Bidding Algorithms for Simultaneous Auctions: A Case Study

Amy Greenwald
Department of Computer Science
Brown University, Box 1910
Providence, RI 02912
amy@brown.edu

Justin Boyan
ITA Software
Cambridge, MA 02139
jab@itasoftware.com

January 10, 2002

Abstract

This paper describes RoxyBot, one of the top-scoring agents in the First International Trading Agent Competition, TAC-2000. A TAC agent simulates one vision of future travel agents: it represents a set of clients in simultaneous auctions, trading complementary (*e.g.*, airline tickets and hotel reservations) and substitutable (*e.g.*, symphony and theater tickets) goods. RoxyBot faced two key technical challenges in TAC: (i) *allocation*—assigning purchased goods to clients at the end of a game instance so as to maximize total client utility, and (ii) *completion*—determining the optimal quantity of each resource to buy and sell given client preferences, current holdings, and market prices. For the dimensions of TAC, an optimal solution to the allocation problem is tractable, and RoxyBot uses a search algorithm based on A^* to produce optimal allocations. An optimal solution to the completion problem is also tractable, but in the interest of minimizing bidding cycle time, RoxyBot solves the completion problem using beam search with a greedy heuristic, producing approximately optimal completions. RoxyBot's completer relies on an innovative data structure called a *priceline*.

1 Introduction

The first international Trading Agent Competition (TAC-2000) challenged its entrants to design an automated trading agent capable of bidding in simultaneous on-line auctions for complementary and substitutable goods [15]. A TAC-2000 agent is a simulated travel agent whose task is to organize itineraries for a group of clients who wish to travel from TACTown to Boston and back again during a five-day period in July.[1] Travel goods, such as airline tickets and hotel reservations, are complementary, and tickets to entertainment events, such as the Boston Red Sox and the Boston Symphony Orchestra, are substitutable. The trading agent's objective is to win items that best satisfy its clients' preferences as inexpensively as possible.

In this paper, we introduce `RoxyBot`, one of the top-scoring TAC-2000 agents. The name `RoxyBot` is short for "ApproximateBot," which suggests our goal of constructing a trading agent whose bidding decisions approximate optimal behavior. `RoxyBot` faced two key technical challenges in TAC-2000: (i) *allocation*—assigning purchased goods to clients at the end of a game instance so as to maximize total client utility, and (ii) *completion*—determining the optimal quantity of each resource to buy and sell given client preferences, current holdings, and market prices. The allocation problem is a form of the winner determination problem, in which an auctioneer seeks to allocate goods to bidders with combinatorial valuations so as to maximize his revenue, and completion can be reduced to winner determination with reserve prices.

For the dimensions of TAC-2000, an optimal solution to the allocation problem is tractable, and `RoxyBot` uses a search algorithm based on A^* to produce optimal allocations. One of the primary foci of this paper is the design `RoxyBot`'s admissible heuristics. An optimal solution to the completion problem is also tractable, but was observed to run for as long as 10 seconds on difficult problem instances. In the interest of minimizing bidding cycle time, `RoxyBot` solves the completion problem using beam search with a greedy heuristic, producing approximately optimal completions. The completion algorithm relies on an innovative data structure called a *priceline*, that reduces completion to *acquisition*—the problem of determining the optimal quantity of each resource to buy, *not sell*, given client preferences, current holdings, and market prices.

This paper is organized as follows. In the following section, we motivate our approach to the design of bidding agent algorithms, and describe `RoxyBot`'s high-level architecture. In Section 3, we describe the TAC-2000 market game, and present examples of allocation and completion. Section 4 presents our allocation algorithm. This algorithm is based on A^* search, and this discussion is therefore dedicated to the description of our admissible heuristics. Section 5 describes our approach to completion. Special emphasis is placed on the *priceline*, a novel data structure which transparently handles either one-sided or double-sided auctions, short-selling of resources, hedging, and both limited and unlimited supply and demand. In Section 6, we describe estimation techniques for building

[1] The TAC-2000 workshop was held at ICMAS '00 in Boston in July, 2000.

pricelines. In Section 7, we present the results of the competition. Lastly, in Section 8, we discuss the general applicability of this work.

2 `RoxyBot`'s Architecture

The primary challenge in TAC-2000 is to bidding agents is to determine how to bid on complementary and substitutable goods that are sold in simultaneous, not combinatorial, auctions. Complementary goods are goods with superadditive utilities: *i.e.*, $u(A\overline{B})+u(\overline{A}B) \leq u(AB)$. In TAC-2000, the utility of airline tickets without hotel reservations (or of hotel reservations without airline tickets) is zero, whereas the utility of complete travel packages is strictly positive. Substitutable goods are goods with subadditive utilities: $u(A\overline{B}) + u(\overline{A}B) \geq u(AB)$. In TAC-2000, the utility of both a theater ticket and a symphony ticket for the same night is bounded above by the higher of the individual utilities attributed to the two separate events. It does not make sense to assign individual utilities to complementary goods (which can be worthless in isolation) or substitutable goods (which can be worthwhile only in isolation). Thus, simple bidding strategies such as "for each good x, bid up to its utility" are not applicable in the TAC-2000 setup.

Instead, `RoxyBot` was built to reason directly about *sets* of goods—the utilities of which are well-defined. `RoxyBot` poses and solves questions such as:[2]

- "Given only the set of goods I already hold, what is the maximum utility I can attain?" Inspired by TAC-2000, we call this problem *allocation*, since in the case of an agent bidding on behalf of multiple clients, the solution is an optimal allocation of goods to clients.
- "Given the set of goods I already hold, and given market prices, supply, and *demand*, on what set of additional goods should I place bids or *asks* so as to maximize my utility plus profits minus costs?" This yet more general problem, *completion*, provides a foundation for bidding strategies in settings with simultaneous single-sided and double auctions.

Given solutions to allocation and completion, a natural architecture for a TAC-2000 agent is to repeatedly compute estimates of market clearing prices, run a completion algorithm to determine target holdings, and bid/ask accordingly. This architecture, which is outlined in Table 1, was employed by `RoxyBot`.

After describing the TAC-2000 market game in the next section, the remainder of this paper focuses on `RoxyBot`'s allocation and completion algorithms and our estimation procedures, all of which are based on heuristic AI search techniques. The strategic timing of bid/ask placement of `RoxyBot` and other TAC-2000 agents is described in [8].

[2] Allocation is a form of winner determination in multi-unit combinatorial auctions, and completion reduces to winner-determination with reserve prices [2]. Therefore, these problems are NP-hard [10].

(A) While some auction remains open, do:

1. **Update** current prices and holdings for each auction.
2. **Estimate** clearing prices, supply, and demand of each good; store information in a **priceline**.
3. Run **completion** to determine the quantity of each good that is ultimately desired; compute the difference between the optimal solution and current holdings.
4. Place **bids** and **asks** strategically (with respect to the current time and the auction mechanisms) to buy and sell goods to reach the desired quantities.

(B) After all auctions have closed, solve the **allocation** problem.

Table 1: `RoxyBot`'s high-level architecture.

3 TAC-2000 Market Game

A TAC-2000 agent is a simulated travel agent whose task is to organize itineraries for a group of clients who wish to travel from TACTown to Boston and back during a five day period. Travel and entertainment goods are traded at simultaneous on-line auctions that run for fifteen minutes. An agent's objective is to secure the goods necessary to satisfy the particular desires of its clients, but to do so as inexpensively as possible. An agent's score is the difference between the utilities it earns for its clients and the agent's expenditures. For details, visit `http://tac.eecs.umich.edu`.

3.1 Supply

The market supply consists of three types of travel goods: (i) flights to and from Boston, (ii) hotel room reservations at two competing hotels, namely, the Grand Hotel and Le Fleabag Inn, and (iii) entertainment tickets for the Boston Red Sox, the Boston Symphony, and Phantom of the Opera. There is a separate auction corresponding to every combination of good and day, yielding twenty-eight auctions in total: eight flight auctions (there are no inbound flights on the fifth day, and there are no outbound flights on the first day), eight hotel auctions (two hotel types and four nights), and twelve entertainment ticket auctions (three entertainment event types and four nights). All twenty-eight auctions are *simultaneous*. The rules of the various auctions are as follows:

(i) An infinite supply of flights is sold by the "TAC-2000 seller", a specially designated supplier, at continuously clearing auctions in which prices follow a random walk. Specifically, prices are initialized between \$250 and \$400, and perturbed every 30–40 seconds by a random value uniformly selected in the range [\$-10, \$10], but confined within the bounds of \$150 and \$600. No resale of flights is permitted.

(ii) The TAC-2000 seller also makes available sixteen hotel rooms per hotel per night, which are sold at open-cry, ascending, multi-unit, sixteenth-price auctions. In other words, the winning bidders are those who bid among the top sixteen, and these bidders uniformly pay the sixteenth-highest price. Transactions clear when the auctions close, which typically occurs at the end of a TAC-2000 game instance, although these auctions are subject to early closing after random periods of inactivity. No bid withdrawal or resale in hotel auctions is permitted.

(iii) Entertainment tickets are traded among TAC-2000 agents in continuous double auctions. Agents can act as either buyers or sellers, and transactions clear continuously. Each agent receives an initial endowment of tickets for each event on each night—zero with probability 1/4, one with probability 1/2, and two with probability 1/4. Ticket resale is permitted.

3.2 Demand

A TAC-2000 game instance pits eight trading agents against one another, with each agent representing eight clients. The market demand is determined by the sixty-four clients' preferences. Each client is characterized by a random set of preferences for ideal arrival and departure dates (IAD and IDD, respectively, which range over days 1 through 4),[3], a grand hotel room reservation value (HV, which takes integer values between 50 and 150), and reservation values for each of the three types of entertainment events (RV, SV, and TV—integers between 0 and 200—for Red Sox, symphony, and theater, respectively). A sample set of preferences appears in Table 2; these preferences were those of the clients assigned to `RoxyBot` during game 3065 of the competition.

Client	IAD	IDD	HV	RV	SV	TV
1	1	2	99	134	118	65
2	1	3	131	170	47	49
3	1	1	147	13	55	49
4	3	3	145	130	60	85
5	1	3	82	136	68	87
6	2	3	53	94	51	105
7	1	2	54	156	126	71
8	1	4	113	119	187	143

Table 2: `RoxyBot`'s client preferences in game 3065.

The job of each TAC-2000 agent is to assemble a feasible package of goods for each of its clients. A *package* is characterized by arrival and departure dates (AD and DD, respectively, ranging over days 1 through 5), a hotel type (H, which takes on value G for Grand Hotel or F for Le Fleabag Inn), and entertainment tickets (I(j,k) is an indicator variable that represents whether or not the

[3]For notational convenience, we remap outbound flight j to $j-1$, for $j \in \{2,3,4,5\}$.

package includes a ticket on night j to event $k \in \{r, s, t\}$; we also write R1, for example, to indicate that the package includes a Boston Red Sox ticket on night 1). In order to obtain positive utility for a client, an agent must construct a *feasible* package for that client; otherwise, the client's utility is zero. A feasible package is one in which (i) the arrival date is strictly less than the departure date, (ii) the same hotel is reserved during all intermediate nights, (iii) at most one entertainment event per night is included, and (iv) at most one of each type of entertainment ticket is included. Given a feasible package, a client's utility for that package is calculated as follows:

$$\text{utility} = 1000 - \text{travelPenalty} + \text{hotelBonus} + \text{funBonus} \tag{1}$$

where

$$\begin{aligned} \text{travelPenalty} &= 100(|\text{IAD} - \text{AD}| + |\text{IDD} - \text{DD}|) \\ \text{hotelBonus} &= \begin{cases} \text{HV} & \text{if H} = \text{G} \\ 0 & \text{otherwise} \end{cases} \\ \text{funBonus} &= \textstyle\sum_j \left[\text{I}(j,r)\text{RV} + \text{I}(j,s)\text{SV} + \text{I}(j,t)\text{TV}\right] \end{aligned}$$

3.3 Allocation

A TAC-2000 agent faces the allocation problem at the end of a game instance when it must find the assignment of goods to clients that maximizes total client utility. At the end of game 3065, `RoxyBot` held the set of goods listed in Table 3. R, S, and T denote tickets to the Red Sox, symphony, and theater, respectively; G and F denote the Grand Hotel and Le Fleabag Inn, respectively; I and O denote inbound and outbound flights, respectively. The optimal allocation, which `RoxyBot` returned after this game, is depicted in Table 4, given the client preferences presented in Table 2. Packages were allocated to clients as shown, and the total utility obtained was 9999.

Good	Day 1	Day 2	Day 3	Day 4
R	2	2	1	2
S	2	0	2	0
T	1	1	1	0
G	4	3	3	1
F	2	2	0	1
I	6	0	2	0
O	1	4	2	1

Table 3: `RoxyBot`'s final set of goods in game 3065.

Client	AD	DD	H	Tickets	Utility
1	1	2	G	S1, R2	1351
2	1	2	G	R1	1201
3	1	1	G	—	1147
4	3	3	G	R3	1275
5	1	2	F	R1, T2	1123
6	3	3	G	T3	1058
7	1	2	F	S1, R2	1282
8	1	4	G	T1, S3, R4	1562

Table 4: `RoxyBot`'s final allocation in game 3065. The total utility is 9999.

3.4 Completion

Good	Day 1	Day 2	Day 3	Day 4
R	1	0	1	0
S	0	2	1	1
T	1	1	0	0
G	0	0	0	0
F	0	0	0	0
I	0	0	0	0
O	0	0	0	0

Table 5: A sample initial set of holdings: no flights or hotels, with the entertainment ticket holdings set at random according to the distribution of initial endowments.

The completion problem, of which we now present an example, is at the heart of `RoxyBot`'s design. A sample set of initial holdings is shown in Table 5: there are no initial holdings of flights and hotels, and the entertainment ticket holdings are randomly set according to the distribution that determines agents' initial endowments. The prices of goods are those listed in Table 6. This table lists only one price for flights and hotels, namely the ask price, but it lists two prices for entertainment tickets, corresponding to bid and ask prices, respectively. (There is a slight bid-ask spread.) The optimal completion computed by `RoxyBot` given the client preferences listed in Table 2 and the input depicted in Table 6 is shown in Table 7. The expected score of 3906.31 is the sum of the individual client's scores—the differences between utilities and expenses computed on a per client basis—and the expected profits from the sale of entertainment tickets.

Good	1		2		3		4		5	
R	72.91	72.91	83.47	83.47	83.47	83.47	19.00	40.00	–	–
S	63.02	63.02	72.00	72.00	39.00	40.00	76.00	76.00	–	–
T	70.03	70.03	80.88	80.88	51.50	174.64	45.58	45.58	–	–
G	47.00		148.50		146.34		24.17		–	–
F	19.26		70.00		36.00		11.17		–	–
I	280.00		339.00		309.00		360.00		–	–
0	–		394.00		297.00		299.00		374.00	

Table 6: Price data: the ask prices for flights and hotels, and the bid and ask prices, respectively, for entertainment tickets.

Client	AD	DD	H	Tickets	Utilities	Expenses	Score
1	1	2	F	R1, S2	1252.00	666.26	585.74
2	1	3	F	R3	1170.00	704.26	465.74
3	1	1	G	—	1147.00	721.00	426.00
4	3	3	G	R3	1275.00	837.81	437.19
5	1	3	F	T1, R2, S3	1291.00	787.73	503.27
6	2	3	F	T2	1105.00	744.00	361.00
7	1	2	F	R1, S2	1282.00	739.17	542.83
8	1	4	F	T1, S3, R4	1449.00	940.46	508.54
Sales	—	—	—	S4	—	−76.00	76.00

Table 7: A sample completion. Expected score equals 3906.31.

4 Allocation

We now describe `RoxyBot`'s allocation and completion algorithms, which are based on heuristic AI search techniques. Although we exploit the special structure of the TAC-2000 game, we believe our approach is sufficiently generic to transfer to other practical problems in the realm of on-line bidding. For example, our algorithms are not wedded to the assumption of linear utility functions, as are competing integer linear programming solutions [3, 13].

The allocation algorithm embedded in `RoxyBot` is an A^* search algorithm, which is well-known to be optimal if its heuristics are admissible (see, for example, [11]). The structure of the search tree is depicted in Figure 1. At each of the 16 depths, some of the final pool of goods are assigned to a client, and those goods are removed from the pool. This setup corresponds to the "branch on bids" formulation of winner determination [12].

There are two stages in the search. The first stage (levels 1 through 8) is dedicated to the assignment of travel packages—combinations of flights and hotel rooms. There are at most 21 such packages, including the null package. The second stage (levels 9 through 16) is dedicated to the assignment of entertainment packages—feasible combinations of entertainment tickets. There are at most 73 such packages, including the null package.

There are two points to note about the division of the search tree into separate travel and entertainment stages. First, the tree is not in fact as large as it appears: in the bottom half of the tree, a client's travel package has already been assigned; thus, the search need only branch on those entertainment packages that are compatible with a client's given travel package. Second, this division facilitates the separate development of travel and entertainment heuristics, which enabled us to exploit the special structure of the TAC-2000 setup while designing heuristics.

During the actual TAC-2000 competition, `RoxyBot` ordered the clients in each stage 1,2,...,7,8. After the competition, inspired by Gonen and Lehmann [7], we experimented with ordering heuristics, and found the simple heuristic *order clients according to the sum of their entertainment ticket reservation values* to be most effective at reducing run-time.

Finally, our implementation of A^* initiates with various greedy searches that are designed to produce a lower bound on the optimal solution. Thereafter, we need only store search nodes whose heuristic values exceed this lower bound. Ultimately, `RoxyBot` achieves substantial pruning of the search space, down from roughly 10^{20} to less than 10^3 search nodes.

4.1 Travel Heuristics

Travel allocation precludes all of the forms of assignment listed in Table 8 at each travel-assignment node. `RoxyBot`'s A^* search algorithm always rules out any packages that violate constraints 5 and 6. In a preprocessing phase that is applied to each node encountered during search, `RoxyBot` computes an upper bound on the remaining number of travel package assignments based on constraints 3 and 4. The admissible travel heuristics are obtained by simultaneously relaxing constraints 1 and 2. One of the travel heuristics relaxes 1 and 2a, the second relaxes 1 and 2b, and the last relaxes 1 and 2c. The remainder of this section describes the preprocessing phase and the admissible heuristics employed by `RoxyBot` in its search for an optimal travel allocation.

4.1.1 Preprocessing

In the preprocessing phase of search for travel package assignments, `RoxyBot` relies on an iterative procedure to prune the set of travel goods, eliminating those goods that (provably) cannot be used in any travel package. An obvious upper bound on the number of travel packages yet to be assigned to clients is simply $I - d$, where I is the number of clients and d is the current depth of the search; in other words, no more packages can be assigned than the number of clients as yet unassigned.

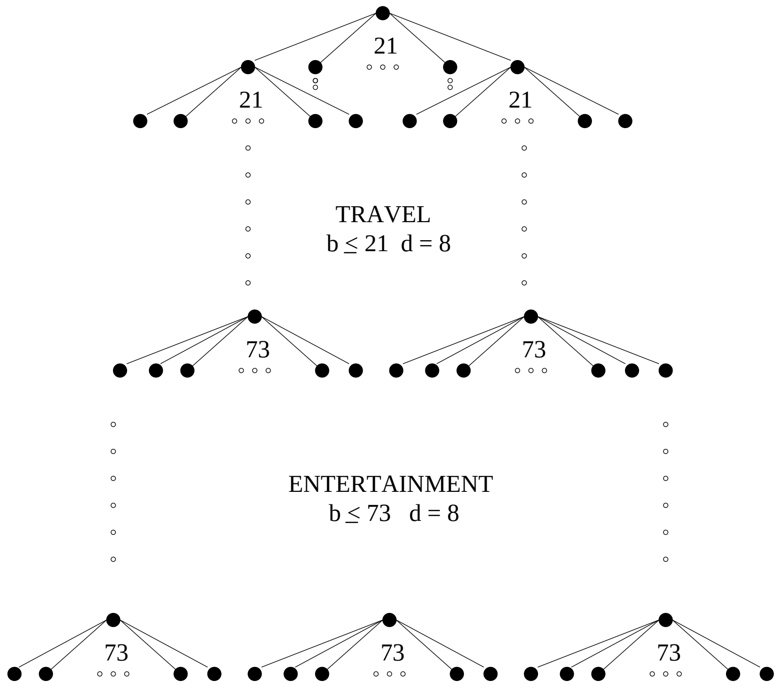

Figure 1: `RoxyBot`'s A^* search tree.

But there exist better bounds. In particular, no more packages can be assigned than the maximum number of packages the current set of travel goods comprise. Copies of goods in excess of this bound can be pruned. Moreover, not all travel goods are feasible components of any travel package—such goods can also be pruned from the current set. In the preprocessing phase, `RoxyBot` repeatedly prunes the set of travel goods, and greedily counts the number of travel packages remaining, until convergence (see Table 9). The pruning and counting algorithms are described presently.

Pruning Algorithm The pruning algorithm is depicted in Table 10. It takes as input an upper bound k on the number of packages and four arrays, namely $\texttt{in}[d]$, $\texttt{out}[d]$, $\texttt{good}[d]$, and $\texttt{bad}[d]$, which store the number of each type of travel good available on day $d \in \{1, 2, 3, 4\}$. It outputs the same four arrays with updated values no greater than the initial values, pruning those travel goods that cannot be used in any travel package. In the first step of initialization, all array values are set equal to the minimum of their initial values and the bound k, since each package uses at most 1 of any good and k is the maximum number of packages. For example, if $\texttt{in}[d] = 4$, but $k = 2$, it suffices to consider $\texttt{in}[d] = 2$, since no more than 2 inbound flights can be part of any allocation. The second initialization step consolidates the $\texttt{good}[d]$ and $\texttt{bad}[d]$ hotel arrays into a single array $\texttt{hot}[d]$.

CANNOT ASSIGN

1. multiple packages to a single client
2. a single package to multiple clients
   i.e., more resources than contained in G
   (a) more hotel rooms than contained in G
   (b) more inbound flights than contained in G
   (c) more outbound flights than contained in G
3. more packages than the set of goods comprise
4. more packages than the number of clients
5. packages based on goods outside G
6. infeasible packages

Table 8: Travel Constraints

Input	initial upper bound $k' = I - d$
	initial set of goods G'
Outputs	final upper bound k
	pruned set of goods G

Repeat

1. $k \leftarrow k'$ and $G \leftarrow G'$.
2. Given k, **prune** the set of travel goods G.
   Store the results in G'.
3. Given k and G', greedily **count** travel packages.
   Store the results in k'.

Until $k = k'$ and $G = G'$

Table 9: Algorithm for preprocessing the set of travel goods.

The intuition for the steps in the intermediate loop of the algorithm are as follows: steps 1–2: cannot assign more arriving/departing flights on day d than the number of hotels on day d; step 3: cannot assign more hotel rooms on day d than the sum of yesterday's hotels and today's arriving

Inputs	integer k
	arrays $\texttt{in}[d]$, $\texttt{out}[d]$, $\texttt{good}[d]$, $\texttt{bad}[d]$
Outputs	arrays $\texttt{in}[d]$, $\texttt{out}[d]$, $\texttt{good}[d]$, $\texttt{bad}[d]$

Initialize

0a. check $\texttt{in}[d]$, $\texttt{out}[d]$, $\texttt{good}[d]$, $\texttt{bad}[d] \leq k$

0b. set $\texttt{hot}[d] = \texttt{good}[d] + \texttt{bad}[d]$

Repeat

for $d \in \{1, \ldots, 4\}$

1. reduce $\texttt{in}[d] \leq \texttt{hot}[d]$
2. reduce $\texttt{out}[d] \leq \texttt{hot}[d]$
3. reduce $\texttt{hot}[d] \leq \texttt{hot}[d-1] + \texttt{in}[d]$
4. reduce $\texttt{hot}[d] \leq \texttt{hot}[d+1] + \texttt{out}[d]$

Until quiescence

Finalize

5. for $d \in \{1, \ldots, 4\}$

   reduce $\texttt{good}[d] \leq \texttt{hot}[d]$

   reduce $\texttt{bad}[d] \leq \texttt{hot}[d]$

Table 10: Pruning algorithm for preprocessing the set of travel goods.

flights; step 4: cannot assign more hotel rooms on day d than the sum of today's departing flights and tomorrow's hotels. These steps are repeated until quiescence. Finally, the $\texttt{good}[d]$ and $\texttt{bad}[d]$ hotel arrays are reset to the minimum of their initial values and the pruned value of $\texttt{hot}[d]$, since no more of each type of hotel can be used on a given day than the maximum number of useful hotels of either type on that day.

An example of the travel goods pruning algorithm appears in Figure 2. The initial values of the arrays (*i.e.*, the set of travel goods, with hotels consolidated) are depicted in Table A. During the first two steps of the inner loop, the number of inbound flights on day 2 is reduced to the number of hotels available on day 2; similarly, the numbers of outbound flights on days 2 and 3 are reduced to the numbers of hotels available on days 2 and 3, respectively (see Table B). The pruning via steps 3 and 4 is depicted in Table C: the number of hotels on day 1 is reduced to the sum of the number of inbound flights on day 1 and the number of hotels on day 0, which is always 0; similarly, the number of hotels on day 4 is reduced to the sum of the number of outbound flights on day 4 and the number of hotels on day 5, which is also always 0. The algorithm now loops back to steps 1 and 2, which prune the number of inbound flights on day 4 and the number of outbound flights on day 1, since no hotels are available on either of these days (see Table D). At this point, no further pruning is possible. The pruned set of travel goods is input to the counting algorithm.

Counting Algorithm The intent of the counting algorithm is to determine the maximum number of packages that can be comprised from the set of travel goods (post-pruning). If this number is less than the current upper bound k, then it becomes the new upper bound on the number of packages yet to assign. The algorithm proceeds by first computing the number of packages that can be comprised using *only* bad hotels (k_F), and then computing this number again using only *good* hotels (k_G). The value $k_F + k_G$ serves as upper bound on the number of legal hotel packages that can be constructed, although flights are double-counted. The counting algorithm also computes the number of packages that can be constructed using *either* good or bad hotels (k_H). The value k_H serves as an upper bound on the number of legal packages that the available flights comprise, allowing for illegal hotel combinations. Finally, the counting algorithm outputs $\min\{k, k_F + k_G, k_H\}$.

The counting itself is accomplished by the following greedy procedure: for $i \in \{F, G, H\}$, compute k_i by greedily counting packages in class i, in order from shortest to longest. Figure 2 presents an example. The table labeled X depicts the given set of travel goods; as above, the numbers of good and bad hotels on day d have been summed to form $\texttt{hot}[d]$. The algorithm begins by counting packages of length 1. There are 2 such packages in the given set, a package on day 2, and another on day 3. After counting, these packages are eliminated from the set of travel goods, yielding Table Y. The counting now continues, but the second time through packages of length 2 are counted. There are again 2 such packages in the set, a package arriving on day 1 and departing on day 2, and another arriving on day 3 and departing on day 4. After eliminating these packages, the set of travel goods is empty and the algorithm terminates. In total, this set of travel goods yields (at most) 4 packages.

The greatest number of packages is produced by using the fewest number of goods per package. Therefore, greedily counting packages from shortest to longest is guaranteed to produce the maximum number of legal packages. Now to determine the maximum number of packages of length 1 is a simple matter, since no packages of length 1 overlap. Moreover, after packages of length 1 are eliminated, no packages of length 2 overlap; thus, it becomes a simple matter to determine the maximum number of packages of length 2. In general, after packages of length l are eliminated, no packages of length $l + 1$ overlap. Therefore, this greedy counting procedure easily computes an upper bound on the total number of legal packages.

4.1.2 Admissible Heuristics

An heuristic is an estimation function. An admissible heuristic is one that produces only optimistic estimates. In a maximization problem such as allocation, admissible heuristics always *overestimate*

A

Day	in	hot	out
1	0	2	1
2	3	1	2
3	2	2	3
4	1	1	0

1–2

↓

B

Day	in	hot	out
1	0	2	1
2	**1**	1	**1**
3	2	2	**2**
4	1	1	0

3–4

↓

C

Day	in	hot	out
1	0	**0**	1
2	1	1	1
3	2	2	2
4	1	**0**	0

1–2

↓

D

Day	in	hot	out
1	0	0	**0**
2	1	1	1
3	2	2	2
4	**0**	0	0

X

Day	in	hot	out
1	1	1	0
2	1	2	2
3	2	2	1
4	0	1	1

↓

Y

Day	in	hot	out
1	1	1	0
2	**0**	**1**	**1**
3	**1**	**1**	**0**
4	0	1	1

↓

Z

Day	in	hot	out
1	**0**	**0**	0
2	0	**0**	**0**
3	**0**	**0**	0
4	0	**0**	**0**

Figure 2: Examples of the pruning (left) and counting (right) algorithms on sets of travel goods.

the value of search nodes. The aim in developing such heuristics is to produce overestimates that are as close to the search nodes' true values as possible, without going under. In our development, we further ruled out the possibility of undertaking any (potentially time-consuming) intermediate search in the computation of heuristic values.

One (search-free) naive, admissible heuristic is arrived at by relaxing the second travel constraint listed in Table 8: *i.e.*, allow a single package to be assigned to multiple clients. Simply assume that any as yet unassigned clients will be assigned their favorite packages *with replacement* and then sum the top k corresponding utility values. Our admissible travel heuristic improves upon this naive idea by considering three specializations. In particular, we compute three travel heuristic values inspired by various relaxations of constraints 2a, 2b, and 2c, and return as our heuristic estimate the minimum of the three values. Note that $\min\{h_1, h_2, h_3\}$ is an admissible heuristic if h_1, h_2, and h_3 are admissible heuristic functions.

Each of our travel heuristics enforces one of the following but relaxes the other two: cannot assign more travel packages than the number contained in the set G that include (i) the Grand Hotel or Le FleaBag Inn, (ii) inbound flights on day 1,...,4, or (iii) outbound flights on day 2,...,5. For example, if G contains only a single inbound flight on day 1, then our inbound flight heuristic ensures that no more than a single package with arrival day 1 be assigned. But now if as in the naive heuristic, we attempt to assign all as yet unassigned clients their favorite packages, and multiple clients' favorite packages specify arrival on day 1, then we cannot determine without further search to which client to assign the single such package constructible from the goods in the set G.

Rather than search, our travel heuristics also relax the first constraint in Table 8: *i.e.*, we allow multiple packages to be assigned to a single client. We combine all client utilities on all packages into a single list, and sort this list from highest to lowest. Next, for each of the three types of resource constraints, we sum the k maximal values that adhere to the constraint under consideration. In this way, we efficiently arrive at heuristic values of the search nodes that are guaranteed to be overestimates of the true values without the need to invoke any intermediate searches. Our travel heuristic dominates the naive heuristic proposed initially.

Our main travel heuristic algorithm is depicted in Table 11. It takes as input the current search node n, the clients' utilities, and the output of the preprocessing phase, namely the upper bounds k, k_F, and k_G, and the pruned set of travel goods G. It outputs the heuristic value $h(n)$ at that node.

The algorithm begins with an initialization phase.[4] The first step is to compute the set of feasible travel packages F that only make use of the travel goods in the set G. For instance, given the set of travel goods depicted in Table 12(a), the set $F = \{$13G,13F,15G$\}$. Travel package 13G, for example, denotes "arrive on day 1, depart on day 3, stay at the Grand Hotel." Next the heuristic partitions the set F along three dimensions to form a set of partitions $\mathcal{P}$. First, F is partitioned according

[4] This initialization phase is for expository purposes only. In practice, we employ caching tricks, rather than explicitly compute the set of feasible travel packages F and the set of partitions $\mathcal{P}$ in an initialization phase.

to whether the package includes the Grand Hotel or Le FleaBag Inn (see Table 13(a)); second, it is partitioned according to the arrival day (see Table 13(b)); and third it is partitioned according to the departure day (see Table 13(c)). In the hotel partition presented in Table 13(a), there are 2 travel packages in class G (the Grand Hotel class) and 1 in class F (the Le FleaBag Inn class).

Given a partition $p \in \mathcal{P}$, the heuristic loops through all classes $c \in p$. In the inner loop, the best $\texttt{num}[c]$ utility values that clients not yet assigned travel packages attribute to the packages in class c are inserted into a priority queue. Now the top k values in the priority queue are summed to yield the heuristic estimate under this partition. This estimate is computed under all three partitions, and the minimum value is returned as the heuristic value of the search node. In Table 13, we instantiate $\texttt{num}[c]$ with $\texttt{hot}[c]$, $\texttt{in}[c]$, and $\texttt{out}[c]$, according to the partition p. The values of $\texttt{in}[c]$ and $\texttt{out}[c]$ are simply the number of copies of good c in the input set G. The values $\texttt{hot}[\text{G}] = k_G$ and $\texttt{hot}[\text{F}] = k_F$ are computed by the counting algorithm.

Inputs	search node n upper bounds k, k_F, k_G pruned set of goods G clients' utilities
Output	estimate $h(n)$

Initialize

compute set of feasible travel packages F, given G
compute set of partitions $\mathcal{P}$, given F

Main Loop

for partition $p \in \mathcal{P}$
    for class $c \in p$
        $\texttt{num}[c]$ = bound on the number of c's at n
        insert top $\texttt{num}[c]$ utilities into a priority queue
    $\texttt{estimate}[p]$ = sum top k entries in the priority queue
return minimum $\texttt{estimate}[p]$

Table 11: Main Travel Heuristic.

Let us now work through an example, given the utility values listed in Table 12(b). According to the departure day partition, there are two feasible packages departing on day 3, but there is only one outbound flight on that day. Thus, only the maximum utility value among all packages departing on day 3 is inserted into the priority queue, namely \$1150. In addition, the utility of package 15G, namely \$975, is inserted into the queue, since there is one flight departing on day 5. Finally, the maximum two utility values in the queue are summed, yielding a heuristic value of \$2150. The heuristic value under the arrival day partition is also $2150. But under the hotel partition, there is 1 Grand Hotel package (insert value $1150), and there is 1 Le FleaBag Inn package (insert value $800). Thus, the hotel partition yields the minimum heuristic value, namely $1950. Note that in general none of these partitionings dominate any of the others.

Good	Day 1	Day 2	Day 3	Day 4
G	1	1	1	1
F	1	1	0	0
I	2	0	0	0
O	0	1	0	1

(a) Travel Goods

Package	Utility
13G	1150
15G	975
13F	800

(b) Utility Values

Table 12: (a) Set of travel goods input to main travel heuristic. (b) Corresponding set of feasible travel packages and sample utility values of clients not yet assigned travel packages.

Hotel	Packages	`hot`
G	{13G, 15G}	1
F	{13F}	1

(a) Hotel

Day	Packages	`in`
1	{13G,13F,15G}	2
2	∅	0
3	∅	0
4	∅	0

(b) Arrival Day

Day	Packages	`out`
2	∅	0
3	{13G,13F}	1
4	∅	0
5	{15G}	1

(c) Departure Day

Table 13: Partitionings of the set of travel goods with upper bounds.

4.2 Entertainment Heuristics

During the entertainment phase of search, `RoxyBot` assigns each client an entertainment package, adhering to the set of constraints listed in Table 14.

`RoxyBot`'s A^* search algorithm immediately rules out any packages that violate constraints 6 and 7; in particular, no entertainment package is ever assigned that is inconsistent with a client's pre-assigned travel package, or for which the tickets that comprise the package are not available. In a preprocessing phase, `RoxyBot` computes an upper bound on the remaining number of entertainment package assignments, which, in accordance with constraints 4 and 5, is the minimum of the total number of entertainment tickets owned (*i.e.*, available) and the number of unassigned hotel rooms.

`RoxyBot`'s admissible entertainment heuristics are inspired by separate relaxations of each of the first three constraints listed in Table 14. The first heuristic loops through the days of the week assigning at most one ticket per client per day, but possibly assigning to a single client multiple

CANNOT ASSIGN
1. single client multiple tickets to the same event
2. single client multiple tickets on the same day
3. single ticket to multiple clients
4. more tickets than we own
5. more tickets than hotel rooms
6. entertainment packages w/o tickets
7. entertainment packages inconsistent w/ travel

Table 14: Entertainment Constraints

tickets to the same event (on different days). The second heuristic proceeds by looping through the three events assigning at most one ticket to each event to each client, but possibly assigning a single client multiple tickets (to different events) on the same day. The third heuristic runs down the list of available packages giving each client the package it most desires, but possibly assigning the same ticket to multiple clients (if a single ticket is contained in the favorite package of multiple clients). As in the case of travel, the minimum of these heuristic estimates is returned as the value of the entertainment heuristic function. The remainder of this section details the admissible heuristics employed by `RoxyBot` in its search for an optimal entertainment allocation.

4.2.1 Admissible Heuristics

We describe our entertainment heuristics in the context of an example. Suppose the search is at depth 14, with two clients, say A and B, yet to be assigned entertainment packages. Assume the entertainment goods that have not yet been assigned to clients and the utilities for clients A and B are those listed in Table 15. In addition, assume clients A and B are both scheduled to be in town on days 1, 2, 3, and 4.

Goods	1	2	3	4
R	0	0	0	1
S	0	0	0	1
T	0	1	0	0

Utilities	`R`	`S`	`T`
A	75	50	25
B	90	60	30

Table 15: Sample set of entertainment goods and utilities.

Notation-wise, let $\texttt{ntix}[d, e]$ denote the number of tickets on day d for event e; let $\texttt{ntix}[d,–] = \sum_e \texttt{ntix}[d, e]$ denote the total number of tickets on day d, and let $\texttt{ntix}[–,e] = \sum_d \texttt{ntix}[d, e]$ denote the total number of tickets to event e.

Our first entertainment heuristic loops through the days, assigning on each day d, at most $\texttt{ntix}[d]$ tickets to clients in town on day d who most value events for which tickets are owned on that day. To implement this heuristic, tables of the form given in Table 16 are cached in sorted order according to the maximum value of each client's preferences for each subset of events. In this way, the list of clients ordered by preference for any of the possible subsets of events owned on day d is readily obtainable. Entertainment heuristic #1 assigns the single ticket on day 2 to client B for 30; in addition, it assigns its first ticket on day 4 to client B for 90 and the second ticket on day 4 to client A for 75. In total, this heuristic estimates the value 195. This heuristic overestimates the true value in its assignment of R4 to both clients.

for all d, assign `ntix`*[d] to clients in town on day d who most value events for which tickets are available on that day*

Table 16: Entertainment Heuristic #1.

Our second entertainment ticket loops through events. For each event e, it assigns at most $\texttt{ntix}[e]$ tickets to clients that are in town on some day for which tickets to event e are owned, in order of preference for event e. Since all clients are in town on all days in our example, this heuristic assigns all tickets to client B (client B's utility values dominate client A's for all events). Thus, this heuristic estimates the value 180. This heuristic overestimates the value of the optimal allocation in its assignment of both R4 and S4 to client B.

for all e, assign `ntix`*[e] to clients who most value event e and whose days in town intersect days for which tickets to event e are available*

Table 17: Entertainment Heuristic #2.

Our final entertainment heuristic is analogous to the naive travel heuristic introduced to motivate our travel heuristic design. Assume that any as yet unassigned clients will be assigned their favorite entertainment packages *with replacement*. In particular, client A is assigned T2 and R4, yielding value 100, and client B is also assigned T2 and R4, yielding value 120. In total, this heuristic estimates the value 200. Overall, our entertainment ticket heuristic returns the the minimum of the three heuristic values, namely 180. Note that the value of the optimal allocation is 170.

5 Completion

5.1 Pricelines

Unlike the allocator, the completer faces the added complexity that the resources being assigned may not yet be in hand; they may still need to be purchased at auction. Furthermore, in the case of entertainment tickets, resources which are in hand might be more profitably sold on the market than allocated to `RoxyBot`'s own clients. To reason about the resource tradeoffs involved, the completer makes use of a data structure called a *priceline* for each resource. A priceline is a list of prices $\vec{p}$, constructed as follows:

- Let $\langle a_1, a_2, \ldots, a_S \rangle$ be the marginal costs (*i.e.*, ask prices) to purchase the first, second, etc. of the resource, up to the total supply S. In general, $a_1 \leq a_2 \leq \ldots a_S$.
- Let $\langle b_1, b_2, \ldots, b_D \rangle$ be the marginal profits (*i.e.*, bid prices) that would be realized by selling the first, second, etc. of an owned resource, up to the total demand D. [5] Note that $b_1 < a_1$, since a bid price that matches an ask price clears immediately. Moreover, $b_D \leq b_{D-1} \leq \ldots b_1$.
- Define the pre-priceline $\vec{q} = \langle \circ, q_1, \ldots, q_{S+D} \rangle$ by concatenating the supply and demand vectors as follows: $\langle b_D, \ldots, b_1, a_1, \ldots, a_S \rangle$, and prepending the pointer $\circ$. Note that $q_1 \leq \ldots \leq q_{S+D}$.
- Finally, define the priceline $\vec{p}$ by shifting the pointer by $D - H$ entries, where H is the quantity of the resource currently held. Note that H may be negative, which represents the short-selling of resources. If $D - H < 0$, then the pointer shifts to the left: *i.e.*, $D - H$ zeroes are inserted to the left of $\vec{q}$. These zeroes represent the *sunk cost* of allocating resources that the agent holds but cannot sell on the open market. If $D - H > 0$, then the pointer shifts to the right. If $D - H$ exceeds $D + S$, then $|S + H|$ ∞s are appended to $\vec{q}$. These ∞s represent the cost of allocating resources that the agent sold short but cannot buy back on the open market.

Examples of pricelines are illustrated in Table 18. Using this construction, the completer's task is much simplified: a package's cost is computed by popping off prices from left to right, starting at the pointer $\circ$, from the relevant pricelines. The value of a package to a client equals the client's utility for that package less its cost.

A strength of the priceline model is its versatility: it transparently handles either one-sided or double-sided auctions, short-selling of resources, and both limited and unlimited supply and demand. A weakness of the priceline model is that it is suited for reasoning only with deterministic prices; it does not explicitly account for variance within auction clearing prices. Nonetheless, `RoxyBot`'s "hedging strategy" heuristically incorporates risk aversion into hotel room pricelines (as described in Table 18).

[5] In the TAC-2000 setup, this aspect of the priceline data structure applies to entertainment tickets only.

- $S = \infty$, $D = H = 0$, $\vec{q} = \vec{p} = \langle \circ, 315, 315, \ldots \rangle$

  This priceline illustrates a typical priceline for flights: an infinite supply is predicted to be available, at an expected price of \$315 each. `RoxyBot` currently holds none of this resource.

- $S = \infty$, $D = 0$, $H = 2$, $\vec{q} = \langle \circ, 315, 315, \ldots \rangle$, $\vec{p} = \langle \circ, 0, 0, 315, 315, \ldots \rangle$

  In this priceline, `RoxyBot` owns two tickets for this flight (which cannot be sold back). The amount spent on those flights is treated as a sunk cost: *i.e.*, the agent need not consider the costs already incurred when allocating them to clients. Allocating more than two of these flights, however, is expected to incur an additional cost of \$315 each.

- $S = 16$, $D = H = 0$, $\vec{q} = \vec{p} = \langle \circ, 105, 155, 205, 255, 305, 355, 405, 455, 505, 555, 605, 655, 705, 755, 805, 855 \rangle$

  `RoxyBot` uses this type of priceline to mitigate risk in hotel auctions. Although the hotel auctions are structured such that a single price is charged to all winning bidders, `RoxyBot` models its own impact on that price by assuming that each additional room is more expensive than the last. This heuristic price-setting mechanism encourages `RoxyBot` to diversify its portfolio of hotel rooms by preventing it from relying too heavily on any one particular hotel room, and therefore bidding for that resource in such a way that `RoxyBot`'s very own bids cause a deadly spike in the hotel auction's clearing price.

- $S = 2$, $D = 2$, $H = 4$, $\vec{q} = \langle \circ, 25, 65, 75, 115 \rangle$, $\vec{p} = \langle \circ, 0, 0, 25, 65, 75, 115 \rangle$

  This priceline reflects a typical scenario in an entertainment market. `RoxyBot` currently holds four of this ticket. The priceline indicates that there is market demand for two of its four tickets, one of which could be sold for \$65, and another of which could be sold for \$25. In addition, there is a supply of two additional tickets on the market, the one of which could be purchased at \$75 and another at \$115. The priceline summarizes all of this information.

  Now if the completer allocates one or two of these tickets to `RoxyBot`'s clients, it incurs no cost, since its first two tickets were not marketable anyway. If the completer allocates four tickets to clients, it incurs a cost of \$90, which represents the opportunity cost of not selling the tickets on the free market. If the completer allocates all six tickets, it incurs the total cost of the priceline, representing both the lost opportunity cost as well as the expense of buying two additional tickets.

- $S = 2$, $D = 2$, $H = -1$, $\vec{q} = \langle \circ, 25, 65, 75, 115 \rangle$, $\vec{p} = \langle 25, 65, 75, \circ, 115 \rangle$

  This priceline corresponds to the situation in which `RoxyBot` has short-sold one copy of this entertainment ticket (*i.e.*, $H = -1$). The cost to the completer of allocating this ticket to one of `RoxyBot`'s clients is the cost of the *second* ticket for sale on the open market, namely \$115. The ticket available at the first price of \$75 will be purchased to replace that which had been sold short, after which the next ticket on the priceline can be allocated.

Table 18: Sample pricelines used by `RoxyBot`'s completer algorithm.

5.2 Beam Search

Like allocation, A^* search can also be used to solve the completion problem; however, most of the A^* heuristics used in `RoxyBot`'s optimal allocator were not applicable in the completer scenario (since the number of goods is bounded only by the number in the marketplace), and running times for an optimal completer occasionally took as long as 10 seconds. Nonetheless, using an approximation technique based on a greedy (non-admissible) heuristic and a variable-width beam search over the same search space,[6] `RoxyBot` finds an optimal completion in less than 1 second of search. Therefore, during the competition, `RoxyBot` used beam search rather than provably optimal A^* search.

Our beam search heuristic $f(n)$ is inspired by the "rollout methods" that have been used in game-tree search (*e.g.*, [1, 14]). It works as follows: a greedy algorithm is run to complete the search from node n down to the bottom of the tree. Specifically, for a client that has thus far been assigned neither a travel package nor an entertainment package, she is assigned the travel and entertainment packages that jointly maximize her travel utility minus cost; for a client that already has a travel package, she is assigned the best entertainment package, namely that which maximizes her entertainment utility minus cost. The time to compute $f(n)$ is linear in the depth of n; therefore, the runtime of our beam search algorithm is quadratic. Since package assignments are made without replacement, this heuristic is inadmissible. It is, however, effective and scalable in practice [3].

Recall that in beam search, search proceeds level by level with no backtracking; at each level, only the top B nodes according to the heuristic are expanded. Since our search tree is of fixed depth $2I$, beam search has the desirable property that it expands no more than $2IB$ nodes in total. Space and time requirements are therefore highly predictable. In a companion paper [3], we compare the performance of our beam search algorithm with an integer linear programming (ILP) solution, which is optimal but for which space and time requirements are not predictable. We found that a beam width of only 1 yielded a median accuracy of 99.4% in the 8 client case, with a median running time of less than 0.01 seconds. In the case of 64 clients, a beam width of 1 achieved a median accuracy of 97.9% in roughly 1 second. In contrast, ILP yielded optimal solutions in the 8 client case, with a median running time of roughly 0.02 seconds; but in one of the 64-client cases, the machine exhausted its 2Gb of RAM after six hours and aborted.

6 Estimation

`RoxyBot`'s pricelines are data structures in which to describe the costs of market resources. In auctions such as those fundamental to the TAC setup, however, costs are not known in advance. Therefore, the actual input to `RoxyBot`'s pricelines are but *estimates* of auction closing prices and *estimates* of future market supply and demand (current holdings are known). Note that it is crucial

[6] Perhaps surprisingly, a 17th level of search, in which we consider selling goods on the open market, is unnecessary. Our priceline design—specifically, that which is exemplified by the the fourth bullet point in our description—enables this optimization. By charging `RoxyBot` opportunity costs whenever it allocates goods it owns to its clients, we reduce the completion problem to that of acquisition—the problem of determining the optimal quantity of each resource to buy, *not sell*—which requires only 16 levels of search. For details, see [2].

to consider *closing* prices and *future* supply and demand, since using current information could lead an agent to make wise short-term decisions that jeopardize long-term success.

To estimate clearing prices in the entertainment ticket auctions, we used an adjustment process based on Widrow-Hoff updating that was inspired by the zero-intelligence plus traders of Cliff and Bruten [4]. `RoxyBot` maintained two internal price estimates for all entertainment tickets, an `ask_est` and a `bid_est`. These estimates were adjusted in the direction of the trade price, if any trades took place. Otherwise, in the presence of a bid-ask spread, the `ask_est` was adjusted in the direction of `lo_ask`, and the `bid_est` was adjusted in the direction of `hi_bid`. This procedure is outlined in Table 19. Since the entertainment auctions clear continuously, market supply and demand were both repeatedly assumed to be 1.

Input	current `ask_est`, `bid_est` current `lo_ask`, `hi_bid` rates of adjustment α, β
Outputs	adjusted `ask_est`, `bid_est`

```
If (a recent trade took place at price p)

   1. ask_est = (1 − α) ∗ ask_est + αp
   2. bid_est = (1 − α) ∗ bid_est + αp

Else (there is a hi_bid−lo_ask spread)

   1. ask_est = (1 − β) ∗ ask_est + βlo_ask
   2. bid_est = (1 − β) ∗ bid_est + βhi_bid
```

Table 19: Setting price estimates for entertainment ticket auctions. During the TAC competition, $\alpha = 0.1$ and $\beta = 0.05$.

The flight pricelines were of the form of the first two examples in Table 18. Supply is infinite and the expected closing price of all flight tickets is precisely the current price. If `RoxyBot` already owned several tickets (as in the second example in Table 18), their costs were sunk: *i.e.*, set to 0. `RoxyBot`'s estimation of hotel pricelines during the 2000 competition was somewhat ad-hoc, since the TAC market game was not suited to the use of automated learning algorithms for price-estimation based on bidding patterns observed during a game instance. Regardless, the main idea of our strategy for building hotel pricelines that naturally lend themselves to hedging is described in the third example in Table 18. The TAC 2001 market game was designed to encourage early bidding in hotel auctions; thus, we implemented hotel price-estimation algorithms in later versions of `RoxyBot`.

7 Results

The results of the TAC competition are depicted in the Figure 3. The first graph depicts the scores in qualifying rounds (90 games, with the lowest 10 scores dropped), and the second graph depicts the scores on competition day (13 games). Along with `RoxyBot`, the other three top-scoring teams were `ATTac`, `Aster`, and `UMBCTac`. `ATTac`, built by a team of researchers at AT&T, is an agent whose functionality is best characterized as adaptable; its flexibility enabled it to cope with a wide variety of scenarios during the competition. `Aster`, developed by `intertrust.com`, is an agent that is neither strictly greedy, nor strictly optimal; scalability, rather than optimality, was foremost among its designers' goals, since they expect many situations of practical interest to be more complex and less structured than TAC. `UMBCTac`'s competitive edge is that it conserves network bandwidth; on average, this agent updates its bidding data every 4–6 seconds, providing a significant advantage over the reported 8–20-second delays experienced by competing agents.

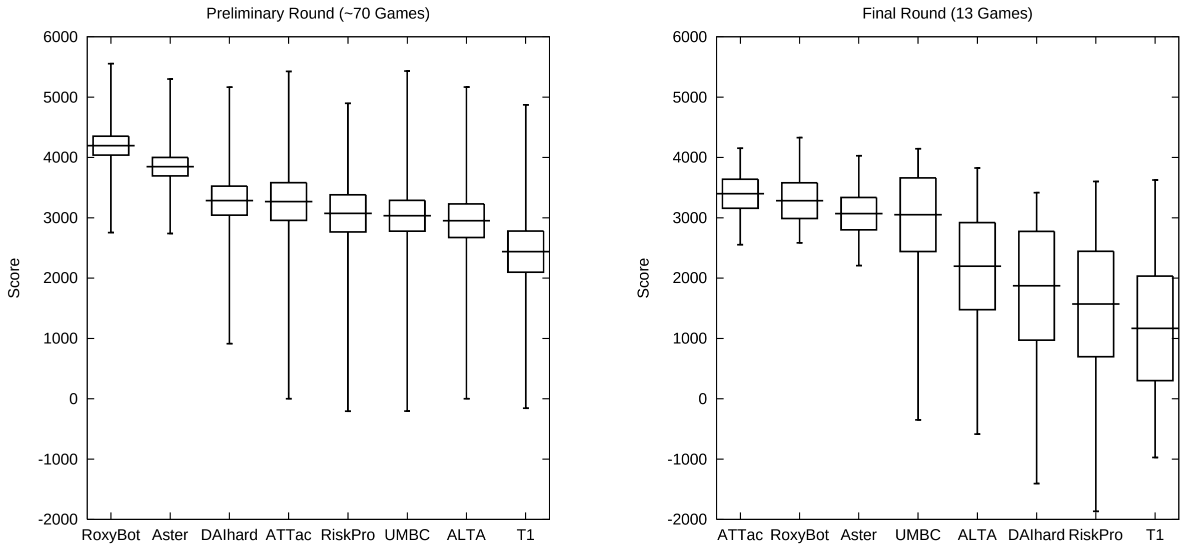

Figure 3: (a) Preliminary Round (90 games, lowest ten scores dropped): horizontal lines indicate mean, minimum, and maximum scores; box delimits 95% Confidence Interval. (b) Final Round (13 games, no scores dropped): horizontal lines indicate mean, minimum, and maximum scores; box delimits 95% Confidence Interval.

8 Summary

This paper described `RoxyBot`, one of the top-scoring agents in the First International Trading Agent Competition. `RoxyBot` faced two key technical challenges in TAC-2000: (i) *allocation*—assigning purchased goods to clients at the end of a game instance so as to maximize total client utility, and (ii) *completion*—determining the optimal quantity of each resource to buy and sell given client preferences, current holdings, and market prices. For the dimensions of TAC-2000, an optimal solution to the allocation problem is tractable, and `RoxyBot` uses a search algorithm based on A^* to produce optimal allocations. An optimal solution to the completion problem is also tractable, but in the interest of minimizing bidding cycle time, `RoxyBot` solves the completion problem using beam search with a greedy heuristic, producing approximately optimal completions.

In related work [2], we have demonstrated the general applicability of the TAC-2000 framework by showing that allocation and completion, which are bid determination (BD) problems in simultaneous auctions, are isomorphic to common variants of the winner determination (WD) problem in combinatorial auctions. The equivalence between BD and WD makes new datasets available for testing by the combinatorial auction community. Implementations of winner determination algorithms are typically evaluated on randomly generated datasets (see, for example, Leyton-Brown, *et al.* [9]), since data from large-scale combinatorial auctions is scarce. (One obvious exception is the FCC spectrum auction.) Unlike randomly generated datasets, the Trading Agent Competition offers an intuitively meaningful dataset. In the future, it would be of interest to compare `RoxyBot`'s algorithmic core to other classic WD algorithms [6, 12], using TAC-2000 and other datasets.

Lastly, we note that AI search techniques like those fundamental to `RoxyBot` and those applied in [6, 12] are more general than traditional integer linear programming (summarized in [5]) solutions, which are applicable only in the case where utilities are linear.

References

[1] B. Abramson. Expected-outcome: A general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):182–193, Feb. 1990.

[2] J. Boyan and A. Greenwald. Bid determination in simultaneous auctions: An agent architecture. In *Proceedings of Third ACM Conference on Electronic Commerce*, 210–212 2001.

[3] J. Boyan, A. Greenwald, R. Kirby, and J. Reiter. Bidding algorithms for simultaneous auctions. In *Proceedings of IJCAI Workshop on Economic Agents, Models, and Mechanisms*, pages 1–11, 2001.

[4] D. Cliff and J. Bruten. Zero is not enough: On the lower limit of agent intelligence for continuous double auction markets. HP Technical Report HPL-97-141, 1997.

[5] S. de Vries and R. V. Vohra. Combinatorial auctions: A survey. *INFORMS Journal on Computing*, 15(3):284–309, 2003.

[6] Y. Fujishima, K. Leyton-Brown, and Y. Shoham. Taming the computational complexity of combinatorial auctions. In *Proceedings of Sixteenth International Joint Conference on Artificial Intelligence*, pages 548–553, August 1999.

[7] R. Gonen and D. Lehmann. Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics. In *Proceedings of Second ACM Conference on Electronic Commerce*, pages 13–29, October 2000.

[8] A. Greenwald and P. Stone. Autonomous bidding agents in the Trading Agent Competition. *IEEE Internet Computing: Special Issue on Virtual Markets*, April 2001.

[9] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of Second ACM Conference on Electronic Commerce*, pages 66–76, October 2000.

[10] M. Rothkopf, A. Pekeč, and R. Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44(8), 1998.

[11] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, 1995.

[12] T. Sandholm and S. Suri. Improved algorithms for optimal winner determination in combinatorial auctions and generalizations. In *Proceedings of AAAI*, pages 90–97, 2000.

[13] P. Stone, M. Littman, S. Singh, and M. Kearns. `ATTac`-2000: An adaptive autonomous bidding agent. In *Fifth International Conference on Autonomous Agents*, 2001.

[14] G. Tesauro and G. R. Galperin. On-line policy improvement using Monte-Carlo search. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in NIPS*, volume 9. MIT Press, 1997.

[15] M. Wellman, P. Wurman, K. O'Malley, R. Bangera, S. Lin, D. Reeves, and W. Walsh. A trading agent competition. *IEEE Internet Computing*, April 2001.