Chapter 6

Negotiation

A negotiation problem is one where multiple agents try to come to an agreement or **deal**. Each agent is assumed to have a preference over all possible deals. The agents send messages to each other in the hope of finding a deal that all agents can agree on. These agents face an interesting problem. They want to maximize their own utility but they also face the risk of a break-down in negotiation, or expiration of a deadline for agreement. As such, each agent must negotiate carefully, trading off any utility it gains from a tentative against a possibly better deal or the risk of a breakdown in negotiation.

DEAL

decide whether the current deal is good enough or whether it should ask for more and risk agreement failure.

Automated negotiation can be very useful in multiagent systems as it provides a distributed method of aggregating distributed knowledge. That is, in a problem where each agent has different local knowledge negotiation can be an effective method for finding the one global course of action which maximizes utility without having to aggregate all local knowledge in a central location. In fact, the metaphor of autonomous agents cooperating in this manner to solve a problem that cannot be solved by any one agent, due to limited abilities or knowledge, was the central metaphor from which the field of distributed artificial intelligence, later known as multiagent systems, emerged (Davis and Smith, 1983). The metaphor is based on the observation that teams of scientists, businesses, citizens, and others regularly negotiate over future courses of action and the result of these negotiations can incorporate more knowledge than any one individual possesses (Surowiecki, 2005). For example, in a NASA rover mission to Mars the various engineering teams and science teams negotiate over what feature to include in the rover. The scientists are concerned with having the proper equipment in Mars so that they can do good science while the engineers try to ensure that everything will work as expected. Often it is the case that one side does not understand exactly why the other wants or rejects a particular feature but by negotiating with each other they arrive at a rover that is engineered solidly enough to survive the trip to Mars and has enough equipment to do useful science while there. Thus, negotiation results in the aggregation of knowledge from multiple individuals in order to make decisions which are better, for the whole, than if they were made by any one individual.

See (Squyres, 2005) for the full story on the MER mission to Mars. An exciting read.

6.1 The Bargaining Problem

A specific version of the negotiation problem has been studied in game theory. It is known as the **bargaining problem** (Nash, 1950). In the bargaining problem, we say that each agent i has a utility function u_i defined over the set of all possible deals Δ. That is, $u_i : \Delta \rightarrow \Re$. We also assume that there is a special deal δ^- which is the no-deal deal. Without loss of generality we will assume that for all agents $u_i(\delta^-) = 0$ so that the agents will prefer no deal than accepting any deal with negative utility. The problem then is finding a protocol f which will lead the agents to the best deal. But, as with all the game theory we have studied, it is not obvious which deal is the best one. Many solutions concepts have been proposed. We provide an overview of them in the next sections.

BARGAINING PROBLEM

See (Osborne and Rubinstein, 1999, Chapter 7) or (Osborne and Rubinstein, 1990) for a more extended introduction to bargaining.

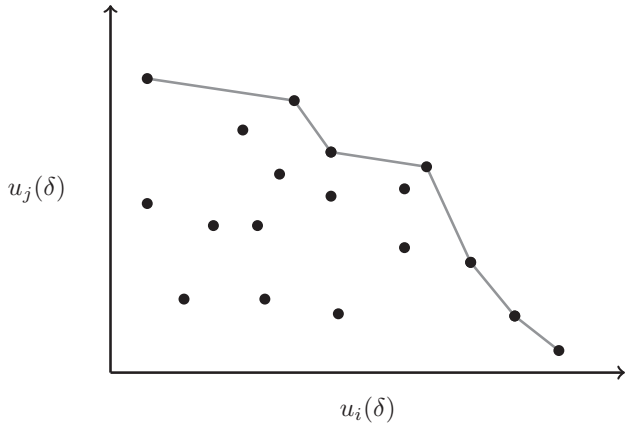


Figure 6.1: The dots represent possible deals. The deals on the gray line are Pareto optimal. The line is the Pareto frontier.

6.1.1 Axiomatic Solution Concepts

AXIOMATIC

An **axiomatic** solution concept is one which we can formally describe and then simply declare it to be our most desirable solution concept simply because we believe it satisfies all the important requirements. There are many possible requirements we might want to impose on a possible solution deal. The most obvious one is Pareto optimality.

Definition 6.1 (Pareto optimal)**.** *A deal δ is Pareto optimal if there is no other deal such that everyone prefers it over δ. That is, there is no δ' such that*

$$\forall_i u_i(\delta') > u_i(\delta).$$

This is an obvious requirement because if a deal is not Pareto optimal then that means that there is some other deal which all of the agents like better. As such, it does not make sense to use the non-Pareto deal when we could find this other deal which everyone prefers.

With only two agents, i and j, we can visualize the set of Pareto-optimal deals by using an xy-plot where the x and y coordinates are the utilities each agent receives for a deal. Each deal then becomes a point in the graph, as seen in figure 6.1. The **pareto frontier** is represented by the line shown in the upper-right which connects all the Pareto deals.

PARETO FRONTIER

We might also want a solution that remains the same regardless of the magnitude of an agent's utility values.

Definition 6.2 (Independence of utility units)**.** *A negotiation protocol is independent of utility units if when given U it chooses δ and when given $U' = \{(\beta_1 u_1, \ldots, \beta_I u_I) : u \in U\}$ it chooses δ' where*

$$\forall_i \; u_i(\delta') = \beta_i u_i(\delta).$$

That is, if an agent in a protocol that is independent of utility units used to get a utility of 10 from the result deal and now has multiplied its utility function by 5 then that agent will now get a utility of 50 from the new resulting deal. So, the utilities the agents receive remain proportional under multiplication.

Definition 6.3 (Symmetry)**.** *A negotiation protocol is symmetric if the solution remains the same as long as the set of utility functions U is the same, regardless of which agent has which utility.*

That is, if two agents swapped utility function then they also end up swapping the utilities they get from the resulting deal. In other words, the specific agents do not matter, the only thing that matters are the utility functions.

Definition 6.4 (Individual rationality)**.** *A deal δ is individually rational if*

$$\forall_i \; u_i(\delta) \geq u_i(\delta^-).$$

So, a deal is individual rational if $u_i(\delta) \geq 0$ since we will be assuming that $u_i(\delta^-) = 0$. That is, a deal is individually rational if all the agents prefer it over not reaching an agreement.

Definition 6.5 (Independece of irrelavant alternatives)**.** *A negotiation protocol is independent of irrelevant alternatives if it is true that when given the set of possible deals Δ it chooses δ and when given $\Delta' \subset \Delta$ where $\delta \in \Delta'$ it again chooses δ, assuming U stays constant.*

That is, a protocol is independent of irrelevant alternative if the deal it chooses does not change after we remove a deal that lost. Only removal of the winning deal changes the deal the protocol chooses.

Given these requirements we can now consider various possible solutions to a negotiation problem. In the **egalitarian solution** the gains from cooperation are split equally among the agents. That is, the egalitarian deal is the one where all the agents receive the same utility and the sum of their utilities is maximal, that is

EGALITARIAN SOLUTION

$$\delta = \arg\max_{\delta' \in E} \sum_i u_i(\delta') \tag{6.1}$$

where E is the set of all deals where all agents receive the same utility, namely

$$E = \{\delta \,|\, \forall_{i,j} u_i(\delta) = u_j(\delta)\}.$$

egalitarian n: one who believes in the equality of all people.

We can find the egalitarian solution visually for the two agent case by simply drawing the line $y = x$ and finding the deal on this line that is farther away from the origin, as seen in figure 6.2. Note that the egalitarian deal in this case is not Pareto optimal. However, if we allowed all possible deals (the graph would be solid black as every pair of x, y coordinates would represent a possible deal) then the egalitarian deal would also be a Pareto deal. Specifically, it would be the point on the $y = x$ line which marks the intersection with the Pareto frontier. Also note that the egalitarian deal does not satisfy the independence of utility units requirement since it assumes the utility units for the agents are comparable, in fact, it assumes that all agents' utility functions use the same units.

A variation on the pure egalitarian deal is the **egalitarian social welfare** solution which is the deal that maximizes the utility received by the agent with the lowest utility. That is, it is the deal δ that satisfies

EGALITARIAN SOCIAL WELFARE

$$\delta = \arg\max_{\delta} \min_i u_i(\delta). \tag{6.2}$$

The egalitarian social welfare solution is especially useful in scenarios where no deal exists which provides all agents with the same utility since every problem is guaranteed to have an egalitarian social welfare solution. However, the solution itself can in some cases seem very un-egalitarian. For example, a deal where two agents receive utilities of 10 and 100 respectively is the egalitarian social welfare solution even when another deal exists which gives the agents 9 and 11 respectively.

The **utilitarian solution** is the deal that maximizes the sum of the utilities, that is

UTILITARIAN SOLUTION

$$\delta = \arg\max \sum_i u_i(\delta). \tag{6.3}$$

The utilitarian deal is, by definition, a Pareto optimal deal. There might be more than one utilitarian deals in the case of a tie. The utilitarian deal violates the independence of utility units assumption as it also assumes utilities are comparable.

utilitarian n : someone who believes that the value of a thing depends on its utility.

We can find the utilitarian deal visually for the two agent case by drawing a line with slope of -1 and, starting at the top right, moving it perpendicular to $y = x$ until it intersects a deal. The first deals intersected by the line are utilitarian deals, as shown in figure 6.3.

The utilitarian and egalitarian solutions both seem like fairly reasonable solutions. Unfortunately, both violate the independence of utility units assumption. It

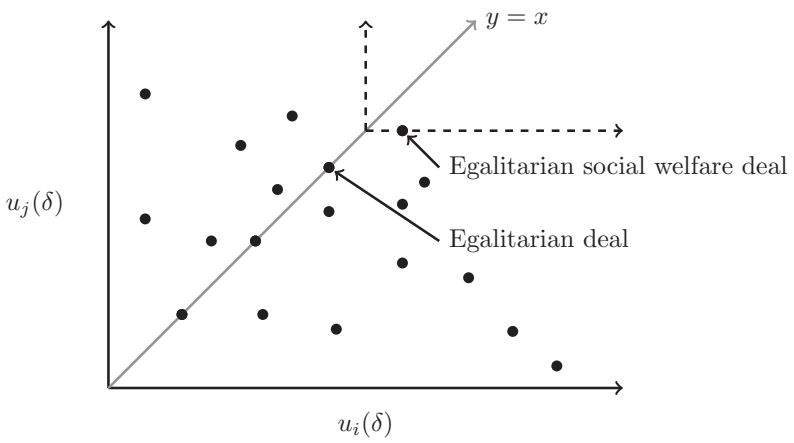


Figure 6.2: Egalitarian and egalitarian social welfare deals. The egalitarian social welfare deal can be found by moving the perpendicular dashed lines from a point high in the $y = x$ line down to $0, 0$ until they touch the first deal.

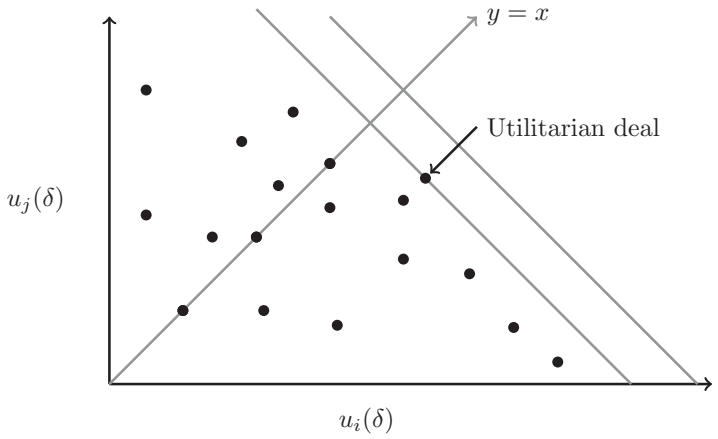


Figure 6.3: Utilitarian deal. The line $y = b - x$ starts with a very large b which is reduced until the line intersects a deal.

was Nash who proposed a solution which does not violate this assumption. The **Nash bargaining solution** is the deal that maximizes the product of the utilities. That is,

Nash bargaining solution

$$\delta = \arg\max_{\delta'} \prod u_i(\delta'). \tag{6.4}$$

The Nash solution is also Pareto efficient (Definition 6.1), independent of utility units (Definition 6.2), symmetric (Definition 6.3), independent of irrelevant alternatives (Definition 6.5). In fact, it is the *only* solution that satisfies these four requirements (Nash, 1950). This means that if we want a solution that satisfies these four requirements then our only choice is the Nash bargaining solution.

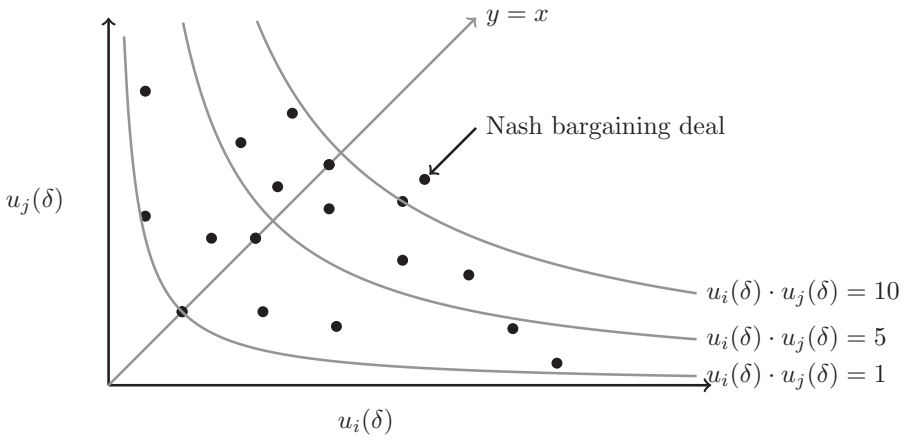


Figure 6.4: Nash bargaining deal.

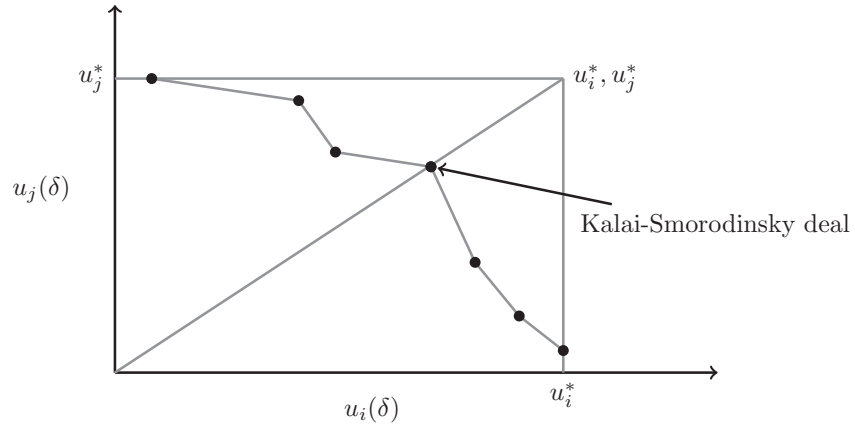


Figure 6.5: Kalai-Smorodinsky bargaining deal. We only show deals on the Pareto frontier.

Figure 6.4 shows a visualization of the Nash bargaining deal. Each curve represents all pairs of utilities that have the same product, that is, the line $y = c/x$. As we move northeast, following the line $y = x$, we cut across indifference curves of monotonically higher products. As such, the last deal to intersect a curve is the one which maximizes the product of the utilities, so it is the Nash bargaining solution.

Yet another possible solution is to find the deal that distributes the utility in proportion to the maximum that the agent could get (Kalai and Smorodinsky, 1975). Specifically, let u_i^* be the maximum utility that i could get from the set of all deals in the Pareto frontier. That is, the utility that i would receive if he got his best possible deal from the Pareto frontier. Then, find the deal that lies in the line between the point δ^- and the point (u_i^*, u_j^*). This solution is known as the **Kalai-Smorodinsky solution**. If all deals are possible then this solution is Pareto optimal. However, if there are only a finite set of possible deals then this solution might not exist. That is, there might not be a point in the specified line. This is a serious drawback to the use of the Kalai-Smorodinsky solution in a discrete setting.

KALAI-SMORODINSKY SOLUTION

The Kalai-Smorodinsky solution, like the Nash bargaining solution, is also independent of utility units requirement. On the other hand, it is not independent of irrelevant alternatives.

There is no general agreement as to which one of these axiomatic solutions is better. The social sciences try to develop experiments that will tell us which one of the solutions, if any, is arrived at by humans engaged in negotiation. Others try to justify some of these solutions as more fair than others. We, as multiagent designers, will probably find that all of them will be useful at some point, depending on the requirements of the system we are building.

6.1.2 Strategic Solution Concepts

Another way to think about what solution will be arrived at in a bargaining problem is to formalize the bargaining process, assume rational agents, and then determine their equilibrium strategies for their bargaining process. That is, define the solution concept to be the solution that is reached by automated rational agents in a bargaining problem. This method does raise one large obstacle: we need to first formally define a bargaining process that allows all the same "moves" as real-life bargaining. If you have ever haggled over the price of an item then you know that it is impossible to formalize all the possible moves of market-vendor. We need a negotiation protocol that is simple enough to be formally analyzed but still allows the agents to use most of their moves.

One such model is the **Rubinstein's alternating offers** model (Rubinstein, 1982). In this model two agents try to reach agreement on a deal. The agents can take actions only at discrete time steps. In each time step one of the agents proposes a deal δ to the other who either accepts it or rejects it. If the offer is rejected then

RUBINSTEIN'S ALTERNATING OFFERS

we move to the next time step where the other agent gets to propose a deal. Once a deal has been rejected it is considered void and cannot be accepted at a later time. The agents, however, are always free to propose any deal and to accept or reject any deal as they wish. We further assume that the agents know each other's utility functions.

The alternating offers models, as it stands, does not have a dominant strategy. For example, imagine that the two agents are bargaining over how to divide a dollar. Each agent wants to keep the whole dollar to itself and leave the other agent with nothing. Under the basic alternating offers protocol the agents' best strategy is to keep proposing this deal to the other agent. That is, each agent keeps telling the other one "I propose that I keep the whole dollar" and the other agent keeps rejecting this proposal. This scenario is not very interesting.

In order to make it more interesting, we further assume that time is valuable to the agents. That is, the agents' utility for all possible deals is reduced as time passes. For example, imagine that instead of haggling over a dollar the agents are haggling over how to split an ice cream sundae which is slowly melting and the agents hate melted ice cream. Formally, we say that agent i's utility at time t for deal δ is given by $\lambda_i^t u_i(\delta)$ where λ_i is i's discount factor, similarly the utility for j is $\lambda_j^t u_j(\delta)$. Thus, the agents' utility for every possible deal decreases monotonically as a function of time with a discount factor given by λ. Note that if $\lambda_i = 0$ then the agent must agree to a deal at time 0 since after that it will receive a utility of 0 for any deal. Conversely, if $\lambda_i = 1$ then the agent can wait forever without any utility loss.

Furthermore, lets assume that the agents' utilities are linear and complementary. Specifically, imagine that the deal δ is simply a number between 0 and 1 and represents the amount of utility an agent receives so that $u_i(\delta) = \delta$ and $u_j(\delta) = 1 - \delta$. In this scenario, it can be shown that a unique subgame perfect equilibrium strategy exists.

Theorem 6.1 (Alternating Offers Bargaining Strategy)**.** *The Rubinstein's alternating offers game where the agents have complementary linear utilities has a unique subgame perfect equilibrium strategy where*

- *agent i proposes a deal*
$$\delta_i^* = \frac{1 - \lambda_j}{1 - \lambda_i \lambda_j}$$
and accepts the offer δ_j from j only if $u_i(\delta_j) \leq u_i(\delta_j^*)$,
- *agent j proposes a deal*
$$\delta_j^* = \frac{1 - \lambda_i}{1 - \lambda_i \lambda_j}$$
and accepts the offer δ_i from i only if $u_j(\delta_i) \leq u_j(\delta_i^*)$.

(Rubinstein, 1982; Muthoo, 1999).

We can understand how these deals are derived by noting that since both agents have utilities that decrease with time the best deal will be reached in the first step. That means that each agent must propose a deal that the other will accept. Specifically, agent i must propose a deal δ_i^* such that $u_j(\delta_i^*) = \lambda_j u_j(\delta_j^*)$ because if it proposes a deal that gives j lower utility then j will reject it and if it proposes a deal that gives j higher utility then i is needlessly giving up some of its own utility. Conversely, j must propose a deal δ_j^* such that $u_i(\delta_j^*) = \lambda_i u_i(\delta_i^*)$. We thus have two equations.

Since $u_i(\delta) = \delta$ and $u_j(\delta) = 1 - \delta$, we can replace these definitions into the above equations to get

$$1 - \delta_i^* = \lambda_j (1 - \delta_j^*) \tag{6.5}$$
$$\delta_j^* = \lambda_i \delta_i^*. \tag{6.6}$$

MONOTONIC-CONCESSION

$\delta_i \leftarrow \arg\max_\delta u_i(\delta)$
Propose δ_i
Receive δ_j proposal
if $u_i(\delta_j) \geq u_i(\delta_i)$
    then Accept δ_j
    else $\delta_i \leftarrow \delta_i'$ such that $u_j(\delta_i') \geq \epsilon + u_j(\delta_i)$ and $u_i(\delta_i') \geq u_i(\delta^-)$
goto 2

Figure 6.6: The monotonic concession protocol.

Upon solving these two equations for δ_i^* and δ_j^* we get the equilibrium values from Theorem 6.1. More generally, it has been shown that a unique subgame perfect equilibrium exists even when the utility functions are not linear but are simply monotonic in δ.

The theorem tells us that the best strategy for these agents is propose a bid on the first time step which will be accepted by the other agent. This action makes sense because we know that utilities only decrease with time so the best possible deal will be had on the first time step. The value of the proposed deal, on the other hand, is very interesting. Notice that the deal i proposes depends only on the agents' discount factor, not on their utility values. The important thing is how fast each agent loses utility over time. For example, notice that if $\lambda_j = 0$ then i will propose $\delta_i^* = 1$. That is, i will propose to keep all the utility to himself. Agent j will accept this proposal since he knows that given that his $\lambda_j = 0$ if he waits for the next time step then he will receive utility of 0 in every possible deal.

Similar techniques have been used to show the existence of equilibrium in other types of bargaining games (Kraus, 2001). These games are all alternating offer games but assume different utility discounts such as a fixed loss utility function that is reduced by a constant amount each time and an interest rate utility which models the opportunity loss, among others. Depending on the type of utility discount and the type of game, some of these games can be proven to have a unique perfect equilibrium strategy that involves only one actions, others have multiple equilibria and thus would require some other coordination method, and yet others can be shown to be NP-complete in the time it takes to find the unique equilibrium. In general, however, the technique used to solve these bargaining games is the same.

1. Formalize bargaining as a Rubinstein's alternating offers games.
2. Generate extended-form game.
3. Determine equilibrium strategy for this game.

In practice, step 3 is often computationally intractable, at least for the general case of the problem.

6.2 Monotonic Concession Protocol

The simplest negotiation protocol following the Rubinstein's model of alternating offers is the **monotonic concession protocol**. In this protocol the agents agree to always make a counter offer that is slightly better for the other agent than its previous offer. Specifically, in monotonic concession the agents follow the algorithm shown in figure 6.6.

MONOTONIC CONCESSION PROTOCOL

Each agent starts by proposing the deal that is best for itself. The agent then receives a similar proposal from the other agent. If the utility the agent receives from the other's proposal is bigger than the utility it gets from its own proposal then it accepts it and negotiation ends. If no agreement was reached then the agent must propose a deal that is at least an increase of ϵ in the other agent's utility. If neither agent makes a new proposal then there are no more messages sent and

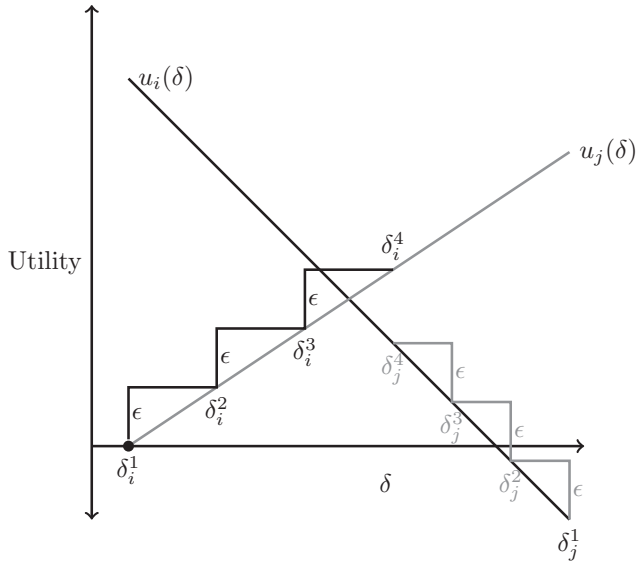


Figure 6.7: Monotonic concession protocol visualization. Both agents have linear utility functions over the set of deals. The superscripts indicate the time so δ_i^1 is i's proposal at time 1. At time 4 we have that $\delta_i^4 = \delta_j^4$ so $u_i(\delta_j^4) = u_i(\delta_i^4)$ and the agents agree on this deal.

negotiations fail, so they implicitly agree on the no-deal deal δ^-. The monotonic concession protocol makes it easy for the agents to verify that the other agent is also obeying the protocol. Namely, if i ever receives a proposal whose utility is less than a previous proposal then it knows that j is not following the protocol.

The protocol can be visualized as in figure 6.7. Here we see a simple example with linear utility functions where the agents reach an agreement (δ^4) after four time steps. Notice that the agreement reached is *not* the point at which the lines intersect. The monotonic concession protocol is not guaranteed to arrive at any particular axiomatic solution concept. All that it guarantees is that it will stop.

A common workaround is to assume its a zero-sum game and use the agent's own utility function.

Monotonic concession has several drawbacks. It can be very slow to converge. Convergence time is dictated by the number of possible deals, which is usually very large, and the value of ϵ. It is also impossible to implement this algorithm if the agents do not know the other agents' utility function. In practice, it is rare for an agent to know its opponent's utility function. Finally, the monotonic concession protocol has a tricky last step. Namely, the two agents could make simultaneous offers where each one ends up preferring the other agent's offer to the one it just sent. This is a problem as both of them now want to accept different offers. This can be solved by forcing the agents to take turns. But, if we did that then neither will want to go first as the agent that goes first will end up with a slightly worse deal.

6.2.1 The Zeuthen Strategy

The monotonic concession protocol we presented implements a simplistic strategy for the agent: always concede at least ϵ to the other agent. However, if an agent knows that the other one will always concede then it might choose to not concede at all. Smarter agents will examine their opponent's behavior and concede in proportion to how much they are conceding.

Zeuthen strategy

This idea is formalized in the **Zeuthen strategy** for the monotonic concession protocol. We start by defining the willingness to risk a breakdown in negotiations to be

$$\text{risk}_i = \frac{u_i(\delta_i) - u_i(\delta_j)}{u_i(\delta_i)}. \tag{6.7}$$

The agent can then calculate the risks for both agents. The Zeuthen strategy tells us that the agent with the smallest risk should concede just enough so that it does not have to concede again in the next time step. That is, the agent that has the least

ZEUTHEN-MONOTONIC-CONCESSION

```
1  δ_i ← arg max_δ u_i(δ)
2  Propose δ_i
3  Receive δ_j proposal
4  if u_i(δ_j) ≥ u_i(δ_i)
5      then Accept δ_j
6  risk_i ← (u_i(δ_i) − u_i(δ_j)) / u_i(δ_i)
7  risk_j ← (u_j(δ_j) − u_j(δ_i)) / u_j(δ_j)
8  if risk_i < risk_j
9      then δ_i ← δ'_i such that risk_i(δ'_i) > risk_j(δ'_j)
10         goto 2
11 goto 3
```

Figure 6.8: The monotonic concession protocol using the Zeuthen strategy.

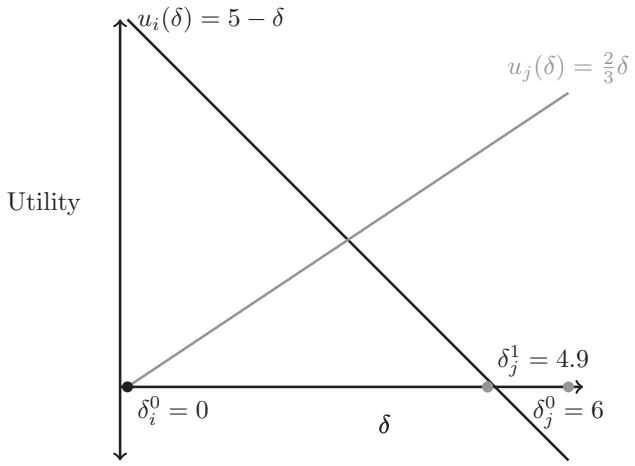


Figure 6.9: Visualization of the Zeuthen strategy at work. The initial proposals are $\delta_i^0 = 0$ and $\delta_j^0 = 6$.

to lose by conceding should concede. More formally, we define the new protocol as shown in figure 6.8.

Figure 6.9 shows a graphical representation of the first step in a Zeuthen negotiation. After the initial proposal the agents calculate their risks to be

$$\text{risk}_i^0 = \frac{5 - (-1)}{5} = \frac{6}{5}$$

and

$$\text{risk}_j^0 = \frac{4 - 0}{4} = 1.$$

Since j has a lower risk it must concede. The new deal must be such that j will not be forced to concede again. That is, it must insure that

$$\text{risk}_i = \frac{5 - (5 - \delta_j)}{5} < \frac{\frac{2}{3}\delta_j - 0}{\frac{2}{3}\delta_j} = \text{risk}_j$$

which simplifies to $\delta_j < 5$. As such, j can pick any deal δ less than 5. In the figure the agent chooses $\delta_j^1 = 4.9$.

The Zeuthen strategy is guaranteed to terminate and the agreement it reaches upon termination is guaranteed to be individually rational and Pareto optimal. Furthermore, it has been shown that two agents using the Zeuthen strategy will converge to a Nash bargaining solution.

Theorem 6.2 (Zeuthen converges to Nash solution)**.** *If both agents use the Zeuthen strategy they will converge to a Nash bargaining solution deal, that is, a deal that maximizes the product of the utilities (Harsanyi, 1965).*

ONE-STEP-NEGOTIATION
1 $E \leftarrow \{\delta \mid \forall_{\delta'} u_i(\delta)u_j(\delta) \geq u_i(\delta')u_j(\delta')\}$
2 $\delta_i \leftarrow \arg\max_{\delta \in E} u_i(\delta)$
3 Propose δ_i
4 Receive δ_j
5 **if** $u_i(\delta_j)u_j(\delta_j) < u_i(\delta_i)u_j(\delta_i)$
6 **then** Report error, j is not following strategy.
7 Coordinate with j to choose randomly between δ_i and δ_j.

Figure 6.10: The one step negotiation protocol (Rosenschein and Zlotkin, 1994).

The proof of the theorem shows how the maximum of both agents' utilities monotonically increases until the protocol reaches an agreement. As such, if there is any other agreement that has higher utility than the one they agreed upon then that agreement would have been proposed in the negotiation. Therefore, there can be no deal with higher utility for any agent than the one they agreed on.

A problem with the Zeuthen strategy might arise in the last step if the agents' risks are exactly the same. Specifically, if both agents realize that their risks are the same and that the next proposal that either of them makes will be accepted by their opponent then both agents will want to wait for the other agent to concede. This is an example of the game of chicken. As such, selfish agents should play the Nash equilibrium strategy which, in this case, is a mixed strategy. Since it is a mixed strategy it means that there remains a possibility for both of them to decide to wait for the other one, thereby breaking down the negotiations when a solution was possible.

The Zeuthen strategy is also attractive because, when combined with the mixed Nash strategy we described, it is in a Nash equilibrium. That is, if one agent declares that it will be using the Zeuthen strategy then the other agent has nothing to gain by using any other negotiation strategy. As such, and agent that publicly states that it will be using the Zeuthen strategy can expect that every agent it negotiates with will also use the Zeuthen strategy.

6.2.2 One-Step Protocol

Once we have a multi-step protocol that (usually) reaches a particular solution, we immediately think about the possibility of skipping all those intermediate steps and jumping right to the final agreement. Specifically, we could define a protocol that asks both agents to send a proposal. Each agent then has two proposals: the one it made and the one it received. The agents must then accept the proposal that maximizes the product of the agents' utilities. If there is a tie then they coordinate and choose one of them at random.

Given the rules established by this protocol, an agent's best strategy is to find of the deals that maximize the product of the utilities and then, from these deals, propose the one that is best for the agent. The agent's algorithm for this **one-step negotiation** protocol is shown in figure 6.10. The strategy implemented by this algorithm is a Nash equilibrium. That is, if one agent decides to implement this algorithm then the other agent's best strategy is to also implement the same algorithm. Thus, when using perfectly rational agents it is always more efficient to use one-step negotiation rather than the more long winded monotonic concession protocol.

ONE-STEP NEGOTIATION

6.3 Negotiation as Distributed Search

If we take a step back and look at the bargaining problem, we realize that the problem of finding a chosen solution deal is in effect a distributed search problem. That is, via their negotiations the agents are effectively searching for a specific solution. A simple and natural way to carry out this search is by starting with some

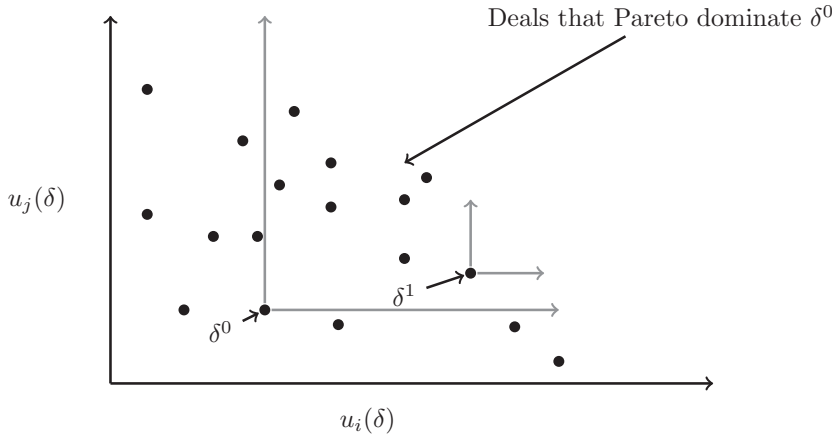


Figure 6.11: Hill-climbing on the Pareto landscape. The quadrant to the top and right of δ^0 contains all the deals that dominate δ^0, similarly for δ^1. There are no deals that dominate δ^1 and yet it is not the social welfare or Nash bargaining solution.

initial deal and at each time step moving to another deal. The process is repeated until there is no other deal which the agents can agree on.

For example, figure 6.11 shows an initial deal δ^0 along with all the other possible deals for a negotiation problem. Any deal that is above and to the right of δ^0, as delineated by the gray lines, dominates δ^0. This means that any deal in this quadrant is preferred by all the agents. Thus, we can expect selfish agents to cheerfully accept any one of those deals. A simple hill-climbing search process will continue moving in this up-and-right direction.

However, the search might get stuck in a local optima. For example, suppose that we go directly from δ^0 to δ^1 as shown in the figure. This new deal is not dominated by any other deal, thus our simple hill-climbing algorithm is now stuck at that deal. Notice also that this deal is not the social welfare maximizing deal nor is it the Nash bargaining deal. Thus, simple hill-climbing among selfish agents can lead us to sub-optimal solutions.

In real applications this search problem is further compounded by the fact that it is often impossible to move directly from any one deal to any other deal because the number of possible deals is simply too large to consider or there are other external limitations imposed on the agents. That is, most applications overlay an undirected graph to the bargaining problem. In this graph the nodes represent the possible deals and an edge between two nodes denotes the fact that it is possible to move between those two deals. Given such a graph, it is possible that the agents would find themselves in one deal that is pareto dominated by a lot of other deals but which are unreachable in one step because there is no edge between those deals and the current deal. Thus, we have even more local optima.

Researchers have tried to circumvent this problem of local optima in bargaining via a number of different methods, as we shall see in the next few sections.

6.4 Ad-hoc Negotiation Strategies

ADEPT

TACTICS

Deployed multiagent negotiation systems, such as **ADEPT** (Faratin et al., 1998; Binmore and Vulkan, 1999), have implemented agents with ad-hoc negotiation strategies. In ADEPT, a handful of selected negotiation **tactics** were tested against each other to determine how they fared against each other. The basic negotiation model used was an alternating offers model with time discounts. The tactics the agents could use included: a *linear* tactic that concedes a fixed amount each time, a *conceder* tactic that concedes large amounts of utility on earlier time steps, and an *impatient* tactic which conceded very little at first and requested a lot. We can think of each one of these tactics as stylized versions of the negotiation strategies people might use in a negotiation.

This ad-hoc approach is often used by multiagent developers who need to build a negotiating multiagent system which achieves a good-enough solution and where

all the agents are owned by the same party and, thus, do not need to behave truly selfishly. It allows us to use somewhat selfish agents that encapsulate some of the system's requirements and then use the negotiation protocol to aggregate these individual selfish interests into a global solution which, although not guaranteed to be the utilitarian solutions, can nonetheless be shown via experimentation to be a good-enough solutions. Nonetheless, we must always remember that these systems are not to be opened up to outside competition. Once we know the other agents' ad-hoc tactics, it is generally very easy to implement an agent that can defeat those tactics, usually at the cost of a much lower system utility. That is, one renegade agent can easily bring down a system of ad hoc negotiating agents, unless the system has been shown to be at an evolutionary stable equilibrium.

6.5 The Task Allocation Problem

TASK ALLOCATION PROBLEM

A common problem in multiagent systems is deciding how to re-allocate a set of tasks among a set of agents. This is known as the **task allocation problem**. In this problem there is a set of tasks T, a set of agents, and a cost function $c_i : s \rightarrow \Re$ which tells us the cost that agent i incurs in carrying out tasks $s \subseteq T$. In some simplified versions of the problem we assume that all agents share the same cost function. Each agent starts out with a set of tasks such that all tasks are distributed amongst all agents. We can think of this initial allocation as δ^- since, if negotiations break down then each agent is responsible for the tasks it was originally assigned. Similarly, every allocation of tasks to agents is a possible deal δ where $s_i(\delta)$ is the set of tasks allocated to i under deal δ. The problem we then face is how to design a negotiation protocol such that the agents can negotiate task re-allocations and arrive at a final re-allocation of the tasks that is one of the axiomatic solution concepts, such as the utilitarian deal.

POSTMAN PROBLEM

An example of this type of problem is the **postman problem** in which a set of postmen are placed around a city each with a bag of letters that must be delivered. A postman prefers to have all his letters with delivery addresses that are very close to each other so as to minimize his delivery costs. The postmen can negotiate with each other in order to make this happen. That is, a postman can trade some of its letters with another postman. In this example the tasks are the letters to be delivered and the cost function is the distance traveled to deliver a set of letters. Different cost functions arise when the postmen have different final destinations, for example, if they are required to go to their respective homes after delivering all the letters, or if they prefer certain areas of town, etc.

Once we are given task allocation problem we must then decide how the agents will exchange tasks. The simplest method is for agents to pair up and exchange a single task. This means that not all deals are directly accessible from other deals since, for example, we can't go from the deal where j does 2 tasks and i does nothing to the deal where i does 2 tasks and j does nothing in one step. We can represent this constraint graphically by drawing an edge between every pair of deals that are reachable from one another by the exchange of a single task. An example of such a graph is shown in figure 6.12. The table in this figure shows the tasks assignments that each deal represents and the costs that each one of the agents incurs for carrying out various subsets of tasks. In order to have positive utility numbers, we let the utility to the agent be 8 minus the cost of carrying out the tasks. Utility is often simply the negative of the cost so these two are used interchangeably. The graph shows all the possible deals as points and the edges connect deals that can be reached from one another by moving a single task between agents.

If the agents are only willing to accept deals that are better for them than their current deal, that is, the accept only Pareto-dominant deals, then every deal in figure 6.12 forms a local maximum. For example, if the agents are in δ^1 then, while j would prefer any one of δ^2, δ^3, or δ^4, agent i would not accept any one of these because u_i is lower for all of them than for δ^1. Thus, if the agents find themselves in δ^1 and they are not willing to lower their utility they will remain there. That is,

δ	$s_i(\delta)$	$s_j(\delta)$	$c_i(\delta)$	$c_j(\delta)$	$u_i(\delta) = 8 - c_i(\delta)$	$u_j(\delta) = 8 - c_j(\delta)$
δ^1	$\emptyset$	$\{t_1, t_2, t_3\}$	0	8	8	0
δ^2	$\{t_1\}$	$\{t_2, t_3\}$	1	4	7	4
δ^3	$\{t_2\}$	$\{t_1, t_3\}$	2	5	6	3
δ^4	$\{t_3\}$	$\{t_2, t_3\}$	4	7	4	1
δ^5	$\{t_2, t_3\}$	$\{t_1\}$	6	4	2	4
δ^6	$\{t_1, t_3\}$	$\{t_2\}$	5	3	3	5
δ^7	$\{t_1, t_2\}$	$\{t_3\}$	3	1	5	7
δ^8	$\{t_1, t_2, t_3\}$	$\emptyset$	7	0	1	8

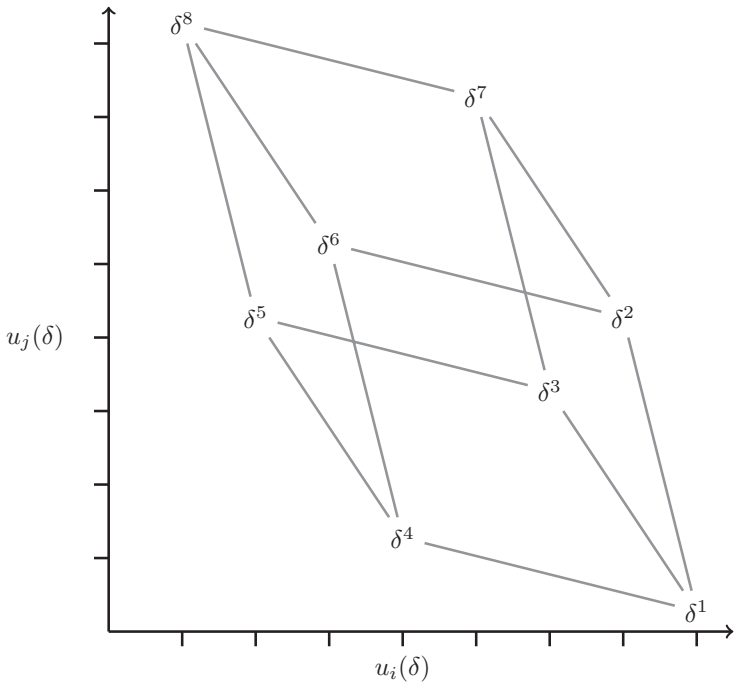


Figure 6.12: An example task allocation problem is shown in the table and its graphical representation as a bargaining problem is shown in the graph. The edges on the graph connect deals that can be reached by moving a single task between the agents. $s_i(\delta)$ is the set of tasks i has under deal δ.

δ^1 is a local maximum, as are all the other deals in this example.

6.5.1 Payments

One possible way to allow the agents to find deals that are closer to the utilitarian deal is by allowing them to use **monetary payments**. For example, agent i might convince j to take a deal that gives j less utility if i can sweeten the deal by giving j some money to take that deal. This basic idea was implemented in the **contract net protocol** (Smith, 1981; Davis and Smith, 1983). In contract net each agent takes the roles of either a contractor or contractee. The contractor has a task that it wants someone else to perform. The contractor is also willing to pay to get that task done. In the protocol, the contractor first announces that it has a task available by broadcasting a call for bids. All agents receive this call for bids and, if they want, reply to the contractor telling him how much they charge for performing the task. The contractor then chooses one of these bids, assigns the task to the appropriate agent and pays him the requested amount.

MONETARY PAYMENTS

CONTRACT NET PROTOCOL

For example, given the current task allocation δ agent i might find that one of his current tasks $t \in s_i(\delta)$ costs him a lot. That is, $c_i(s_i(\delta)) - c_i(s_i(\delta) - t)$ is a large number. In this case i will be willing to pay up to $c_i(s_i(\delta)) - c_i(s_i(\delta) - t)$ in order to have some other agent perform t—the agent will pay up to the utility he will gain from losing the task, any more than that does not make sense as the agent can simply do the task himself. Similarly, any other agent j will be willing

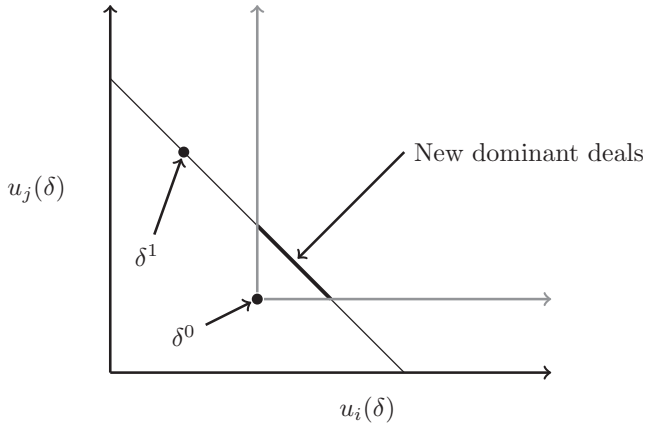


Figure 6.13: Deal δ^1 is turned into an infinite number of deals, represented by the line that intersects δ^1, with the use of payments. Some of those new deals Pareto dominate δ^0, as shown by the thicker part of the line.

to perform t as long as it gets paid an amount that is at least equal to any cost increase he will endure by performing the task, that is, the payment must be at least $c_j(s_j(\delta)) - c_j(s_j(\delta) + t)$. Note that in the general case some of these numbers could be negative, for example, an agent could actually have lower costs from performing an extra task. However, these sub-additive cost functions are rare in practice.

The use of monetary payments has the effect of turning one deal into an infinite number of deals: the original deal and the infinite number of payments, from $-\infty$ to ∞ that can be made between the agents. Figure 6.13 shows a graphical representation of this process. Here we see deal δ^1 transformed into the set of deals represented by the line going thru δ^1. The figure also shows us how this transformation creates new deals that dominate another existing deal δ^0. Thus, if the system was in δ^0 and we were doing hill-climbing then we would be stuck there as δ^1 does not dominate δ^0. But, if we used the contract net then agent j could make a payment to i, along with the task transfer, which would give i a higher total utility. This new task allocation plus payment is a deal that lies within the thick area of the line that intersects δ^1. Thus, contract net allows us to reach a task allocation that is the utilitarian solution. Unfortunately, the addition of payments does not guarantee that we will always reach the utilitarian solution, this depends on the particular characteristics of the cost function.

One type of cost functions that have been found to be easy to search over are additive cost functions where the cost of doing a set of tasks is always equal to the sum of the costs of doing each task individually.

ADDITIVE COST FUNCTION

Definition 6.6. *A function $c(s)$ is an **additive cost function** if for all $s \subseteq T$ it is true that*

$$c(s) = \sum_{t \in s} c(t).$$

In these scenarios it has been shown that the contract net protocol, or any other protocol that uses payments and always moves to dominant deals, will eventually converge to the utilitarian social welfare solution.

Theorem 6.3. *In a task allocation problem with an additive cost function where we only allow exchange of one task at a time, any protocol that allows payments and always moves to dominant deals will eventually converge to the utilitarian solution (Endriss et al., 2006).*

For example, figure 6.14 reproduces the previous figure 6.12 but now we draw some lines to represent all the possible deals that are possible from the various deals using payments. The new deals are represented by the parallel lines which intersect each of the original deals. It is easy to confirm that all deals except for the utilitarian deal δ^7 are dominated by an adjacent deal. Thus, if we continue moving to successively dominant deals we are guaranteed to end up at δ^7. The proof of Theorem 6.3 follows the same line of thought; it shows that any deal (with

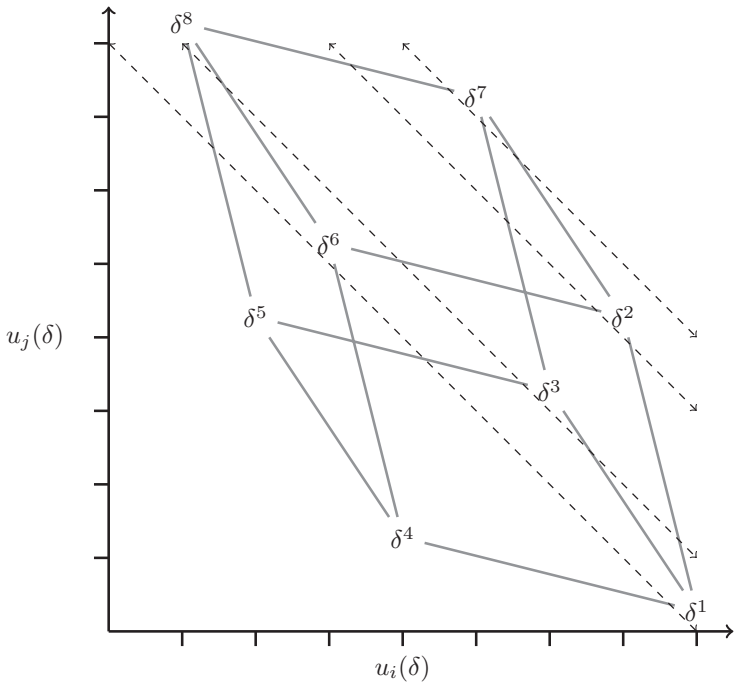


Figure 6.14: Monetary payments with an additive cost function. The deals expand to an infinite set of deals, represented by dashed lines intersecting the original deals. We show lines for deals δ^7, δ^2, δ^8, and δ^1 only. Note that the utilitarian deal δ^7 is the only one not dominated by any other deal.

payments) that is not dominated by some other deal must be the utilitarian social welfare solution.

If we allow arbitrary cost functions then there is very little we can say about which solution a hill-climbing protocol will reach, except for one scenario. In the case where every pair of deals is connected by an edge, we can get to any deal from every other deal in one step, imagine that the graph in figure 6.12 fully connected, then using payments guarantees we will converge to the utilitarian social welfare (Sandholm, 1997; Sandholm, 1998; Andersson and Sandholm, 1998). We can easily verify this by looking again at figure 6.14. Here we can see how each deal is converted into a line of slope -1 and the utilitarian deal will always be the one with the line that is furthest to the top right. Thus, any protocol that successively moves via dominant deals (hill-climbing towards the top right) will always arrive at the utilitarian solution because, as we assumed, every deal is always reachable from every other deal. We can directly connect every deal to every other deal by using a very flexible contracting language, such as **OCSM contracts** which allows all possible task transfer combinations.

OCSM contracts

Unfortunately, being able to move from every deal to every other deal in one step means that the agents will need to consider a vast number of possible deals at each time step. Thus, this approach merely replaces one intractable problem with another. In general, what we want is to limit the deal accessibility—the number of edges in the graph—such that there is a manageable number of deals accessible from every deal and there are few local optima where a hill-climbing negotiation could get stuck. We currently lack any general tools for achieving this result but, for any specific problem it is likely that a good multiagent designer can engineer a viable solution.

6.5.2 Lying About Tasks

In some cases it might be worthwhile for an agent to hide from other agents some of the tasks it has to perform, or to make up tasks and tell others that it has perform them, or to make up these tasks and, if someone else offers to perform the phony tasks then actually create the new tasks. All these methods of cheating result in the modification of the set of existing deals. For example, if an agent creates a new

δ	$s_i(\delta)$	$s_j(\delta)$	$u_i(\delta)$	$u_j(\delta)$
δ^1	$\emptyset$	$\{t_1\}$	1	3
δ^2	$\{t_1\}$	$\emptyset$	2	1

δ	$s_i(\delta)$	$s_j(\delta)$	$u_i(\delta)$	$u_j(\delta)$
δ^1	$\emptyset$	$\{t_1, t_2\}$	1	5
δ^2	$\{t_1\}$	$\{t_2\}$	2	3
δ^3	$\{t_2\}$	$\{t_1\}$	2	3
δ^4	$\{t_1, t_2\}$	$\emptyset$	8	1

Figure 6.15: Lying by task creation example. The top table shows the original utility values, where δ^1 is the Nash bargaining solution. The bottom table shows the values after agent i creates phony task t_2. Here, δ^4 is the Nash bargaining solution.

task then we now have to consider all the possible ways in which the new bigger set of tasks can be distributed among the agents. If an agent hides a task from the agents then the system has fewer possible allocations to consider. We can then ask the question, when is it in an agent's best interest to lie?

That is the question asked in *Rules of Encounter* (Rosenschein and Zlotkin, 1994). In it, the authors assume that the agents will use a bargaining protocol which will arrive at the Nash bargaining solution, presumably by either using alternating offers with the Zeuthen strategy or the ONE-STEP-NEGOTIATION protocol from figure 6.10. Once an agent knows that the final agreement deal will be the Nash bargaining solution then all it has to do is check each possible lie to see if the solution reached when it tells that lie gives him a higher utility than he would have received by telling the truth. Fortunately for the agent, and unfortunately for us, it has been shown that such lies do generally exist and will benefit the agent.

Figure 6.15 shows an example where an agent has an incentive to create a phony task. The table on the left shows the initial utility values for a simple game with only one task. In this game the Nash bargaining solution is δ^1 in which i does not perform any task and receives a utility of 1. Noticing that it would get a utility of 2 if it did perform t_1, i decides to create a phony task t_2 whose utility values are given on the table. In this new game the Nash bargaining solution is δ^4 which gives both tasks to agent i. Thus, by creating this task i was able to get a utility of 2 instead of the original 1 by getting t_1 allocated to itself. That is, since t_2 was assigned to i, the agent does not have to worry about perform this task nor does he have to worry about other agents trying to perform a task he made up. Thus, it can lie and get away with it.

6.5.3 Contracts

In our discussion of payments we have thus far assumed that if one agent says that he will pay another agent to perform some tasks then both of them will abide by that contract. The tasks will be performed and the money will be paid. Of course, things are not so simple for real applications. Specifically, in a dynamic environment an agent that is given a set of tasks to perform might find that right after he agrees to perform the tasks a new deal appears which will give him much higher utility. In this case the agent will be tempted to de-commit on his current contract and establish a new one. Similarly, it is possible that right after an agent agrees to pay a certain amount to get some tasks done that a new agent appears which is willing to perform those tasks for less. Again, the agent will be tempted to de-commit on his payment.

For example, in figure 6.16 the agents start out with δ^0. Lets say that in this scenario there is only one task to perform and, under δ^0, agent j is going to perform it. Both agents get a utility of 2 under this initial deal. However, j notices that if he can give the task to i then they will be at δ^1 which gives j a utility of 4 and i a utility of 1. Thus, j reasons that he could pay i the amount of \$2 to perform the task. This new deal is δ^2 which gives both agents a utility of 3. δ^2 dominates δ^0 and will thus be accepted by both. The only problem in moving from δ^0 to δ^2 is

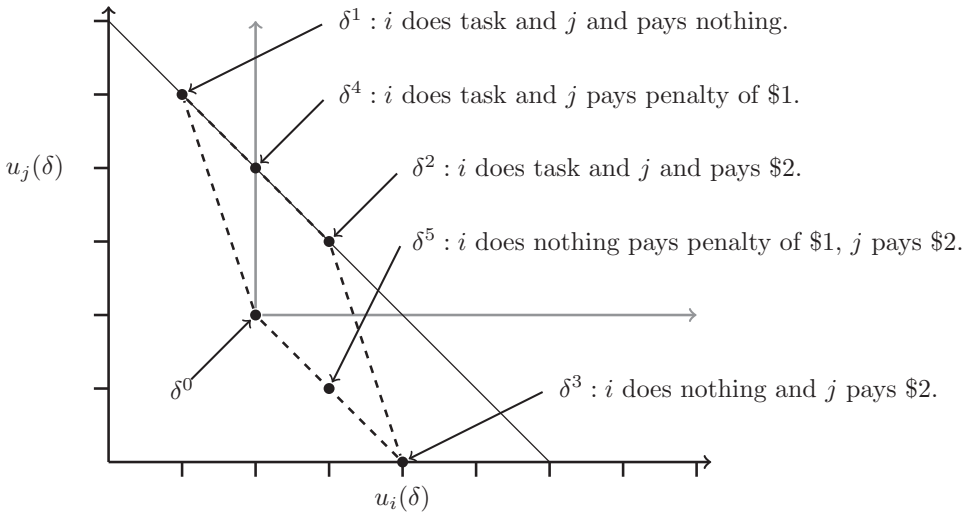


Figure 6.16: Leveled commitment contracts illustration. The agents start at δ^0 where j is performing the one available task and i is idle. δ^1 and δ^3 are the two de-commitment contracts. When we add a de-commitment penalty of 1 then these two become δ^4 and δ^5 respectively.

the risks involved with this deal. If j fails to pay i then we are at δ^1 where i loses a utility of 1 as compared to δ^0. Similarly, if j pays but i fails to perform the task then j has to perform the task and we end up at δ^3 where j has lost 2 in utility.

A simple way to fix this problem is to enforce all contracts. If the agents agree to a deal then each one must deliver on that promise. However, there are cases when this might be impossible or undesirable. Specifically, in dynamic environments where new opportunities can appear at any time allowing agents to de-commit might increase social welfare, in other applications the cost of policing the systems might be higher than the gains from having enforcable rules. In these cases we might wish to use more sophisticated contracts.

LEVELED COMMITMENT CONTRACTS

One solution is to use **leveled commitment contracts** (Sandholm and Lesser, 2002; Sandholm and Zhou, 2002) which have the added stipulation that if an agent breaks the contract then it must pay a penalty. For example, we could add a penalty of \$1 to anyone who breaks the contract in figure 6.16. By adding this penalty we have that, instead of the de-commitments leading us to δ^1 and δ^3 they will lead us to δ^4 and δ^5. Note how these new deals are closer to δ^0 thus reducing the risks of loss for both agents. Specifically, if j were to de-commit and fail to pay the \$2, he must still pay the \$1 penalty to i so i's final utility is 2 even if j de-commits, the same as $u_i(\delta^0)$. This means that there is no risk for i to take this contract. Similarly, if i de-commits and fails to do the task then we end up at δ^4 because i pays a penalty of \$1 to j who must perform the task. In this case, j's utility is 1 which is greater than $u_j(\delta^3) = 0$ so j has reduced his risk. Such risk reduction means that in dynamic situations where the agents are constantly trying to tradeoff current income against future possibilities—deciding whether or not to wait for a better deal to come along—the agents will be able to accept more contracts and, hopefully, the system will perform better.

Note that the de-commitment decision itself is not obvious, even when the agent receives a better offer from somewhere else. For example, say that after agreeing to δ^2 agent i receives another offer that will give him a utility of 10 but in order to perform that task he must de-commit on the task from j. Clearly i wants to de-commit but, a strategic agent i would also realize that there is a change that j will de-commit first. If j de-commits then i gets the penalty payment from j and still gets to perform to other task. The agents thus face a version of the game of chicken (figure 3.5) with uncertainty about the other agent's payoff. Namely, i does not know if j wants to de-commit or not. If i has some probabilistic data on j's offers then these could be used to generated an extended-form game. Agent i can then find the Nash equilibrium of this game and play that strategy.

Another solution is to extend the negotiation protocol to include tentative con-

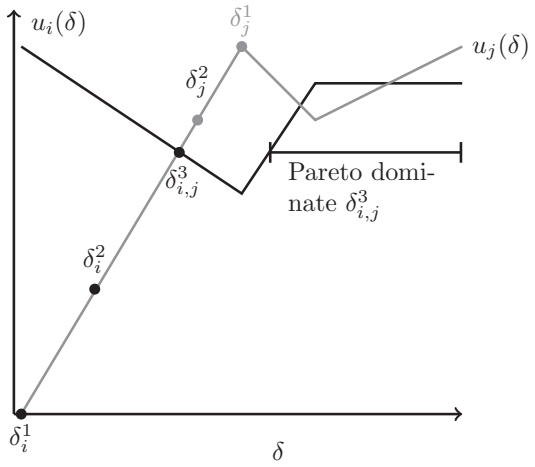


Figure 6.17: Example of convergence to non-Pareto deal. The agents converge to deal $\delta_{i,j}^3$ which is Pareto-dominated by all the deals indicated on the right.

tracts. For example, we could extend the contract net protocol to include pre-bid and pre-assignment steps (Aknine et al., 2004). This allows agents to make tentative commitments which they can make binding at a later time when they are more certain that no new offers will arrive from other agents.

6.6 Complex Deals

Thus far we have considered deals to be atomic units that cannot be broken down into smaller pieces. In contrast, real world deals are known to be composed of many different items such as price, warranty, delivery time, color, etc. For example, two agents negotiating over the purchase of a car will need to agree on the price to be paid, the color of the car, the number of years in the warranty, the value of the trade-in car, the type of financing and interest rate provided, and many other possible parameters. We describe these complex preference languages in Section 6.7. These dimensions inevitably lead to an explosion in the space of possible deals.

MULTI-DIMENSIONAL DEAL

More formally we define a **multi-dimensional deal** as one that is composed of a set of variables $x_1, x_2, \ldots, x_n$ with domains $D_1, D_2, \ldots D_n$, respectively. For example, one variable could correspond to price and its domain could be all integer numbers in the range 0 to 100. We can also re-define the agent's utility function so that it explicitly represents each variable. For example, instead of an opaque $u_i(\delta)$, we could have a more expressive $u_i(\delta) = c_1 u_i^1(x_1) + c_2 u_i^2(x_2) + \cdots + c_n u_i^n(x_n)$ or some other combination of the individual variables. The negotiation problem remains that of finding a deal that corresponds to a chosen solution concept, such as the utilitarian deal or the Nash bargaining deal.

The astute reader will notice that this is nearly the exact same problem as the distributed constraint optimization problem from Chapter 2. In fact, the only difference between multi-dimensional negotiation and the constraint optimization problem is that they assume different data distribution requirements. In distributed constraint optimization it is assumed that each agent owns one or more variables whose value it can change and the agents generally only care about the value of their variable and a few other ones. In negotiation the variables are not owned by any agent and the agents generally have preferences over all the variables. Finally, in constraint optimization there is a clear solution concept—minimize constraint valuations—while in negotiation each agent is assumed to have a utility function and there is no one obviously better solution concept.

When the problem uses multi-dimensional deals we often find that the total number of deals is extremely large—the number of possible deals grows exponentially with the number of variables. Thus, it becomes even more likely that a hill-climbing negotiation algorithm will get stuck in a local optima. For example, figure 6.17 shows how we could converge to a non-Pareto deal via protocol similar to mono-

tonic concession. The agents start out by proposing deals that are the best possible for them and then concede to the other agent. However, the agents only consider deals that differ from the last proposed deal by changing only a few variables—they are "near" the last proposed deal if we define the distance to be the number of variables that have changed. Since they only consider a subset of the deals they might end up ignoring better deals. Specifically, the agents in the figure ignore the deals on the right part of the chart. As such, they end up converging on a deal which is Pareto-dominated by many other possible deals.

The Hamming distance, a similar concept from information theory, between two binary strings is the number of positions which are occupied by different values. Thus the distance between 101 and 110 is 2.

The reason for ignoring the deals is often the simple fact that the space of possible deals is so large that there is not enough time for the agent to sample it well enough to find other deals which are better than the ones it has proposed. In these cases agents choose a few attributes and change their values monotonically. For example, when negotiating a complex car deal you might wish to keep all other attributes at fixed values and negotiate only over price. In that case you might end up at a non-Pareto deal if your chosen values for the other attributes are not the right ones, such as the case where the dealer would be willing to accept a much lower price for a car with a one-year warranty rather than the three-year warranty you are asking for.

We can formalize this strategy by having the agents agree on one dimension at a time. For example, we could declare that the agents first agree on the price at which the car will be sold, then go on to haggle on the warranty, then on the color, and so on until all the dimensions are covered. The order in which these negotiations are carried out is called an **agenda**. As you might expect, ordering the negotiation in this way can sometimes lead to sub-optimal outcomes but does make the negotiation problem tractable for a lot more cases. Comparison of the results from agenda-based negotiations versus full negotiation, using specific negotiation strategies have shown that a social welfare deal is reached under certain combination of strategies (Fatima et al., 2004) .

AGENDA

6.6.1 Annealing Over Complex Deals

A common solution to the problem of searching for an optimal solution over a very large space of possible answers is to use an **annealing** method. Simulated annealing algorithms start out with a randomly chosen deal which becomes the best deal found thus far. New possible deals are generated by mutating this deal and are accepted as the new best deal if they are better than the current best deal or, with a certain probability, they are accepted even if they are worse than the current best. The probability of accepting a worse deal and the severity of the mutations both decrease with time. In this way the algorithms is guaranteed to converge to a locally optimal solution. In practice it has been found that this solution is often also the global optimum.

ANNEALING

Simulated annealing has been used to implement several simple negotiation protocols (Klein et al., 2003). The protocol uses a **mediator** agent instead of having agents negotiate with each other. At each step the mediator presents a deal to both agents. The agents can either accept the deal or reject it. If both of them accept the deal the mediator mutates the deal slightly and offers the new deal to both agents, who can once again either accept it or reject it. If one or more of the agents rejected the proposed deal then a mutation of the most recently accepted deal is used instead. The protocol implemented by the mediator is shown in figure 6.18.

MEDIATOR

We can then implemented two types of agents.

Hill Climber Accepts a deal only if it gives him a utility higher than its reservation price $u_i(\delta^-)$ and higher than that of the last deal it accepted. That is, it monotonically increases it reservation price as it accepts deals with higher utility.

Annealer Use a simulated annealing algorithm. That is, it maintains a temperature T and accepts deals worse than the last accepted deal with probability

ANNEALING-MEDIATOR

```
1  Generate random deal δ.
2  δ_accepted ← δ
3  Present δ to agents.
4  if both accept
5      then δ_accepted ← δ
6           δ ← mutate(δ)
7           goto 3
8  if one or more reject
9      then δ ← mutate(δ_accepted)
10 goto 3
```

Figure 6.18: Procedure used by annealing mediator.

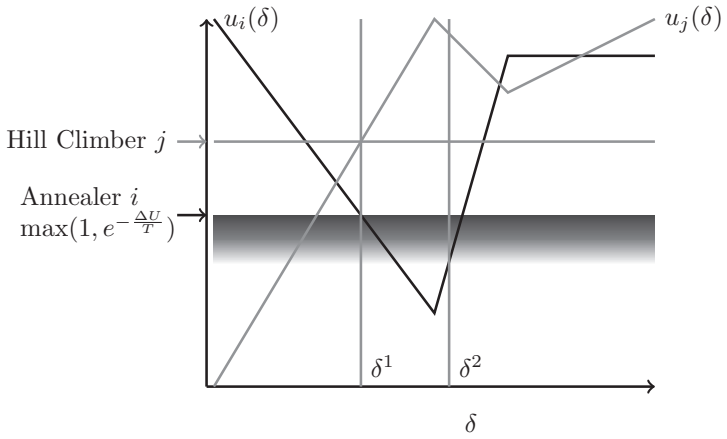


Figure 6.19: Two steps in the annealing algorithm with one hill climber j and one annealing agent i. After the mediator presents δ^1 both agents set up their new reservation prices shown by the line, for j, and the fuzzy bar for i.

$\max(1, e^{-\frac{\Delta U}{T}})$, where ΔU is the utility change between the deals.

Annealer agents, along with the annealing mediator, effectively implement a simulated annealing search over the set of possible deals. Meanwhile, hill climbing agents implement a hill climbing search over the set of possible deals.

For example, in figure 6.19 agent j is a hill climber and i is an annealer. After the mediator presents δ_1 both of them accept since its utility is higher than their reservation prices. The hill climber sets a new reservation price below which he will not accept any deal. The annealer sets up a probability of accepting deals that are below his new reservation price. When δ_2 appears, j will accept it because it is above its reservation price and i might also accept it, but with a small probability as $u_i(\delta_2)$ is slightly below its reservation price.

This type of annealing algorithm is a good example of how we can take a standard search technique like annealing and apply it to a distributed negotiation problem. Annealing works because there is some randomness in the sequence of deals that are explored. The annealing negotiation protocol places this randomness within a negotiation. Unfortunately, generating proposal deals purely at random means ignoring any knowledge that the agents have about the shape of their utility function. Also, an annealing agent is acting irrationally when it accepts a deal that is worse for it than a previous deal. We would need some way to justify why this agent occasionally acts irrationally.

6.7 Argumentation-Based Negotiation

We can further relax the idea of an agenda, where agents negotiate sequentially over each dimension of the deal, and instead define a more complex negotiation language which agents can use for negotiation. That is, thus far we have only considered the use of a **1-sided proposal** where a deal is proposes and it is either accepted

or rejected. In **argument-based protocols** the agents use a more sophisticated language for communications. There are currently no standard languages for argumentation although there is a lot of work being done on preference languages, as we will see in Chapter 7. Still, we can categorize the various types of utterances that an argumentation language might support (Jennings et al., 2001).

ARGUMENT-BASED PROTOCOLS

Specifically, an agent might be able to **critique** the proposal submitted by the other agent. Critiques provide the agents with information about others' utility functions, specifically with respect to the previous proposal. This information can be used to rule out some deals from consideration. For example, agent i and j might engage in the following negotiation:

CRITIQUE

i**:** I propose that you provide me with $x_1 = 10$ under conditions $x_2 < 20$ and delivery $x_3 < 20061025$.

j**:** I am happy with the price of $x_2 < 20$ but the delivery date x_3 is too late.

i**:** I propose that I will provide you with service $x_1 = 9$ if you provide me with $x_1 = 10$.

j**:** I don't want $x_1 = 9$.

An agent might come back with a **counter-proposal** which are new deals, just like in the alternating offers models, but which are generally assumed to be related to the last offer. For example,

COUNTER-PROPOSAL

i**:** I propose that you provide me with service $x_1 = 10$ under conditions $x_2 < 20$ and delivery $x_3 < 20061025$.

j**:** I propose that you provide me with service $x_1 = 10$ under conditions $x_2 < 30$ and delivery $x_3 \geq 20061025$.

An agent might be able to **justify** his negotiation stance with other statements. Statements it hopes will convince the other agent to accept his proposal. These statements are just ways of giving more knowledge to the other agent in the hopes that it will use that knowledge to eliminate certain deals from consideration. For example,

JUSTIFY

i**:** My warehouses is being renovated and it will be impossible to deliver anything before the end of the month, that is $x_3 > 20061031$.

An agent might try to **persuade** the other agent to change its negotiation stance. Persuasion is just another way of giving more knowledge tot the other agent. For example,

PERSUADE

i**:** Service $x_1 = 9$ is much better than you think, look at this report.

Finally, an agent might also employ **threats**, **rewards**, and **appeals** in order to convince the others to accept his proposal. These techniques help agents build better models of the others' utility functions, eliminate sets of deals from consideration, and change the agents utility functions in order to make them better reflect the reality of the situation.

THREATS
REWARDS
APPEALS

Argument-based negotiation closely matches human negotiation. Unfortunately, that also means that it is very hard to build agents that can understand these complex negotiation languages. A common approach is to build the agent using Prolog or some other logic-based language. The agent can then keep a database of the messages (facts) sent by the opponent and try to infer the set of possible contracts. Even when exploiting the inference powers of a logic-based programming language the problem of implementing correct argument-based negotiators is still very hard. No one has been able to implement a general argument-based negotiator. Even when limiting the problem to a specific domain, there are very few examples of successful programs that do argument-based negotiation. Still, there is a growing

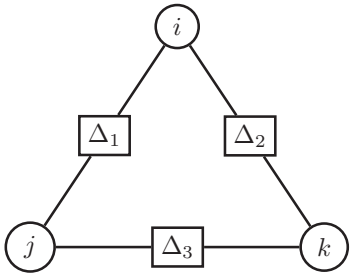


Figure 6.20: Graphical representation of a negotiation network with agents i, j, and k, where Δ_1 is a set of deals that i and j can agree upon.

research community developing new preference description languages to be used by auctions, as we will see in Chapter 7. These languages could just as easily be used by an argumentation-based agent as by a centralized auctioneer.

There is a community of agent-based argumentation researchers working on developing a standardized language for arguments (nevar et al., 2006; Rahwan et al., 2004). They distinguish between the communication and domain languages used for representing the arguments and the negotiation protocol which constraints which type of messages can be sent at which time and what they mean. For example, a protocol could have rules on when a negotiation must end such as when one of the agents says "that is my final offer". Similarly, a protocol could have commitement rules which specify whether or not an agent can withdraw from a previous commitment it made.

6.8 Negotiation Networks

CONCURRENT NEGOTIATIONS

It is possible that an agent might be involved in **concurrent negotiations** with several other agents where the resulting deals might not be compatible with each other. For example, you might be negotiating with several car dealers at the same time, as well as negotiating a car loan with several lenders and car insurance with several insurance agencies. Once you make a purchasing deal with a specific dealer then you can stop negotiating with the other dealers. Furthermore, if the deal you made includes a loan agreement you can also stop negotiating with the lenders. Thus, as you negotiate with the various parties their last offers will tend to have an effect on how you negotiate with the others. For example, if a lender offers you a very low interest rate then you might be able to afford a more expensive car and you will not be swayed by the dealer's offer that includes a higher interest rate loan.

More formally, we can define a negotiation network problem as one were a set of agents negotiate over a set of deals such that all agents end up agreeing to a set of deals that are compatible with each other.

NEGOTIATION NETWORK

Definition 6.7. *A **negotiation network** problem involves a set of agents A and set of sets of deals. Each set of deals Δ_i involves only a subset of agents $\Delta_i^a \subseteq A$ and always includes the no-deal deal δ^-. A solution $\vec{\delta}$ to the problem is a set of deals, one from each Δ_i set, such that all the deals that each agent is involved in are compatible with each other. Compatibility is captures by the c function, where*

$$c_i(\delta, \delta') = \begin{cases} 1 & \text{if } \delta \text{ and } \delta' \text{ are compatible} \\ 0 & \text{otherwise.} \end{cases}$$

Figure 6.20 shows a graphical representation of a negotiation network problem with three agents: i, j and k, and where i and j can enter into any of the deals in Δ_1, similarly i and k can enter into a deal from Δ_2 and j and k can enter into a deal from Δ_3. We also need to define the boolean functions c_i, c_j, and c_k which tell us which pair of deals are compatible.

In the negotiation networks the standard negotiation problem is further exacerbated by the fact that each agent must maintain simultaneous negotiations with several other agents and these negotiations impact each other. The approaches at solving these type of problem has thus far consisted of using ad-hoc negotiation

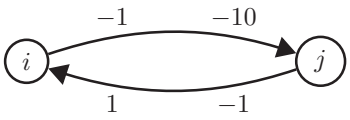


Figure 6.21: The coercion network.

Figure 6.22: A network with three agents. Agent j can only split 10 dollars with either i or k, but not both.

strategies and running tests to determine how these fare against each other (Nguyen and Jennings, 2004; Zhang et al., 2005a; Zhang et al., 2005b).

6.8.1 Network Exchange Theory

Another way to approach the problem of negotiation networks is to look at what humans do and try to imitate it. This is known as the **descriptive approach**. Luckily for us, researchers in Sociology have developed and studied a model which they call **network exchange theory** (NET) that is very similar to the negotiation networks problem (Willer, 1999). In this model humans are represented by nodes in a graph. The annotated edges between nodes represent the possibility of a negotiation between those two humans. Sociologists have run tests with human subjects in these networks in order to record how they behave. They then found equations that can be used to predict the deals that people eventually agree on, on average, based on the initial structure of the problem.

DESCRIPTIVE APPROACH

NETWORK EXCHANGE THEORY

Figure 6.21 shows a simple NET network. The nodes represent agents i and j. The edges are directional and represent interaction possibilities from the source agent to the destination agent where only the source agent can decide whether to enact the interaction. Specifically, if i decides to enact its edge it would give i a utility of -1 and would give j a utility of -10. Imagine that i is threatening to harm j if i does not relinquish his wallet. Similarly j could decide to enact its edge which would give it a utility of -1 and give i a utility of 1. This particular network is known as a coercion scenario because i can threaten j to give him the 1 otherwise i will punish j by giving it -10. Even thought it is not rational for i to want go punish j for it is punishing itself at the same time, the threat works in the real world. Another type of edge used is the resource pool edge where an amount of utility is to be divided among the agents but the agents must decide how it is to be distributed. A sample is shown in figure 6.23. This network shows two agents, i and j, who are negotiating over how to divide 10 dollars.

NET can also represent various deal compatibility requirements. One such example is exclusive connections where one agent can exchange with any one of a number of other agents but exchange with one of them precludes exchange with any of the other ones. This is typically simply represented by just adding more nodes and edges to the graph, as shown in figure 6.22.

Based on test results, NET tells us that each person has a **resistance** to each particular payment p given by a resistance equation. i's resistance to payment p is given by

RESISTANCE

$$r_i = \frac{p_i^{\max} - p_i}{p_i - p_i^{\text{con,}}} \tag{6.8}$$

where $p_i^{\max}$ is the maximum i could get and p_i^{con} is the no-deal deal. j's resistance is similarly defined. If we further know that i and j are splitting 10 dollars then we know that $p_i + p_j = 10$.

The resistance equation is meant to capture the person's resistance to agreeing to a deal at any particular price. The higher the resistance the less willing the person is to agree to the deal. Note how this equation is not linear, as might be expected if people were rational, but has an exponential shape. This shape tells us a lot about our irrational behavior.

Figure 6.23: A sample exchange network with 10 units to be distributed among two agents.

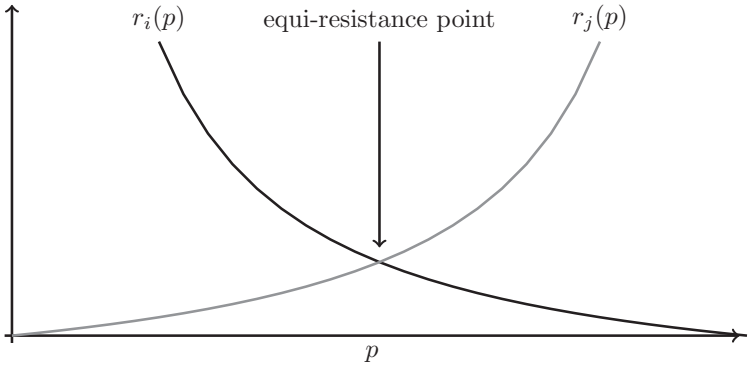


Figure 6.24: Resistance of two agents to each possible deal p.

EQUI-RESISTANCE POINT

NET tells us that exchange happens at the **equi-resistance point** where both agents have equal resistance, that is where

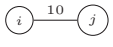
$$r_i = \frac{p_i^{max} - p_i}{p_i - p_i^{con}} = \frac{p_j^{max} - p_j}{p_j - p_j^{con}} = r_j. \tag{6.9}$$

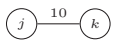
Notice that the equi-resistance equation tells us the agreement that people will eventually reach via negotiation, but it does not tell us how the negotiation took place, that is, it does not tell us their negotiation tactics.

We can represent the equi-resistance point graphically by simply replacing p_j with $10 - p_i$ in j's resistance equation r_j and plotting the two curves r_i and r_j. The point at which the curves cross is the point of exchange, as shown in figure 6.24. Of course, the exchange does not always happen at the midpoint. For example, if i had an offer from some other agent for 6 dollars if it refused to negotiate with j, this would mean that $p_i^{con} = 6$ which would change the equi-resistance point.

ITERATED EQUI-RESISTANCE ALGORITHM

We can solve complex NETs using the **iterated equi-resistance algorithm**. The algorithm simply uses the equi-resistance equation repeatedly on each edge of the graph in order to calculate the payments that these agents can expect to receiver. This is repeated for all edges until payments stop changing. For example, for the graph in figure 6.22 we would follow these steps.

1. Apply Equi-resistance to 

. Gives us $p_j = 5$.
2. Apply Equi-resistance to 

. Let $p_j^{con} = 5$ and apply equi-resistance again.
3. Repeat until quiescence.

The iterated equi-resistance algorithm is not guaranteed to converge and it might converge to different solutions when the edges are sorted in differently. Even when it does converge the deal it reaches is not guaranteed to be the one that humans would reach. However, many experiments have been performed with humans in *small* networks (less than 12 nodes) which have shown that the iterated equi-resistance algorithm correctly predicts the resulting deal. This algorithm gives a new solution for the negotiation problem which, unlike the solutions in Section 6.1.1, is not based on some desirable properties of the solution but is based on evidence from human negotiation, that is, it gives us a descriptive solution to the negotiation problem. If you want to implement agents that reach the same solution as humans then you probably want to use the iterated equi-resistance solution.

Exercises

6.1 Given the following utility values for agents i and j over a set of possible deals δ:

δ	$u_i(\delta)$	$u_j(\delta)$
δ^1	1	0
δ^2	0	1
δ^3	1	2
δ^4	3	1
δ^5	2	2
δ^6	1	1
δ^7	8	1

1. Which deals are on the Pareto frontier?
2. Which one is the egalitarian social welfare deal?
3. Which one is the utilitarian deal?
4. Which one is the Nash bargaining deal?
5. Which one is the Kalai-Smorodinsky deal?

6.2 In a distributed workflow enactment application, a workflow is defined as a set of tasks all of which must be performed in order to complete the workflow and collect the payment. The following table lists the available workflow instances, their tasks and their payments:

Workflow	Tasks	Payment
w_1	t_1, t_1, t_1	6
w_2	t_1, t_1, t_2	5
w_3	t_1, t_1, t_3	3
w_4	t_1, t_2, t_2	1
w_5	t_2, t_2, t_3, t_3	8
w_6	t_2, t_2	1

There are three agents. An agent can perform at most two tasks at a time, as long as those tasks are **different**. Also, a task is performed for only a single workflow. That, if an agent performs t_1 for w_1 then this t_1 cannot be used for any other workflow. The goal is for the agents to find the set of workflows that maximizes the total payment.

1. Re-state this problem as a negotiation problem, show the set of possible deals. (Hint: note that the agents have identical capabilities, so you do not need to worry about which one does which task).
2. We further constrain this negotiation by only allowing agents to either drop one workflow, add one workflow, or exchange one workflow for another. But, they can only do so if this increases the total payment (thus, you quickly conclude that they will never drop a workflow). Which deals become local optima under these constraints?

Chapter 7

Auctions

Auctions are a common and simple way of performing resource allocation in a multiagent system. In an auction, agents can express how much they want a particular item via their bid and a central auctioneer can make the allocation based on these bids. Of course, this generally requires the use of a centralized auctioneer but there are techniques for reducing this bottleneck. Still, even centralized auctions can be very complex and produce unexpected results if one does not understand all the details.

7.1 Valuations

Before we begin to talk about the various types of auctions we must first clarify how people value the items being sold. We have used the notation $u_i(s)$ to refer to the utility that agent i derives from state s. Similarly, if s is instead an item, or set of items, for sale we can say that $v_i(s)$ is the **valuation** that i assigns to s. We furthermore assume that this valuation is expressed in a common currency, thus $v_i(s)$ then becomes the maximum amount of money that agent i is willing to pay for s. When studying auctions we generally assume that all agents have a valuation function over all the items being sold.

VALUATION

In the simplest case this valuation function reflects the agent's utility of owning the given items. For example, if you plan to eat a meal then the amount you are willing to pay for each item in the menu depends solely on how hungry you are and how much you like each item. In these cases we say that the agent has a **private value** function.

PRIVATE VALUE

On the other hand, there are items which you cannot consume and gain no direct utility from but which might still have a resale value. The classic example are stocks. When you buy a share in some company you cannot do anything with that share, except sell it. As such, your valuation on that share depends completely on the value that others attribute, and will attribute, to that share. These are known as **common value** functions.

COMMON VALUE

Most cases, however, lie somewhere in the middle. When you buy a house you take into consideration the value that you will derive from living in that house as well as its appreciation prospects: the price you think others will pay when you finally sell it. This is an example of a **correlated value** function and is very common in the real world with durable high priced items.

CORRELATED VALUE

The type of valuation function that the agents use changes their optimal behavior in an auction. For example, if an agent has a common value function then it will likely pay more attention to what the other agents are bidding. Most multiagent implementations use agents with private value functions as most systems do not want to waste the time required to implement secondary markets for the items being sold. Still, in open multiagent systems it might be impossible to prevent secondary markets from appearing.

7.2 Simple Auctions

There are times when there are many agents and the only thing they need to negotiate over is price. In these occasions it makes sense to use an auction since they are fast and require little agent communication. However, auctions are not as simple as they might appear at first and there are many ways in which things can go wrong when using them.

English auction
first-price open-cry ascending

The actual mechanisms used for carrying out an auction are varied. The most common of all is the **English auction**. This is a **first-price open-cry ascending** auction. It is the standard one used in most auction houses. In it, the auctioneer raises the price as people yell higher bids. Once no one is willing to bid higher, the person with the highest bid gets the item and pays his bid price. These auctions sometimes have an initial or **reservation price** below which the seller is not willing to sell. The dominant strategy in an English auction, with private value, is to bid the current price plus some small amount until either the auction is won or one's reservation price is reached.

reservation price

winner's curse

If an English auction is common or correlated value then it suffers from the **winner's curse**. For example, when you buy a stock in an English auction it means that you paid more than anyone else was willing to pay. As such, your valuation of that share must now be less than what you paid for it. In the real world we gamble that at some point in the future the others will realize that this stock really is worth that much more.

First-price sealed-bid

A similar auction type is the **First-price sealed-bid** auction. In this auction each person places his bid in a sealed envelope. These are given to the auctioneer who then picks the highest bid. The winner must pay his bid amount. These auctions have no dominant strategy. The buyer's best strategy is to spy on the other bidders in order to determine what they are going to bid and then bid slightly higher than that, as long as that is less than one's reservation price. If spying is impossible then the agent has no clearly superior strategy. Because of the incentive for spying, these auctions lead to a lot of inefficiencies when paired with intelligent agents.

Ontario Flower Growers Co-op, an example of a Dutch auction at work. The two large circles in the back are used to show the descending price.

Dutch
open-cry descending price

The **Dutch** auction is an **open-cry descending price** auction. In it the seller continuously lowers the selling price until a buyer hits a buzzer, agreeing to buy at the current price. The auction's name comes from its use by the Dutch to sell flowers. The Dutch flower markets have been around for centuries and are still thriving. Every morning carts of flowers are paraded before eager flower shop owners who are equipped with a buzzer. Each cart stops before the buyers and a clock starts ticking backwards from a high number. When the first buyer hits his buzzer the flowers are sold to him at the current price. Analysis of the Dutch auction shows that it is equivalent to a first-price sealed-bid auction in terms of strategy. That is, it has no dominant strategy. However, it has the nice property of being real-time efficient. The auction closes quickly and the auctioneer can make it move even faster by lowering the price faster. This real-time efficiency makes it a very attractive auction for selling cut flowers as these lose their value quickly after being harvested.

Vickrey
second-price sealed-bid

The **Vickrey** auction is a more recent addition and has some very interesting properties. It is a **second-price sealed-bid** auction. All agents place their bids and the agent with the highest bid wins the auction but the price he pays is the price of the second highest bid. Analysis of this auction has shown that bidding one's true valuation, in a private value auction, is the dominant strategy. For example, let your valuation for the item being sold be v. If you bid less than v then you are risking the possibility that some other agent will bid $w < v$ and get the item even though you could have won it. Moreover, since w is less than v you could have bid v and paid only w. As such, you have nothing to gain by bidding less than v but risk the possibility of losing an auction that you could have won at an acceptable price. On the other hand, if you bid $v' > v$ then you are risking that some other agent will bid w, where $v' > w > v$, and thereby cause you to pay more than your reservation price v. At the same time, you do not gain anything by bidding higher

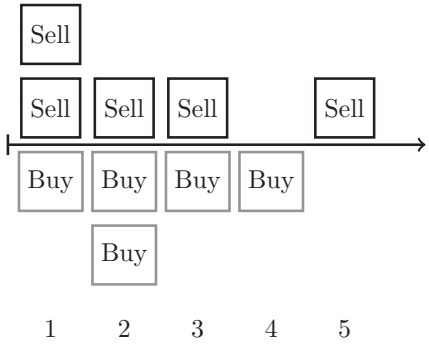
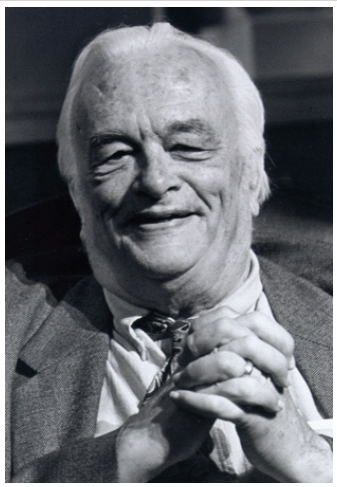


Figure 7.1: Graphical representation of a double auction. The x-axis represents prices. Each box represents one buy or sell order at the given price.

than v because the only auction that you might win by bidding v' instead of v are those where you have to pay more than v.

As such, the Vickrey auction eliminates the need for strategizing. Since there is an easy dominant strategy the agents do not have to think about what they should do. They just play their dominant strategy and bid their true valuation. Thus makes it a very attractive auction in terms of its efficiency but it is also for this reason that most people don't like Vickrey auctions. People are often hesitant about revealing their true valuations for an item because we know that the world is an iterated game and this information could be used against us in some future auction. As such, Vickrey auctions are seldom used in the real world.



William Vickrey. 1914–1996. Nobel Prize in Economics.

DOUBLE AUCTION

Finally, the **double auction** is a way of selling multiple units of the same item. It is the auction used in stock markets. Each buyer places either a buy or a sell order at a particular price for a number of instances of the item (number of shares in the stock-market). The buy and sell bids can be visualized in a simple graph such as the one shown in Figure 7.1. Here, the x-axis represents a price and each box represents an offer to buy or sell a share at the given price.

Once we have all the bids then it is time to clear the auction. There are many different ways to clear a double auction. For example, if figure 7.1 we could match the sell order for 1 with the buy for 5 then pocket the difference of $5 - 1 = 4$ for ourselves, or we could clear it at 3 and thus give both bidders a deal, or we could match the seller for 1 with the buy for 1, and so on. As can be seen, there are many different ways to match up these pairs and it is not clear which one is better.

One metric we might wish to maximize is the amount of surplus, known as the spread by traders. That is, the sum of the differences between the buy bids and the sell bids. In some auctions this surplus is kept by the auctioneer who then has an incentive to maximize it. Another option is to use it to enable more bids to clear. In the example above the total supply and demands are 12, therefore all bids should clear. One way to do this is to pair up all the small sell bids and give these sellers exactly what they asked for then give the surplus to the sell bid of 5 in order to clear it with the remaining buy bid.

Another metric we could use is a uniform price. That is, rather than giving each seller and buyer the exact price they asked for, we give them all one uniform clearing price. Clearly, the only buy bids that will clear are those that are above the clearing price and the only sell bids that clear are those below the clearing price.

7.2.1 Analysis

Now that we know the various auction types, there is an obvious question that we must ask ourselves. On which auction do sellers make more money? This question is answered by the following theorem.

Theorem 7.1 (Revenue Equivalence)**.** *All four single-item auctions produce the same expected revenue in private value auctions with bidders that are risk-neutral.*

Table 7.1: Example of inefficient allocation.

	tasks	Agent 1	Agent 2
Costs of Doing Tasks	t_1	2	1.5
	t_2	1	1.5
	t_1,t_2	2	2.5

We also know that if the bidders are risk-averse then the Dutch and first-price are better. A risk-averse bidder is willing to pay a bit more than their private valuation in order to get the item. In a Dutch or First-price auction a risk-averse agent can insure himself by bidding more than would be required.

In common or correlated value cases the English auction gives a higher revenue to the seller. The increasing price causes others to increase valuation. That is, once the agent sees others bidding very high for the item the agent realizes that the item is really worth more to the other agents so it also raises its valuation of the item.[1]

BIDDER COLLUSION

As it is often the case when money is involved, we have to be on the look out for ways in which the agents might cheat. The problem of **bidder collusion** affects all 4 auctions. In bidder collusion the bidders come to an a-priory agreement about what they will bid. They determine which one of them has the higher valuation and then all the other bidders refrain from bidding their true valuation so that the one agent can get it for a much lower price. The winner will then need to payback the others. The English and Vickrey auctions are especially vulnerable to bidder collusion as they **self-enforce collusion** agreements. For example, say there are 10 buyers and they realize that one of them has a valuation of 100 while the others have a valuation of 50 for the item. They agree to let him buy it for 1. In an English auction one of the 99 agents could defect and bid 2. However, this would only make the high-valuation agent bid 3, and so on until they get to 51. The high-valuation agent will get the item for 51 so the other agent gets nothing by defecting. The same logic applies in a Vickrey auction.

SELF-ENFORCE COLLUSION

LYING AUCTIONEER

Another problem might be that a **lying auctioneer** can make money from a Vickrey auction. All he has to do is to report a higher second-price than the one that was announced. The winner then pays this higher second price to the auctioneer who gives the buyer the real second price and pockets the difference. This requires that the bids are not revealed and that the buyer does not pay the seller directly. If the buyer paid the seller directly then a lying auctioneer and the seller could collude to charge the buyer a higher price. A lying auctioneer can also place a shill in an English auction. That is, assuming that the auctioneer gets paid a percentage of the sales price. If the auctioneer gets paid a fixed amount then there is no incentive for him to increase the sales price.

shill: a decoy who acts as an enthusiastic customer in order to stimulate the participation of others.

When auctioning items in a series when their valuations are interrelated, such as chairs in a dining set or bandwidth rights in adjacent counties, it is possible to arrive at inefficient allocations. For example, the problem in Table 7.1 leads to an inefficient allocation if we first auction t_1 and then t_2. Specifically, if we auction t_1 first then Agent 2 will get it as it has the lower cost. When we then auction t_2 both agents have the same cost (1) but, no matter who gets it the total cost is 2.5. If, on the other hand agent 1 had won both tasks then the total cost would be 2. matter who gets it the total cost is 2.5. This is the problem of **inefficient allocation**.

INEFFICIENT ALLOCATION

We could solve this problem if we made the agents use full lookahead effectively building an extended-form game from their possible interactions. With full lookahead the agents can build a tree where each level corresponds to a task being auctioned. In this way agent 1 can see that it will only cost him 2 to do t_1 and t_2 so it can reduce its cost for t_1 from 2 to 1. Of course, this puts agent 1 at risk of not getting t_2 since agent 1 generally will not know agent 2's cost for t2 so it does not know if it will win that auction. Another much better way of solving the problem of

[1] An interesting example of this was a British auction for 3G bandwidth licenses. The standard English auction was modified so that everyone must agree to buy at the current price or leave the room. This led to the licenses selling for 1000 times the expected amount (Harford, 2005).

inefficient allocations is to use a combinatorial auction, which we will learn about in Section 7.3.

7.2.2 Auction Design

When designing an auction for a multiagent system you must make many decisions. You must first determine what kind of control you have over the system. It is possible that you control only the bidding agent and the auction is already implemented, as when building agents to bid on Ebay. It is possible that you control only the auction mechanism, as when building an auction website. Finally, it is possible that you might control both agents and mechanism, as when building a closed multiagent system.

The case where you control the mechanism is especially interesting. You must then decide what bidding rules you will use: when bids are to be placed, when they can no longer be placed, what form can these bids takes, and what rules they must follow. For example, you might set a rule that a new bid has to always be for a higher value. You also set up clearing rules which determine when the items are sold. We explained some of the problems with various clearing rules in the double auction. The four standard auction types already have clearing rules but you might want to modify these. Finally, you must decide on information rules: how much information the agents are to know about what the other agents bid, whether to reveal information during the bidding process itself or after clearing (Wurman et al., 2002).

Currently all online auctions are implemented as centralized web applications but it is not hard to imagine a future where the auctions are freed from the constraints of a central hub and become a protocol enacted by buying and selling agents.

7.3 Combinatorial Auctions

Arguably, the **combinatorial auction** has been the most widely used auction in multiagent systems. In it agents can place bids for sets of items instead of just placing one bid for each item for sale. In many systems we have the problem that there is a set of tasks or jobs that needs to be distributed among the agents but the agents have complex preferences over the set of tasks. For example, in a workflow application we have a set of workflows, each composed of a set of web services, which must be performed by certain deadlines. Each agent can perform a subset of the services but each agent has different costs which might depend on the agent's type, its current load, the services it has done before, etc. Our problem as system designers is to allocate the workflows to agents so that we maximize the total number of workflows completed. Another example of combinatorial auctions is the selling of broadcasting rights by the federal government where cellular companies prefer to own the same frequencies in nearby locations, or at least to own some bandwidth in all the neighborhoods of a city. A final example is the buying of parts to put together a PC which requires a motherboard, CPU, ram, etc. Each part can be bought independently but only some bundles work together. These problems, and all problems of this type, can be solved by a combinatorial auction.

COMBINATORIAL AUCTION

Formally, we define a combinatorial auction over a set of items M as being composed of a set of bids, where each agent can supply many different bids for different subset of items. Each bid $b \in B$ is composed of b^{items}, which is the set of items the bid is over, b^{value} the value or price of the bid, and b^{agent} which is the agent that placed the bid.

For example, say you had a set of 5 figurines, one each of a different Teen Titan and you received 6 different combinatorial bids, as shown in Figure 7.2. The question you then face is how to determine which are the winning bids so as to maximize the amount of revenue you receive. Note that you can sell each item only once since you

Price	Bid items
\$1	Beast Boy
\$3	Robin
\$5	Raven, Starfire
\$6	Cyborg, Robin
\$7	Cyborg, Beast Boy
\$8	Raven, Beast Boy

Figure 7.2: Teen Titans figurines: (from top left) Beast Boy, Cyborg, Robin, Raven, and Starfire. The set of combinatorial bids received for them in on the table at the right.

only have one of each. This is the problem of winner determination. In the figure, the correct solution would be to accept the \$3, the \$5 and the \$7 bids.

7.3.1 Centralized Winner Determination

WINNER DETERMINATION

The **winner determination** problem is finding the set of bids that maximizes the seller's revenue. Or, more formally, find

$$X^* = \arg\max_{X \subseteq C} \sum_{b \in X} b^{\text{value}} \tag{7.1}$$

A variation on this problem is when agents can submit XOR bids. That is, when an agent can say that it wants only one of his bids to win. Computationally, both problems are similar.

where C is a set of all bid sets in which none of the bids share an item, that is

$$C = \{Y \subseteq B \,|\, \forall_{a,b' \in Y} a^{\text{items}} \cap b^{\text{items}} = \emptyset\}. \tag{7.2}$$

Unfortunately, this is not a simple problem as there are, in the worst case, many possible bidsets. Specifically, if bids exists for all subsets of items then X is a way of partitioning the set of items S into non-overlapping subset. That is, take the set of items S and figure out how many ways it can be split into smaller sets. We can calculate this number by remembering that the **Stirling number of the second kind** gives us the number of ways to partition a set of n elements into k non-empty sets. Namely,

STIRLING NUMBER OF THE SECOND KIND

$$S(n,k) = \frac{1}{k!} \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-i)^n. \tag{7.3}$$

Using this formula we can easily determine that the total number of allocations of m items is given by

$$\sum_{i=1}^{m} S(m,i), \tag{7.4}$$

which is bounded by

$$O(m^m) \text{ and } \omega(m^{m/2}).$$

This means that a brute force search of all possible allocations of items to agents is computationally intractable. In fact, no approach will work in polynomial time.

Theorem 7.2. *Winner Determination in Combinatorial Auction is NP-hard. That is, finding the X^* that satisfies* (7.1) *is NP-hard (Rothkopf et al., 1998).*

Even simplifying the problem does not make it easier to solve. For example, say that instead of trying to find the best allocation we simply want to check if there exists an allocation with total revenue of at least w. We call this the **decision version** of the winner determination problem. Lets also further restrict the types of bids the agents can submit. Even under these circumstances the problem remains hard.

DECISION VERSION

Theorem 7.3. *The decision version of the winner determination problem in combinatorial auctions is NP-complete, even if we restrict it to instances where every bid has a value equal to 1, every bidder submits only one bid, and every item is contained in exactly two bids (Cramton et al., 2006, Chapter 12).*

```
BUILD-BRANCH-ON-ITEMS-SEARCH-TREE
1  Create a singleton bid for any item that does not have one
2  Number items from 1 to m
3  Create empty root node
4  for n ∈ M in order
5       do Add as its children all bids that
6            include the smallest item that is not an ancestor of n but
7            that do not include any item that is an ancestor of n.
```

Figure 7.3: Algorithm for building a branch on items search tree. This algorithms does not find a solution, it only builds a tree for the purpose of illustration.

Thus, the problem is very hard, even when we try to limit its complexity. But, there is some hope. The winner determination problem in combinatorial auctions can be reduced to a **linear programming** problem and, therefore, solved in polynomial time with well-known algorithms but only if prices can be attached to single items in the auction (Nisan, 2000). That is, there needs to be a singleton bid for every item. In many cases we can satisfy this requirement by simply adding the missing singleton bids, each with a value of 0. Specifically, the linear program which models the winner determination problem is to find the x that satisfies the following:

LINEAR PROGRAMMING

Simplex is the most widely used linear programming algorithm. It has worst-case exponential time, but in practice it is much faster. Other algorithms exist that are guaranteed polynomial.

Maximize:

$$\sum_{b \in B} x[b] b^{\text{value}}$$

Subject to:

$$\sum_{b \mid j \in b^{\text{items}}} x[b] \leq 1, \forall j \in M$$

$$x[b] \in \{0, 1\}, \forall b \in B,$$

where $x[b]$ is a bit which denotes whether bid b is a winning bid. That is, maximize the sum of the bid values given that each item can be in, at most, one winning bid. It has also been shown that the linear programming problem will solve a combinatorial auction when the bids satisfy any one of the following criteria (Nisan, 2000):

1. All bids are for consecutive sub-ranges of the items.
2. The bids are hierarchical.
3. The bids are only OR-of-XORs of singleton bids.
4. The bids are all singleton bids.
5. The bids are downward sloping symmetric.

A different approach to solving the winner determination problem is to conduct one of the standard AI-searches over all possible allocations, given the bids submitted. The advantage over using a linear programming solver is that we can tweak our AI search algorithms and optimize them to solve our specific problem. That is, we can put some of our domain knowledge into the algorithm to make it run faster, as we shall see.

Before we can do search we need to define our search tree. One way we can build a search tree is by having each node be a bid and each path from the root to a leaf correspond to a set of bids where no two bids share an item. The algorithm for building this tree is shown in figure 7.3. We refer to this tree as a **branch on items** search tree. Figure 7.4 shows an example tree built in this fashion. In this case we have five items for sale, numbered 1–5. The column on the left lists all the bids received. We omit the bid amount and only show the set of items for each bid. The search algorithm uses these bids to build the tree shown on the right of the figure. We start at the first level from the top. All the children of the root are bids that have item 1 in them. Then, we proceed to add children to each node. The

BRANCH ON ITEMS

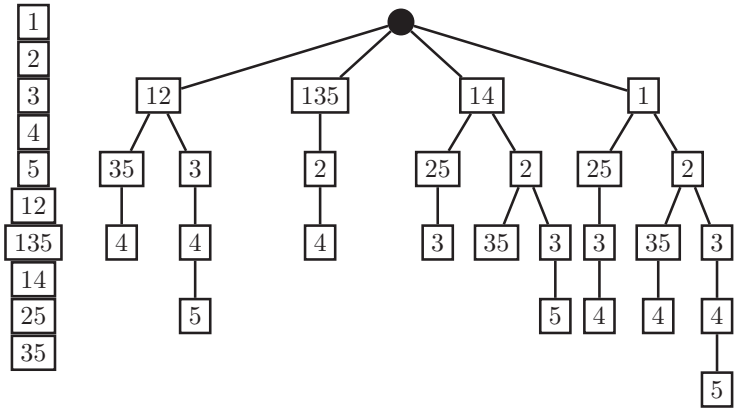


Figure 7.4: Branch on items search tree for winner determination in combinatorial auctions. Note that this tree has 9 leafs (9 possible ways of selling all items given the bids) while the total number of dividing 5 items into subsets is 52.

children of every node will be all the bids that contain the smallest number that is **not** on the path from the root to the node. Since the algorithm has the provision of adding a singleton bid with value 0 for every item, we are guaranteed to find a suitable bid as a children of every node. The only time we cannot find such a bid is when the path from the root to the node contains all items. In this case the node is a leaf and the set of bids from root to leaf constitutes a possible bid set.

The speedup of this search over the brute force method of considering all possible ways of breaking up 5 items into subsets can be confirmed by the fact that this tree has 9 leafs, therefore only 9 working bid sets exists. Meanwhile, the application of the Stirling formula gives us

$$\sum_{i=1...5} S(5,i) = 52,$$

which means that there are 52 ways to break up 5 items into subsets. Clearly, fewer bids means faster run time which is the central idea of the search algorithm. In general, we know that the number of leafs in the tree is bounded.

Theorem 7.4. *The number of leaves in the tree produced by* BUILD-BRANCH-ON-ITEMS-SEARCH-TREE *is no greater than* $(|B|/|M|)^{|M|}$*. The number of nodes is no greater than* $|M|$ *times the number of leaves plus 1 (Sandholm, 2002).*

BRANCH ON BIDS

We can also build a binary tree where each node is a bid and reach edge represents whether or not that particular bid is in the solution. We refer to this tree as a **branch on bids** search tree, an example is shown in Figure 7.5. Each edge on the tree indicates whether the parent node (bid) is to be considered as part of the final bidset. For example, the rightmost branch of the tree consists of all "In" branches so the rightmost leaf node corresponds to the bidset (35)(14)(2) which forms a complete allocation. In practice, the branch on bids search tree is often faster than the previous tree because it gives us more control over the order of bids so we can better optimize the bid order. Also, the branch on bids search does not require us to add dummy singleton bids.

We now have to decide how to search our chosen tree. Since both trees have a number of nodes that is exponential on the number of bids a breadth first search would require too much memory. However, a depth first search should be possible, but time consuming. A branch and bound algorithm like the one we used for DCOP in Chapter 2.2 further helps reduce the search space and speed up computation. In order to implement it we first need a function h which gives us an upper bound on the value of allocating all the items that have yet to be allocated. One such function is h

$$h(g) = \sum_{j \in M - \bigcup_{b \in g} b^{\text{items}}} \max_{b | j \in b^{\text{items}}} \frac{b^{\text{value}}}{|b^{\text{items}}|}, \tag{7.5}$$

where g is the set of bids that have been cleared. The function h simply adds up the maximum possible revenue that each item not in g could contribute by using the

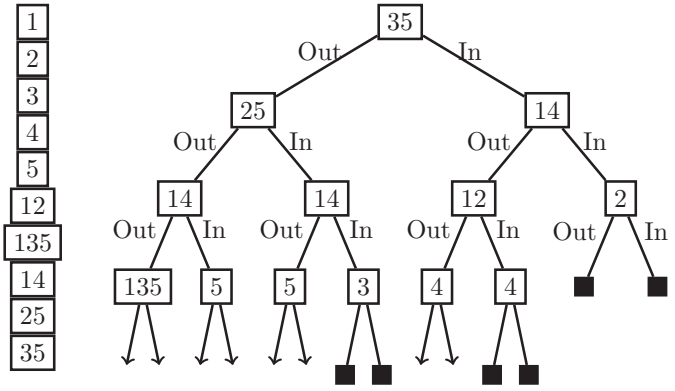


Figure 7.5: Branch on bids partial tree. The black boxes indicate search paths that have terminated because they denote a complete set of bids, that is, no more bids can be added because they contain items already sold.

```
BRANCH-ON-BIDS-CA()
1  r* ← 0        ▷ Max revenue found. Global variable.
2  g* ← ∅        ▷ Best solution found. Global variable.
3  BRANCH-ON-BIDS-CA-HELPER(∅, B)
4  return g*

BRANCH-ON-BIDS-CA-HELPER(g, available-bids)
 1  if available-bids = ∅
 2     then return
 3  if ⋃_{b∈g} b^items = M                          ▷ g covers all items
 4     then if Σ_{b∈g} b^value > r*                 ▷ g has higher revenue than r*
 5             then g* ← g
 6                  r* ← Σ_{b∈g} b^value
 7          return
 8  next ← FIRST(available-bids)
 9  if next^items ∩ ⋃_{b1∈g} b1^items = ∅}          ▷ next's items do not overlap g
10     then g' ← g + next
11          if Σ_{b1∈g'} b1^value + h(g') > r*
12             then BRANCH-ON-BIDS-CA-HELPER(g', REST(available-bids))
13  BRANCH-ON-BIDS-CA-HELPER(g, REST(available-bids))
```

Figure 7.6: A centralized branch and bound algorithm that searchers a branch on bids tree and finds the revenue maximizing solution given a set B of combinatorial bids over items M.

bid that pays the most for each item, divided by the number of items on the bid. This function provides an upper bound since no feasible bidset with higher revenue can exist.

Given the upper bound $h(g)$ we can then implement the branch and bound algorithm shown in figure 7.6. This algorithm searches the branch on bids tree. It maintains a partial solution g to which it adds one bid on each recursive call. Whenever it realizes that partial solution will never be able to achieve revenue that is higher than the best revenue it has already found then it gives up on that subtree, see line 8 of BRANCH-ON-BIDS-CA-HELPER. This algorithm is complete and thus guaranteed to find the revenue maximizing bidset.

We can also use the same heuristic function to do an A^* search. Unfortunately, since A^* acts much like a breadth first search it generally consumes too much memory. A viable solution is to use iterative deepening A^*. IDA^* guesses how much revenue we can expect and runs a depth-first search that prunes nodes that have used more than that. If a solution is not found then the guess is reduced and we try again. IDA^*, with some optimizations, was implemented by the **Bidtree** algorithm (Sandholm, 1999) on the branch on items search tree. In practice, this approach was found to often be slower than a branch and bound search.

BIDTREE

The BRANCH-ON-BIDS-CA algorithm is the basic framework for the Combinatorial Auction Branch on Bids (**CABOB**) algorithm (Sandholm et al., 2005). CABOB

CABOB

```
BRANCH-ON-ITEMS-CA()
1  r* ← 0        ▷ Max revenue found. Global variable.
2  g* ← ∅        ▷ Best solution found. Global variable.
3  BRANCH-ON-ITEMS-CA-HELPER(1, ∅)
4  return g*

BRANCH-ON-ITEMS-CA-HELPER(i, g)
1  if i = m                                          ▷ g covers all items
2     then if Σ_{b∈g} b^value > r*                   ▷ g has higher revenue than r*
3             then g* ← g
4                  r* ← Σ_{b∈g} b^value
5          return
6  for b ∈ {b ∈ B | i ∈ b^items ∧ b^items ∩ ⋃_{b1∈g} b1^items = ∅}  ▷ b's items do not overlap g
7        do g' ← g + b
8           if Σ_{b1∈g'} b1^value + h(g') > r*
9              then BRANCH-ON-ITEMS-CA-HELPER(i + 1, g')
```

Figure 7.7: A centralized branch and bound algorithm that searchers a branch on items tree and finds the revenue maximizing solution given a set B of combinatorial bids over items M.

improves the performance of the basic algorithm in several ways, one of which is by improving the search for new bids to add to the partial solution. Specifically, we note that a naive implementation of line 6 of BRANCH-ON-BIDS-CA-HELPER would mean that we would build this set on each recursive call to the function. That would be very time consuming as there are an exponential number of bids in B. CABOB handles this problem by maintaining graph data structure which has all the bids that can still be used given g. The nodes in the graph are the bids that are still available and the edges connect every pair of bids that share an item. In this way when a new bid is added to g it is removed from the graph as well as all the other bids that are connected to it.

CASS

We can also perform the branch and bound search on the branch on items search tree, as shown in Figure 7.7. This algorithm is the basis for the **CASS** (Combinatorial Auction Structured Search) algorithm which also implements further refinements on the basic algorithm (Fujishima et al., 1999).

Most algorithms for centralized winner determination in combinatorial auction expand on the basic branch and bound search by using specialized data structures to speed up access to the information need—the viable bids given the current partial solution—and implement heuristics which have been shown to reduce the size of the search space, especially for certain popular bid distributions. Some heuristics that have been found useful include the following:

- Keep only the highest bid for any set. That is, if there is a bid of \$10 for items 1,2 and another bid of \$20 for 1,2 then we get rid of the \$10 bid.

- Remove provably noncompetitive bids, that is, those that are dominated by another bid or sets of bids. For example, if there is a bid for \$10 for item 1 and another bid for \$5 for items 1,2 then the \$10 bid dominates the \$5 bid—any situation in which we choose the \$5 bid would be made better if we changed that bid for the \$10 bid.

- Decompose bids into connected sets, each solved independently. If we can separate the set of bids into two or more sets of bids where all bids for any item are to be found in only one of the sets then this set of bids becomes a smaller, and independent, winner determination problem.

- Mark noncompetitive tuple of bids. For example, if there are bids \$1:(1,2), \$1:(3,4), \$10:(1,3), \$10:(2,4) then the pair of \$10 bids dominates the pair of \$1 bids, so we can get rid of them.

- In the branch-on-items tree place the items with the most bids first on the tree. This way the most constrained items are tried first thereby creating fewer leafs.
- If the remaining search subtree is the same for different nodes in the search tree, as can happen when different items are cleared but by different bids, then these subtrees can be cached. The subtree is solved once and the answer, that is, the best set of bids found in it, is saved for possible future use.

In general the best speed attainable by the best algorithms varies greatly depending on the type of bids submitted. For example, if the number of items in each bid is chosen from a flat probability distribution then we can solve problems with thousands of items and tens of thousands of bids in seconds. On the other hand, if each bid contains exactly five randomly chosen items and a price of 1 then we can only solve problems with tens of items and hundreds of bids in a minute. The Combinatorial Auction Test Suite (**CATS**) can generate realistic types of bid distributions so new algorithms can be compared to existing ones using realistic bid sets (Leyton-Brown et al., 2000). It generates these bids by using several sample scenarios. In one scenario there is a graph where the nodes represent cities and the edges are railroad tracks that connect these cities. The items for sale are the tracks between cities, that is, the edges. The agents are given pairs of host/destination cities and are told that they need to establish a train route between their city pairs. Thus, each agent determines all the possible sets of edges which connect his city pairs and submits XOR combinatorial bids on them. The value of each path depends on the total distance; shorter routes are preferred.

CATS

7.3.2 Distributed Winner Determination

One problem with the centralized winner determination algorithms, aside from the bottleneck, is that they require the use of a trusted auctioneer who will perform the needed computations. Another option is to build a peer-to-peer combinatorial auction protocol which lets the sellers themselves determine the winning set of bids and discourages them from cheating.

Incremental Auctions: Distribute over Bidders

One way to distribute the winner determination calculation is by offloading it on the bidding agents. We can do this by using an increasing price auction and making the bidders figure out which bids would be better than the current standing bid. This is the approach taken by the Progressive Adaptive User Selection Environment or **PAUSE** combinatorial auction (Kelly and Stenberg, 2000) (Cramton et al., 2006, Chapter 6).

PAUSE

A PAUSE auction for m items has m stages. Stage 1 consists of having simultaneous ascending price open-cry auctions for each individual item. During this stage the bidders can only place individual bids on items. At the end of this state we will know what is the highest bid for each individual item and who placed that bid. In each successive stage $k = 2, 3, \ldots, m$ we hold an ascending price auction where the bidders must submit sets of bids that cover all items but each one of the bids must be for k items or less. The bidders are allowed to use bids that other agents have placed in previous rounds. Also, any new bid set has to have a sum of bid prices which is bigger than the currently winning bid set.

At the end of each stage k all agents know the best bid for every subset of size k or less. Also, at any point in time after stage 1 has ended there is a standing bid set whose value increases monotonically as new bid sets are submitted. Since in the final round all agents consider all possible bid sets, we know that the final winning bid set will be one such that no agent can propose a better bidset. Note, however, that this bid set will generally not be X^* since we are using ascending price auction so the winning bid will be only slightly bigger than the second highest bid for the particular set of items.

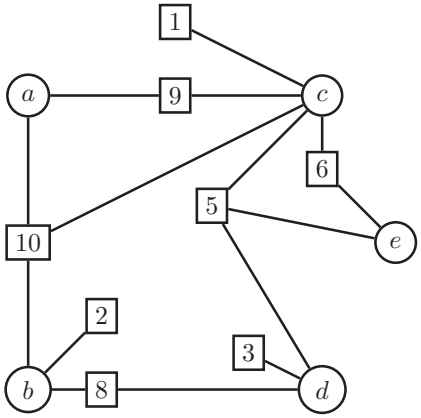


Figure 7.8: Graphical representation of a distributed winner determination problem. The circles represent agents/items while the squares represent combinatorial bids.

ENVY-FREE

In the general case, the PAUSE auction has been shown to be **envy-free** in that at the conclusion of the auction no bidder would prefer to exchange his allocation with that of any other bidder. However, it is not guaranteed to find the utilitarian solution (7.1).

The PAUSE auction makes the job of the auctioneer very easy. All it has to do is make sure each new bidset adds up to a number that is bigger than the current best as well as make sure that any bids an agent places that are not his do indeed correspond to other agents' bids. The computational problem shits from one of winner determination by the auctioneer to one of bid generation by the prospective buyer agents. Each agent must search over the space of all bid sets which contain at least one of its bids. The search is made easier by the fact that the agent need only consider the current best bids and that in stage k all bid sets must contain at least one bid of size k since they would have otherwise been bid in a previous stage. The **pausebid** algorithm uses the same branch and bound techniques used in centralized winner determination but expands them to include the added constraints an agent faces. As such, the pausebid algorithm can be used to find the myopically optimal bid for agents in the PAUSE auction (Mendoza and Vidal, 2007). The research also reveals that agents using pausebid reach the same allocation as the utilitarian solution, assuming all the agents bid their true valuations, at least 95% of the time.

PAUSEBID

Another way of distributing the winner determination problem among the bidders is provided by the Virtual Simultaneous Auction (**VSA**) (Fujishima et al., 1999) which is based on market-oriented programming ideas (Wellman, 1996). The VSA is an iterative algorithm where successive auctions for the items are held and the bidders change their bids based on the last auction's outcome. The auction is guaranteed to find the optimal solution when the bidding terminates. Unfortunately, there is no guarantee that bidding will terminate and experimental results show that in most cases bidding appears to go on forever.

VSA

Distribute over Sellers

Another way to distribute the problem of winner determination is to distribute the actual search for the winning bid set among the sellers. Imagine a distributed system where each person that wants to sell an item runs an agent. The agent advertises that the item is for sale. Every person who wants to place a, possibly combinatorial, bid does so by telling all the agents present in the bid about it. After some time the agents have gathered some bids and begin the clearing process. The set of agents and bids can be visualized in a graph such as Figure 7.8.

Here we see that agent b has received three bids. One of them is a singleton bid worth \$2, the other two are combinatorial bids one of them for \$8 and including agent d and the other for \$10 and including agents a and c. The problem we face is how can the agents a–e determine the set of winning bids.

The simplest solution is to do a complete search via sequentialized ordering. In

this method we use the same search tree as before but instead of implementing it centrally we pass messages among the agents. Agent 1 handles the top level of the tree. It tentatively sets one of its bids as cleared and sends a message to agent 2. In this way, each agent (roughly) handles one level of the tree. Note the agents are sequentialized so there is no parallelism, even though it is distributed.

Another option is to partition the problem into smaller pieces then have sets of agents to a complete search via sequentialized ordering on each of the parts. That is, we first partition the set of agents then do a complete search on each subset. This method means that each subset works in parallel. However, if there is any bid that contains agents from more than one subset then the solution found is no longer guaranteed to be optimal.

Another option is to maximize the available parallelism by having the agents do **individual hill-climbing**. Each agent starts by ordering all its bids based on their price divided by the number of agents in the bid under the assumption that the agent gets $1/n$ of a bid that includes n agents. The agent picks the first bid in the list (the highest valued) and tells all the other agents in the bid that it wants to clear it. If the agent receives similar messages from all the agents in the bid this means that they all wanted to clear it so the bid is considered cleared and all the agents in it are removed from the protocol. The remaining agents repeat the same process with their next highest bid and so on until they run out of bids. This algorithm is guaranteed to terminate but will only find, at best, a local optima.

INDIVIDUAL HILL-CLIMBING

Winner Determination as Constraint Optimization

It is interesting to note that we can reduce the winner determination problem to a constraint optimization problem as described in Chapter 2.2 in two different ways. One way is to let the variables $x_1, \ldots, x_m$ be the items to be sold and their domains be the set of bids which include the particular item. That is, each item k is represented by a variable x_k with domain D_k which contains all the bids that involve k. The constraints are given by the bids. Every bid is replaced with a constraint which returns the value of the bid if all the items in the bid have a value equal to that bid. That is, if all the items are cleared for that bid/constraint then that constraint returns its value, otherwise the constraints returns a value of zero. In this problem we now want to maximize the value returned by the constraints.

We can also reduce the winner determination problem by letting the variables be the bids themselves with binary domains which indicate whether the bid has been cleared or not. We then need two sets of constraints. One set of constraints has to be generated such that they give a very large penalty if any two bids with items in common have been cleared, so as to make sure that never happens. The other set simply gives a value for every cleared bid equal to the value of that bid.

Since both of these are standard constraint optimization problems we can use the algorithms from Chapter 2.2 to solve them in a distributed manner. However, as those algorithms are generic, it seems likely that they will not perform as well as the specialized algorithms.

7.3.3 Bidding Languages

We have thus far assumed that each buyer can submit a set of bids, where each bid b specifies that he is willing to pay b^{value} for the set of items b^{items}. Implicit in the set of bids is the fact that the agent is also willing to accept winning two or more of his bids. That is, if b and b' are two bids for non-overlapping sets of items then any agent that places them should also be happy to win both bids. This bidding language is known as **or bids**, because agents can place multiple **atomic bids** and they can win any feasible combination of the bids they submit. That is, the agent expresses his valuation as b_1 OR b_2 OR. . . OR b_k.

OR BIDS
ATOMIC BIDS

A limitation of OR bids is that they cannot represent sub-additive valuations. For example, a sub-additive valuation arises if you have a value of 10 for a red hat and 10 for a blue hat but together value them at 11 because you really only need

one hat. In this scenario if you placed the individual bids as OR bids it could be that you end up paying 10 for both hats. We thus say that OR bids are not a complete bidding language since they cannot represent all possible valuations.

XOR BIDS

Xor bids, on the other hand, can represent all possible valuations. An XOR bid takes the form of a series of atomic bids joined together by exclusive-or operations: b_1 XOR b_2 XOR... XOR b_k. This bid represents the fact that the agent is willing to win any *one* of the bids, but not more than one. Thus, you can use it to place a bid that says you are willing to pay 10 for a red hat or 10 for a blue hat or 11 for both but want to buy at most one of them.

One problem with XOR bids is that they can get very long for seemingly common valuations that can be more succinctly expressed using the OR bids. For example, say an agent has a purely additive valuation function over a set of items, that is if $s = s' \cup s''$ then $v(s) = v(s') + v(s'')$. This agent could have expressed this valuation by placing an OR bid where each atomic bid was just for one item. Implicit in this bid is the fact that the agent would be willing to accept any subset of the items as long as he gets paid the sum of the individual valuation. If the same agent had to place an XOR bid it would have to place an atomic bid for every subset of items, and there are $2^{|s|}$ such subsets.

OR* BIDS

Another practical problem with XOR bids is that most of the winner determination algorithms are designed to work with OR bids. The problem can be solved by adding dummy items to OR bids, these bids are known as **or* bids**. For example, if an agent wants to place a bid for item a and b, but not both, it could generate a dummy item d and place an OR bid for items $\{a, d\}$ and $\{b, d\}$. This way the agent guarantees that it will not win both a and b. OR* combines the completeness of XOR bids with the succinctness of OR bids without adding too many new dummy items. In fact, any bid that can be expressed by OR or XOR using x atomic bids can be represent using an OR* bids with at most x^2 dummy items (Nisan, 2000). Thus, all the winner determination algorithms we have studied can also be used with XOR bids as long as you first convert them to OR* bids.

7.3.4 Preference Elicitation

We can also try to reduce the amount of information the bidders must supply by trying to only ask them about those valuations that are important in finding the utilitarian solution (Hudson and Sandholm, 2004). This can best be achieved in the common case of **free disposal** where there is no cost associated with the disposal of an item, that is, if $S \subseteq T$ then $v(S) \leq v(T)$. For example, if we know that an agent values item a at 10 then we know that it must also value the set (a, b) at *least* at 10. Similarly, if the agent values items (a, b) at 5 then we know that its value for item a is at *most* 5.

FREE DISPOSAL

Given free disposal, an auctioneer can incrementally explore a network like the one in Figure 7.9 which shows all the possible subsets of items and associates with each one a upper bound (UB) and a lower bound (LB) on the valuation that the agent has for that particular set of items. The directed edges indicate which sets are preferred over other ones. For example, the agent will always prefer the set $\{a, b, c\}$ over the set $\{a, b\}$ and even, transitively, over the set $\{a\}$. The graph also makes it easy to see how the auctioneer can propagate the bounds he learns on one set to the others. Namely, the lower bounds can be propagated upstream and the upper bounds can be propagated downstream. For example, in the figure there is a lower bound of 5 the set $\{b\}$, knowing this the auctioneer can immediately set the lower bounds of $\{b, c\}$ and $\{a, b, c\}$ to 5. Similarly, the upper bound of 9 for the set $\{a, c\}$ can be propagated down to $\{a\}$, $\{c\}$, and $\emptyset$. Note also that if the agent tells the auctioneer its exact valuation for a particular set then the auctioneer will set both the upper and lower bounds of that set to the given value.

The goal of an elicitation auctioneer is to minimize the amount of questions that it asks the bidders while still finding the best allocation. If we limit the auctioneer to only asking questions of the type *"What is your n^{th} most preferred set?"* then

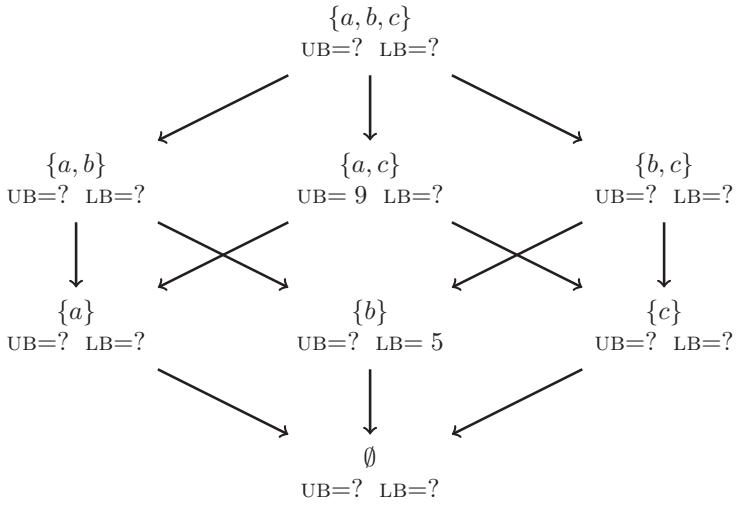


Figure 7.9: Constraint network for determining an agent's valuation. Directed edges indicate preferred sets.

```
PAR()
 1  fringe ← {{1, ..., 1}}
 2  while fringe ≠ ∅
 3        do c ← first(fringe)
 4           fringe ← rest(fringe)
 5           successors ← CHILDREN(c)
 6           if FEASIBLE(c)
 7              then pareto-solutions ← pareto-solutions ∪ c
 8              else for n ∈ successors
 9                         do if n ∉ fringe ∧UN-DOMINATED(n, pareto-solutions)
10                               then fringe ← fringe ∪ n

CHILDREN({k_1, ..., k_n})
1  for i ∈ 1...n
2        do c ← {k_1, ..., k_n}
3           c[i] ← c[i] + 1
4           result ← result ∪ c
5  return result
```

Figure 7.10: The PAR algorithm. The procedure FEASIBLE($\{k_1, \ldots, k_n\}$) asks each bidder i for its k_i most valued set, if we haven't asked before. It uses these sets to determine if, together, they form a feasible allocation. The CHILDREN procedure returns a set of possible solutions, by adding 1 to each position in $\{k_1, \ldots, k_n\}$.

we will be unable to find the revenue maximizing allocation. However, we can still find the Pareto optimal allocations.

PAR ALGORITHM

The **PAR algorithm**, shown in figure 7.10, allows an elicitation auctioneer to find a Pareto optimal solution by only using rank questions (Conen and Sandholm, 2002). It does this by incrementally building a solution lattice for the bidders. Figure 7.11 shows an example of a complete lattice for two bidders. The PAR algorithm maintains a set variable, called the *fringe*, of possible Pareto optimal allocations. At the first time step the auctioneer adds the solution {1,1} to the *fringe*, where {1,1} represents the solution formed by using both agents' most preferred solution. In every succeeding step the auctioneer picks some solution from the *fringe* and asks the agents for those sets, if it does not already know them. This communication with the bidders occurs within the FEASIBLE procedure. In the example figure both agents prefer the set $\{a, b\}$ the most so they will both respond with this set. Since the set of bids $(\{a, b\}, \{a, b\})$ is not feasible the algorithm checks to make sure that each one of the CHILDREN of {1,1}, in this case {1,2} and {2,1}, is not dominated by any other allocation in the set *pareto-solutions* and adds it to the *fringe* if its not already there. The algorithm continues in this way until the *fringe* is empty, at that point the variable *pareto-solutions* contains all the Pareto allocations for the

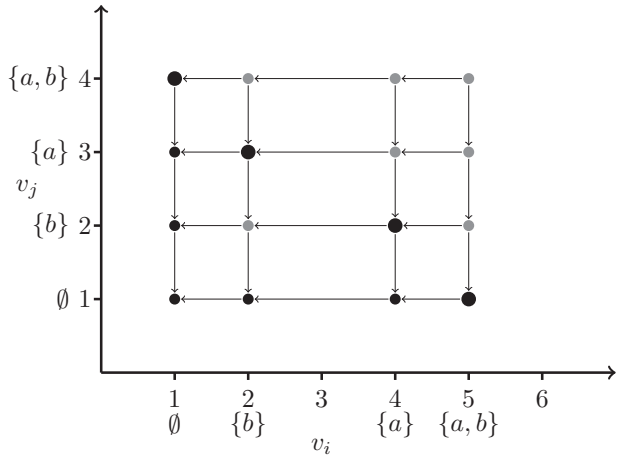


Figure 7.11: Rank lattice. The dots represent all possible allocations. Directed edges represent Pareto dominance. Grey dots are infeasible while black dots are feasible allocations. The PAR search starts at the top rightmost point and stops when it has identified the complete Pareto frontier of feasible allocations—the larger black dots.

```
EBF()
 1  fringe ← {{1, ..., 1}}
 2  if |fringe| = 1
 3     then c ← first(fringe)
 4     else  M ← {k ∈ fringe | v(k) = max_{d∈fringe} v(d)}
 5           if |M| ≥ 1 ∧ ∃_{d∈M} FEASIBLE(d)
 6              then return d
 7              else  c ← PARETO-SOLUTION(M)
 8  if FEASIBLE(c)
 9     then return c
10  successors ← CHILDREN(c)
11  for n ∈ successors
12        do if n ∉ successors
13              then fringe ← fringe ∪ {n}
14  goto 2
```

Figure 7.12: The EBF algorithm. FEASIBLE(d) returns true if d is a feasible allocation. PARETO-SOLUTION(M) returns one allocation from M which is not Pareto-dominated by any other allocation in M.

problem.

Since the PAR algorithm does not ask the agents for their valuation values it cannot determine which one of the Pareto allocations is the utilitarian allocation. Of course, once we start asking for valuations we have a better idea of which bids will likely be part of the utilitarian allocation, namely those sets that have the highest value per item.

EFFICIENT BEST FIRST

The **efficient best first** (EBF) algorithm performs a search similar to the one that PAR implements but it also asks for the values of the sets and always expands the allocation in the *fringe* which has the highest value (Conen and Sandholm, 2002). Figure 7.12 shows the algorithm. This algorithm will find the utilitarian allocation.

Unfortunately, both PAR and EBF have worst case running times that are exponential in the number of items. They also perform rather poorly in practice. PAR does not make any effort at trying to pick a item set to ask about, it simply chooses randomly, so its bad performance is not surprising. EBF's elicitation strategy has also been shown to perform poorly in experiments—it asks too many questions. Its performance also degrades as more agents are added.

A more general framework for elicitation is to maintain a set of allocations which are potentially optimal. Initially, this set would contain all allocations in the general case. In cases where we can assume some structure for the value function, such as free disposal, then it contains all those allocations that are not dominated by another. The elicitation algorithm can then choose one allocation from this set and ask the agents their values for the sets they receive in that allocation. These

values can then be propagated to other sets and a new allocation chosen (Conen and Sandholm, 2001b; Conen and Sandholm, 2001a).

Within this general framework there are various elicitation strategies we could try. The simplest one is to randomly choose an allocation from the set. This technique has been shown to require a number of elicitations that is proportional to $n2^m$ where n is the number of agents and m is the number of items. Another strategy is to choose the candidate with the highest lower bound value on the expectation that a candidate with a higher value is more likely to be the optimal choice, or at least will need to be eliminated from competition. Experiments have shown that this strategy performs better than random elicitations.

7.3.5 VCG Payments

VCG payments, which we will see in Chapter 8.2, can be applied to combinatorial auctions (MacKie-Mason and Varian, 1994). In a VCG combinatorial auction the bid set with maximum payment is chosen as the winner but the bidders do not have to pay the amounts they bid. Instead, each bidder pays the difference in the total value that the *other* bidders would have received if he had not bid (and the best set of bids was chosen from their bids) minus the total value the other bidders receive given his bid. Each bidder thus get a payment that is proportional to his contribution to everyone else's valuation. This has the desirable effect of aligning the bidders' incentives with the utilitarian allocation and thus eliminating the incentive to lie about their true valuation. However, it increases the computational requirements as we now have to also solve a winner determination problem for every subset of $n-1$ agents in order to calculate the payments. That is, VCG payments increase the work by a factor of n, where n is the number of agents.

Exercises

7.1 The branch and bound algorithm for the branch on bids search tree, seen in figure 7.6, does not specify in which order the bids should be searched. Give a item heuristic for this ordering and explain why it should, on average, help find a solution quicker than expanding bids in a random order.

7.2 What is the set of winning bids given the following bids?

Price	Bid items
\$1	Beast Boy
\$3	Robin
\$5	Raven, Starfire
\$6	Cyborg, Robin
\$4	Cyborg, Beast Boy
\$3	Raven, Beast Boy

7.3 Show how we can reduce the problem of winner determination in a combinatorial auction to a constraint optimization problem.

7.4 Can the problem of winner determination in a combinatorial auction be reduced to a constraint satisfaction problem? Show your proof.

7.5 You are given a painting to sell at an auction and wish to maximize its sale price. What type of auction should you use? Explain.

7.6 In Chapter 6.5 we mentioned the postman problem. How can the postmen use a combinatorial auction to solve their problem? Explain how the bids are to be generated.

7.7 Why is a branch on bids search faster than a branch on items search for winner determination in combinatorial auctions?

7.8 In a combinatorial auction with 50 items and using a computer that takes 1 millisecond to explore each bidset, what is an upper bound on the amount of time it would take to find a solution to the winner determination problem?

7.9 The winner determination problem in combinatorial auction seeks to maximize revenue, that is, maximize the amount paid by the buyers. Provide three reasons or situations in which this solution might not be the most desirable on.