Intelligent Agents and Robotic Systems

Ascending auction algorithm for decentralized scheduling

Petros Mitseas

December 28, 2021

Abstract

In this assignment we examine the application of the ascending auction algorithm in solving a certain task allocation problem, known as the scheduling problem. We introduce the theoretical background of the algorithm, as well as experimental results and comparisons with other algorithms in terms of optimality and speed.

1 Introduction

Task allocation is a common problem found in everyday life. Examples include organizing a project's plan to meet specific deadlines or allocating CPU time for executing user programs. The problem essentially boils down to orchestrating the allocation of a shared resource used to carry out tasks, in such a way that the value produced by completing these tasks is maximized. The word *value* is used here in a generic fashion and depends on the kind of application. Many types of allocation problems exist: Tasks may require single or multiple time units to be completed, with or without deadlines and there may be multiple agents competing for the resource.

In the next paragraph we formulate a specific type of allocation problem called the *Scheduling problem*. In section 3 we introduce the *Ascending Auction* algorithm Wellman et al. [2001] and the relationship between the scheduling problem and the Competitive equilibrium. In section 4 we apply the algorithm to an actual scenario and present the experimental results.

2 The Scheduling Problem

This problem considers the allocation of a single shared resource among multiple agents. Each agent $a_i \in A$ (where A is the agent set) has a unique task to accomplish that requires multiple consecutive time slots. The task also has a deadline and a value gained for its completion. The task is considered completed if all of its required time units have been allocated, and the deadline is not due. Therefore each agent and its corresponding task can be described with the following tuple:

$$agent_i = (duration_i, deadline_i, value_i) \tag{1}$$

The shared resource has a number of available time slots T to be allocated to the agents. A $slot_i \in T$ can be occupied only by a single agent. A slot can be represented by the following tuple:

$$slot_i = (agent, bid_i) \tag{2}$$

where $agent \in A$ is the allocated agent and bid_i a value which is used by the ascending auction process. bid_i is initialized with a entry value $entry_i$.

Table 1: A configuration of agents and tasks

Agent	Duration	Deadline	Value
0	4	4	24
1	2	8	16
2	5	6	6

Suppose that C is the set of agents that have been granted their corresponding consecutive slots on the shared resource, where $C \subseteq A$. There may also be time slots in $Q \subseteq T$ that haven't been allocated to an agent. The total value of the allocation is:

$$V(C) = \sum_{i \in C} value_i + \sum_{i \in Q} entry_i \tag{3}$$

The goal is to find the allocation C^* that maximizes $V(C)$.

$$C^* = \arg\max_C V(C) \tag{4}$$

Table 2: A potential time slot allocation. This particular allocation is also the optimal. The total value is 35 (considering an entry price of 1 per slot)

Slot	Agent
0	0
1	0
2	0
3	0
4	1
5	1
6	-
7	-
8	-

3 Ascending Auction

Solving the problem of optimal allocation turns out to be NP-hard and therefore computationally expensive when it comes to large numbers. In this section we describe the ascending auction algorithm which can be used to provide sub-optimal solutions to the problem, in favor of speed and scalability.

3.1 The Algorithm

Consider S a subset of consecutive time slots. The utility of the agent is equal to the valuation minus the sum of bids for these time slots. At each iteration of the algorithm each agent calculates the best subset, which maximizes the utility.

$$S^* = \arg\max_S u(S) - \sum_{i \in S} bid_i \tag{5}$$

If the utility is positive then the agent increases the bid for these slots by a small number ϵ, and the slots are now assigned to this agent. The agents cannot withdraw their bids, which can only be replaced by higher ones. The process ends when no further changes occur.

Algorithm 1 The ascending auction algorithm

```
for slot_i ∈ T do
    bid_i ← entry_i                          ▷ Initialize bids with entry values
end for
for agent_i ∈ A do
    F_i ← ∅                                  ▷ F_i denotes the set of slots allocated to the agent i
end for
while F changes do
    for agent_i ∈ A do
        for slot_j ∈ T do                    ▷ Raise the bid by a small value, for unallocated slots
            if slot_j ∈ F_i then
                p_j ← bid_j
            else
                p_j ← bid_j + ε
            end if
        end for
        S* ← arg max_S value_i − Σ_{j∈S} p_j     ▷ Find subset that maximizes utility
        for slot_j ∈ S* do                   ▷ Update bids and remove other agent's allocations for the slots
            bid_j ← p_j
            for k ∈ A − {i} do
                if slot_j ∈ F_k then
                    F_k ← F_k − {slot_j}
                end if
            end for
        end for
        F_i ← S*                             ▷ Allocate the slots to agent i
    end for
end while
```

3.2 Competitive Equilibrium

The definition of Competitive Equilibrium as stated in Shoham and Leyton-Brown [2008], suggests that a feasible assignment S and a price vector p are in competitive equilibrium when for every pairing $(i, j) \in S$: $\forall k, u(i, j) \geq u(i, k)$, where

- $u(i, j) = v(i, j)$ - p_j
- $v(i, j)$, the valuation of agent i for resource j
- p_j, the price of resource j

If a feasible assignment S and a price vector p satisfy the competitive equilibrium condition then S is an optimal assignment.

The competitive equilibrium can be generalized to fit the scheduling problem as such: Given a scheduling problem, a solution F is in competitive equilibrium at prices p if and only if:

- For all $i \in A : F_i = \arg\max_S value_i - \sum_{j \in S} p_j$
- For all time slots $j \in F_\emptyset : p_j = entry_j$
- For all time slots $j \notin F_\emptyset : p_j \geq entry_j$

Table 3: A competitive equilibrium, with parameters $entry_i = 1.00, \epsilon = 0.25$

Slot	Agent	Bid
0	0	1.25
1	0	1.25
2	0	1.25
3	0	2.50
4	1	1.00
5	1	1.00
6	-	1.00
7	-	1.00
8	-	1.00

Consider the setup of Table 1. A potential allocation that arises when applying the ascending auction algorithm is shown in Table 3. We can verify that this allocation satisfies the competitive equilibrium:

- Agent 0 gets a utility of $24 - (1.25 + 1.25 + 1.25 + 2.50) = 17.75$. There is no other subset that offers a greater utility for agent 0.
- Agent 1 gets a utility of $16 - 2 = 14$
- For agent 2 there is no subset that satisfies (5)
- The price of all unallocated slots is equal to the entry price.

The total value of the solution is equal to 43.

Although the allocation that satisfies a Competitive Equilibrium is optimal, the latter doesn't always exist. Consider the following agents and a shared resource of 2 slots:

Table 4: A configuration of agents and tasks with no equilibrium

Agent	Duration	Deadline	Value
0	2	2	4
1	1	2	3

If agent 0 has placed bids in both slots, then the sum cannot surpass the agent's valuation i.e. 4. This means that one slot will definitely have a bid ≤ 2. Therefore agent 1 is eligible to claim this slot, placing a bid ≥ 2. The utility of agent 0 is now negative thus breaking the equilibrium.

3.3 Optimality

Since a competitive equilibrium may not exist in an agent configuration, applying the ascending auction algorithm may produce suboptimal solutions. In the case of 4 the algorithm completes with the following solution:

Table 5: Applying the ascending auction, with parameters $entry_i = 1.00, \epsilon = 0.25$

Slot	Agent	Bid
0	0	2.25
1	1	2.00

The equation (3) gives a total value of 2 for this allocation. If agent 0 was allocated both slots, the total value would be equal to 4. Clearly we can observe that the allocation produced by the algorithm was far from optimal.

3.4 Convergence

Though optimality is not guaranteed, the algorithm does manage to converge in a predictable manner. At each iteration each agent that bids for a slot, increases its price by ϵ. At some point the price becomes high enough so that any subset containing the slot is not affordable. Therefore other agents would settle to other subsets or choose the empty subset. The running time of the algorithm is bounded as such:

$$O(n \max_{F_i} \frac{\sum_{i \in A} v_i(F_i)}{\epsilon}) \tag{6}$$

, where $v_i(F_i)$ is the total value of the allocation F_i of agent i and n the number of agents.

4 Experimental Results

In this section we evaluate the performance of the ascending auction algorithm under actual tests. We are interested in the following metrics:

- Solution optimality, comparison with brute force and other algorithms
- Running time
- Inter-agent communication

The tests were performed on a single machine in Python. Each agent has a single task, where the duration and deadline are sampled randomly from a uniform distribution between 1 and 10.

4.1 Comparison with other algorithms

We compare the ascending auction algorithm with the Weighted Job Schedule, Priority First and Brute Force approaches.

4.1.1 Weighted Job Schedule

For each agent, assume that the task with $end\_time = deadline$ and $start\_time = deadline - duration$. The objective is to find the resource allocation that maximizes the sum of utilities. The algorithm uses Dynamic Programming to solve the problem, using the following condition:

- Given $S : \{s_0, s_1, ..., s_n\}$: A sorted list of agent tasks, based on deadline (Earliest first).
- $p(j)$: The largest integer such that $finish_{p(j)} < start_j$, where $start_j$ is the start time of task j, and $finish_j$ the finish time. In other words, this number is the index of the latest task that can be allocated without overlapping with task j.
- The optimal allocation $opt(i)$ is given by the recursive equation (for $i \neq 0$):

$$opt(i) = \max \begin{cases} opt(i-1) & \text{(don't include task i)} \\ value_i + opt(p(i)) & \text{(include task i)} \end{cases} \tag{7}$$

, and $opt(0) = 0$

4.1.2 Priority First

This algorithm follows a greedy approach to solving the problem:

- Sort tasks based on utility earned (Highest first).
- Starting with the task with the highest utility, find all slot subsets that can be allocated to the task.
- Find the subset that is closest to the deadline and hasn't been already allocated to another agent. Allocate it to the current agent.
- Continue the process for all agents.

4.1.3 Brute Force

The algorithm works by generating all the potential allocations and selecting the best one. Obviously the algorithm has an exponential complexity. Therefore it's only used here to compute the optimal allocation for small problem sizes, in order to provide a baseline for measuring the other algorithms' performance.

4.2 Optimality

In the first experiment, we measure the error between the total value of the solution generated from the ascending auction algorithm versus the brute force algorithm (which provides the globally optimal solution). We also vary the number of agents at play.

Examining the error distribution, it is clear that the possibility of a non-optimal solution increases with the number of agents. The ascending auction behaves relatively consistently, in contrast with the weighted job first and priority-based algorithms.

Table 6: Results for 10 agents / 200 iterations

Algorithm	Mean score	% Error from optimal	Running time (seconds)
Brute force	56.91	0%	6.78
Ascending auction	53.05	7.94%	0.052
Priority first	46.36	18.67%	0.010
Weighted job schedule	45.65	18.54%	-

Figure 1: The mean difference from the optimal solution, expressed as percentage.

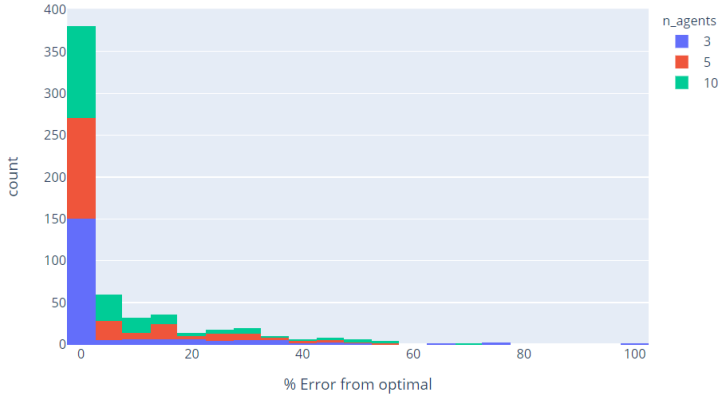

Figure 2: The error distribution between the ascending auction and the optimal solution, for different number of agents.

4.3 Messages exchanged

Ascending auction is a decentralized algorithm in the sense that every agent holds the logic for bidding and selecting time slots. Suppose that an agent transmits a message to the auction board every time it places a new bid (new allocation).

The experiment is run 200 times for every combination of agent number / slot number and the

total number of messages is displayed in the following heatmap.

The number of messages grows linearly with the number of agents. Therefore the algorithm can be scaled without heavily affecting the network traffic. Fewer slots result in more messages, because the agents are competing for less resources.

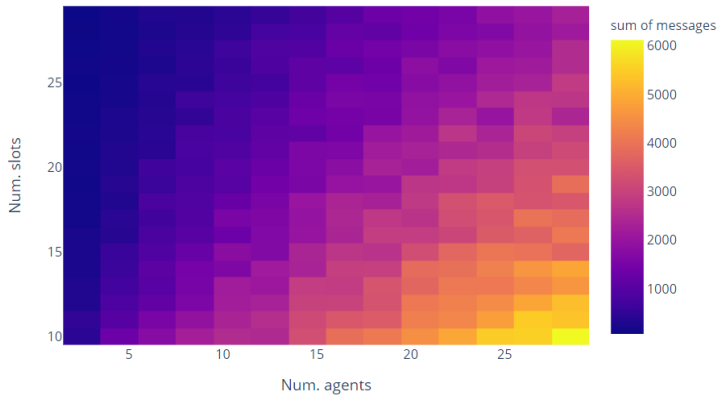

Figure 3: Number of messages sent from the agents. Fewer slots result in increased number of messages, due to the agents competing for fewer resources.

4.4 Running time

We measure the execution time of ascending auction with different number of agents. As shown in figure 4, the metric holds a linear relationship with the number of agents, as mentioned in paragraph 3.4.

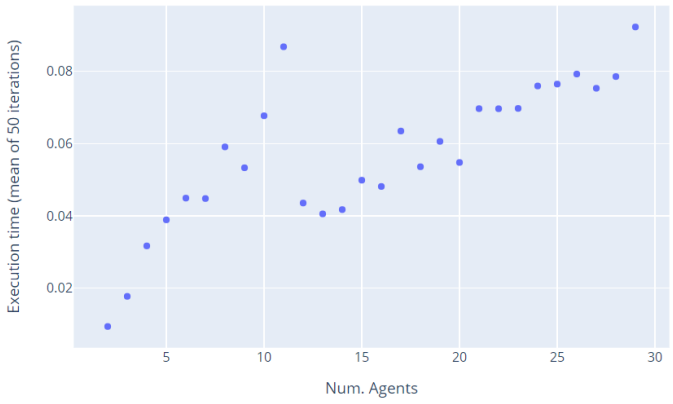

Figure 4: Running time versus number of agents.

Examining figure 5, it may seem that priority-first algorithm runs faster than ascending auction. While this is true when running on a single machine, it is worth noting that the ascending auction algorithm can easily run in a distributed fashion. In that case, each agent runs on a separate

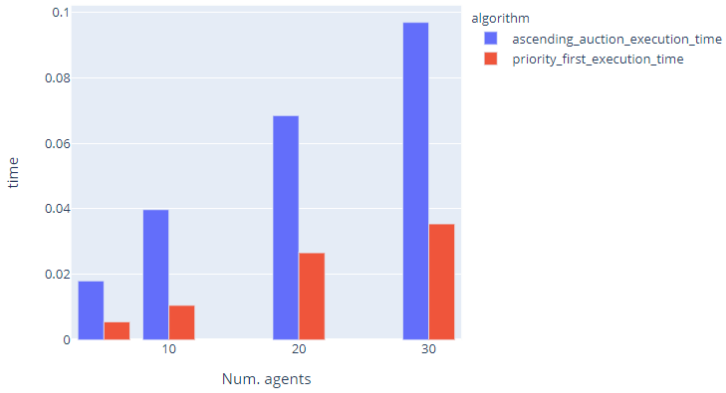

Figure 5: Comparison between ascending auction and priority first. Both scale linearly with the number of agents.

machine, thus the required computations are divided among the agents, resulting in much better performance.

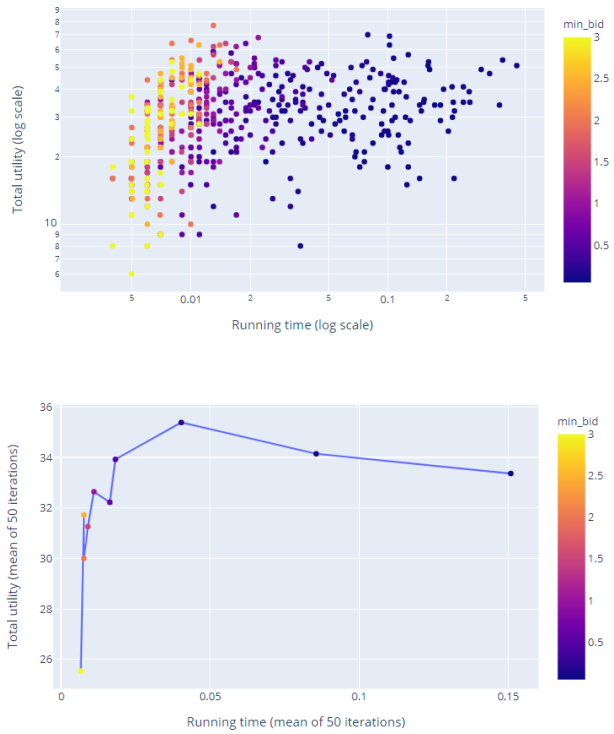

Figure 6: Total utility vs running time for different values of ϵ. The first figure shows the raw data from different experiments in log scale. The second figure summarizes the values by taking the mean value for each ϵ

4.5 Tuning the ϵ parameter

As described each agent, in its bidding round, raises the price of the chosen slots by ϵ. The value of this parameter, has an impact both on the convergence speed and the optimality of the solution. A small value dramatically increases the running time, whereas if the value is too high, the algorithm may miss the optimal solution.

In our experiments, the value $\epsilon = 0.25$ turned out to strike a balance between speed and optimality.

5 Conclusion

Ascending auction is a distributed algorithm that can be applied to solve the scheduling problem. While optimality is not guaranteed, in our experiments we observed that most of times the algorithm converges to an adequate solution. Combined with the distributed nature and easy scaling, the algorithm stands as a strong alternative among others, in solving such type of problems.

You may find the code used for the experiments, in the GitHub repo: https://github.com/petros94/ascending-auction

References

Michael P Wellman, William E Walsh, Peter R Wurman, and Jeffrey K MacKie-Mason. Auction protocols for decentralized scheduling. *Games and economic behavior*, 35(1-2):271–303, 2001.

Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.