A constraint programming agent for automated trading

Erik Aurell, Magnus Boman, Mats Carlsson, Joakim Eriksson, Niclas Finne, Sverker Janson, Per Kreuger & Lars Rasmusson
SICS Swedish Institute of Computer Science AB
P.O. Box 1263, SE-16429 Kista, Sweden

8th February 2002

Abstract

The Trading Agent Competition (TAC) combines a fairly realistic model of the Internet commerce of the future, including shopbots and pricebots, with a challenging problem in automated reasoning and decision making. Automated trading via auctions under severe time constraints are to be conducted by entering autonomous agents into TAC, assuming the role of travel agents. The TAC game rules, as well as a description of the discrete optimization problem faced by an agent that wishes to allocate goods to its clients, are described. The TAC'01 entry "006", encapsulating a constraint programming solution, is explained in some detail.

1 Introduction

SICS, Swedish Institute of Computer Science AB, entered an agent named "006" in TAC'01, the 2001 Trading Agent Competition, the second in a series of open-invitation events organized by the University of Michigan. The main goal of TAC is to stimulate research into, and design of, shopbots: agents as representatives of clients seeking to acquire goods in competitive Internet auctions. An overview of the first event (TAC'00) and the general philosophy and motivation has been published in [32]. There has also been published a description of the agents that reached the finals [12], as well as detailed presentations of the best performing agents [29, 11], and some of the others [10].

A presentation of the second event, which was considerably more challenging, can be found on the TAC home page [31], with presentations by many of the agents that reached the semi-finals. Several more detailed descriptions are presently being prepared, and at least one is widely circulated [35], although apparently not yet available on the web at this time. Both the first and the second event have generated significant attention in the general press [5, 20, 13, 7, 16, 30, 6, 22].

The TAC scenario is on the one hand a prototype for problems that will come up as e-commerce advances into more complex domains: an agent represents eight clients, with characteristics known at run-time, and has been assigned the task of delivering to each of them a travel package. This is a bundle of goods consisting of an in-flight (mandatory), and out-flight (mandatory), rooms in one hotel during the nights on location (mandatory) and entertainment tickets (optional).

The agent receives from each client a lump sum for a delivered travel package, and bonuses for the choice of hotel and entertainment tickets. All the goods to be bundled have to be acquired on 28 separate on-line auctions, one for each good, and all running on the Michigan Internet AuctionBot server [36]. One run of this game, with eight agents each representing eight clients, is known as a TAC *instance*. The actual competitions, both TAC'00 and TAC'01, have taken place over many round-robin TAC instances, first in large number of qualifying and seeding rounds, and then in a smaller number of semi-final and final rounds.

On the other hand, the TAC scenario can be seen as a general procedure of automatically distributing goods to clients that hold non-additive utilities. As such it can be contrasted with other procedures in several ways. One is privacy: a TAC client transmits its preferences to its agent, but the agent should transmit them no further, and only interacts with the auction server by placing bids. The agent therefore reveals relatively little of the aggregated demand curve of its clients to the server and the surrounding world. Another is economic efficiency: if the clients transmit their preferences to a server, the server could then instead compute the optimum. This would solve the resource allocation problem with one large combinatorial auction. The outcome of a TAC instance is seldom optimal. Computing the optimal allocation is a computationally difficult problem [25, 27, 1, 28], and it is an open question how well such an approach would scale up in practice. It also requires that the clients reveal more of their demand curve to the server, and therefore goes against the privacy dimension. For a discussion, see [2].

Another aspect of TAC is that of distributed computation. The intelligence in TAC resides in the agents. Relative to their clients they have to solve a *combinatorial optimization problem*, where goods acquired by the agent are packaged into bundles, and delivered as such to the clients. In fact, the problem facing the agent in this respect is completely equivalent to that of an auctioneer determining winning

bids in a combinatorial auction. A successful TAC agent must therefore make use of similar integer programming techniques, as in [29, 11], or, alternatively, as here, of techniques of constraint programming.

Relative to other agents, the TAC agents instead have a *strategy problem*. To determine which strategy to use in a competitive situation is not easy [23]. If a game is played out many times among many players using deterministic strategies, and if the weaker players are weeded out, then a collection of players should evolve to a Nash equilibrium according to the "mass-action interpretation" of the Nash theory [21, 17]. There can however be many different equilibria, and which strategy to choose hence depends on which other strategies are present. In TAC the number of players is not large, and as long the agents do not use generalized strategies, a Nash equilibrium is not assured; furthermore, the weaker players are not weeded out, so the mass action interpretation does not apply. This essentially means that there is no known way to compute *a priori* the best course of action.

Agent methodology presents mechanisms for encapsulation of preferences. The encoding of information as server-readable data, which can then serve as input to an optimization algorithm, is an interesting feature not only of TAC, but of any agent approach to automated trading. While the determinism of the basic matching service cannot always be allowed to be disturbed by such novel approaches, electronic trade is steadily increasing. Definitions of agents also present Quality-of-Service in exchange systems in general, through the formulation and evaluation of complex orders.

There is an intellectual tradition of computer experiments in competitive strategy games, and in these experiments simple and robust strategies often do quite well. In the first Prisoner's Dilemma competition, the best strategy was the very simple Tit-for-Tat [3]. In the Santa Fe computer double auction tournament, the best was the also very simple "Steal-the-Deal" [26]. In retrospect, TAC'01 offers a further example. The winning strategy was "livingagents", which was perhaps the simplest to reach the finals, and the strategy of which could actually be deduced from the bids it made. 006 used a more complicated strategy, and did much worse.

The paper is organized as follows: in section 2 we describe the idea of the strategy used by 006, and in sections 3 and 4 we describe the constraint programming solver and the overall architecture of the agent as implemented. Section 5 tells the story of how 006 did up to the finals, and why its strategy was relatively unsuccessful. Section 6 sums up the presentation, and discusses the choices made and the lessons learned for the future. For completeness we summarize the most important information on the rules of TAC'01 in appendix A.

2 006 – the strategy

2.1 The execution loop

The main strategy of the agent was to only make repeated locally optimal adjustments to the goods it currently held. This was later adjusted for flights, as the agent ended up with unused flights. The strategy was as follows:

- **Collect price quotes**
  Get updated price quotes from the auctions.
- **Compute Marginal Costs**
  For each type of goods, calculate a marginal cost vector $\mathbf{p} = (p_0, ..., p_8)$ where p_i is the cost of changing from the current amount to i. If we already have i units of the goods, $p_i = 0$. If we need to sell goods to get i units, then the cost is negative, otherwise it is positive. When the goods cannot be bought (perhaps because the auction has closed), the cost is set to infinity, and when it cannot be sold, it is set to zero. Since the agent does not know the true marginal costs p_i, the costs are somewhat naively estimated from a mix of historical data (mean) and current prices (see below).
- **Compute Allocation**
  The constraint solver, described in section 3, computes the optimal amounts of goods to have, given that the goods can be acquired at the marginal costs above. The solver is an anytime algorithm that produces increasingly better allocations. We run the solver for about 10 seconds, which works reasonably well.
- **Bid**
  The bids are computed from what we already hold, what is currently bid for, and what is required from the allocation step above. The maximal price to bid for the resources is the amount that was specified in the marginal cost vector. For event auctions we increase the bid toward the maximal price exponentially, but for hotel auctions we bid the maximal price immediately. This is because hotel quotes only are updated once per minute, and one hotel auction closes every minute.

  The flight auctions are handled differently as they pose a kind of real option problem, i.e. act now or act later. Several different heuristics were tried. The one that was used in the competition tried to avoid buying flights that would be unused by finding the flights that were most likely to be needed. This was done by calling the solver several times with both high and low

price estimates. Flights that were always allocated were bought early on. The rest of the flights were bought as normal goods, but only after five minutes of play.

- **Iterate**
  When the bids are placed, monitor the auctions to see if we are allocated resources, bids become invalid, or if an auctions closes. If so, return to the first step. When the game has ended, terminate the loop.

2.2 Comments

We did not realize at first that the price quotes from the hotel auctions were not updated continuously. We therefore initially believed that hotel prices would go up, and that one would be able to track these prices closely. If an optimal allocation did not use all hotel rooms for which the agent had valid bids, we assumed that the agent could simply stop raising its bid on these rooms, and that they would then drop out of these auctions after a while. Although hotel rooms can not be sold and bids for hotel rooms not withdrawn, we hence assumed that superfluous hotel rooms was not going to be a problem.

Since flight tickets can not be resold, we wanted to test various variants to the above, as concerns these. We tried

- treat flight tickets like any other good, i.e. buy them whenever the solver says so. This generally led to the agent buying many flight tickets in the beginning, and then buying more at a later stage, when the solver had decided another allocation pattern was better.
- buy flight tickets according to a delayed tactic; generally only at specified times, and then only a fraction of what the solver told us. This often led us to buying flight tickets somewhat late in the instances, and paying comparatively high prices.

In the final stages of the competition we used a variant of the delayed tactic, as described above.

As noted above, our estimates of expected prices were not very sophisticated. We used the following:

- the expected prices of entertainment tickets were the latest available prices, separately for bid and ask.

- the expected price of a flight ticket was the expected price at the time of purchase. If we were using a strategy that called for purchasing the ticket at a time T in the future, we used the expected price at T, given all that was known up to the current time t.

- in the beginning of an instance the expected prices of hotels were the average prices of these hotels in previous rounds. As time (t) progressed we monitored the actual price P_t in the current instance. We assumed that this price would grow linearly until that auction closed, which provided an extrapolated expected price T_c/tP_t, if T_c was the expected closing time. We compared this estimate, from the current instance, with the average from previous rounds, and used the largest of the two as the expected prices of the hotels.

  It is an easy observation from the data, that if the auction of rooms in one hotel on one day closes, the prices of the rooms in the other hotel on the same day tend to rise. Hence, as soon as one hotel had closed on some day d, the average price of the other hotel on that day was taken to be the conditional average in previous rounds when the hotels closed in the same order. No attempt was made to extract more information on hotel room prices from the order of closing of the auctions.

3 The Solver

The task of the solver module is to compose a trip package for the clients so that the net agent utility is maximized. That utility is the difference between the total client benefit and the total cost. The cost and benefit functions are given to the solver module extensionally (as tables). We use a constraint programming over finite domains (CLP(FD)) [34] approach to this optimization problem.

3.1 Variables

All variables X range over an interval r of integers, denoted $X \in r$.

3.1.1 Problem Variables

One of each variable per client c.

- $T_c \in 0..1$ 1 denotes a trip was allocated,

- $A_c \in 0..4$ arrival day (0 denotes no trip),
- $D_c \in 1..5$ departure day (1 denotes no trip),
- $H_c \in 0..1$ 1 denotes good hotel,
- $E_{d,c} \in 0..3$ event on day $d \in 1..4$ (0 denotes no event).

3.2 Cumulated variables

These variables are simply functions of the problem variables, and range over $0..8$. $d \in 1..4$ denotes a day.

- $\bar{A}_d$: number of arrivals on day d,
- $\bar{D}_d$: number of departures on day $d+1$,
- $\bar{C}_d$: number of cheap hotel rooms on day d,
- $\bar{G}_d$: number of good hotel rooms on day d,
- $\bar{E}_{e,d}$: number of event e tickets on day d.

3.3 Objective function

Let $\beta : c \times x \rightarrow \mathcal{Z}$ denote the *benefit* of commodity x for client c, and let $\pi : d \times n \times x \rightarrow \mathcal{Z}$ denote the *price* of n items of commodity x on day d. These functions are given as input data. The objective is to maximize the following function:

$$\begin{array}{l} \sum_{c \in 1..8}(1000 \cdot T_c + \beta(A_c) + \beta(D_c)) \\ + \sum_{c \in 1..8}(\beta(E_{1,c}) + \beta(E_{2,c}) + \beta(E_{3,c}) + \beta(E_{4,c})) \\ - \sum_{d \in 1..4}(\pi(\bar{A}_d) + \pi(\bar{D}_d) + \pi(\bar{C}_d) + \pi(\bar{G}_d)) \\ - \sum_{d \in 1..4}(\pi(\bar{E}_{1,d}) + \pi(\bar{E}_{2,d}) + \pi(\bar{E}_{3,d})) \end{array} \tag{3.1}$$

3.4 Constraints

In this section, $[E = y]$ denotes 1 if the equation $E = y$ holds, and 0 otherwise.

Link constraints for tickets. These constraints link problem variables for flights and events to cumulated variables. They are encoded as six *global cardinality* constraints [24], but are given below in a more basic form.

$$\forall d \in 1..4 : \bar{A}_d = \sum_{c \in 1..8} [A_c = d] \tag{3.2}$$

$$\forall d \in 1..4 : \bar{D}_d = \sum_{c \in 1..8} [D_c = d + 1] \tag{3.3}$$

$$\forall d \in 1..4 \forall e \in 1..3 : \bar{E}_{e,d} = \sum_{c \in 1..8} [E_{d,c} = e] \tag{3.4}$$

Link constraints for hotel rooms. These constraints link problem variables for hotel rooms to cumulated variables. They are encoded as a single *cumulatives* constraint [4], but are given below in a more basic form. For this purpose, we first need to introduce new cumulated variables:

- $\bar{A}^c_d$: number of arrivals on day d for clients staying in the cheap hotel,
- $\bar{A}^g_d$: number of arrivals on day d for clients staying in the good hotel,
- $\bar{D}^c_d$: number of departures on day d for clients staying in the cheap hotel,
- $\bar{D}^g_d$: number of departures on day d for clients staying in the good hotel.

$$\forall d \in 1..4 : \bar{A}^c_d = \sum_{c \in 1..8} [A_c = d] \cdot (1 - H_c) \tag{3.5}$$

$$\forall d \in 1..4 : \bar{A}^g_d = \sum_{c \in 1..8} [A_c = d] \cdot H_c \tag{3.6}$$

$$\forall d \in 1..4 : \bar{D}^c_d = \sum_{c \in 1..8} [D_c = d + 1] \cdot (1 - H_c) \tag{3.7}$$

$$\forall d \in 1..4 : \bar{D}^g_d = \sum_{c \in 1..8} [D_c = d + 1] \cdot H_c \tag{3.8}$$

$$\bar{C}_1 = \bar{A}^c_1 \tag{3.9}$$

$$\bar{C}_2 = \bar{A}^c_1 + \bar{A}^c_2 - \bar{D}^c_1 \tag{3.10}$$

$$\bar{C}_3 = \bar{D}^c_3 + \bar{D}^c_4 - \bar{A}^c_4 \tag{3.11}$$

$$\bar{C}_4 = \bar{D}^c_4 \tag{3.12}$$

$$\bar{G}_1 = \bar{A}^g_1 \tag{3.13}$$

$$\bar{G}_2 = \bar{A}^g_1 + \bar{A}^g_2 - \bar{D}^g_1 \tag{3.14}$$

$$\bar{G}_3 = \bar{D}^g_3 + \bar{D}^g_4 - \bar{A}^g_4 \tag{3.15}$$

$$\bar{G}_4 = \bar{D}^g_4 \tag{3.16}$$

Arrival precedes departure.

$$\forall c \in 1..8 : A_c < D_c \tag{3.17}$$

Trip constraints. If no trip was allocated for the client, then $T_c = 0, A_c = 0, D_c = 1, E_{1,c} = E_{2,c} = E_{3,c} = E_{4,c} = 0$, otherwise $T_c = 1, A_c > 0, D_c > 1$.

$$\forall c \in 1..8 : T_c = 0 \leftrightarrow A_c = 0 \leftrightarrow D_c = 1 \tag{3.18}$$

$$\forall c \in 1..8 : T_c = 0 \rightarrow H_c = E_{1,c} = E_{2,c} = E_{3,c} = E_{4,c} = 0 \tag{3.19}$$

Event constraints. A client can visit an event once only, and only between the arrival and departure dates.

$$\forall c \in 1..8 \forall e \in 1..3 : [E_{1,c} = e] + [E_{2,c} = e] + [E_{3,c} = e] + [E_{4,c} = e] \leq 1 \tag{3.20}$$

$$\forall c \in 1..8 \forall d \in 1..4 : E_{d,c} > 0 \rightarrow A_c \leq d \wedge D_c > d \tag{3.21}$$

3.5 Search procedure

CLP(FD) is an incomplete constraint solver. Having declared all variables and posted all constraints, the constraint solver may not be able to detect global infeasibility, and we must resort to search, usually some form of backtrack search. This is true also for the solver module's search procedure, the details of which are given below.

Search for optimality. For TAC, the search procedure should have the property that it quickly finds a first solution, and then keeps finding progressively better and better solutions. In a CLP(FD) setting, this is usually done with a variant of branch-and-bound search: Given a solution that is best so far, the search is limited to better solutions by bounding the benefit function (Eqn. 3.1) below. For each new solution, the search is restarted with a tighter lower bound.

Avoiding useless work. When the procedure finds a solution, it also records a trace [15] of the choices made on the way from the root of the search tree to the solution. The idea is that bad choices, i.e. ones that did not lead to a solution, should not be remade. When the search is restarted, this information is used to avoid remaking bad choices.

Escaping from the trap of depth-first search. The main advantage of depth-first search is its great memory efficiency: the search tree can be explored in space proportional to a single branch. The main drawback is its sensitivity to search order: the first choice made is truly critical, for it will not be undone until the rest of the search space is exhausted. A popular way of mitigating this drawback is Limited Discrepancy Search (LDS, [14]), which is simply depth-first search with an upper bound to the number of backtracks on any path from the root. The solver module uses LDS with an iteratively incremented bound.

Variable order. Let the *rank* of a client denote the client's maximal utility. The problem variables are assigned in the order: (i) arrival and departure dates; (ii) hotels rooms; and (iii) event tickets. Within each of these three sets, the variables are ordered by descending rank of the respective client.

Value order. Let the *estimated value* $t(X = v)$ of an assignment $X = v$ be computed as the average of the lower and upper bounds of the benefit function

(Eqn. 3.1) under the assumption $X = v$. In a CLP(FD) setting, this is straightforward to compute.

When a hotel room or event ticket variable X is going to be assigned, the solver module tries the values v in the domain of X by descending $t(X = v)$. When arrival and departure dates (A, D) are going to be assigned, the solver computes the estimated value of each *combination* of values for A and D, and tries the value tuples (a, d) by descending $t(A = a, D = d)$.

4 The Agent as Implemented

The 006 agent that finally played in the 2001 Trading Agent Competition was written somewhat quickly during the seeding rounds. Parts of the design was however made earlier, in the work on the agent that participated in the qualification rounds. 006 was developed in SICStus Prolog [8], including the solver (TAC Optimizer), described in section 3.

4.1 The 006 Architecture

Figure 4.1 shows the architecture of 006. This section will describe the implementation of all parts except the solver.

4.1.1 Communication and Scheduling

This component handles the communication with the TAC server and the solver. It also handles scheduling of other components that need to be active during TAC games (auction handlers, etc). This is necessary because Prolog is single threaded.

4.1.2 Game Handler

This component is responsible for handling a specific game and is not initialized until a game is started and the game's client preferences have been received by the agent. It setups the auction handlers for the game and handles the communication between the auction handlers and the solver. The game handler also

- makes sure the solver is not started before quotes have been received from all auctions. This ensures that the agent does not place any bids before the TAC server has started its auctions.

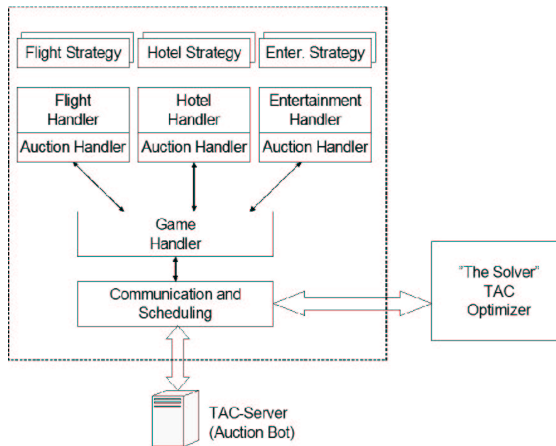

Figure 4.1: The 006 agent architecture

- handles the retrieval of transactions and reports them to the auction handlers.
- restarts the solver to calculate a new client allocation whenever an allocation has failed (or probably will fail).
- determines when a game has ended and handles the final score calculation using the solver.

4.1.3 Auction Handler

This component is the super object of the flight, hotel, and entertainment handlers, and contains all common code such as handling quote information, allocation, bids, etc.

4.1.4 Flight, Hotel and Entertainment Handlers

Each instance of *Auction Handler* is specific to the type of auction and the type of item sold at the auction. They will convert information between the internal format and the solver format when interaction with the solver take place.

All handlers have a default bidding strategy implemented, but most important is that they enable plug-in of strategy handlers making it easy to experiment with different strategies.

4.1.5 Strategy Components

Each strategy component implements a strategy for one type of auction and must implement the following predicates (methods):

`calculate_solver_prices_impl` - calculates the (expected) prices that is to be sent to the solver for a new optimization.

`calculate_new_bid_impl` - calculates a new bid based on desired number of items, number of owned, current price. This happens when the solver gives the agent "new orders" to follow (and there has not been any previous bids on this item).

`calculate_replacement_bid_impl` - calculates a bid to replace an old bid based on desired number of items, number of owned, current price, and the last bid placed. This happens when the solver gives the agent "new orders" and there has been a previous bid.

`update_bid_list_impl` - update the current bid or signal that it is time to send new price expectations to the solver. This happens when the current bid is no longer valid (because of price increase, etc).

5 Results in the TAC'01 competition

SICS entered an agent in the preliminary rounds of TAC'01 that was incomplete, and did quite badly. The organizers of TAC'01 nevertheless generously enough invited us to partake in the seeding rounds. After a walk-through of the existing code, it was decided to redo the implementation from scratch. Work began on September 21, development continuing through the seeding rounds than began on September 24. This agent eventually finished overall as number 17 out of 19, but did relatively better toward the end of the seeding rounds.

For the semifinal heats, 16 agents were divided in quarters, with respectively the top and bottom quarters fighting for half of the seats in the finals, and the two middle quarters for the other half. 006 placed fifth in its own heat, and did therefore not advance to the finals. The semifinal results are summarized in Table 5.

In points, 006 beat the next agent in its own heat by 1500, and the first agent in the other heat that did not advance by 350. As discussed in the caption to Table 5, differences by about 300 in the semifinals are not likely to be statistically significant, and the scores in the "top and bottom" heat also tended to be a little higher than those in the "middle" heat. All that can be said is that if 006 would have been ready earlier, and would have qualified to the middle heat from the seeding rounds, it would have had a better chance of reaching the finals.

The agent 006 was reasonably stable at the end of the seeding rounds. Its results can therefore be taken to be as much due to the defects of the strategy, as to remaining implementation problems. In retrospect, 006 as described above in in section 2, was partially built on a misunderstanding of the competition rules. Only when starting the second implementation did we realize that hotel price quotes were not up to date, and not very indicative for what one should have to pay to get the room. To do reasonably well we had to add heuristic corrections. These were essentially of two kinds: i) estimates of expected prices, based on history and how the current instance was played out; and ii) fudge factors to compensate that the price quotes for hotels were old, and usually too low. The first is described above, and could have been made more systematic, if we would have had more time. For the second we never had time to consider seriously what should have been the best choice, but just settled for an overbid factor, usually a fraction of the averaged final price from earlier rounds, so that we seldom dropped out of these auctions.

6 Discussion

The Trading Agent Competition is a challenging market game. It is interesting for purely practical and commercial reasons, as a prototype for e-commerce of the future. Possible application areas include automatic decision making and allocation of more fine-grained commodities and on shorter time-scales than would be possible with human traders. Examples are markets in bandwidth, electric power consumption, and other communication services prone to overloading and congestion. Services such as TAC agents may also be of potential interest to human end-users, as shopbots looking for, e.g. travel packages for business travelers.

Results in TAC'01 semifinal heats

Agent	Heat	Score	Std Dev	Min,Max
livingagents	**1**	3660.2	893.8	(2057.54, 4982.63)
SouthamptonTAC	**1**	3614.5	747.3	(2373.81, 4658.15)
Urlaub01	**1**	3484.8	924.1	(1864.88, 4874.64)
whitebear	**1**	3469.7	1043.0	(1579.2, 4645.88)
Retsina	**2**	3293.5	630.9	(2306.17, 4116.57)
ATTac	**2**	3249.2	407.9	(2535.69, 3941.2)
006	**1**	3240.8	1108.1	(962.09, 4196.88)
CaiserSose	**2**	3038.1	640.9	(1898.67, 3806.1)
TacsMan	**2**	2966.1	595.2	(2331.27, 4236.4)
PainInNEC	**2**	2905.9	540.6	(2142.72, 3585.82)
polimi_bot	**2**	2834.7	1102.1	(0, 3796.55)
umbctac	**2**	2772.9	813.5	(844.83, 3880.15)
RoxyBot	**2**	2112.4	1478.7	(-1778.29, 3977.33)
arc−2k	**1**	1746.3	1948.7	(-1038.31, 4153.12)
jboadw	**1**	1716.7	1281.3	(0, 4005.92)
harami	**1**	94.4	2537.2	(-6534.55, 3380.82)

Table 1: The results of the semifinal heats of TAC'01. Affiliations of the agents and team members can be found on the TAC home page [31]. Heat 1 was composed of the agents placing in the top and the bottom quarter in the seeding rounds, while heat 2 contained the agents from the middle half. The top four agents in each heat advanced to the finals. The score was computed as the average result over 11 instances, and is an unbiased estimate of how the agent would play in many iterations, always restarting from scratch after 11 instances, with estimated error on the order of the agent's standard deviation, divided by $\sqrt{10}$ (3.16). Differences in score by less than about 300 points in the same heat should therefore not be taken to be statistically significant.

Results in TAC'01 finals

Agent	Score	Std Dev	{ Min,Max }	High/Low
livingagents	3670.0	622.3	{ 2331.99, 4631.84}	**5, 1**
ATTac	3621.6	691.6	{ 2079.15, 4877.1}	**1, 3**
whitebear	3513.2	700.1	{ 1563.85, 4695.49}	**2, 5**
Urlaub01	3421.2	698.3	{ 1923.09, 4640.21}	**4, 4**
Retsina	3351.8	668.2	{ 2150.35, 4643.28}	**3, 2**
SouthamptonTAC	3253.5	1466.9	{-3119, 4485.62}	**6, 8**
CaiserSose	3074.1	656.2	{ 1696.46, 3965.41}	**8, 6**
TacsMan	2859.3	1054.3	{ 0, 4230.26}	**7, 7**

Table 2: The results of the final heats of TAC'01. Table entries as in Table 5, except last column that ranks the agents by, respectively, their highest and lowest results in the finals. *SouthamptonTAC* had a crash in one game; if not their score would have been 3530.6, *i.e.* third place, and its standard deviation more or less as in the semifinal heats. The number of instances in the final was 24; differences by more than a standard deviation divided by about five should not be significant. By this token, agents could switch order pairwise, *e.g. livingagents* against *ATTac* but would be significantly ahead or behind all others. Exceptions are *CaiserSose* and *TacsMan*, the placing of which would be significant by this measure. It is interesting that the ranking of agents by score is also partially reproduced in a ranking by highest and lowest result, although the match is not perfect. Another robustness test was performed by the TAC'01 organizer [18], subtracting the fluctuations in "desirability" of the random preferences of the actual clients. That re-analysis essentially reproduces the order above.

The TAC set-up is also interesting in its own right. It has stimulated many groups to implement or adapt suitable anytime optimization packages, to be able to compute what a collection of goods is really worth to an agent. In TAC'00 only two teams had such packages, that actually computed the optimum, while in TAC'01 most agents had, at least among those reaching the semifinals. This problem of complementary utilities, and how to calculate them efficiently, is certainly of wider interest in computational economics. TAC has also stimulated thinking about strategies. In the first game, TAC'00, the rules were slightly different than described here, and strongly favored taking decisions as late as possible. It was therefore, that year, important to monitor network characteristics, and adapt the strategy to response times. In TAC'01 the upward trend of flight prices made waiting to the last moment a bad strategy. The uneven updating of hotel prices also meant that the quality of network connections probably only mattered, and not by much, to the event tickets auctions. The highest scoring agent, livingagents from Living Agents AG, made all its bids on flights and hotels immediately at the start of a game instance, and therefore played close to the opposite strategy. In passing, one may note that livingagents placed very high bids on the hotel rooms it wanted, and therefore always got them. The price it had to pay was, however, according to the game rules, only the 16th highest bid, generally much lower. The success of livingagents hence depended entirely on the other agents making reasonable bids, and is hence a nice illustration of the Efficient Market Hypothesis [9], in a fully deterministic setting.

The SICS agent, 006, did reasonably well. It did not crash during the semifinals, but twice did expensive re-allocations that an ocular review of the game history data suggest were unnecessary. Its results therefore probably reflect the strategy as much, or more, than the implementation. Since the basic idea of the strategy was partially based on a misunderstanding of the competition rules, one lesson is to investigate them more carefully another time.

A more far-reaching lesson is that it would have helped to have a local test environment, in which to run TAC agents against each other in a controlled manner [1]. Anecdotal evidence suggest that several successful teams did develop such platforms. After the finals we therefore decided to re-engineer a TAC server of our own, of which an alpha version is available [33]. It is our intention that this server, written in SICStus Prolog, be released as open software in the public domain. Discussions are under way with the University of Michigan team to hold at least one TAC event on this server in 2002.

In conclusion, the TAC concept has quickly become a benchmark in automatic

[1] All test, qualifying, seeding, semifinal and final games in TAC'00 and TAC'01 have been run on the TAC server at University Michigan, which is built on proprietary software (AuctionBot), and not available in source code or as executable.

market research. We look forward to participating in the next event.

Acknowledgement

This work was supported by the Swedish Board for Technical Development (NUTEK) through project B1EFIN, and by the Swedish IT Institute (SITI) through project EMARKETS.

A Summary of TAC'01 rules

For completeness, we summarize here the rules of TAC'01. This description is not complete, particularly as to how bids are expressed to the auction server, see instead [31].

A.1 Utilities

A feasible travel package consists of an in-flight, an out-flight, and rooms at one hotel during all nights between, and (optionally) tickets to entertainment events. The client utility of one feasible travel package is

$$u = 1000 - \text{travel_penalty} + \text{hotel_bonus} + \text{fun_bonus} \tag{A.1}$$

The utility to the agent of a collection of goods is the sum of the utilities to clients of feasible travel packages delivered to them.

Every client has a set of preferences, that are its preferred arrival date (PD), its preferred departure date (PD), how much it wants to pay extra for the good hotel (HP), and how much it wants to pay for each entertainment event (AW, AP and MU). The *travel_penalty* is $100 * (|AA - PA| + |AD - PD|)$, where AA and AD are its *actual* arrival and departure dates, i.e. 100 for every day away from PD and PD. The *hotel_bonus* is payed if the client gets to stay at the better hotel. It is payed per trip, irrespective of its length.

For the entertainment tickets, a *feasible entertainment package*, ancillary to a feasible travel package, is one which contains any event at most once, never two events on the same day, and no event on AD, the day of departure. The *fun_bonus* is the sum of what the client wants to the pay for the events in its feasible entertainment package.

The utility to an agent of a collection of goods is highly non-linear, since e.g. two flight tickets are not worth anything if they cannot be be combined with hotel rooms and delivered to a client as a feasible travel package. Given a collection of goods, there are a typically many ways of dividing them into packages. Computing which such combination of packages maximizes agent utility is a problem of combinatorial optimization.

A.2 Auctions

All goods are distributed in separate markets called auctions. An instance of the TAC game runs over 12 minutes. Some of the auctions run for the full time, some for less.

The auctions for flight tickets are more properly just over-the-counter markets, where a seller (the auction server) always holds an unlimited supply of tickets. There are eight flight auctions, one for every possible in-flight day (day 1-4), and one for every possible out-flight day (day 2-5). These auctions are open for the full instance and clear continuously. The price charged by the seller changes in time, somewhat like the way an airline sells seats on a given flight in tranches released at different dates. The price changes are random, but the general trend is upward.

Hotel rooms are sold on ascending multi-unit Mth price auctions, a kind of generalized Vickrey auctions. There are 16 items (hotel rooms) of each kind. The price payed for one item is the 16th highest bid. There are eight hotel auctions in every TAC game instance, one for every night (1-4) and every type of hotel (good or bad). The hotel auctions start at the beginning of the game. All hotel auctions close between 4.00 (four minutes into the game) and 11.00, according to the scheme that one, randomly chosen, closes on every minute from 4.00 to 11.00.

The event tickets initially distributed randomly among the agents. They can then be bought and sold by the agents among each other in standard continuous double auctions. There are twelve auctions of this kind, one for each day (1-4) and for every kind of entertainment. These auctions clear continuously, and are open for a whole instance.

A.3 Available information

The TAC API allows agents to query status of auctions, for e.g. last bid price, last ask, and more. For flight auctions and entertainment auctions the auction server answers with updated information, with some delay.

A curious feature of TAC'01 is that the server does not instantaneously process bids on hotel rooms. Only once a minute are the available bids collected, and the 16 highest bids selected. For the full following minute, the server will answer that the 16th highest price so calculated is the updated price quote – no account is taken of higher bids which may have arrived in the meantime, so that the price one would have to beat to get the hotel in the end could be higher than the quote. Similarly, the agents holding the 16 bids that were highest on the minute will for the full following minute receive the reply that these bids are active, although they may have been overbid by others.

In summary, it is an important feature of TAC'01 that the agents can not get information on current prices of hotel rooms, because these prices are not continuously computed.

References

[1] A. Andersson, M. Tenhunen, and F. Ygge. Integer programming for combinatorial auction winner determination. In *Proc. Fourth International Conference on Multiagent Systems (ICMAS-00)*, 2000.

[2] A. Andersson and F. Ygge. Managing large-scale computational markets. In *Proc. 31st Hawaiian International Conference on System Sciences*, volume VOL VII—Software Technology Track, pages 4–13. IEEE Computer Society, 1998.

[3] Robert Axelrod. *The Evolution of Co-operation*. Basic Books, Inc., New York, USA, 1984.

[4] Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. Technical Report T2001-11, Swedish Institute of Computer Science, 2001.

[5] Finding something smart on the web is about to get easier. Boston Globe, 2001.

[6] Doug Bryan. First-ever e-commerce trading competition pits bot against bot. Center for Strategic Technology Research, 2000.

[7] Anne Eisenberg. In online auctions of the future, it'll be bot vs. bot vs. bot. New York Times, 2000.

[8] Mats Carlsson et al. *SICStus Prolog User's Manual*. Swedish Institute of Computer Science, release 3 edition, 1995. ISBN 91-630-3648-7.

[9] Eugene F. Fama. Efficient capital markets: A review of theory and empirical work. *Journal of Finance*, 25:383–417, 1970.

[10] Nicoletta Fornara and Luca Maria Gambardella. An autonomous bidding agent for simultaneous auctions. In *Proceedings of the Fifth International Workshop on Cooperative Information Agents (CIA-2001)*, volume 2182 of *Lecture Notes on Artificial Intelligence LNAI*, pages 130–141, 2001.

[11] Amy Greenwald and Justin Boyan. Bidding algorithms for simultaneous auctions: A case study. In *Proc. of the Third ACM Conference on Electronic Commerce*, pages 115–124, 2001.

[12] Amy Greenwald and Peter Stone. Autonomous bidding agents in the trading agent competition. *IEEE Internet Computing*, 5(March-April, 2):52–60, 2001.

[13] Dave Gussow. Tampa plays host to battling bots. St. Petersburg Times, 2001.

[14] William Harvey and Matthew Ginsberg. Limited discrepancy search. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, 1995.

[15] Pascal Van Hentenryck. Incremental constraint satisfaction in logic programming. In D.H.D. Warren and P. Szeredi, editors, *ICLP'90, 7th Int. Conf. on Logic Programming*, Series in Logic Programming, pages 189–202, Jerusalem, 1990. The MIT Press.

[16] Mo Krachnal. Ladies and gentlemen, start your shopbots. TechWeb, 2000.

[17] Harold W. Kuhn, John C. Harsanyi, Reinhard Selten, Joergen W. Weibull, Eric van Damme, and Peter Hammerstein. The work of john nash in game theory. In T[] Persson, editor, *Nobel Lectures Economic Sciences 1991–95*, pages 155–172, 185–190. World Scientific, Singapore, 1994.

[18] Kevin Lochner. TAC finals analysis. Mail from M. Wellman to the tacdiscuss list, sent on 18 Nov 2001 13:38:15 -0500., 2001.

[19] Tim McDonald. Robot agents: Coming soon to software near you. Yahoo News, 2001.

[20] J Nash. *Non-cooperative games*. PhD thesis, Princeton University, 1950.

[21] Kevin O'Malley. Agents & automated online trading. Dr. Dobb's Journal, 2001.

[22] Eric Rasmussen. *Games and Information, 2nd ed.* Blackwell Publ., Cambridge, MA, USA, 1994.

[23] J.C. Régin. Generalized arc consistency for global cardinality constraint. In *Proc. 13th AAAI*, pages 209–215. The MIT Press, 1996.

[24] Michael H. Rothkopf, Aleksandar Pekec, and Ronald M. Harstad. Computationally manageable combinatorial auctions. Technical Report RRR 13-95, Rutgers University, NJ, USA, 1995.

[25] John Rust, John H. Miller, and Richard Palmer. Characterizing effective trading strategies; insights from a computerized double auction tournament. *Journal of Economic Dynamics and Control*, 18:61–96, 1994.

[26] Tuomas Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, pages 542–547, Stockholm, Sweden, 1999.

[27] Tuomas Sandholm. CABOB: A fast optimal algorithm for combinatorial auctions. In *Proc. International Joint Conference on Artificial Intelligence (IJCAI)*, Seattle, WA, 2001.

[28] Peter Stone, Michael L. Littman, Satinder Singh, and Michael Kearns. Attac-2000: An adaptive autonomous bidding agent. *Journal of AI Research (JAIR)*, 15:?, 2001.

[29] Betsy Strother. The trading agent competition. SIGecom Exchanges, 2000.

[30] EC-01 trading agent competition. `http://auction2.eecs.umich.edu/`, 2001.

[31] TAC Team, Michael P. Wellman, Peter R. Wurman, Kevin O'Malley, Roshan Bangera, Shou de Lin, Daniel Reeves, and William E. Walsh. Designing the market game for a trading agent competition. *IEEE Internet Computing*, 5(March-April, 2):43–51, 2001.

[32] TAC 2002. `http://www.sics.se/~market/tac/`, 2002.

[33] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[34] Michael P. Wellman, Amy Greenwald, Peter Stone, and Peter R. Wurman. The 2001 trading agent competition. Unpublished manuscript.