Web Interfaces for Human Bidding Agents and General Auction Scheduling with the Java Auction Configuration Kit (`JACK`)

Andrew Loomis
Department of Computer Science
Brown University

May 14, 2013

1 Introduction

The simulation of real-world economic markets is a crucial capability for the experimental economists who are trying to understand them. In particular, the ability to perform user studies can lead to important insights into the behavior of humans in these domains [1, 2, 3, 4]. In many of these markets, auctions are a significant mechanism used to distribute goods and services. The goal of this work is to extend the functionality of the Java Auction Configuration Kit (`JACK`) [1], a general purpose auction simulator, to more easily support large user studies in a variety of auction domains and configurations. To that end, this work is divided into improving `JACK` in two key areas: the first area is the addition of a web interface for human bidders, and the second area is the implementation of a robust mechanism for general auction scheduling.

The current `JACK` implementation does not provide an extensible solution for running auction simulations with large numbers of human participants. It requires that all participants download and install a client application capable of interfacing with the `JACK` server running the simulation. The addition of a web interface for human participants has many advantages over the current design. Such an interface would make running large scale experiments much easier. Tightly integrating this interface with Amazon Mechanical Turk (Mturk) and other online crowdsourcing facilities could create near on-demand auction simulations with human participants. Such a setup would be extremely valuable to the researchers and educators trying to better understand human behavior in complex games [5]. Section 2 covers the implementation of such an interface in more detail.

In addition to providing convenient and easy to use interfaces for auction simulations, flexible configuration is a key feature for any simulator. In particular, the scheduling of multiple auctions distinguishes many types of real-world markets from one another. Sequential and simultaneous schedules are common, but more complicated schedules also exist such as those that can be found on eBay, or in the Dutch flower auctions. `JACK` does not have a convenient way to express these diverse schedules. In fact, the only way they can currently be expressed is by implementing them manually in code. Section 3 will cover an implementation of general auction scheduling through configuration that attempts to solve this problem for many different types of auction schedules.

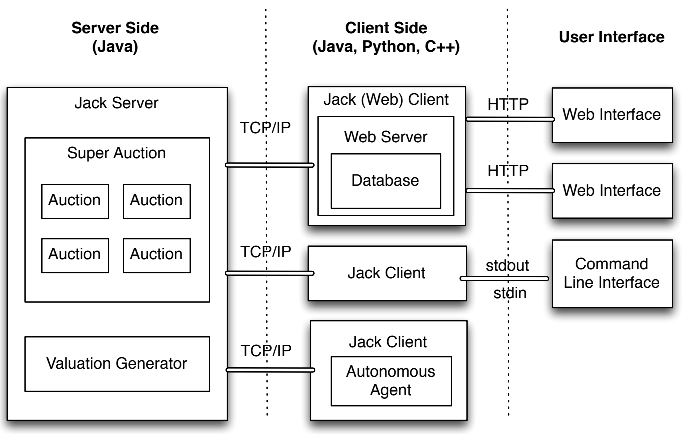


Figure 1: Client-server architecture of JACK.

2 Web Interface

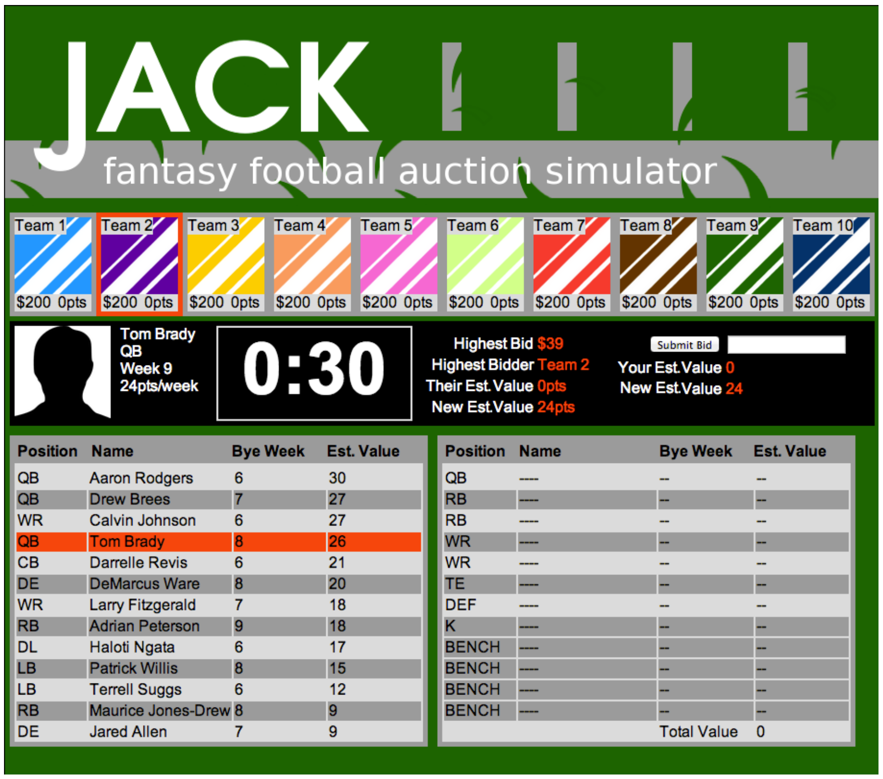
The architecture of the JACK framework follows the client server model. At a high level, the server is responsible for executing auction simulations, while the clients are responsible for participating in them. The communication protocol between the server and clients is well defined for each auction type, and so clients can be written in any language that support TCP/IP. These clients generally fall into one of two categories. They are either autonomous agents or interfaces for human participants. The current JACK library comes with an implementation of a command line interface for human participants, however it is not particularly convenient or efficient for conducting human experiments. We can still use this implementation as a model for implementing a much more powerful web interface for humans that requires almost no changes to the JACK server backend. An overview of new architecture with such an interface is shown in Figure 1.

The JACK web client is implemented using Python's Django framework and uses a sqlite database as its backend. Like other clients, the web client communicates with the server using TCP/IP. When a typical client receives a message from the server it either passes that information to an autonomous agent or presents that information directly to a user. In contrast, the web client stores this information in a database. The web interface, which is implemented in HTML and Javascript, constantly polls this database through HTTP requests looking for new information from the JACK server. When new information is detected, it is forwarded to the web interface where it is presented to the user. Messages from the user to the JACK server, which predominantly consist of bids, are returned in much the same way. When a user places a bid in the web interface an HTTP post is sent to the web server. The web server updates the database, and when the web client detects that a new bid has been placed it forwards that bid to the JACK server.

2.1 Fantasy football auction draft

To demonstrate the capabilities of this design, this work includes an implementation of a web interface for the fantasy football auction draft. The popularity of fantasy sports has continued to increase in recent years, and many large media corporations such as CBS, ESPN, FOX, and Yahoo! offer their services to run online leagues. In this game, as in all fantasy sports, players acting as virtual General Managers (GMs) run teams composed of real athletes that compete against each other in predetermined statistical categories. At the beginning of a season, players are distributed among the teams by means of one of several types of drafts. In the auction draft, GMs take turns nominating players to go up for bid in an ascending auction. The bidding is constrained by a fixed salary cap or budget that is used to pay for a manager's entire team.

At its core the fantasy football auction draft consists of a set of sequential ascending common-value



Tom Brady
QB
Week 9
24pts/week

0:30

Highest Bid $39
Highest Bidder Team 2
Their Est. Value 0pts
New Est.Value 24pts

Your Est.Value 0
New Est.Value 24

Position	Name	Bye Week	Est. Value
QB	Aaron Rodgers	6	30
QB	Drew Brees	7	27
WR	Calvin Johnson	6	27
QB	Tom Brady	8	26
CB	Darrelle Revis	6	21
DE	DeMarcus Ware	8	20
WR	Larry Fitzgerald	7	18
RB	Adrian Peterson	9	18
DL	Haloti Ngata	6	17
LB	Patrick Willis	8	15
LB	Terrell Suggs	6	12
RB	Maurice Jones-Drew	8	9
DE	Jared Allen	7	9

Position	Name	Bye Week	Est. Value
QB	----	--	--
RB	----	--	--
RB	----	--	--
WR	----	--	--
WR	----	--	--
TE	----	--	--
DEF	----	--	--
K	----	--	--
BENCH	----	--	--
BENCH	----	--	--
BENCH	----	--	--
BENCH	----	--	--
		Total Value	0

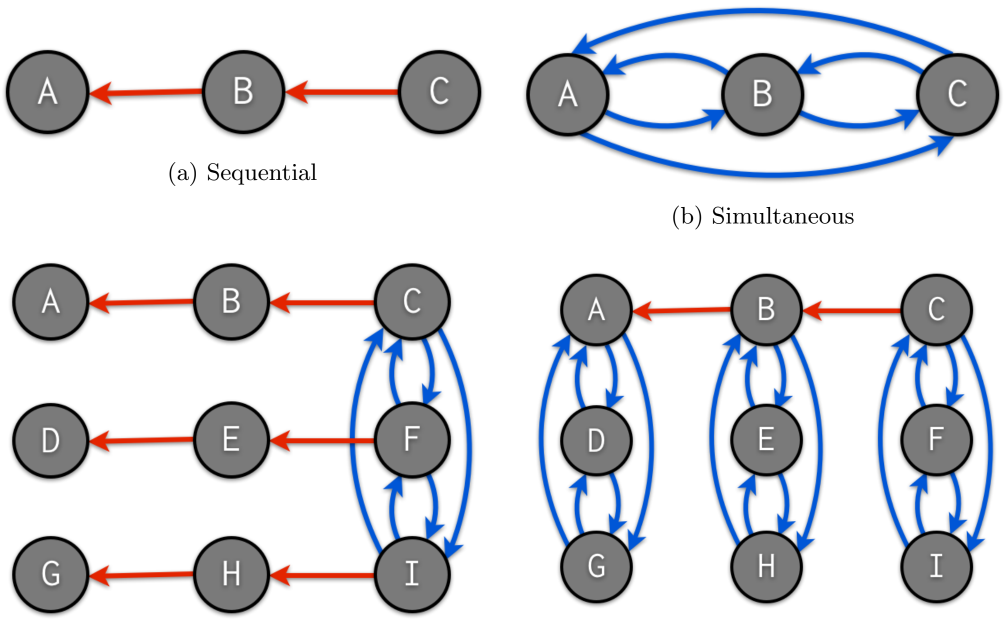
Figure 2: Fantasy football auction draft web interface. Here, bidders compete in sequential ascending auctions. The current player up for auction is in the center of the screen, the schedule of players going up for auction is shown in the left pane, and the bidder's current team is shown in the right pane.

auctions. The goods sold in these auctions are players, whose true value is unknown (as the season has not yet been played) but can be estimated based on their past performance. The managers attempt to assemble the best virtual football team as determined by their players' combined statistical performances. They are constrained not only by their budgets but also by a quota, which determines the number of players at each position that they can draft. A given draft generally consists of ten managers that compete in over one hundred consecutive auctions. The size of the auction and the complexity of the valuation functions for each manager make this an interesting problem to study. A screenshot of the web interface for the fantasy football auction draft is show in Figure 2.

3 General Auction Scheduling

Many of the games that JACK is used to simulate consist of multiple auctions executed according to some predetermined schedule. Some common types of auction scheduling include sequential, simultaneous, and sequential-simultaneous. In a sequential schedule, auctions are executed one after the other. Conversely, in a simultaneous schedule, auctions are executed at the same time. This often means that they not only start at the same time, but also that they end at the same time. In a sequential-simultaneous schedule sets of simultaneous auctions are executed sequentially. While these schedules are relatively simple, more complicated schedules can be found in the real world. For example, the Dutch Flower Auctions (DFAs) consist of several sequential schedules that are executed asynchronously.

Previously, the implementation of any auction schedule within JACK was the responsibility of the game designer. This resulted in inflexible simulations, whereby making small changes to the schedule required



(a) Sequential

(b) Simultaneous

(c) Simultaneous-sequential

(d) Sequential-simultaneous

Figure 3: Example dependency graphs for some common auction schedules. The nodes in each graph represent an auction, the red edges represent starting dependencies, and the blue edges represent ending dependencies.

making changes to the code itself. In addition, more complicated schedules, such as those where auctions are not executed synchronously, were nearly impossible to implement in the supporting framework. To solve these issues, this project generalizes the auction scheduling problem by representing all schedules as dependency graphs. This representation is flexible enough to describe many different types of complicated schedules. In addition, we have integrated an implementation of this scheduling mechanism into `JACK` that has many advantages such as simple configuration and multi-threaded execution.

3.1 Dependency Graph Representation

In our representation, an auction schedule is described by a directed graph with two different types of edges or dependencies. Each node of the graph represents a single auction. This could be a single sealed bid auction, ascending auction, descending auction, or even a multi-unit auction. The first type of dependency that can exist between two auctions is a starting dependency. An auction which has a starting dependency on another auction cannot be started before that auction has been completed. Sequential auctions can be described entirely by their starting dependencies as shown in Figure 3a. The second type of dependency that can exist between two auctions is an ending dependency. An auction which has an ending dependency on another auction cannot end until that auction has also ended. Two auctions which have ending dependencies on each other must then end at the same time. These types of dependencies are necessary to represent simultaneous auctions. In addition, simultaneous auctions must also have the same starting dependencies. An example of the dependency graph for simultaneous auctions is shown in Figure 3b. By combining both types of dependencies into a single graph we can obtain more complication schedules. Example of simultaneous-sequential and sequential-simultaneous auctions are shown in Figures 3c and 3d respectively.

3.2 Implementation

The implementation of general auction scheduling in `JACK` consists of two parts: dependency graph construction and dependency graph execution. Dependency graph construction takes an XML configuration file as input and outputs an adjacency list representation of the graph. Dependency graph execution takes this representation and executes each auction in a separate thread according to the constraints specified by the schedule.

The XML schema used to specify an auction execution schedule is designed to be simple and flexible. At the outermost level is the *schedule* element. This element contains one or more *task* elements, which are each associated with a specific auction. The auctions to which each task is referring are specified in a different location. This makes it easy to swap out different schedules with the same auctions at configuration time without the need to recompile code. Additionally, *task* elements can optionally contain one or more dependency elements. The *startDepend* elements explicitly specify the starting dependencies of an auction, and likewise the *endDepend* elements explicitly specify its ending dependencies. In this format the XML configuration almost exactly mirrors the adjacency list representation used to execute the schedule.

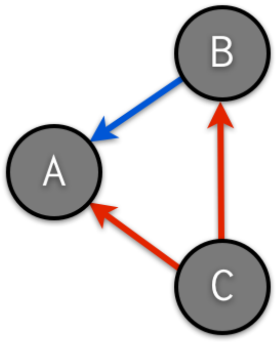
However, explicitly specifying dependencies for many simple auction schedules can be cumbersome. Our implementation provides an alternative to this approach by introducing the *sequential* and *simultaneous* XML elements. These two additional tags are used to implicitly imply dependencies on the tasks that they contain. Tasks encompassed in a sequential block are executed sequentially by adding a starting dependency from each task in the block to the task before it. Tasks encompassed by a simultaneous block are executed simultaneously by ensuring that they have identical starting and ending dependencies. The *sequential* and *simultaneous* elements can also be arbitrarily nested to form sequential-simultaneous, simultaneous-sequential, and other complex schedules. When both explicit and implicit dependencies are used in the schedule configuration, the explicit dependencies are applied first, and the implicit dependencies are applied second. Some example configurations that use both explicit and implicit dependencies are given in Figure 4.

Given the fully specified dependency graph of the auction schedule the `JACK` framework must now execute it. Our implementation takes a simple approach. At each iteration of the main execution loop, it searches for all of the auctions that have met their starting dependencies, and it starts them in their own thread. Then it searches for all auctions that have been started and met their ending dependencies, and it ends them. It repeats this process until either all auctions have finished or there are no auctions still running and no others that can be started. This approach has the advantage that every auction that has met its starting dependencies is started as soon as possible. It also guarantees to execute the schedule successfully as long as that schedule is not logically invalid.

4 Conclusion

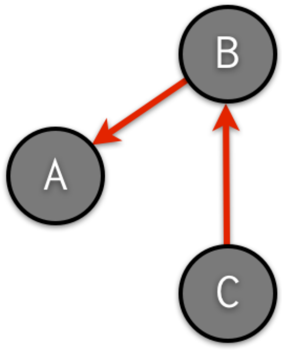
This project extended the `JACK` auction simulator in several key areas. First, it outlined a design for integrating web interfaces for human bidders into the existing framework. Second, it introduced the concept of general auction scheduling using dependency graphs. And finally, it provided an implementation of both of these improvements in the form of the fantasy football auction draft and a flexible and easy to configure scheduler. These additions have made `JACK` a more powerful and flexible tool for the researchers and educators interested in learning more about human behaviors in complex markets. However, this work is far from complete. The creation of simple to use interfaces for human bidders is only one step in the direction of performing large scale user studies in these domains. Future work in this area should continue to focus on these interfaces as well as the actual execution of user studies with them. In the area of auction scheduling, there is still a lot of work to be done. More efficient algorithms for executing schedules based on dependency graphs can certainly be achieved by topologically sorting these graphs. Simple problems such as how to validate the dependency graphs and how to integrate these schedules with addition types of dependencies (such as time) have not yet been implemented and are left as future work.

```
<schedule>
   <task auctionId="A"/>
   <task auctionId="B">
      <endDepend auctionId="A"/>
   </task>
   <task auctionId="C">
      <startDepend auctionId="A"/>
      <startDepend auctionId="B"/>
   </task>
</schedule>
```



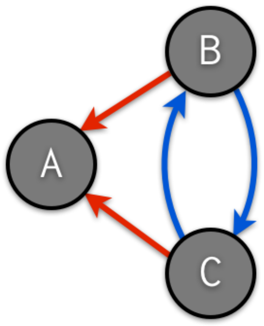
(a) Explicit dependencies

```
<schedule>
   <sequential>
      <task auctionId="A"/>
      <task auctionId="B"/>
      <task auctionId="C"/>
   </sequential>
</schedule>
```



(b) Implicit dependencies

```
<schedule>
   <task auctionId="A"/>
   <simultaneous>
      <task auctionId="B">
         <startDepend auctionId="A"/>
      </task>
      <task auctionId="C"/>
   </simultaneous>
</schedule>
```



(c) Explicit and implicit dependencies

Figure 4: This figure depicts several XML schedule configurations and their corresponding dependency graphs. When building the graph from the configuration in 4c, the explicit starting dependency of auction B on A is added first. Next the implicit dependencies are added to enforce the constraint that auction B and auction C are simultaneous. This includes adding an additional starting dependency to C, as well as making B and C ending dependencies of each other.

References

[1] T. Goff, A. Greenwald, E. Hilliard, W. Ketter, and E. Sodomka. Jack: A java auction configuration kit. *AAMAS-12 Workshop on Agent-Mediated Electronic Commerce (AMEC) and Trading Agent Design and Analysis (TADA)*, June 2012.

[2] Robert Dorsey and Laura Razzolini. Explaining overbidding in first price auctions using controlled lotteries. *Experimental Economics*, 6(2):123–140, 2003.

[3] R Mark Isaac. Just who are you calling risk averse? *Jornal of Risk and Uncertainty*, 20(2):177–187, 2000.

[4] John H Kagel, Dan Levin, and Arps Hall. Auctions: A survey of experimental research, 1995-2008. *Handbook of Experimental Economics*, 2, 2008.

[5] Winter Mason and Siddharth Suri. Conducting behavioral research on Amazons Mechanical Turk. *Behavior research methods*, 44(1):1–23, 2011.