TRADING AGENTS

Lutfi Ilke Kaya
Department of Computer Science
Brown University

Introduction

In this paper, I am going to report my work and achievements on three different Trading Agent Competition (TAC) projects: TAC Supply Chain Management (SCM), TAC Ad Auctions (AA) and TAC Prediction Challenge (PC).

Supply Chain Management

1. Replacement Costs

Replacement cost is the cost of replacing a component in the inventory, after it is used to build a computer. Its necessity and implementation were one of the longest and most important discussions in our group in Fall 2008.

a. Necessity

I prepared several tests and analyzed their results to show the necessity of these artificial costs. Results showed that replacement costs are extremely important because of our bidding strategy.

Our bidder considers 3 costs for each increment: cost of having the computer in the inventory, cost of having the components in the inventory, and cost of procuring the components. Minimum of these three costs is considered to be the cost of that increment. Let's say in an increment, we decided to procure components. When these arrive, they may sit in the inventory for days before they are used. As they are sitting in the inventory, for another increment, the bidder may use all or some these components (because the cost of that increment will be 0, as components are already in the inventory).

This is a main source of sub-optimality, causing negative millions of dollars of profit and dozens of unused components sitting in the inventory. After analyzing these results, we decided that replacement costs are necessary for our greedy algorithm.

b. New Configuration

While analyzing the implementation of replacement costs; I noticed that only the running-out-of-stock dates of components are considered while calculating replacement costs. However, a component may be replaced anytime between the procurement and the running out of stock date. I modified the code to work this way. This change fixed the sub-optimality, however the calculations became computationally more demanding. Because of that, I made this new change configurable, in a simple model, the previous implementation may be used.

c. Redundant Computations

Already calculated replacement costs were recalculated every day, even though they do not change. I fixed this problem and saved some computation time.

2. Overlapping Parts

a. Optimality

During our discussions about replacement costs and optimality of the greedy algorithm, we ended up with settings that caused our algorithm to work sub-optimally with overlapping parts. I implemented these settings to prove our claims and ran games. Results proved their sub-optimal effects.

We did not find a solution to this problem, however we did some brainstorming and we had the idea of performing local search after marginal procurer returns its solution to see if any increments could be given up for two more profitable increments. We did not implement this idea and decided to address other sources of sub-optimality first; also ILP does not have this problem.

b. No Overlapping Parts Configuration

After we proved the effect of overlapping parts, we decided to take it out of the scene so that we can have a simpler setting to analyze and find possible flaws in the algorithm. I implemented a configuration that products have no overlapping parts. We used this configuration to check the optimality of our other algorithms.

3. ILP and Greedy Algorithm Comparisons

We decided to compare ILP and Greedy Algorithm in our bidder and procurers. Expected Bidder and Procurer use ILP and Marginal Bidder and Procurer use Greedy Algorithm. I created 8 unique settings, each one using either marginal or expected bidder, marginal or expected procurer and replacement costs or not. I ran 8 games with these settings. Greedy algorithm did really bad in these tests, it completed games with -400 million dollars and the inventory filled with unused components, ILP was doing much better than the greedy algorithm, but it was still far away from being optimal. In the light of these tests, we decided to focus on the sources of sub-optimality in our greedy algorithm.

4. Sources of Sub-optimality in Marginal Procurer

We decided to address the sources of sub-optimality in marginal procurer. We ended up with 7 problems: infinite supplier RFQs, past increments, future increments, late lead-time increments using procurement, late lead-time increments using inventory, overlapping parts and initial inventory. We created and implemented several tests and example cases and analyzed their results. Then, we wrote a report on each of these items.

<u>Ad Auctions</u>

A detailed report of our group's work on our agent, Shlemazl, can be found in our paper "Autonomous Bidding in Ad Auctions".

1. Bid to Position Models

My first responsibility in this project was designing and implementing Bid to Position models. These models return the expected average position for a given query and bid. Here I will explain two models, a third model, Ensemble Model, is also present in one of the following items.

a. Linear Bid to Position Model

A very simple model, which was intended to use as a test case at first. It keeps the previous days bid and position for each query, and calculates the position of a given bid with the formula (q: query)

bid(q) = lastBid(q) + lastBid(q) * m * (lastPos(q) – pos(q))

m is the slope here, and I used 5 different linear bid to position models with m = 1/3, 1/4, 1/5, 1/6 and 1/7.

This model does not require training, inserting a point to the model implicitly trains it.

If the predicted position is smaller than 1, its set to 1. If its smaller than the number of slots, its set to Double.NaN.

b. Bucket Bid to Position Model

This is a model that clumps up bids into buckets and map buckets to positions instead. Bucket size is a constant in the code and its default value is 0.4. An improvement to this model would be to find the optimal bucket size empirically, but 0.4 worked fine.

Every bid is assigned to a bucket with the following formula:

bucket(query, bid) = (int) bid / BUCKET_SIZE

Every query has its own buckets and there is not a fixed number of buckets, i.e. if bid = 4 and BUCKET_SIZE = 0.5, that bid will be assigned to bucket 8. However, if we never see a bid as high as 4, there will not be an 8th bucket.

After a bid is assigned to a bucket, the bid’s corresponding position is added to that bucket. In other words, buckets carry the sum of positions:

bucket += position(query, bid)

bucketAppearance(query, bucket) += 1

When we are given a query and bid to predict a position, we simply get the bucket of the bid, return the sum of positions in the bucket divided by the number of times that bucket was incremented.

position(query, bid) =

content(bucket(query, bid)) / bucketAppearance(query, bucket)

In this model, “no position” is represented as (“number of slots” + 1). Position is returned as Double.NaN (which is used in case of no position in the actual game) if we get a position higher than the number of slots.

Of course, it is possible that the bucket of the bid that the model needs to predict does not exist. In this case, the model uses the closest buckets on its left and right and returns an average distance according to the distance.

2. Position to Bid Models

These models can be used to predict Position – Bid relations.
They are almost the same with bid to position models except some minor changes. Because of this, I am not going into details. To give an idea about the changes, in Bucket Position to Bid Model, bucket size is set to 1.0 instead of 0.4 to represent a position. Positions are assigned to buckets and buckets carry the sum of bids for that position.

3. Model Evaluation

I implemented a comparable model infrastructure that can be extended by any model. This way, a user can evaluate their models by directly passing them to the evaluator without changing any part of the code, provided their models extend the comparable structure. This made it very easy to use the evaluators; furthermore it gave the freedom of implementing new evaluators easily by just extending the basic evaluator. Also, this structure made it possible to develop the ensemble model.
I implemented two simple evaluators using this structure:

a. Absolute Error Model Comparison

This evaluator calculates the absolute errors of the models.

In my implementation, TEST_SIZE was 5 and it worked fine. As an improvement, an optimal size can be found empirically.

Input
List<ModelDataPoint> datapoints (from game log)
List<AbstractComparableModel> models

Output
Best model & Absolute error of each model

```
foreach model in Candidate Models
    for d = datapoints.size – TEST_SIZE; d < datapoints.size; d++
        for t = 0; t < d;  t++
            insert data[t] to model
        end
        train model
        get the prediction for data point at d
        increment error of model by absolute (prediction – actual)
        reset model
        end
    end
```

b. Norm-Squared Error Model Comparison

This evaluator is the same as Absolute Error Model Comparison, except it computes norm-squared error.

c. Evaluate From Log

This may be handy if users would like to evaluate their models using game logs. It loads model data points from a (already parsed) log file, creates the objects that the evaluator requires, passes them to the evaluator and evaluates them.

4. Ensemble Model

This model utilizes the model evaluator and basically evaluates a set of candidate models on the fly and uses the one that fits the current data the best. The best model is selected by the evaluator and is basically the model that gave predictions with the least error for the last TEST_SIZE days.
This is evaluation process is done every day with updated data and a new model is selected every day (the same model can be selected consecutively of course). Then, the position is computed with:

position(query, bid) = bestModel.getPosition(query, bid)

This is a very good model because we do not have to stick with a single model but use the best fitting one instead. Of course, this does not necessarily mean that it will perform better than any candidate model it considers. Because, a model may perform poorly in the last 5 days, therefore not be selected by this model, but still get the best predictions for the next day.
We entered TAC AA 2009 with this model and a set of 25 candidate models.

Prediction Challenge Modules

Prediction Challenge is a TAC game where teams focus on the prediction part of TAC SCM. Our Prediction Challenge models make almost perfect predictions and we would like to use these predictions in TAC SCM. In other words, we wanted to create two models in TAC SCM, which are modules that feed daily information to Prediction Challenge (in regular usage, PC gets this information from parsed game logs) and get predictions from it.
These models needed to access to Prediction Challenge code, so Prediction Challenge source is now referenced from TAC SCM.

1. Component Prices Predictor Module

This module passes daily information from TAC SCM to a component prices predictor in Prediction Challenge and returns component price predictions. It is used as a Supply Model in TAC SCM, therefore it extends the SupplyModeler class and implements its abstract methods. These methods are where we fill in the in and

out message vector to Prediction Challenge. These vectors represent the information in the game logs, the messages received and sent by the agent, respectively. This was one of the most important parts of this implementation; any problem here would affect the predictions in a very bad way.
Another challenge I faced during the implementation was the lack of deep copying in Prediction Challenge classes. After I implemented the initial module and ran tests to make sure that daily data is passed correctly, I noticed that the predictions were wrong. After analyzing some reports, I noticed that it was caused by the "backtracking" nature of SCM Supply Model, basically it deep copies itself before each try and then backtracks to that copy, thus lack of deep copying in Prediction Challenge caused faulty copies of the predictor in the module. Fixing this problem was very problematic because no class in Prediction Challenge had deep copy methods and had a very deep structure, so I had to add deep copies to almost every class in Prediction Challenge. I made the deep copies working for SimplestComponentPricesPredictor, and ran tests to make sure that data flow is correct. I could not run a comparison between predictions using the logs and predictions using the module, because SimplestComponentPricesPredictor could not make any predictions using the game log, even though other predictors could. Unfortunately I could not fix the parser because I had no documentation on how it worked (and it wouldn't worth the time because the problem was caused by just a dummy predictor) and did not have time to get another price predictor to work with the module because of the necessity of implementing deep copies.
Instead, I wrote a report about what I did and what needs to be done. Basically, all that needs to be done is just implementing deep copies to the predictors the users would like to use. Then, the module can use these predictors, and the success of the predictions can also be checked.

2. Computer Prices Predictor Module

Initially I was only planning to implement Component Prices Predictor but as I understood the code and noticed that some parts of the modules are very similar, I decided to begin working on this as well.
The way that this module works is very similar with the module above, except it predicts product prices instead thus has a computer prices predictor in it. This one did not require any deep copies, so implementation was much easier.
After finishing the implementation, I made sure that data flow is correct and everything is working properly. Here I started my work on the simplest predictor as well, and during the tests I noticed that it was so simple that it did not allow me to make a healthy comparison between the predictions from the log and the ones from the module. As far as I checked, everything is working properly, but other predictors may be modified to be compatible with the module so that we are 100% sure the predictions are correct.
Again, I wrote a report explaining this part, even though what is left to do is very straightforward: users need to make sure that the predictors they would like to use are compatible with the module (SimplestComputerPricesPredictor can be used as an example).