed time interval, announcing them globally. A task consists of a *skill set*, an *expiration time*, and a *duration*. The skill set is a list of the skills that are necessary to successfully complete the task; this list can include duplicates, indicating that multiple same-skill agents are required. After a task is announced, any agents possessing a skill in the skill set may volunteer to join the team, subject to certain network constraints, as described below. Once the team has formed (i.e., the set of agents that have joined the team matches the skill set), the task runs until the duration is reached, or the expiration time is reached. In the former case, it is considered a success; otherwise, it is a failure.

An agent may only join a new task when it is in the

*This material is based upon work supported by the National Science Foundation under Grant No. #0545726.

UNCOMMITTED state. After joining a team, the agent moves into the COMMITTED state, where it remains until the skill set has filled, or the time expires, whichever occurs first. When the skill set is complete, the agents on the team move into the ACTIVE state. Once the task is complete, or the time expires, all associated agents return to the UNCOMMITTED state.

The social network that connects the agents constrains the agents in two ways. First, agents may only join a team if they are the first to join, or if they have a neighbor already on the team. The resulting teams always induce a connected subgraph of the social network. Second, agents only receive information about the skills and states of adjacent neighbors, and do not know what other agents are in the network.

The social network can also change over time. In our competition, a small number of edges were rewired at each time step. An alternative class project would have students implement a strategy for the agents to control this rewiring process locally.

Each agent has an associated *policy* that it uses to decide whether or not to join a team working on a specific task. This strategy is the focus of the class project, and is implemented by each student.

A straightforward measure of local performance for each agent is the ratio of successful tasks to the number of tasks that the agent joined:

$$Y(a_i) = \frac{\text{\# successful teams joined}}{\text{\# teams joined}}$$

Globally, the performance of the entire system can simply be measured as the ratio of number of tasks completed to the number of tasks generated:

$$\text{Performance} = \frac{\text{Tasks successfully completed}}{\text{Tasks generated}}$$

Other performance measures are possible; this topic was the subject of much discussion within the class, and is discussed more in the next section.

Implementation and Deployment

We summarize here the features of Repast that facilitated the development of TFE, and briefly describe the TFE implementation itself. The TFE software can be found at `http://maple.cs.umbc.edu/mates/teamformation.html` . The course website is `http://www.cs.umbc.edu/courses/graduate/677/spring07/` .

Implementation within the Repast 3 toolkit The Repast (Repast 2003) toolkit is an open-source agent simulation development environment that includes tools for implementing common agent tasks such as movement, network formation, and communication. Repast has several features that facilitated the implementation of TFE:

- Management of a global clock,
- Execution of specified actions at each time step,
- Randomization of action execution order to simulate parallel operation,
- A social network data structure, including link generation tools,
- Visualization tools for viewing network activity and performance, and
- A simulation control interface through which model parameters could be changed and the clock could be started, paused, stepped, or reset.

We designed the system using Repast's network data structure, in which each node in the network corresponds to an agent. At each time step (or at fixed intervals) the simulation rewires a small number of edges, generates tasks, and permits each agent to execute its team joining program. We created and integrated new data structures to model the tasks and agents. Tasks have a simple design: each task is modeled as an array of integers that represent required skills and a method that determines whether or not a particular agent may join the task. The agent class was designed with the goal of minimizing the students' learning curve, so that they could quickly design and implement their own agents.

Abstract agent design An agent in our model must be able to perform the following actions: identify eligible tasks, select a task to complete (if any), commit to a task, and change state. Identifying eligible tasks, committing to tasks, and changing states are behaviors that are identical for all agents. The only part of the agents that differ is the policy that determines which task to join when the agent is in the UNCOMMITTED state. Therefore, we provided the former policies, and each student only had to implement a joining policy.

Figure 1 shows a partial source listing of the agent class that students must modify. One useful feature of the model is that students may experiment with both very simple and very complex agents. The `decideAction` method is the only required function. It returns the index of the input task list (-1 if none selected) for the task that the agent wishes to commit to. Policies can be as simple as always selecting the first task in the list, or as complex as comparing all of the tasks and analyzing the commitments of neighbors to decide which task to choose. An optional method is `free`, which is called by the simulation manager when a task has expired or been completed. This provides an opportunity to analyze the outcome of the previous team joining decision; some students chose to use reinforcement learning or statistical methods to guide future decisions.

Visual display of agent actions Repast provides several useful visualization and control components to enable interaction during simulation. The main component is the network visualization, which can be seen in Figure 2. Nodes have two colors: the center color indicates the agent's state and the colored border uniquely identifies agents of a particular type. (Each student's set of agents has a corresponding color.) The thickness of the border is determined by the

```
public class AgentStub extends DefaultAgent {
        public AgentStub () { }

        public int decideAction (NodeViewable thisAgent, ArrayList<NodeViewable> neighbors,
                double currentTime, ArrayList<TeamFormationModel.Task> tasks)
        {
                for (int i = 0; i < tasks.size(); i++) {
                        TeamFormationModel.Task temp = tasks.get(i);
                        //always accept
                        if (/*here is where you can start to modify your selection strategy*/ true)
                        {
                                return i;
                        }

                }

                return -1;
        }

        public void free (NodeViewable thisAgent, ArrayList<NodeViewable> neighbors,
                double currentTime, boolean lastTaskSuccessful, TeamFormationModel.Task previousTask)
        { }
}
```

Figure 1: Agent class that students modify to create their agent.

agents' performance; the best agents have the thickest border. Repast also has several built-in functions for displaying real-time data, which are discussed below.

A competitive environment Finally, to facilitate the operation of a tournament-style competition, the program was designed so that any number of agents of each user-defined type could be inserted into the model. Each agent is a subclass of the default agent type; we used an input file to specify the class names of the agents we wanted to simulate, the color of the agents, and how many agents of that type to use. This makes the initial configuration of the program simpler for in-class use.

Example Agents

As mentioned above, students designed agents with a variety of policies. These policies included basic probabilistic choices and also more elaborate post-processing policies in which learning was performed as each task was completed. Below are descriptions of some student-designed agents.

Agent A Agent A had a policy that involved both an initial pass/fail decision and a secondary learning process that learned which task skill sets had a high success rate. In the decideAction method, agent A first considered its node degree in deciding whether or not to accept a task. If the node degree was twice the number of agents needed to complete the task, the agent accepted. Otherwise, it ran a second check. In the free method, if the agent is in the `COMMITTED` state when the method is called, this implies a failure. The agent hashes the task's required skill set and decrements its score by 1 (with an initial score of 0). If it wasis in the `ACTIVE` state, it increments the skill set's score by 1. If the score for this task is greater than -3, the agent will accept the task the next time an identical skill set is considered.

Agent B Agent B's design was thoroughly deterministic. Its free method was empty, so no learning techniques were used. The student designed this agent with the motivation of having an agent that only made good decisions. Its first check was to see if joining the team would guarantee success by verifying that its skill was required and that there was only one vacant slot left on the team. Second, it checked the skills of its neighbors. If the neighbors could potentially fill more than 80% of the task requirements over time, then Agent B accepts. This simple but insightful agent design resulted in the highest performance of the student submissions.

Agent C Agent C was similar to agent B. However, instead of profiling whether its neighbors could fill out the task set, agent C merely checked to see if (a) it was the last necessary member of the team, and (b) 80% of the task set was already filled. Agent B and agent C both had higher performance than other agents, most likely because they were the only agents with the low-risk, high payoff check for teams with all but one slot filled. However, agent B was far superior to agent C because it made much safer decisions by taking its neighbors' skills into account. Interestingly, if these two agents were the only types of agents present in an environment, they would never complete any tasks (since they only join once a task is nearly complete). This "freeloading" behavior led to some lively discussions among the students, and numerous suggestions for score modifications to punish such behavior.

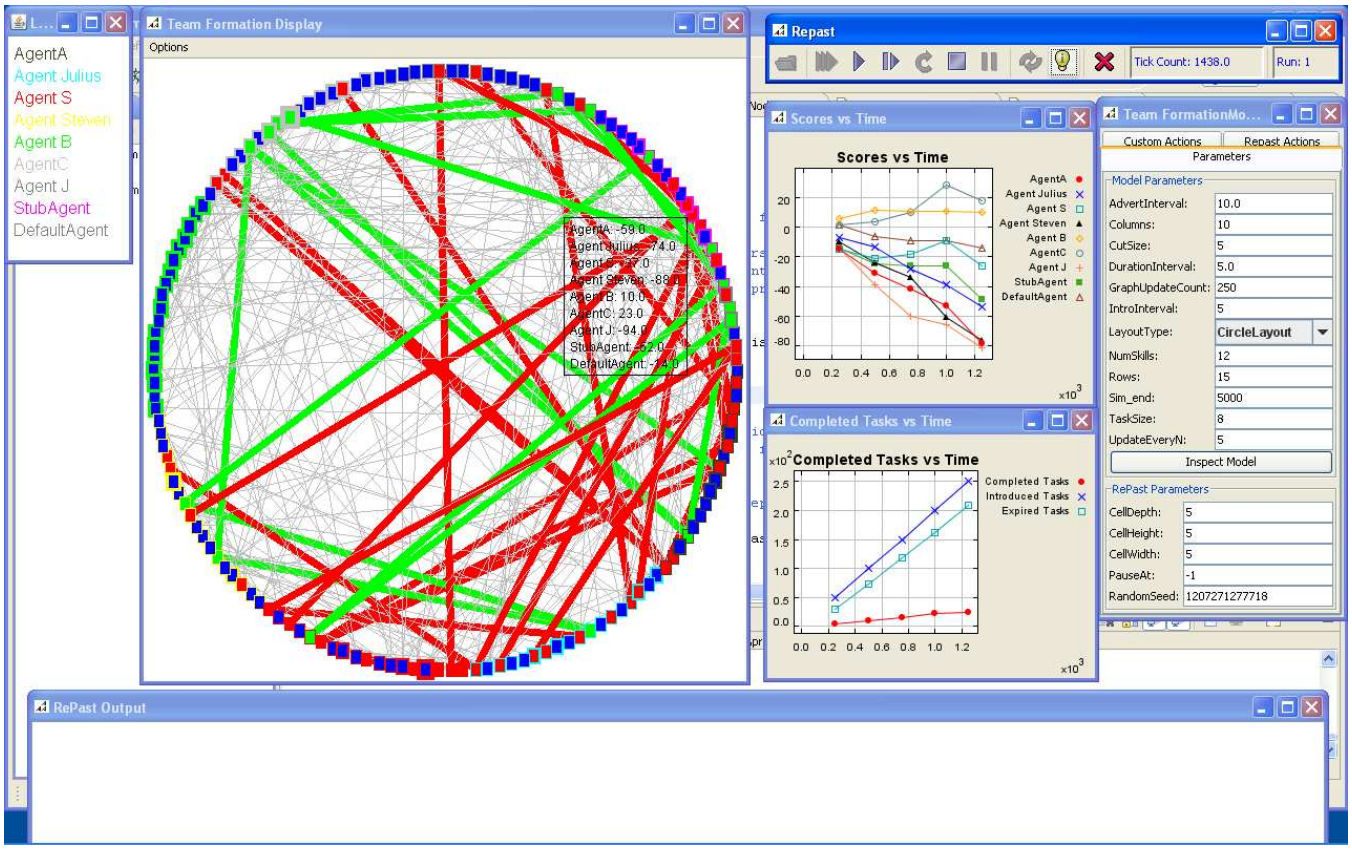

Figure 2: Screen shot of the simulator. Thicker, colored lines indicate links between teammates. The legend shows the team color of each agent (there is more than one of each agent type). The plots in the center indicate local agent performance, and global performance. The panels on the top and right are for simulation control.

Discussion

The project was very successful at engaging the students in a process of learning and exploration. The iterative development process, with a "dry run" tournament a few weeks before the final due date, gave the students practical experience with the nuances of multi-agent systems. Most importantly, by actually designing their own agents to operate in a heterogeneous agent society, they learned first-hand how sensitive agent performance is to the specific environment in which the agent is embedded. The parameters of the environment itself (number of agents, frequency of tasks, scoring function used to reward the agents) had a strong effect on the performance of individual agents, as did the behaviors and decision-making strategies that were being used by the other agents. The TFE provides a highly dynamic, non-stationary environment, with many similar properties to a real-world multi-agent trading environment. Many students experimented with methods they had learned in the class (reinforcement learning, trust management, statistical modeling), allowing them to explore these topics in more depth.

Naturally, as with any first-time implementation, many unforeseen problems presented themselves. We summarize here some of the problems, and our solutions, from the first classroom deployment of the TFE.

Scoring A main issue of debate was a fair measure for scoring the performance of the agents. The initial scoring method used was the ratio of successful tasks to total accepted tasks. From a research perspective, this score reflects the quality of the team-joining policy. A high ratio corresponds to agents who choose good teams to join. In order for global performance to be high, individual agents must continue to join teams, or no tasks will be completed. However, in a competitive environment, there is no incentive to increase global performance. An agent scores well as long as they have the highest local ratio for tasks completed. Therefore, the best strategy under this scoring method is to maximize the probability of completing the first task, and then reject tasks for the remainder of the simulation.

This incentive can be mitigated by forcing agents to accept at least a minimum number of tasks, penalizing agents who accept fewer tasks. However, this still provides an incentive for agents to be risk-adverse, in the sense that it is better to complete fewer tasks while maintaining a high success ratio, rather than completing more tasks with a lower success ratio. Cautious agents do well with respect to this score when they refuse to join tasks early (long before the deadline), since the only way for a task to fail is for a team to not be completely formed. For example, some students' policies were to only join teams that were nearly complete. Given this behavior, a risk-seeking agent may perform well with other risk-seeking agents, but perform very poorly with cautious agents. (The risk-seeking agents will accept tasks and then fail, because the cautious agents are not willing to commit.) Furthermore, a community consisting only of cautious agents will have poor global performance, since they tend to join tasks too late for a complete team to ever form.

After observing this problem, we had a class discussion and allowed the students to democratically choose a new scoring method. They decided on a points-based system, rather than a ratio system. Agents received two points for every completed task, and lost one point for every failed task. This makes taking risks more attractive, since the only way to advance in score is to complete more tasks, not necessarily by only joining tasks which will succeed. Joining tasks that are likely to fail is still a bad idea, since being in a task that fails lowers the agent's score. However, this new scoring system has flaws as well. Without a heavy price for joining tasks that fail, the point system now favors risky agents over cautious ones.

Since the scoring method has such a significant impact on the performance of the agents, choosing a scoring method actually turned out to be one of the most interesting and educational parts of the project. The next time we teach the class, we plan to incorporate a preliminary development phase, after which we will again have an open discussion of different scoring methods, and select a scoring method based on a class vote. Another option would be to develop a parameterized scoring function, and not announce the parameters until the day of the competition. This would make the agent design process significantly more difficult, since the team joining strategy could depend on the scoring parameters.

Task Distribution In a preliminary competition that we held as a "shake-down" of the agent designs, the winning agent had the simplest strategy: "always accept." Although the competition was designed to give students an opportunity to test their agent strategies, it did not reward students who put more effort into more sophisticated agent strategies. The reason the always-accept agent performed so well was because the task generator was using a long task interval and a small task size (number of agents per task). The result task environment only had a few open tasks at any given time, and these tasks only required a few agents each. As a result, most tasks could be completed successfully—there was not much "competition" for the available agents. After this preliminary match, we modified the task generation parameters, decreasing the average probability of success; in this environment, always-accept was no longer a dominant strategy.

Information Hiding Another issue that arose was that in our initial implementation, all of the methods in the simulator were public. No students cheated deliberately, but because we were not explicit about which parts of the simulation students were allowed to touch, there was a gray area about which methods and information were "fair game" for the agents to access, and which were not.

In our ongoing development, we have made several modifications to TFE to prevent this issue from arising. In particular, much of the development since the deployment has focused on securing the code to prevent students from modifying or accessing functions or parameters unintended for the agent. Another major design change is that in the new version of the system, interaction with the agent is always initiated by the simulation manager. The agent cannot act

except when the simulation manager permits it, so the agents are unable to impede the model's proper functioning. In fact, the agent can only provide one value to the simulator that influences the behavior of the system: which task, if any, it wishes to accept.

Furthermore, agents are provided with all of the information they need as parameters in the `decideAction` function, so that students do not need to use methods in other classes to create a policy. The information given to agents is locally scoped, so changing the values passed to the agent has no effect on the actual values in the model. This prevents the agent from cheating by accessing information it is not intended to know. This design also has the benefit of easing development for students. Students do not have to search through documentation or source code functions to determine what their agents are capable of knowing, reducing the learning curve associated with using the implementation. Only two classes from the model are passed into the agents; therefore, the students do not have to worry about the details of the implementation of the model to successfully create agents. Both of the classes passed into the agents only have accessor functions that are publicly available.

Visualizations A final complaint from the students is that the contest was not interactive enough. The network display showed the actions as they were being taken; it was colorful and created a sense of action. Nevertheless, it was too complex and moved too quickly for the students to really understand what was happening. Since the original deployment, we have added two graphs that show the model's performance over time. They can be seen in Figure 2. The first is the individual agent type (student) performance, so students can see how their own agent policy compares to others. Second, a global view of the performance of all agents is presented, with three plots: tasks generated, tasks failed, and tasks completed. Finally, after the simulation concludes, the score, average score, and standard deviation of the scores of each agent type are automatically recorded, then printed in the output window, at the bottom of Figure 2.

Related Work

Repast was one of several simulators we could have used as the foundation for TFE. Other examples include Swarm (Swarm 2008), Ascape (Inchiosa & Parker 2002), Mason (Luke *et al.* 2004), NetLogo (Wilensky 1999), and TeamBots (TeamBots 2000). We chose Repast primarily because it had several desirable features such as built-in visualization, and also because its broad applicability made it a good fit for other lab projects besides TFE.

In terms of educational environments that emphasize ease of development, few exist, and to the authors' knowledge, there are no publicly available team formation simulators intended for classroom use. However, there are some AI-related projects that abstract away from the simulation environment as we have. One such simulator is Robocode (RoboCode 2008), a two-dimentional robotic tank simulator in which students must design motion and weapon algorithms for an arena-style competition. The complexity of RoboCode agents can be much higher than the agents in TFE, but RoboCode, like TFE, is accessible to novice programmers as most of the advanced features are optional. TFEcan also be viewed as being similar to the various simulators for the iterated prisoner's dilemma (Axelrod & Hamilton 1981), although our environment is significantly more complex and the payoff matrix for TFEis unpredictable because of the variable factors in the simulation, such as varying network connectivity and the uncertainty of other agents' policies.

Conclusions

We have developed, implemented, and deployed the TFE, a team formation testbed that is appropriate for classroom usage at both the undergraduate and the graduate level. It is simple enough that it could be used as a component of a broad introductory AI course, but has enough challenge that it makes an interesting and dynamic course project for an upper-level or graduate class on intelligent agents and multi-agent systems. Because the implementation presents an abstraction of the problem to the user, students have a short learning curve, and can make their agents as simple or as complex as they wish. The requirements for a working agent are simple, yet offer the opportunity to implement complex policies. Our improvements since the first classroom deployment should make future use smoother from an instructor's standpoint, and should also make it more interesting from a student's point of view.

Several open questions remain. The first is the continuing discussion about what the "best" scoring method is in this competitive setting. Resolving this question was actually a surprisingly valuable learning experience for the students, and in future deployments, we would probably again allow the students to democratically choose the scoring method, after some initial period of development and testing. Second, the range of agent policies we received from students was quite broad, and in some cases the strategies were quite unexpected. An interesting study would be to ask students why they chose a specific policy, which may give insight into how humans view cooperation in a mixed environment. Finally, we wish to investigate the power of our model as a research tool. Implementing features that allow easier manipulation of agent skill distribution, task size and difficulty, network connectivity, and stochastic effects in team success could all lead to the identification of interesting problems in team formation research.

References

Axelrod, R., and Hamilton, W. 1981. The evolution of cooperation. *Science* 211(4489):1390.

Gaston, M. 2005. *Organizational Learning and Network Adaptation in Multi-Agent Systems.* Ph.D. Dissertation, University of Maryland Baltimore County.

Inchiosa, M., and Parker, M. 2002. Overcoming design and development challenges in agent-based modeling using ASCAPE. *Proceedings of the National Academy of Sciences* 99(90003):7304.

Luke, S.; Cioffi-Revilla, C.; Panait, L.; and Sullivan, K. 2004. MASON: A New Multi-Agent Simulation Toolkit. *Proceedings of the 2004 SwarmFest Workshop* 8.

Repast. 2003. Repast organization for architecture and development. http://repast.sourceforge.net.

RoboCode. 2008. Robocode: Build the Best, Destroy the Rest! http://robocode.sourceforge.net.

Swarm. 2008. Main page—swarm wiki. http://www.swarm.org.

TeamBots. 2000. Teambots 2.0. http://www.cs.cmu.edu/~trb/TeamBots/.

Wilensky, U. 1999. Netlogo. http://ccl.northwestern.edu/netlogo/.