CM50206 Intelligent Agents: TAC Report

Thomas Smith

Centre for Digital Entertainment
University of Bath
taes22@bath.ac.uk

Abstract. TACCA is an autonomous bidding agent designed to compete in Trading Agent Competition tournaments. The final implementation is a hybrid approach melding techniques from a number of previous agents and custom improvements. This paper details the overall architecture of the Tacca implementation, and comments on the outcome of a sample tournament against other agents implemented in a similar context.

1 Introduction

This paper describes the design and implementation of 'TACCA': a Trading Agent Competition Competitive Agent. The purpose of the competition is to make a number of informed decisions about the purchase of commodities in three varied markets, within an environment of limited time and information, and in competition with other agents pursuing similar objectives. It is an open academic research problem with annual competitions and multiple dedicated teams. Given the wealth of published papers available describing possible (and occasionally successful) approaches to the challenge, Tacca is based on a number of known good approaches, most significantly that of RoxyBot [3], the winner of the 2006 TAC tournament.

1.1 Overview

This paper begins with a very brief description of the Trading Agent Competition, in order to give context for the high-level description of RoxyBot, the agent on which the initial approach was modelled. After explaining the approach taken to high-level choices of algorithm, the actual implementation of modules within Tacca is then detailed, followed by a critical analysis of the performance of the final approach, a discussion of the competition results, and a number of suggestions for possible future improvements to the agent. Finally, the entire source of the modified DummyAgent.java file is presented in Appendix A.

2 Background

The TAC consists of three main challenges: purchasing flights (which generally increase in price over time), bidding on hotels (sixteenth-price auctions which close randomly through the round), and buying and selling entertainment tickets (continuous double auctions between agents). The overall challenge is to purchase from all three markets sufficient commodities to construct valid trips, which are scored according to client preferences. A typical approach is to develop methods for predicting opportune moments to purchase each of the three types of commodity [5], with optionally some overall optimisation method used to allow predictions about each market to inform purchasing decisions in the other two. This optimisation approach is the one taken by RoxyBot [4].

Greenwald, Lee, & Naroditskiy describe an implementation that uses tailored methods for each market to produce predicted future pricelines, and then a two-stage stochastic optimiser in order to select reasonable prices to bid 'now' and 'later' [3]. Three different approaches are used to the priceline generation problem:

Flights: Though flight prices vary according to a semi-random walk, the TAC model for generating this walk is well understood [1], and depends on a single hidden variable z for each flight. RoxyBot uses Bayesian updating to model a probability distribution for z, and hence predict estimated minima in the expected random walk.

Hotels: RoxyBot simulates Simultaneous Ascending Auctions in order to predict competitive equilibrium prices for each hotel, and hence appropriate maximum bids [2]. A number of minor optimisations are implemented in order to provide more accurate long-term predictions.

Entertainment: Optimistic predictions are made using sampled historical data from the logs of the last 40 games. Estimates at any given time are based upon historical data for that period, and estimates for future times are based on the minima across all future times.

The generated pricelines from these algorithms are used as input to an n-stage stochastic optimiser, which for simplicity is collapsed to a two-stage problem – "current" and "future". Sample average approximation is used as a reasonable heuristic for solving the bidding problem, and bidding decisions and bid amounts are predicated on the output of the optimiser.

3 Approach

According to an analysis of previous international TAC tournament competitors, 'taking into account game-specific information about flight prices is a major distinguishing factor [in the relative efficacy of alternative approaches].' [6]. Therefore initial implementation efforts were directed towards reimplementing the Bayesian updating method used by RoxyBot to estimate probability distributions for z, and generate future minima for each flight to inform bidding decisions. This constrained the time available to reimplement the other RoxyBot modules (hotel and entertainment priceline generation, overall optimisation), and so less complex approaches were chosen to improve hotel and entertainment bidding decisions.

4 Implementation

Using the provided `DummyAgent.java` as an initial implementation of the three bidding strategies, incremental improvements were constructed and tested on the public practice servers. Mercurial was used for version control in order to allow easy reversion of approaches that proved unhelpful overall.

4.1 Flights

The flight prediction approach was modelled on the implementation in RoxyBot [4], which uses Bayesian updating to model and refine a probability distribution for the possible values of the hidden variable z, in the range $[-10, 30]$. The TAC server uses z to influence the progress of the random price walk of each flight - generally for values of $z < 0$, the price will tend to decrease over time, while for values $z > 0$ the reverse is true [1]. For each possible value of z the price may be randomly perturbed within a given range; therefore Tacca defines a Range class in order to model the expectation of these possibilities, compare them to known pricing data received from the server, and use Bayesian inference to update the probability distribution for z.

In order to provide predictions of future minima, an expected walk is generated using the current price of the flight and each of the remaining possible values for z. A weighted sum of the minimum of each walk is then calculated, using the individual probability of each value of z. Finally, this expected minimum is compared to the current price, and a bid placed once sufficient confidence is reached that the current price is less than probable future prices.

4.2 Hotels

The hotel allocation and purchase approach that Tacca uses is very closely modelled on the one supplied in the default implementation – ideally dynamic reallocation would be possible based upon the flight pricelines, but time constraints made this infeasible.

4.3 Entertainment

Tacca improves upon the naïve entertainment allocations made by the `DummyAgent` class. In the original implementation, each client is assigned only their most preferred entertainment type if it is available within the initial allocation of tickets, or an attempt is made to purchase it for the first day of their visit if it is not already owned. Tacca uses a more complex approach which allocates each client at least one ticket from the initial allocation in order of preference, and then additional allocations are made so long as there are entertainments available on valid days. Finally, tickets are selected for purchase to ensure that each client receives at least one entertainment ticket – preferably their first choice.

5 Critical analysis

5.1 Competition Results

Despite maintaining reasonably good standings on the available practice servers, Tacca performed relatively poorly in the overall competition rankings coming in 23rd out of 30 competitors, with an average score just under a third of the highest-performing agent's. There appear to be a number of contributing factors to this result. As described above, in the absence of hotel price prediction and overall purchasing optimisation, the flight price minima prediction provides a small overall benefit at best. The combination of the minima prediction and the improved entertainment allocation was enough to ensure consistently good performance on the available public practice servers, however logs from public practice indicate that only a small sample of the other agents competed on the public servers contemporaneously with Tacca, and this sample was skewed as it contained none of the ultimately top-performing agents in the competition proper. Competition logs reveal that Tacca's low average score is largely driven by a significant number of games with negative outcomes - where flights were purchased and entertainments retained, but insufficient hotel nights purchased to complete valid packages. This indicates that other agents with more aggressive or intelligent hotel purchasing strategies were more successful in the competition for limited hotel availability. Ultimately, Tacca achieved a reasonable positive score, though did not score as well as the majority of the other agents.

5.2 Possible Improvements

Given the low overall placement of Tacca in the competition rankings, it is apparent that a number of improvements are possible. Evidently, given RoxyBot's track record in the international competitions, complete reimplementation of RoxyBot would have been likely to perform well. Specifically, it would also have greatly improved the utility of the flight prediction model as without the ability to optimise hotel purchases around cheap flights, the benefit of the minima prediction is reduced solely to the ability to buy decreasing flights (roughly $\frac{1}{4}$of the total flights) cheaply. Unfortunately, the limited timescale made achieving a complete reimplementation challenging, resulting in the hybrid agent presented. Looking specifically at the approaches for each market that were eventually implemented, there are a number of small but significant improvements that could still have been implemented.

5.2.1 Flights In some cases Tacca delayed the purchase of flights beyond the point necessary to have reasonable confidence in the probability distribution of z. A heuristic approach could be taken to analyse whether any particular value was probable beyond a certain threshold or number of standard deviations. Given appropriate tuning, this could allow Tacca to bid earlier on flights that were most likely to solely increase in price throughout the game.

5.2.2 Entertainment Though the entertainment allocation and purchasing approach works well in the early stages of the game, making prudent use of the initially supplied tickets, there are a number of minor alterations that would have resulted in more efficient purchasing and selling. The clients' desires are only considered during the initial allocation phase, and in the case of no relevant entertainments existing in the initial allocation a single preferred entertainment type and date is naïvely selected for future purchase. Rather than a single allocation, a structure representing the client's assigned value for each entertainment could be constructed and queried whenever entertainment prices changed, and a relevant entertainment purchased only whenever it was available below the client's relevant cost. Similarly, once the initially allocated entertainments are purchased, they are retained for the remainder of the game regardless of the market fluctuations. The same structure of clients' proposed prices could be used to calculate the marginal utility of selling any given entertainment at any given time, and then make them available on the market whenever the predicted utility passes a certain threshold. These improvements to the purchase and sale of entertainment tickets would likely result in fewer allocations of owned entertainments to clients, and greater overall profit due to cheaper purchases and sale of valuable tickets.

5.2.3 Hotels The generation of future pricelines by the flight price prediction module should in theory allow optimisation over hotel selection by enabling dynamic reallocation of trip dates and duration in order to take advantage of savings available from cheap flights – wherever these outweigh the transport penalty for alteration from client preferences. This would also work in reverse – if hotels were available unusually cheaply due to low demand, an optimiser could also dynamically reallocate the flight selection. Unfortunately the time constraints did not allow for successful implementation of RoxyBot's optimisation strategy.

References

[1] Cheng, S.F., Leung, E., Lochner, K.M., O'Malley, K., Reeves, D.M., Schvartzman, J.L., Wellman, M.P.: Walverine: A walrasian trading agent. Decision Support Systems 39(2), 169–184 (2005)

[2] Cramton, P., Shoham, Y., Steinberg, R.: Simultaneous ascending auctions (2006)

[3] Greenwald, A., Lee, S.J., Naroditskiy, V.: Roxybot-06: Stochastic prediction and optimization in TAC travel. Journal of Artificial Intelligence Research 36(1), 513–546 (2009)

[4] Lee, S.J., Greenwald, A., Naroditskiy, V.: Roxybot-06: An (saa) 2 tac travel agent. In: IJCAI. pp. 1378–1383 (2007)

[5] Stone, P., Schapire, R.E., Csirik, J.A., Littman, M.L., McAllester, D.: Attac-2001: A learning, autonomous bidding agent. In: Agent-Mediated Electronic Commerce IV. Designing Mechanisms and Systems, pp. 143–160. Springer (2002)

[6] Wellman, M.P., Reeves, D.M., Lochner, K.M.: Price prediction in a trading agent competition [extended abstract]. In: Proceedings of the 4th ACM conference on Electronic commerce. pp. 216–217. ACM (2003)

A Source Listing

Modified DummyAgent.java, licensing and comment header skipped

```java
package se.sics.tac.aw;
import se.sics.tac.util.ArgEnumerator;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.Map.Entry;
import java.util.TreeMap;
import java.util.Map;
import java.util.logging.*;

public class DummyAgent extends AgentImpl {

  private static final Logger log =
    Logger.getLogger(DummyAgent.class.getName());

  private static final boolean DEBUG = false;

  private static final int FLIGHTS = 8;
  private static final float FLIGHT_MIN = 150.0f;
  private static final float FLIGHT_MAX = 800.0f;

  //------------------ int z = ??;    // z per flight : bound on final peturbation. Unknown
  private static final int c = 10;    // -c : lower bound of possible z values
  private static final int d = 30;    //  d : upper bound for
  private static final float T = 540.0f;  //  T : total game time in seconds

  private float[] bidPrices;
  private float[] currPrices;
  private float[] flightDeltas;

  private List<Map<Integer, Float>> Pz;
  private ArrayList<int[]> clientEntPrefs;

  //INVESTIGATE: this code doesn't get called between games.
  //         (re-)initialisation of values moved to gameStarted()
  protected void init(ArgEnumerator args) {
    bidPrices = new float[agent.getAuctionNo()];
    currPrices = new float[agent.getAuctionNo()];
    flightDeltas = new float[FLIGHTS];

```

```java
    Pz = new ArrayList<Map<Integer, Float>>();

  }

  public void quoteUpdated(Quote quote) {
    int auction = quote.getAuction();
    int auctionCategory = agent.getAuctionCategory(auction);
    if (auctionCategory == TACAgent.CAT_HOTEL) {
      int alloc = agent.getAllocation(auction);
      if (alloc > 0 && quote.hasHQW(agent.getBid(auction)) && quote.getHQW() < alloc) {
        Bid bid = new Bid(auction);
        // Can not own anything in hotel auctions...
        bidPrices[auction] = quote.getAskPrice() + 50;
        bid.addBidPoint(alloc, bidPrices[auction]);
        if (DEBUG) {
          log.finest("submitting bid with alloc="
                 + agent.getAllocation(auction)
                 + " own=" + agent.getOwn(auction));
        }
        agent.submitBid(bid);
      }
    } else if (auctionCategory == TACAgent.CAT_ENTERTAINMENT) {
      int alloc = agent.getAllocation(auction) - agent.getOwn(auction);
      if (alloc != 0) {
        Bid bid = new Bid(auction);
        if (alloc < 0)
          bidPrices[auction] = 200f - (agent.getGameTime() * 120f) / 720000;
        else
          bidPrices[auction] = 50f + (agent.getGameTime() * 100f) / 720000;
        bid.addBidPoint(alloc, bidPrices[auction]);
        if (DEBUG) {
          log.finest("submitting bid with alloc="
                 + agent.getAllocation(auction)
                 + " own=" + agent.getOwn(auction));
        }
        agent.submitBid(bid);
      }
    } else if (auctionCategory == TACAgent.CAT_FLIGHT) {
      // calculate delta from last know price (ternary guard for initialisation spike)
      flightDeltas[auction] = quote.getAskPrice() - ((currPrices[auction] > 0.0f)? currPrices[auction] : quote.getAskPrice());
      currPrices[auction] = quote.getAskPrice();
```

```
      log.fine("got quote for auction " + auction + " with price " + currPrices[auction] + "( delta: " + flightDeltas[auction] + ")");
    }
  }

  public void quoteUpdated(int auctionCategory) {
    log.fine("All quotes for "
        + agent.auctionCategoryToString(auctionCategory)
        + " have been updated");
    if (auctionCategory == TACAgent.CAT_FLIGHT) {
      long seconds = agent.getGameTime();
      log.fine("Predicting future flight minima after " + seconds/1000 + " seconds");
      flight_predictions((int) (seconds/1000));
      expected_minimum_price((int) (seconds/1000));
      for (int i = 0; i < FLIGHTS; i++) {
        log.fine("Flight " + i + ": current price is " + currPrices[i] + ", expected minimum is " + bidPrices[i]);
        // if (the game is ending soon, or current price is within 5% of the expected minimum) and we need the flight and we've had time to study trends
        if ((seconds > 500*1000 || currPrices[i] < 1.05 * bidPrices[i]) && agent.getAllocation(i) - agent.getOwn(i) > 0 && seconds > (20 + 10*i) * 1000) {
          log.fine("Bidding.");
          Bid bid = new Bid(i);
          bid.addBidPoint(agent.getAllocation(i) - agent.getOwn(i), currPrices[i]);
          if (DEBUG) {
            log.fine("submitting bid with alloc=" + agent.getAllocation(i)
                + " own=" + agent.getOwn(i));
          }
          agent.submitBid(bid);
        }
      }
    }
  }

  public void bidUpdated(Bid bid) {
    log.finer("Bid Updated: id=" + bid.getID() + " auction="
        + bid.getAuction() + " state="
        + bid.getProcessingStateAsString());
    log.finer("       Hash: " + bid.getBidHash());
  }

  public void bidRejected(Bid bid) {
    log.warning("Bid Rejected: " + bid.getID());
```

```java
    log.warning("       Reason: " + bid.getRejectReason()
    + " (" + bid.getRejectReasonAsString() + ')');
  }

  public void bidError(Bid bid, int status) {
    log.warning("Bid Error in auction " + bid.getAuction() + ": " + status
    + " (" + agent.commandStatusToString(status) + ')');
  }

  public void gameStarted() {
    log.fine("Game " + agent.getGameID() + " started!");

    //reinitialise prices, deltas and z-probabilities
    bidPrices = new float[agent.getAuctionNo()];
    currPrices = new float[agent.getAuctionNo()];
    flightDeltas = new float[FLIGHTS];

    Pz = new ArrayList<Map<Integer, Float>>();
    // cache the value of the uniform initial probability of any z
    // INVESTIGATE: is there an off-by-one error here? surely there are 41 z values
    float uniformP = 1.0f / (c+d);
    for (int flight = 0; flight < FLIGHTS; flight++) {
      TreeMap<Integer, Float> m = new TreeMap<Integer, Float>();
      for (int z = 0-c; z <= d; z++) {
        m.put(z, uniformP);
      }
      Pz.add(m);
    }
    log.fine("Initialised variables. Pz.size(): " + Pz.size() + " Pz[3].size(): " + Pz.get(3).size());
    log.fine("    Pz[3].get(-2): " + Pz.get(3).get(-2));

    calculateAllocation();
    sendBids();
  }

  public void gameStopped() {
    log.fine("Game Stopped!");
    for (int i = 0; i < FLIGHTS; i++) {
      log.fine("bidPrices[" + i + "]: " + bidPrices[i] + "\t currPrices[" + i + "]: " + currPrices[i]);
    }

  }

  public void auctionClosed(int auction) {
```

```
    log.fine("*** Auction " + auction + " closed!");
  }

  // Nested class to represent ranges for flight value
      peturbations
  class Range {

      private float low, high;

      public Range(float l, float h){
          this.low = l;
          this.high = h;
      }

      public boolean contains(float number){
          return (number >= low && number <= high);
      }

      //generates a valid range given hypothetical z and t in
           seconds
      // (c and d are constants used in the generation of z
          by the server; 10 and 30 respectively)
      public Range(int t, int z) {
        float x = c + (t/T)*(z-c);
        if (x > 0) {
          this.low = 0-c;
          this.high = x;
          return;
        } else if (x < 0) {
          this.low = x;
          this.high = c;
          return;
        }
        this.low = 0-c;
        this.high = c;
      }

      // return the uniform probability of any int within the
           range
      public float uniformP() {
        return 1.0f / (high - low);
      }

      // return the midpoint of the range, used for expected
          values
      public float getMid() {
        return (high - low) / 2.0f;
      }

      public String toString() {
```

```java
      return "(" + this.low + ") - (" + this.high + ")";
    }

}

// for each possible value of z for each flight, calculate the likelihood that that value
//  is the one the server is using to generate the prices
private void flight_predictions(int t) {
  int flightNo = 0;
  // for each flight (each initialised with possible values of z from -c to d [-10,30])
  for (Map<Integer, Float> flight : Pz) {

    log.fine("Calculating for flight " + flightNo + "; " + flight.size() + " values for z remain.");
    float runningTotal = 0;

    Iterator<Entry<Integer, Float>> z = flight.entrySet().iterator();
    Range r;

    // for each remaining possible value of z
    while (z.hasNext()) {
      Entry<Integer, Float> p = (Entry<Integer, Float>)z.next();
      r = new Range(t, p.getKey());   // calculate the range of possible values for y
      if ( r.contains(flightDeltas[flightNo]) ) {          // if y is within range for this z
        p.setValue( r.uniformP() * p.getValue());
        runningTotal += p.getValue();
      } else {
        log.finest("" + currPrices[flightNo] + " is outside probable range: " + flightDeltas[flightNo] + " exceeds " + r.toString());
        z.remove(); //this value of z cannot explain observed prices, discard it.
      }
    }

    // normalise the probablilities of each z value remaining plausible for this flight
    for (Entry<Integer, Float> p : flight.entrySet()) {
      flight.put(p.getKey(), p.getValue()/runningTotal);
    }

    flightNo++;
  }

```

```
    return;
  }

  // for each possible value of z for each flight, calculate the minima along an expected walk
  //  take a weighted average of these minima according to the probabilites of z
  private void expected_minimum_price(int t) {
    int flightNo = 0;
    // for each flight
    for (Map<Integer, Float> flight : Pz) {

      float runningTotal = 0.0f;
      //for each plausible value of z
      for (Map.Entry<Integer, Float> z : flight.entrySet()) {
        float min = Float.POSITIVE_INFINITY;
        float p = currPrices[flightNo]; //current price for this flight
        //simulate forwards to the end of the game
        for (int tau = t; tau <= T; tau+=10) {
          //peturbing by naive expectations of delta
          float delta = new Range(tau, z.getKey()).getMid();
          p = Math.max(FLIGHT_MIN, Math.min(FLIGHT_MAX, p + delta ));
          //track the minimum price observed
          if (p < min) {
            min = p;
          }
        }
        // multiply min by the probability that this is the one true z
        runningTotal += min * z.getValue();
      }
      // set our expected minimum for this flight to the weighted average
      bidPrices[flightNo] = runningTotal;
      flightNo++;
    }

  }

  private void sendBids() {
    for (int i = 0, n = agent.getAuctionNo(); i < n; i++) {
      int alloc = agent.getAllocation(i) - agent.getOwn(i);
      float price = -1f;
      switch (agent.getAuctionCategory(i)) {
      case TACAgent.CAT_FLIGHT:
        // don't bid on flights at the start of the game - we wait for the opportune moment
        break;
```

```java
    case TACAgent.CAT_HOTEL:
      if (alloc > 0) {
        price = 200;
        bidPrices[i] = 200f;
      }
      break;
    case TACAgent.CAT_ENTERTAINMENT:
      if (alloc < 0) {
        price = 200;
        bidPrices[i] = 200f;
      } else if (alloc > 0) {
        price = 50;
        bidPrices[i] = 50f;
      }
      break;
    default:
      break;
    }
    if (price > 0) {
      Bid bid = new Bid(i);
      bid.addBidPoint(alloc, price);
      if (DEBUG) {
        log.finest("submitting bid with alloc=" + agent.getAllocation(i)
             + " own=" + agent.getOwn(i));
      }
      agent.submitBid(bid);
    }
  }
}

//store the client preferences as allocated desires in particular auctions.
private void calculateAllocation() {
  clientEntPrefs = new ArrayList<int[]>();

  for (int i = 0; i < 8; i++) {
    int inFlight = agent.getClientPreference(i, TACAgent.ARRIVAL);
    int outFlight = agent.getClientPreference(i, TACAgent.DEPARTURE);
    int hotel = agent.getClientPreference(i, TACAgent.HOTEL_VALUE);
    int type;

    // Get the flight preferences auction and remember that we are
    // going to buy tickets for these days. (inflight=1, outflight=0)
```

```java
      int auction = agent.getAuctionFor(TACAgent.CAT_FLIGHT, TACAgent.TYPE_INFLIGHT, inFlight);
      agent.setAllocation(auction, agent.getAllocation(auction) + 1);
      auction = agent.getAuctionFor(TACAgent.CAT_FLIGHT, TACAgent.TYPE_OUTFLIGHT, outFlight);
      agent.setAllocation(auction, agent.getAllocation(auction) + 1);

      // if the hotel value is greater than 70 we will select the
      // expensive hotel (type = 1)
      if (hotel > 70) {
        type = TACAgent.TYPE_GOOD_HOTEL;
      } else {
        type = TACAgent.TYPE_CHEAP_HOTEL;
      }
      // allocate a hotel night for each day that the agent stays
      for (int d = inFlight; d < outFlight; d++) {
        auction = agent.getAuctionFor(TACAgent.CAT_HOTEL, type, d);
        log.finer("Adding hotel for day: " + d + " on " + auction);
        agent.setAllocation(auction, agent.getAllocation(auction) + 1);
      }

      //calculate the client's ordered preferences
      clientEntPrefs.add(getClientEntPrefs(i));
      //allocate them their first choice - from what we own if possible
      bestEntDay(inFlight, outFlight, i, 0);

    }

    //loop through all of the clients, allocating them their second and third preferences if possible
    for (int pref = 1; pref <= 2; pref++) {
      for (int client = 0; client < 8; client++) {
        bestEntDay(agent.getClientPreference(client, TACAgent.ARRIVAL), agent.getClientPreference(client, TACAgent.DEPARTURE), client, pref);
      }
    }
  }

  private void bestEntDay(int inFlight, int outFlight, int client, int pref) {
    //retrieve the type of entertainment we're looking for
```

```
    int type = clientEntPrefs.get(client)[pref];
    for (int i = inFlight; i < outFlight; i++) {
      //skip this date if the client already has allocated
          entertainment
      if (0-i == clientEntPrefs.get(client)[0] || 0-i ==
          clientEntPrefs.get(client)[1]) {
        continue;
      }
      int auction = agent.getAuctionFor(TACAgent.
          CAT_ENTERTAINMENT, type, i);
      if (agent.getAllocation(auction) < agent.getOwn(auction
          )) {
        log.finer("Adding entertainment " + type + " on " +
            auction);
        agent.setAllocation(auction, agent.getAllocation(
            auction) + 1);
        //double up on the prefs to store which days are
            already allocated
        clientEntPrefs.get(client)[pref] = -i;
        return;
      }
    }

    // If none left and needy, just take the first...
    if (pref == 0) {
      int auction = agent.getAuctionFor(TACAgent.
          CAT_ENTERTAINMENT, type, inFlight);
      agent.setAllocation(auction, agent.getAllocation(
          auction) + 1);
      clientEntPrefs.get(client)[0] = -inFlight;
      return;
    }
  }

  // return a short ordered list for the order of client
      entertainment type preferences
  private int[] getClientEntPrefs(int client) {
    int e1 = agent.getClientPreference(client, TACAgent.E1);
    int e2 = agent.getClientPreference(client, TACAgent.E2);
    int e3 = agent.getClientPreference(client, TACAgent.E3);

    int orderedPrefs[] = {0,0,0};

    orderedPrefs[0] = (e1 > e2 && e1 > e3)? TACAgent.
        TYPE_ALLIGATOR_WRESTLING : (e2 > e3)? TACAgent.
        TYPE_AMUSEMENT : TACAgent.TYPE_MUSEUM;
    orderedPrefs[1] = (e1 < Math.max(e1, Math.max(e2, e3)) &&
         e1 > Math.min(e1, Math.min(e2, e3)))? TACAgent.
        TYPE_ALLIGATOR_WRESTLING : (e2 < Math.max(e1, Math.
```

```
        max(e2, e3)) && e2 > Math.min(e1, Math.min(e2, e3)))?
         TACAgent.TYPE_AMUSEMENT : TACAgent.TYPE_MUSEUM;
    orderedPrefs[2] = (e1 < e2 && e1 < e3)? TACAgent.
        TYPE_ALLIGATOR_WRESTLING : (e2 < e3)? TACAgent.
        TYPE_AMUSEMENT : TACAgent.TYPE_MUSEUM;
    log.fine("client " + client + ": " + orderedPrefs[0] + "
        " + orderedPrefs[1] + " " + orderedPrefs[2]);
    return orderedPrefs;
  }



  //
     -------------------------------------------------------------------

  // Only for backward compability
  //
     -------------------------------------------------------------------


  public static void main (String[] args) {
    TACAgent.main(args);
  }

} // DummyAgent
```