CS 396 Special Topics in Artificial Intelligence
Project Deliverable 5

Ari Wilson

April 10th, 2007

Implementation report

In the last few weeks, I have made a significant amount of progress on my project. My implementation has moved from a primarily Java code base to C++ in the interests of speed and language familiarity. I realized that the emphasis of my project was not on determining near-optimal strategies for resource gathering, but rather was on strategies designed to operate well during auctioning. None of the existing domains, agents, or classes in TeamBots had much to do with this kind of simulation, and only the Forage domain came close to meeting my needs for agent resource gathering. Even Forage was designed for a more complex resource collection scenario than I was envisioning and modifying it to fit my needs would have taken more time than I had available. Essentially, I was spending about twice as much time as I would spend developing new solution looking for hacks that would operate somewhat like what I envisioned in the sparse documentation of the TeamBots communication and simulation library in order to get everything to work. I also plan to run many tests of 1,000 or more games, which would not be feasible to do quickly in TeamBots.

For these reasons, I constructed the entire simulation environment and agent code from scratch. One disadvantage of not using TeamBots has been that there no graphical display of game progress, but I have created a rudimentary textual interface to compensate. Automatic simulation statistics gathering is fairly rudimentary and is one of the few implementation details that needs to be finished. Depending on the level of output requested, the simulation will currently print out the agents, agent types, agent priorities and resources involved in a trial. 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 statistics. I have broken up the statistics gathered and the output generated while the simulation is running so that the simulation environment can be examined on the metagame, game, and turn level, depending on the degree of granularity desired.

The structure of my code as it is executed during a typical test run is as follows. Most parameters that I will wish to vary in evaluations are defined in the code as constants that can be quickly identified and changed. These parameters include things like the simulation field size, the number of agents, move speed, and various other simulation and agent limits.

At the beginning of execution, the main function simply creates a `Metagame` object and lets it run a trial, recording the results of the trial to a log file. The `Metagame` class controls the number of agents, their normally-distributed priorities, and the number of resources that will be 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. For fairness, order of turns is

determined randomly for each iteration.

It is within this context that the `ResourceGathering` and `Auctioning` classes determine whether particular actions are allowable and update the game world accordingly. 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()`). To this end, I make the simulation environment's full observable nature explicit by passing a global vector of all agent resources and positions to each agent. Similarly, agent communication in 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 (starting an auction, inquiring on the current highest bid, inform previous bidders of a new bid, placing a bid, or announcing a winner), content, a source, and a destination (or multicast). 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.

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 below its maximum or gathering a predetermined number of units of resources if they are available at its location. A basic example of utility calculations to determine which action to take can be seen in either `BasicGreedyAgent` or `BasicHoardingAgent`. This stage runs until all resources have been gathered.

During the auction stage, each agent can send a large number of messages each turn. Agents are responsible for internally maintaining the state of auctions and bids relevant to themselves, as well as ensuring that they do not bid more than their available resources. Unfortunately, due to time concerns, much low-level auction message interpretation functionality has not been encapsulated appropriately and resides as spaghetti code in the `auction()` function of each agent. To avoid having to learn internal auction and resource gathering details when creating a new agent in my framework, existing agent code should be used as a template. 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 internally referred to as quiescence.

Two existing agents have already been mentioned. `BasicGreedyAgent` and `BasicHoardingAgent` have both been created according to the description given in deliverable 3. There are two new agents, `FictitiousAgent` and `QLearnAgent`. These two agents share their resource gathering logic with `BasicGreedyAgent`. This is done to limit the number of variables present in the simulation and discern the effects of learning on auction outcomes, where predictions of other players utilities can have more effect than in resource play. Extending these algorithms to resource play is left to future work. Based on their interactions with creators of auctions, these two learning policies attempt to learn between games what the priorities of other agents are for each resource. Starting from the statistical average for each resource priority and each agent, `FictitiousAgent` plays a best response to each action based on the previous bids that have been accepted, rejected, or overturned in favor of higher bids. Supposed priorities will be updated according to Bayes' rule. Again starting from the statistical average for each resource priority and agent, `QLearnAgent` will similarly use the q-learning algorithm with reinforcement learning to build a model of other agent characteristics.

The remaining work to be done on these last two agent types and with statistics gathering will be done before this weekend, which will be devoted to evaluations and analysis.

Data analysis

The data gathered from my experiments with varying types of agents (homogeneous, partially mixed, fully mixed) will be conducted over ten 2,000 game trials for each type of experiment. The

data can be analyzed to give the winning percentages of agent types, the standard deviation on the average chance of winning, along with similar statistics given over the auction results of each experiment: which agents won the highest percentage of auctions. The two learning agents will be compared using the two basic agents as baselines against which to measure improvement, using chi-square tests. A detailed discussion of the reasoning behind the selection of other simulation parameters, including the learning parameters of Q-learning and fictitious play, will be given.

Some further qualitative evaluations will be given about the performance of the baseline agents during resource play and the effectiveness of the basic auctioning mechanism found in `BasicGreedyAgent` versus the lack of auction participation by `BasicHoardingAgent`. As these are not vital to my project, they will not be given too much attention in the final report.

Evaluation and results analysis

After all data has been appropriately analyzed, summaries and charts will be created for use in the final report and presentation. Only then can conclusive results be given about the efficacy of the agent designs constructed for this project scenario.