FalloutBots

Multi-agent discovery, cooperation, and competition in a resource-constrained environment

Ari Wilson

CS396 Distributed Artificial Intelligence
Project Final Report

April 18, 2007

Introduction

`FalloutBots` is a project that explores the performance of different methods of learning in a particular multi-agent system. This system has a fully-observable environment but the nature of other agents is not known. Automated systems have many of the same problems as humans do when making decisions with risks, uncertainty, and limited resources in the presence of others. *Game theory* and *Bayesian decision-making* are models that can be used to analyze these types of systems and develop appropriate responses to the situations in which agents find themselves.

The classic theories of game theory and statistical behavior, however, do not readily lend themselves to the analysis of the complex, possibly adaptive behavior of adversaries and collaborators. Near-optimal play in such scenarios is generally not possible due to extremely large state spaces. Therefore, I have used basic machine learning techniques to enable agents to make better predictions in their relatively complex world. *Q-learning* and *fictitious play* are two well-explored methods developed in the domains of reinforcement learning and game theory for agents to learn about their environment and fellow agents. Q-learning agents focus on learning about rewards by modeling and updating the state of the environment. Fictitious learning agents concentrate on learning the strategy of opposing agents and formulating a best response to their actions in order to maximize their own rewards. These methods are especially useful when little domain-specific knowledge is available. A number of hypotheses relating to the effectiveness of these methods and other aspects of multi-agent systems are explored in the experiments analyzed in this report. A broad overview of multi-agent learning methods is given in [PL05].

The inspiration for the market- and learning-based algorithms used in `FalloutBots` was the methods used for Trading Agent Competition agents as described in [GB05] and extended in [BT06]. However, the details of my system differ significantly from any scenario I have thus far examined.

`FalloutBots` is developed as a simplified model of multi-agent discovery, cooperation, and competition in a resource-constrained environment. For example, robotic platforms attempting to build a lunar colony on the dark side of the moon may encounter limited resources in the form of water, certain metals, and gases. All of these materials may be needed in certain measure by agents tasked with loosely-coupled operations. Agents will be required to collect these resources in an expeditious fashion and distribute them in an appropriate fashion so that each agent can complete its task to the best of its ability with the available resources. Agents may or may not have *a priori* knowledge of each others' priorities. In a competitive environment, software agents may have to find, share, and utilize limited processing time, memory, and other resources in host networked systems. These examples were generalized to create the scenario described in the following section.

Scenario description

At the beginning of a trial, a certain number of types of resources are created. Each type of resource is present in a certain quantity in the game world. Each agent is given priorities in the range $[0, 1]$ for the resources that will be placed in each game. Each trial is composed of some number of games, which are the basic structure for agent action in this scenario. Games can be divided into two subcomponents: resource-gathering and auctioning. In the resource-gathering phase, each agent placed on a two-dimensional plane must navigate and collect as many of the high priority

resource piles as quickly as possible. After all resources have been collected, the game moves into the auctioning phase. In this phase, agents can attempt to bargain resources they have for the resources of others in a series of time-limited English multi-unit combinatorial auctions. This phase ends when no agent wishes to begin a new auction.

Agents are assigned performance values (or *rewards*) by two primary metrics in this system. One of these metrics is *individual utility*, given for a single game by $u(a) = \sum_{r \in R} n_a(r) p_a(r)$, where R is the set of all types of resources in a trial, $n_a(r)$ is the number of resources of type r possessed by a, and $p_a(r)$ is a's priority for resource r. Individual utility is a type of *local reward* and rewards competitive efforts to gain as much utility as possible in any given game. The number of wins for each agent in a trial can be used to compute an overall agent rating for a trial.

The other method by which agents are rated is the *allocative efficiency* of the group of agents in a game. The *optimal allocation utility* of a trial can be calculated starting from an empty allocation for every agent in the set of agents A. For each resource type r, add all of r to the agent a with $p_a(r) = \max_{i \in A} p_i(r)$. The optimal allocation utility is then given by $\sum_{i \in A} u(i)$. This value will remain constant throughout a trial, as no priorities or overall resource amounts are changed. The *allocation utility* of a group of agents is simply the sum of utilities for each agent after a game, again given by $\sum_{i \in A} u(i)$. The allocative efficiency is then simply the allocation utility divided by the optimal allocation utility. This is a global reward, in that it measures how closely resources were distributed according to maximum utility. It primarily rewards cooperation between agents during the auctioning stage of a game, as resource gathering is mostly a race to get all the resources collected as quickly as possible. The *average allocative efficiency* is an equivalent metric that can be used to measure the success of cooperation between agents over an entire trial.

System description

The simulation environment and agent implementations for `FalloutBots` were created in Visual Studio 2005 and total slightly over 2,000 lines of C++ code. An overview of the data structures involved in this implementation is given in figure 1.

Environment design

The structure of the environmental and monitoring code as it is executed during a typical test run is as follows. At the beginning of execution, the main function will run several trials by creating a number of `Metagame` objects to oversee individual trials. Statistics for each trial are recorded to a separate log file. The `Metagame` class controls the number of agents, their priorities, and the types and number of resources available during a trial. The results of a `Metagame` consist of the results of a certain number of games.

The `Game` class is initially responsible for determining agent positions and placing resource piles. Once a `Game` starts, it is responsible for giving each agent its turn, as well as determining when the resource-gathering stage ends and the auctioning stage begins. If desired, each agent could be made to run in a separate thread for simultaneous play, but some mutex conditions would be needed on shared resource variables. By using a turn-based system, however, I can ensure that the the length and number of turns for each agent are equal to those of other agents. This fairness is partially achieved by giving a random order to the turns for each iteration and restricting agent actions during a turn.

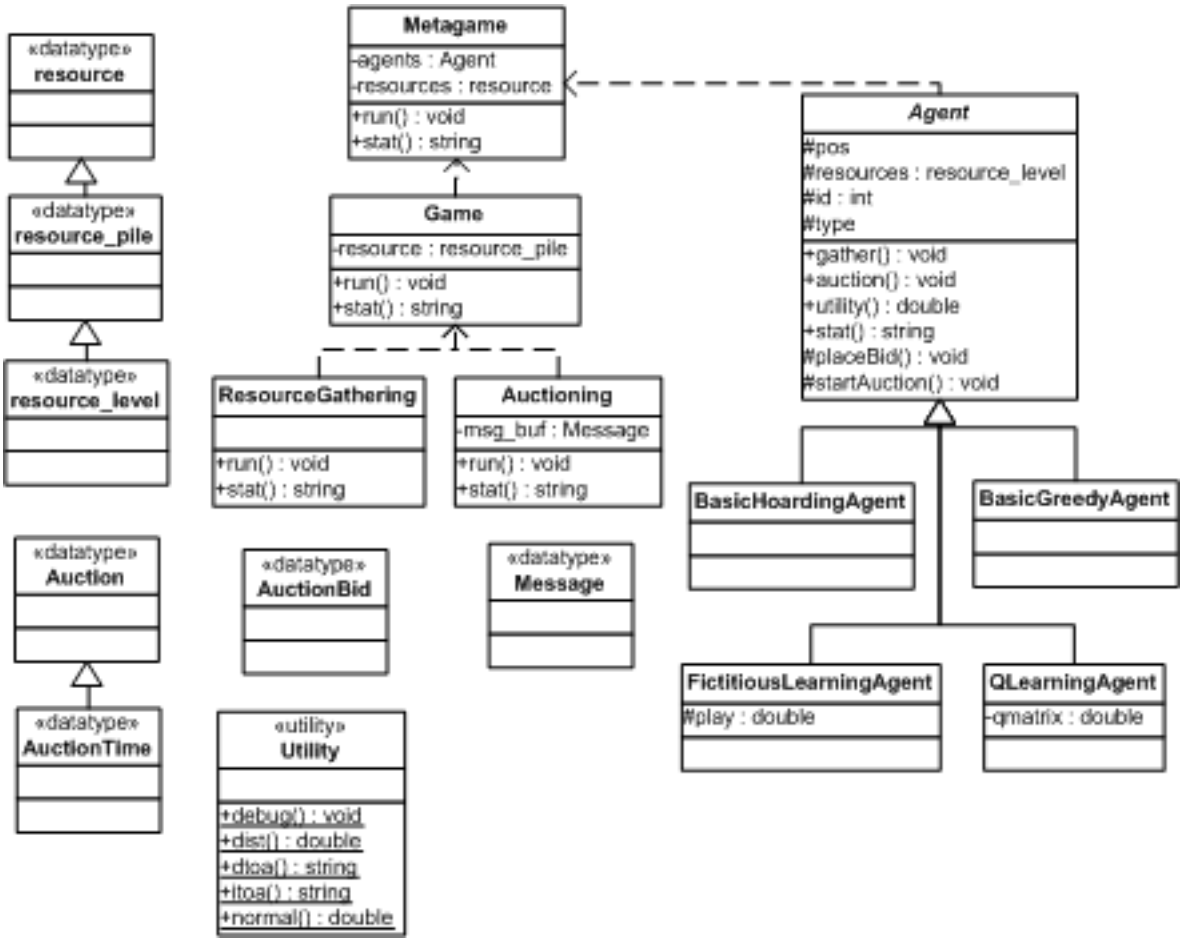

Figure 1: Implementation design

Table 1: Auction message types

Type	*Number of receivers*	*Meaning*
START_AUCTION	All	I am starting an auction.
INQUIRE_BID	One	I want to know the high bid on an auction.
TELL_BID	One	I am replying to your INQUIRE_BID.
PLACE_BID	One	I am placing a bid on an auction.
ANNOUNCE_BID	One or all	I am announcing the new high bid on an auction.
ANNOUNCE_WINNER	All	I am announcing the winner of an auction.

It is within this turn-based context that the ResourceGathering and Auctioning classes determine whether particular actions are allowable and update the game world accordingly. The environment's fully-observable nature is made explicit by passing a global vector of all agent resources and positions to each agent. Agent communication during auctioning is achieved through a globally accessible message buffer in the Auctioning data structure that is passed to all agents during their turn. A message consists of a type, content, a source, and a destination. A data structure representing the auction related to a given message is passed along with the message. After a certain amount of time, messages expire from the buffer and are removed. Table 1 gives a complete listing of the types of messages that can be sent in the auction stage.

During the resource-gathering stage, each agent is responsible for performing exactly one of the available moves in a given turn: moving at a speed at or below its maximum or gathering a predetermined number of units of resources if they are available at its location. This stage runs until all resources have been gathered.

During the auction stage, each agent can send and respond to a number of auction-related messages each turn. Agents are responsible for internally maintaining the state of auctions and bids relevant to themselves. The auctioning stage runs until there are no unread messages on the message buffer and a certain number of agent turns has passed. This time limit is sometimes called quiescence.

Agent design

Internally, each particular type of agent is derived from an abstract Agent class and has a strategy during both stages of a game, respectively referred to as gather() and auction().

It was decided early on in project development that the resource-gathering stage of a game was not as interesting to analyze as the auctioning stage in terms of immediate feedback and direct interaction with other agents. That is, it would be more complicated to develop an effective approximation to the solution of the *credit assignment problem* for resource-gathering when agent intentions and capabilities are initially completely unknown to other agents. Market simulations, on the other hand, have been well-analyzed in the literature[GB05] and thus can be better represented in a theoretical framework for agent design. For this reason, with the exception of BasicHoardingAgent, which has minor modifications to make it play more conservatively, all agent strategies during the resource-gathering phase are identical.

BasicGreedyAgent design

`BasicGreedyAgent` is a simple agent that attempts to maximize its utility in a given game without considering anything about the other agents operating in its environment. It gathers resources by examining all resource piles within a given distance, sorting these resources by the utility they give minus the amount of time needed to move to that location and process that pile. In the bidding round, it operates by checking the current resources and quantities being sold against the resources and quantities in its inventory. The agent will randomly choose items worth less than the auction to bid. If it has no resources with value less than the resources up for sale, it will not place a bid. It will advertise an auction for an item in its possession randomly if the auction's utility is less than some small fixed value and accept bids for quantities of items that will increase its overall utility.

BasicHoardingAgent design

`BasicHoardingAgent` operates on the principle that a bird in the hand is worth two in the bush. In most cases, it will operate similarly to `BasicGreedyAgent`. However, when deciding whether or not to move to a location to collect a resource, it will weight the distance to a resource pile as a more heavily negative factor by some constant scale factor. The reasoning behind this decision is that another agent probably wants the resource, and if it is far away, it is more likely to get it than our agent. As well, it will not create its own auctions or bid on others' auctions on the principle that even its lesser valued resources are probably very valuable to other agents.

QLearningAgent design

Q-learning as a technique relies on the abilities of agents to accurately model their state, rewards received, and actions. The last of these is not too difficult to define in `FalloutBots` - an action in the auctioning stage of a game consists of either placing a bid or starting an auction. However, determining how many resources to bid or sell in an auction is still a difficult question for a learning agent. It is not too difficult to make a fair approximation for the types of reward discussed in the scenario description section. It is done in `QLearningAgent` by making winning an auction or having one of its auctions complete successfully count as a reward. The value of this reward is the utility of the items traded or purchased minus the utility of the items bid or put on sale.

The most difficult part of developing a q-learning agent for the auctioning phase of my scenario is determining how to appropriately model the state of an agent in its environment. Many factors come into play here of which an agent is initially completely unaware, including the priorities, types, and tendencies of other agents. As well, it is not at all clear how to factor in currently-existing auctions and bids into the state calculation. `QLearningAgent` approximates state by representing it as a matrix of the current market value of all resources for all agents. According to the q-learning update equation, these values are updated when auctions are won or lost either by ourselves or by others.

Effectively, this simple approximation of state makes `QLearningAgent` into a level-0 agent, as described in [Wei99]. It utilizes only its knowledge of market values to make decisions, but better cannot be expected without more initial knowledge of other agents.

FictitiousLearningAgent design

Fictitious play was one of the first game theoretic learning rules developed. In repeated or stochastic games, fictitious learning makes the assumption that one's opponent is playing a stable mixed strategy. Using some initial beliefs as a basis, an agent using fictitious learning attempts to find the best response for a given game history and presumed strategy. In many trivial and nontrivial games, fictitious learning can be proved to converge to Nash equilibria, but there are examples where fictitious learning will never converge [SLB06]. Since fictitious learning models other agents in the system statically, any agent using it is automatically level-0.

`FictitiousLearningAgent` models its environment and other agents in an attempt to find a best response. Since the space of combinations of unknown variables involved is extremely large, it is necessary to approximate this response in order to minimize calculation time. A `FictitiousLearningAgent`will act similarly in most instances to a `BasicGreedyAgent`. However, when deciding whether or not to create an auction or to bid on an auction, the agent will attempt to continue to play using resources that have been observed in the past to be widely-desired. It is in the best interest of both agents to have such negotiations for highly-valued items. When no agent wants to further pursue auctions or make bids, the system has reached a possible equilibrium state and `FictitiousLearningAgent` will wait to continue the auctioning process until a new game has begun.

Simulation logging capabilities

The simulation keeps track of the the agents, agent types, agent priorities and resources involved in each trial and will log this information in a human-readable format to a file in the same directory as the main executable. If desired, for each game, the simulation will log the initial positions of agents and resource piles and compile individual agent resource-gathering, auctioning, and utility statistics. The simulation will average these results over the course of an entire trial to produce a final set of logged statistics. The simulation can be paused during execution to see current agent statistics and actions undertaken on the trial, game, and turn level, depending on the level of feedback desired.

Experimental design

In order to validate my system and agent designs, many simulations assessing the performance of the different types of agents were conducted. Several additional experiments were designed to examine some hypotheses I had regarding issues that cropped up while developing `FalloutBots`. Unless otherwise noted, the environmental parameters used during the trials were those given in table 2.

Performance measurements were conducted on the following combinations of agents: BasicGreedyAgents; BasicHoardingAgents; QLearningAgents; FictitiousLearningAgents; BasicGreedyAgents and BasicHoardingAgents; BasicGreedyAgents and QLearningAgents; BasicGreedyAgents and FictitiousLearningAgents; BasicHoardingAgents and QLearningAgents; BasicHoardingAgents and FictitiousLearningAgents; QLearningAgents and FictitiousLearningAgents; BasicGreedyAgents, QLearningAgents and FictitiousLearningAgents; BasicHoardingAgents, QLearningAgents and FictitiousLearningAgents; and a free-for-all.

Table 2: Default experiment simulation parameters

Name	*Value*	*Meaning*
AGENT_GATHER_SPEED	1	Agents can gather a max amount of resources per turn.
AGENT_MOVE_SPEED	1	Agents can move a max distance per turn.
DISCOUNT_RATE	0.8	QLearningAgent agent parameter.
FIELD_*	Varies	Gives the dimensions of the playing field.
HOARDING_FACTOR	2	BasicHoardingAgent agent parameter.
ITER_LIMIT	5	Number of turns a message will stay in the buffer.
LEARNING_RATE	0.2	QLearningAgent agent parameter.
MAX_AGENTS	15	The maximum number of agents possible in a trial.
MAX_AMOUNT_RESOURCE	20	The maximum amount of any resource in a trial.
MAX_GREEDY_AUCTION_UTIL	8	BasicGreedyAgent agent parameter.
MAX_RESOURCE	20	The maximum number of different resources in a trial.
NUM_GAMES	2000	The number of games in a trial.
NUM_TRIALS	10	The number of trials in a simulation.
PLAY_SPEED	0.25	FictitiousLearningAgent agent parameter.
PLAY_START	1	FictitiousLearningAgent agent parameter.
QUIESCENCE	5	Number of turns to wait until ending auction stage.
Priority generation	avg=0.5 std dev=0.1	Values taken from a clamped normal distribution.

During project development, severe restrictions were placed on my design by the necessity of achieving acceptable run times for lengthy trials. To test the limitations of my system as designed, I relaxed several key system limitations, including MAX_AGENTS, MAX_RESOURCE, MAX_AMOUNT_RESOURCE, and FIELD_*, and ran several trials to compare these run times to those obtained in less strenuous testing. This test was conducted with all agent types present.

One of the largest issues I faced during project development was that of generating fair priority values. Initially, priority values were assigned uniformly from [0,1]. However, I quickly realized that this was not even remotely close to fair as an unintelligent agent could dominate others that had lower priorities for every resource in utility calculations.

I changed this scheme to one where, although priority values were generated uniformly, the sum of priority values for an agent had to be one. In this way, no agent could strictly dominate any other for all resources just based on its priorities. However, this led to situations where the winning agents that had a relatively high priority for a small number of resources that were plentiful in the game world would succeed in gathering most of those while others fought over the scraps. Although the auctioning process helped to distribute resources more equitably, certain agents with this property were still winning a disproportionate share of all games.

The priority generation method finally adopted for use in this simulation was to take values from a normal distribution restricted to values in [0,1]. Since I kept the standard deviation relatively small compared to the average, no agent had a large advantage over others in its priority values. In the end, this turned out to be more fair than any other method. An unfortunate side effect of this change was to reduce the number of successful auctions in a game, because the small relative differences in value for items made it unprofitable for agents to exchange most combinations of resources.

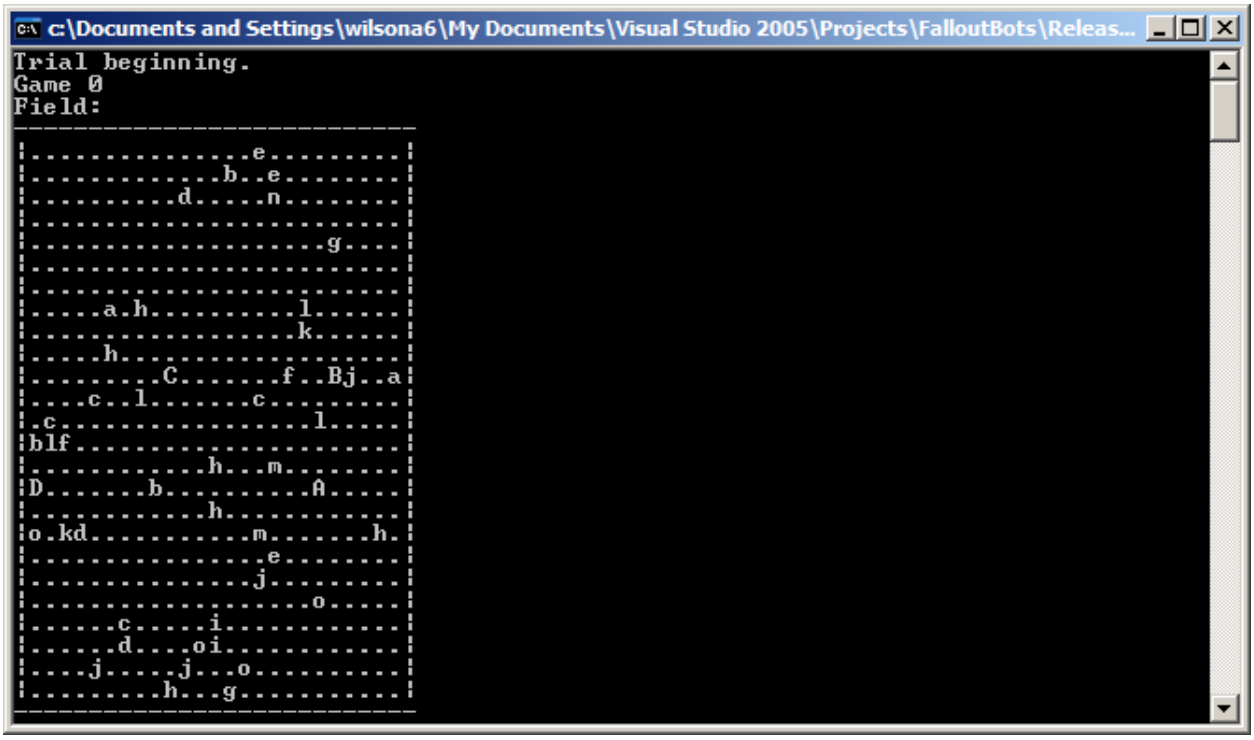

Figure 2: Initial resource-gathering stage layout. Agents are represented by capital letters, piles of resources are given in lower-case.

Two experiments were performed relating to these issues. In one, the standard deviation of the normal distribution from which I draw priority values was increased to 0.2. My hypothesis is that agents will tend to have a poorer allocative efficiency than before, because some agents now have no incentive to trade with any others as they already possess most of their high value resources. My second experiment replaces the current priority generation mechanism with the original scheme, a uniform distribution on [0,1], and seeks to verify my original finding that many agents get shut out entirely from winning due to their lower average priority values. In both of these experiments, I use a homogeneous population of the simplest agent type, `BasicHoardingAgent`, to ensure that agent trading will not lessen the significance of results.

Experimental results

During these experiments, 400,000 games were played and 500 megabytes of data was generated on 10 computers over the course of several hours. This raw data is available on a supplemental CD that will be turned in on Thursday. Representative screen captures of the system in action are given in figures 2 and 3.

As the experiments conducted were on such a large scale, instead of giving a full summary of all results, I will summarize key facts and findings.

The majority of experiments conducted related to comparisons between different types of agents. Not surprisingly, in homogeneous testing on average allocation efficiency, `QLearningAgent` did a little bit better than `FictitiousLearningAgent` who did a lot better than `BasicGreedyAgent`,

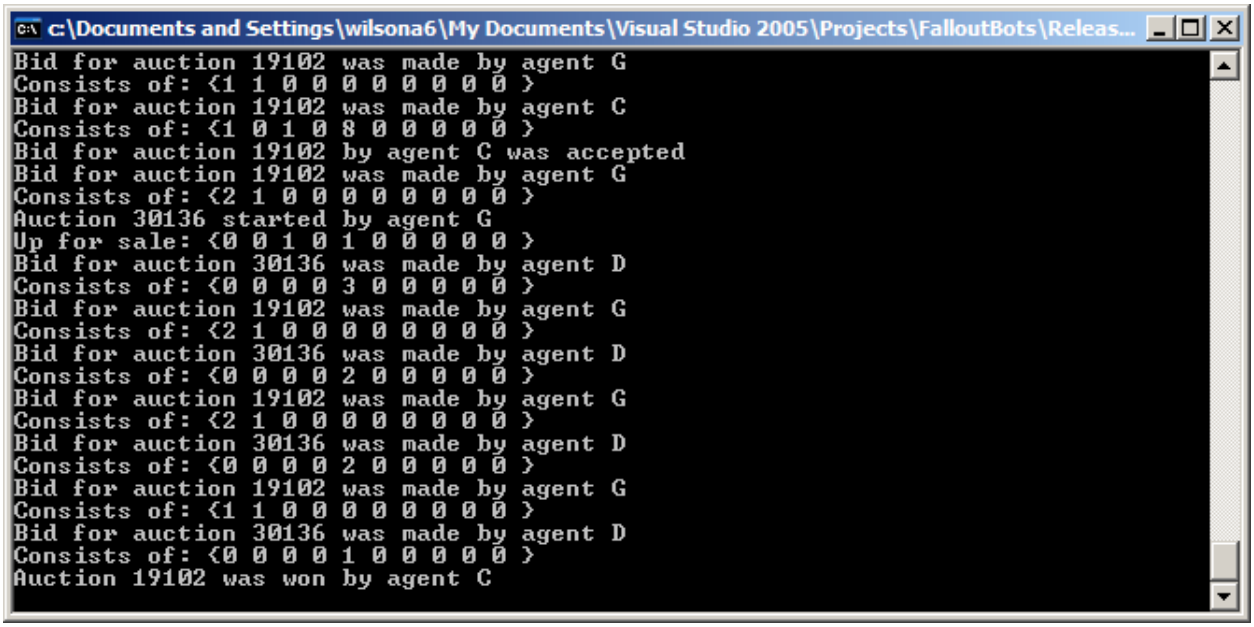

Figure 3: Auctioning stage in progress

which did better than `BasicHoardingAgent`. The learning algorithms of the two learning agents enabled them to better allocate resources among themselves, while the fact that `BasicGreedyAgent` participates in *some* auctioning gave it an advantage over `BasicHoardingAgent`. When comparing overall utility between agents, it was noted with unpleasant surprise that there was not much statistical difference between the different kinds of agents. I conjecture that this is due to the resource-gathering stage determining to a large degree the overall utilities of agents. All types of agent have near-identical resource-gathering implementations.

In the experiments that introduced two or three agent types into the mix, most of the statistical patterns noticed in homogeneous populations held. It was noted, however, that when only one or two agents of a given type was in a simulation that they tended to do more poorly on both metrics than when they were more plentiful. I conjecture that this is due to the lack of agent modeling done by my agents - they assume everyone else is like them, and do not perform well when most were not.

In the final free-for-all experiment, the trends noticed in the one-, two-, and three- agent types experiments once again held.

The experiment conducted on timing revealed that, with the original simulation parameters, trial run times were, on average 43.21 s with a standard deviation of 15.23 s. With relaxed simulation parameters, the average trial run time jumped to 225.96 s with a standard deviation of 79.64 s. With simulation parameters only increased an average 75%, trial run time shot up by 523%, indicating that a linear increase in the simulation space was correlated with an exponential increase in simulation run times.

The first experiment conducted on priority generation validated my hypothesis that relative priority distance has a significant effect on allocation efficiency. With the altered priority mechanism, average allocation efficiency stood at 81.78% with standard deviation 5.23%. With the original simulation parameters, average allocation efficiency dwarfed that at 87.39% with standard

deviation 4.73%.

The final experiment dealt with the distance between game winners and losers in two different priority generation mechanisms. With a uniform priority-generating mechanism, the average standard deviation of number of games won was 201.23 with a standard deviation of 15.13. With a normally-distributed priority generating mechanism, this same figure was 94.76 with a standard deviation of 9.59. Statistically speaking, the standard priority mechanism used in `FalloutBots` is much more fair than the initial method developed for that purpose.

Discussion

I had an unfair advantage when I came up with my hypotheses. I wrote the whole system and agents. I performed all the debugging. I have probably run the system nearly as many times in the development process as was done in the entirety of experimentation. For these reasons and the solid DAI foundation behind my questions and guesses, most of my hypotheses were borne out by experiment.

However, I am disappointed that there remain areas of performance in which the more intelligent agents did not significantly beat what are essentially random agents. I do not feel that these results are in fact inherent faults in the learning mechanisms behind the agents. My approximations to the learning mechanisms are at least somewhat accurate according to the results, but might be inaccurate in a large class of situations. More extensive testing on scenarios in which they can fail would be necessary to discover their limitations.

I believe, however, that the major contributing factor to the lack of a performance gap is that the entire resource-gathering stage of my scenario is played essentially blindly by all agents. Logically, this stage should have a huge effect on the outcomes of a game as agents who dominate it do not have to do much negotiation or auctioning at all. If I were to have more time for this project, I would improve on this area of my agent designs first.

Project process

For the most part, thanks to the relatively well-spaced stream of due deliverables, the project development process was fairly relaxed. However, it may be to students' benefit if the project demo date were moved up. This would be done in order to ensure that running code of some sort is present at an earlier stage in the development process than was the case this semester. My realization that I would have to write my entire project environment and agent code from the ground up and my subsequent move from Java to C++ necessitated scrunching my schedule in before the due date. I was forced to leave many avenues of more detailed analysis and exploration of my system unexamined. The month-long break in deliverables was also problematic in that picking up where I left off proved to be difficult. As a personal note, even with examples of projects from previous classes, finding an interesting project of sufficient but not overwhelming difficulty in the field of DAI is difficult when one does not have existing research projects to draw on. I feel that, at the beginning of the semester, more time should be devoted to an overview of what is to be covered so students can have more idea of what they will be interested in investigating, especially if they are undergraduates or beginning graduate students.

Conclusions and future work

Both q-learning and fictitious learning proved effective in increasing the allocative efficiency for each agent. Although some warnings were given in [Wei99] that concurrent independent reinforcement learning would not perform well in tightly-coupled tasks, even a crude approximation of true learning proved effective in aiding agent results in all tested metrics.

I have only begun to explore the boundaries of the `FalloutBots` scenario. Many more elaborate learning paradigms and algorithms than the ones used have been developed. Their feasibility has not been assessed in terms of `FalloutBots`.

As well, the agents currently implementing learning only attempt to learn during the auctioning stage of a game. From my experiments with altering priority and comparing agents, it seems that the majority of utility for a given agent is acquired in resource-gathering. Therefore, extending the learning algorithms in `QLearningAgent` and `FictitiousLearningAgent` to deal with issues in resource-gathering, including coordination, organizing against competitors, and maintaining an effective message exchange system, should be an important aspect of subsequent research in this scenario.

As another avenue for future work, a more formalized model of the information available to an agent in a game should be developed. This modeling should be done in order to assess what kind of higher-level modeling agents can do with the information they have on their fellow agents. There has been no rigorous analysis done on this topic so far.

More speculatively, this scenario may prove to be a good testbed for the coalition formation algorithms developed in [VA07]. I constructed the `FalloutBots` environment in such a way that an effective translation to physically-situated robots should not be extremely difficult. Coalitions constructed to gather resources or to act in auctions may well be able to achieve much higher levels of efficiency than the agents presented here.

Bibliography

[BT06] Anthony Bagnall and Iain Toft. Autonomous Adaptive Agents for Single Seller Sealed Bid Auctions. *Autonomous Agents and Multi-Agent Systems*, 12:259–292, 2006.

[GB05] Amy Greenwald and Justin Boyan. Bidding algorithms for simultaneous auctions: A case study. *Autonomous Agents and Multi-Agent Systems*, 10:67–89, 2005.

[PL05] Liviu Panait and Sean Luke. Cooperative Multi-Agent Learning: The State of the Art. *Autonomous Agents and Multi-Agent Systems*, 11:387–434, 2005.

[SLB06] Yoav Shoham and Kevin Leyton-Brown. *Multi Agent Systems*. 2006.

[VA07] Lovekesh Vig and Julie Adams. Coalition Formation: From Software Agents to Robots. *Journal of Intelligent Robotic Systems*, 2007.

[Wei99] Gerhard Weiss, editor. *Multiagent Systems: A modern approach to distributed artificial intelligence*. 1999.