2

Distributed Optimization

In the previous chapter we looked at distributed ways of meeting global constraints. Here we up the ante; we ask how agents can, in a distributed fashion, optimize a global objective function. Specifically, we consider four families of techniques and associated sample problems. They are, in order:

- distributed dynamic programming (as applied to path-planning problems);
- distributed solutions to Markov Decision Problems (MDPs);
- optimization algorithms with an economic flavor (as applied to matching and scheduling problems); and
- coordination via social laws and conventions, and the example of traffic rules.

2.1 Distributed dynamic programming for path planning

Like graph coloring, path planning constitutes another common abstract problem-solving framework. A path-planning problem consists of a weighted directed graph with a set of n nodes N, directed links L, a weight function $w : L \mapsto \mathbb{R}^+$, and two nodes $s, t \in N$. The goal is to find a directed path from s to t having minimal total weight. More generally, we consider a set of goal nodes $T \subset N$, and are interested in the shortest path from s to any of the goal nodes $t \in T$.

This abstract framework applies in many domains. Certainly it applies when there is some concrete network at hand (e.g., a transportation or telecommunication network). But it also applies in more roundabout ways. For example, in a planning problem the nodes can be states of the world, the arcs actions available to the agent, and the weights the cost (or, alternatively, time) of each action.

2.1.1 *Asynchronous dynamic programming*

principle of optimality

Path planning is a well-studied problem in computer science and operations research. We are interested in distributed solutions, in which each node performs a local computation, with access only to the state of its neighbors. Underlying our solutions will be the *principle of optimality*: if node x lies on a shortest path from s to t, then the portion of the path from s to x (or, respectively, from x to t) must also be the shortest paths between s and x (resp., x and t). This

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

dynamic programming

allows an incremental divide-and-conquer procedure, also known as *dynamic programming*.

asynchronous dynamic programming

Let us represent the shortest distance from any node i to the goal t as $h^*(i)$. Thus the shortest distance from i to t via a node j neighboring i is given by $f^*(i, j) = w(i, j) + h^*(j)$, and $h^*(i) = \min_j f^*(i, j)$. Based on these facts, the AsynchDP algorithm has each node repeatedly perform the following procedure. In this procedure, given in Figure 2.1, each node i maintains a variable $h(i)$, which is an estimate of $h^*(i)$.

Figure 2.2 shows this algorithm in action. The h values are initialized to ∞, and incrementally decrease to their correct values. The figure shows three iterations; note that after the first iteration, not all finite h values are correct; in particular, the value 3 in node d still overestimates the true distance, which is corrected in the next iteration.

procedure AsynchDP (node i)
if *i is a goal node* **then**
  $h(i) \leftarrow 0$
else
  initialize $h(i)$ arbitrarily (e.g., to ∞ or 0)
repeat
  forall *neighbors j* **do**
    $f(j) \leftarrow w(i, j) + h(j)$
  $h(i) \leftarrow \min_j f(j)$

Figure 2.1 The asynchronous dynamic programming algorithm.

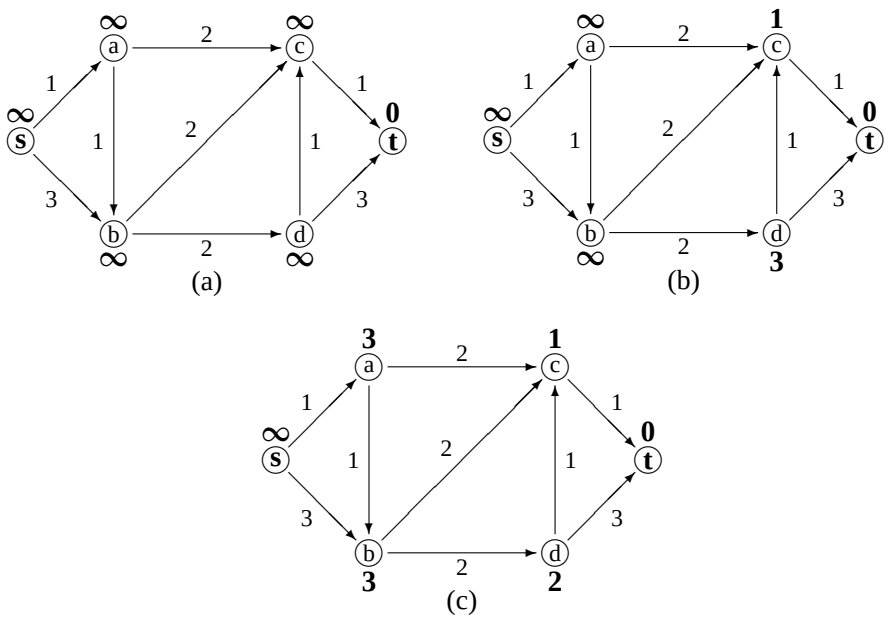

Figure 2.2 Asynchronous dynamic programming in action.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

```
procedure LRTA*
i ← s                                              // the start node
while i is not a goal node do
    foreach neighbor j do
        f(j) ← w(i, j) + h(j)
    i' ← arg min_j f(j)                            // breaking ties at random
    h(i) ← max(h(i), f(i'))
    i ← i'
```

Figure 2.3 The learning real-time A* algorithm.

One can prove that the AsynchDP procedure is guaranteed to converge to the true values, that is, h will converge to h^*. Specifically, convergence will require one step for each node in the shortest path, meaning that in the worst case convergence will require n iterations. However, for realistic problems this is of little comfort. Not only can convergence be slow, but this procedure assumes a process (or agent) for each node. In typical search spaces one cannot effectively enumerate all nodes, let alone allocate them each a process. (For example, chess has approximately 10^{120} board positions, whereas there are fewer than 10^{81} atoms in the universe and there have only been 10^{26} nanoseconds since the Big Bang.) So to be practical we turn to heuristic versions of the procedure, which require a smaller number of agents. Let us start by considering the opposite extreme in which we have only one agent.

2.1.2 *Learning real-time A**

learning real-time A* (LRTA*)

In the *learning real-time A**, or LRTA*, algorithm, the agent starts at a given node, performs an operation similar to that of asynchronous dynamic programming, and then moves to the neighboring node with the shortest estimated distance to the goal, and repeats. The procedure is given in Figure 2.3.

As earlier, we assume that the set of nodes is finite and that all weights $w(i, j)$ are positive and finite. Note that this procedure uses a given heuristic function $h(\cdot)$ that serves as the initial value for each newly encountered node. For our purposes it is not important what the precise function is. However, to guarantee certain properties of LRTA*, we must assume that h is *admissible*. This means that h never overestimates the distance to the goal, that is, $h(i) \leq h^*(i)$. Because weights are nonnegative we can ensure admissibility by setting $h(i) = 0$ for all i, although less conservative admissible heuristic functions (built using knowledge of the problem domain) can speed up the convergence to the optimal solution. Finally, we must assume that there exists some path from every node in the graph to a goal node. With these assumptions, LRTA* has the following properties:

admissible heuristic

- The h-values never decrease, and remain admissible.
- LRTA* terminates; the complete execution from the start node to termination at the goal node is called a *trial*.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

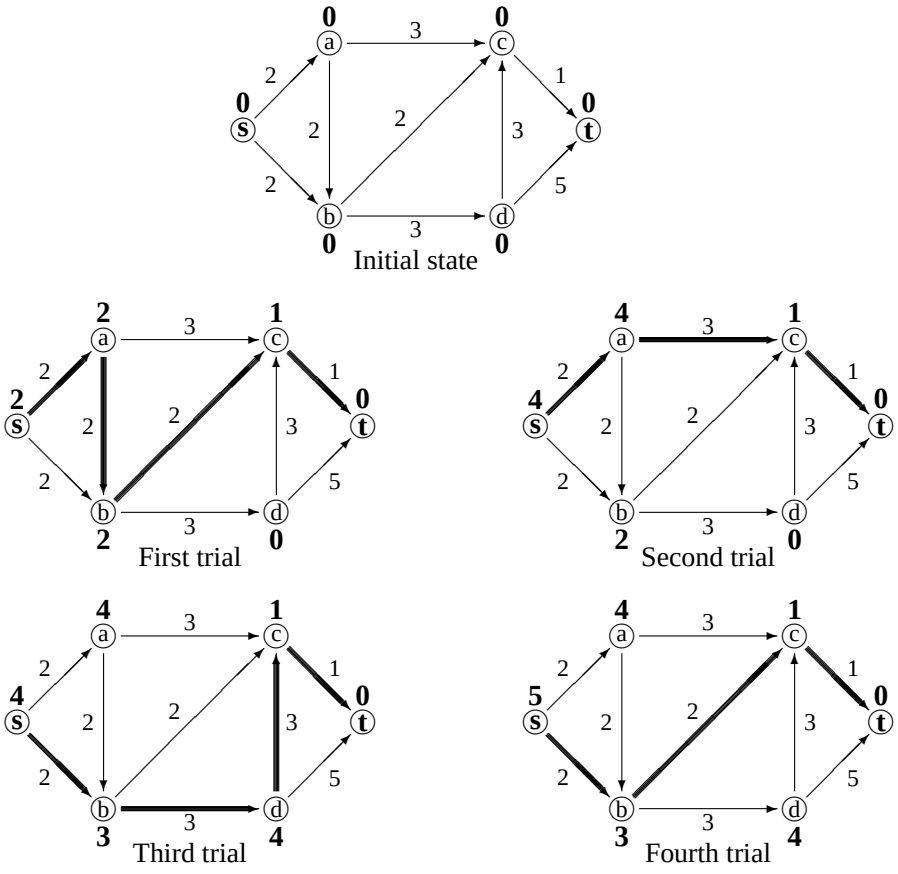

Figure 2.4 Four trials of LRTA*.

- If LRTA* is repeated while maintaining the h-values from one trial to the next, it eventually discovers the shortest path from the start to a goal node.
- If LRTA* find the same path on two sequential trials, this is shortest path. (However, this path may also be found in one or more previous trials before it is found twice in a row. Do you see why?)

Figure 2.4 shows four trials of LRTA*. Do you see why admissibility of the heuristic is necessary?

LRTA* is a centralized procedure. However, we note that rather than have a single agent execute this procedure, one can have multiple agents execute it. The properties of the algorithm (call it LRTA*(n), with n agents) are not altered, but the convergence to the shortest path can be sped up dramatically. First, if the agents each break ties differently, some will reach the goal much faster than others. Furthermore, if they all have access to a shared h-value table, the learning of one agent can teach the others. Specifically, after every round and for every i, $h(i) = \max_j h_j(i)$, where $h_j(i)$ is agent j's updated value for $h(i)$. Figure 2.5 shows an execution of LRTA*(2)—that is, LRTA* with two agents—starting from

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

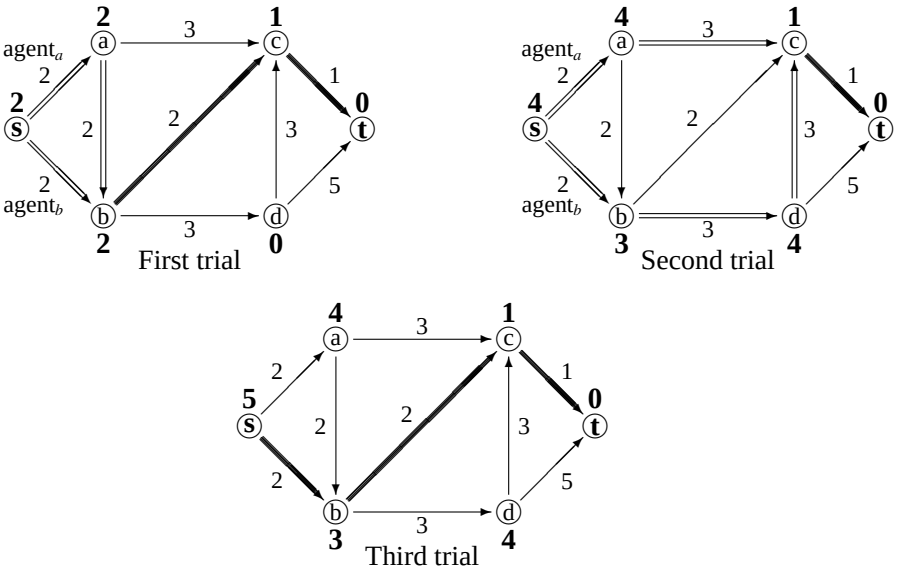

Figure 2.5 Three trials of LRTA*(2).

the same initial state as in Figure 2.4. (The hollow arrows show paths traversed by a single agent, while the dark arrows show paths traversed by both agents.)

2.2 Action selection in multiagent MDPs

In this section we discuss the problem of optimal action selection in multiagent MDPs.[1] Recall that in a single-agent MDP the optimal policy π^* is characterized by the mutually-recursive Bellman equations:

$$Q^{\pi^*}(s,a) = r(s,a) + \beta \sum_{\hat{s}} p(s,a,\hat{s}) V^{\pi^*}(\hat{s})$$

$$V^{\pi^*}(s) = \max_a Q^{\pi^*}(s,a)$$

value iteration

Furthermore, these equations turn into an algorithm—specifically, the dynamic-programming-style *value iteration algorithm*—by replacing the equality signs "=" with assignment operators "←" and iterating repeatedly through those assignments.

However, in real-world applications the situation is not that simple. For example, the MDP may not be known by the planning agent and thus may have to be learned. This case is discussed in Chapter 7. But more basically, the MDP may simply be too large to iterate over all instances of the equations. In this case, one approach is to exploit independence properties of the MDP. One case where this arises is when the states can be described by feature vectors; each feature can take on many values, and thus the number of states is exponential in the number

1. The basics of single-agent MDPs are covered in Appendix C.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

of features. One would ideally like to solve the MDP in time polynomial in the number of features rather than the number of states, and indeed techniques have been developed to tackle such MDPs with factored state spaces.

multiagent MDP

We do not address that problem here, but instead on a similar one that has to do with the modularity of actions rather than of states. In a *multiagent MDP* any (global) action a is really a vector of local actions $(a_1, \ldots, a_n)$, one by each of n agents. The assumption here is that the reward is common, so there is no issue of competition among the agents. There is not even a problem of coordination; we have the luxury of a central planner (but see discussion at the end of this section of parallelizability). The only problem is that the number of global actions is exponential in the number of agents. Can we somehow solve the MDP other than by enumerating all possible action combinations?

We will not address this problem, which is quite involved, in full generality. Instead we will focus on an easier subproblem. Assuming that the Q values for the optimal policy have already been computed. How hard is it to decide on which action each agent should take? Since we are assuming away the problem of coordination by positing a central planner, on the face of it the problem is straightforward. In Appendix C we state that once the optimal (or close to optimal) Q values are computed, the optimal policy is "easily" recovered; the optimal action in state s is $\arg\max_a Q^{\pi^*}(s, a)$. But of course if a ranges over an exponential number of choices by all agents, "easy" becomes "hard." Can we do better than naively enumerating over all action combinations by the agents?

In general the answer is no, but in practice, the interaction among the agents' actions can be quite limited, which can be exploited both in the representation of the Q function and in the maximization process. Specifically, in some cases we can associate an individual Q_i function with each agent i, and express the Q function (either precisely or approximately) as a linear sum of the individual Q_is:

$$Q(s, a) = \sum_{i=1}^{n} Q_i(s, a).$$

The maximization problem now becomes

$$\arg\max_a \sum_{i=1}^{n} Q_i(s, a).$$

This in and of itself is not very useful, as one still needs to look at the set of all global actions a, which is exponential in n, the number of agents. However, it is often also the case that each individual Q_i depends only on a small subset of the variables.

For example, imagine a metal-reprocessing plant with four locations, each with a distinct function: one for loading contaminated material and unloading reprocessed material; one for cleaning the incoming material; one for reprocessing the cleaned material; and one for eliminating the waste. The material flow among them is depicted in Figure 2.6.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

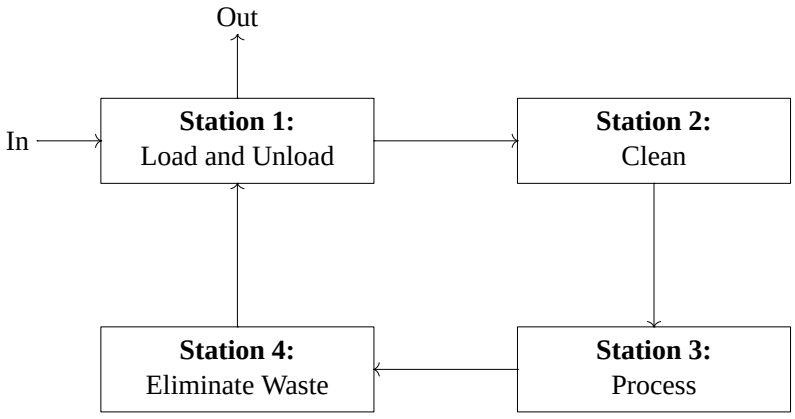

Figure 2.6 A metal-reprocessing plant.

Each station can be in one of several states, depending on the load at that time. The operator of the station has two actions available: "pass material to next station in process," and "suspend flow." The state of the plant is a function of the state of each of the stations; the higher the utilization of existing capacity the better, but exceeding full capacity is detrimental. Clearly, in any given global state of the system, the optimal action of each local station depends only on the action of the station directly "upstream" from it. Thus in our example the global Q function becomes

$$Q(a_1, a_2, a_3, a_4) = Q_1(a_1, a_2) + Q_2(a_2, a_4) + Q_3(a_1, a_3) + Q_4(a_3, a_4)$$

and we wish to compute

$$\arg\max_{(a_1,a_2,a_3,a_4)} Q_1(a_1, a_2) + Q_2(a_2, a_4) + Q_3(a_1, a_3) + Q_4(a_3, a_4).$$

Note that in the preceding expressions we omit the state argument, since that is being held fixed; we are looking at optimal action selection at a given state.

variable elimination

In this case we can employ a *variable elimination* algorithm, which optimizes the choice for the agents one at a time. We explain the operation of the algorithm via our example. Let us begin our optimization with agent 4. To optimize a_4, functions Q_1 and Q_3 are irrelevant. Hence, we obtain

$$\max_{a_1,a_2,a_3} Q_1(a_1, a_2) + Q_3(a_1, a_3) + \max_{a_4}[Q_2(a_2, a_4) + Q_4(a_3, a_4)].$$

conditional strategy

We see that to make the optimal choice over a_4, the values of a_2 and a_3 must be known. Thus, what must be computed for agent 4 is a *conditional strategy*, with a (possibly) different action choice for each action choice of agents 2 and 3. The value that agent 4 brings to the system in the different circumstances can be summarized using a new function $e_4(A_2, A_3)$ whose value at the point a_2, a_3 is the value of the internal max expression

$$e_4(a_2, a_3) = \max_{a_4}[Q_2(a_2, a_4) + Q_4(a_3, a_4)].$$

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

Agent 4 has now been "eliminated," and our problem now reduces to computing

$$\max_{a_1,a_2,a_3} Q_1(a_1, a_2) + Q_3(a_1, a_3) + e_4(a_2, a_3),$$

having one fewer agent involved in the maximization. Next, the choice for agent 3 is made, giving

$$\max_{a_1,a_2} Q_1(a_1, a_2) + e_3(a_1, a_2).$$

where $e_3(a_1, a_2) = \max_{a_3}[Q_3(a_1, a_3) + e_1(a_2, a_3)]$ Next, the choice for agent 2 is made:

$$e_2(a_1) = \max_{a_2}[Q_1(a_1, a_2) + e_3(a_1, a_2)].$$

The remaining decision for agent 1 is now the following maximization:

$$e_1 = \max_{a_1} e_2(a_1).$$

The result e_1 is simply a number, the required maximization over $a_1, \ldots, a_4$. Note that although this expression is short, there is no free lunch; in order to perform this optimization, one needs to iterate not only over all actions a_1 of the first agent, but also over the action of the other agents as needed to unwind the internal maximizations. However, in general the total number of combinations will be smaller than the full exponential combination of agent actions.[2]

We can recover the maximizing set of actions by performing the process in reverse. The maximizing choice for e_1 defines the action a_1^* for agent 1:

$$a_1^* = \arg\max_{a_1} e_2(a_1).$$

To fulfill its commitment to agent 1, agent 2 must choose the value a_2^* that yielded $e_2(a_1^*)$,

$$a_2^* = \arg\max_{a_2}[Q_1(a_1^*, a_2) + e_3(a_1^*, a_2)].$$

This, in turn, forces agent 3 and then agent 4 to select their actions appropriately:

$$a_3^* = \arg\max_{a_3}[Q_3(a_1^*, a_3) + e_4(a_2^*, a_3)];$$

$$a_4^* = \arg\max_{a_4}[Q_2(a_2^*, a_4) + Q_4(a_3^*, a_4)].$$

The actual implementation of this procedure allows several versions. Here are a few of them:

A quick-down, slow-up two-pass sequential implementation: This follows the example in that variables are eliminated symbolically one at a time

2. Full discussion of this point is beyond the scope of this book, but for the record, the complexity of the algorithm is exponential in the tree width of the *coordination graph*; this is the graph whose nodes are the agents and whose edges connect agents whose Q values share one or more arguments. The tree width is also the maximum clique size minus one in the triangulation of the graph; each triangulation essentially corresponds to one of the variable elimination orders. Unfortunately, it is NP-hard to compute the optimal ordering. The notes at the end of the chapter provide additional references on the topic.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

starting with a_n. This is done in $O(n)$ time. When up to a_1 the actual maximization starts; all values of a_1 are tried, alongside all values of the variables appearing in the unwinding of the expression. This phase requires $O(k^n)$ time in the worst case, where k is the bound on domain sizes.

A slow-down, quick-up two-phase sequential implementation: A similar procedure, except here the actual best-response table is built as variables are eliminated. This requires $O(k^n)$ time in the worst case. The payoff is in the second phase, where the optimization requires a simple table-lookup for each value of the variable, resulting in a complexity of $O(kn)$.

Asynchronous versions: The full linear pass in both directions is not necessary, given only partial dependence among variables. Thus in the down phase variables need await a signal from the higher-indexed variables with which they interact (as opposed to all higher-indexed variables) before computing their best-response functions, and similarly in the pass up they need await the signal from only the lower-indexed variables with which they interact.

One final comment. We have discussed variable elimination in the particular context of multiagent MDPs, but it is relevant in any context in which multiple agents wish to perform a distributed optimization of an factorable objective function.

2.3 Negotiation, auctions, and optimization

In this section we consider distributed problem solving that has a certain economic flavor. In the first section below we will informally give the general philosophy and background; in the following two sections we will be more precise.

2.3.1 *From contract nets to auction-like optimization*

contract net

Contract nets were one of the earliest proposals for such an economic approach. Contract nets are not a specific algorithm, but a framework, a protocol for implementing specific algorithms. In a contract net the global problem is broken down into subtasks, and these are distributed among a set of agents. Each agent has different capabilities; for each agent i there is a function c_i such that for any set of tasks T, $c_i(T)$ is the cost for agent i to achieve all the tasks in T. Each agent starts out with some initial set of tasks, but in general this assignment is not optimal, in the sense that the sum of all agents' costs is not minimal. The agents then enter into a negotiation process which improves on the assignment and, hopefully, culminates in an optimal assignment, that is, one with minimal cost. Furthermore, the process can have a so-called *anytime property*; even if it is interrupted prior to achieving optimality, it can achieve significant improvements over the initial allocation.

anytime property

The negotiation consists of agents repeatedly contracting out assignments among themselves, each contract involving the exchange of tasks as well as money. The question is how the bidding process takes place and what contracts hold based on this bidding. The general contract-net protocol is open on these

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

issues. One particular approach has each agent bid for each set of tasks the agent's marginal cost for the task, that is, the agent's additional cost for adding that task to its current set. The tasks are allocated to the lowest bidders, and the process repeats. It can be shown that there always exists a sequence of contracts that result in the optimal allocation. If one is restricted to basic contract types in which one agent contracts a single task to another agent, and receives from him some money in return, then in general achieving optimality requires that agents enter into "money-losing" contracts in the process. However, there exist more complex contracts—which involve contracting for a bundle of tasks ("*cluster contracts*"), or a swap of tasks among two agents ("*swap contracts*"), or simultaneous transfers among many agents ("*multiagent contracts*")—whose combination allows for a sequence of contracts that are not money losing and which culminate in the optimal solution.

cluster contracts

swap contracts

multiagent contracts

At this point several questions may naturally occur to the reader.

- We start with some global problem to be solved, but then speak about minimizing the total cost to the agents. What is the connection between the two?
- When exactly do agents make offers, and what is the precise method by which the contracts are decided on?
- Since we are in a cooperative setting, why does it matter whether agents "lose money" or not on a given contract?

We will provide an answer to the first question in the next section. We will see that, in certain settings (specifically, those of linear programming and integer programming), finding an optimal solution is closely related to the individual utilities of the agents.

Regarding the second question, indeed one can provide several instantiations of even the specific, marginal-cost version of the contract-net protocol. In the next two sections we will be much more specific. We will look at a particular class of negotiation schemes, namely (specific kinds of) auctions. Every negotiation scheme consists of three elements: (1) permissible ways of making offers (*bidding rules*), (2) definition of the outcome based on the offers (*market clearing rules*), and (3) the information made available to the agents throughout the process (*information dissemination rules*). Auctions are a structured way of settling each of these dimensions, and we will look at auctions that do so in specific ways. It should be mentioned, however, that this specificity is not without a price. While convergence to optimality in contract nets depends on particular sequences of contracts taking place, and thus on some coordinating hand, the process is inherently distributed. The auction algorithms we will study include an auctioneer, an explicit centralized component.

bidding rule

market clearing rule

information dissemination rule

The last of our questions deserves particular attention. As we said, we start with some problem to be solved. We then proceed to define an auction-like process for solving it in a distributed fashion. However it is no accident that this section precedes our (rather detailed) discussion of auctions in Chapter 11. As we see there, auctions are a way to allocate scarce resources among *self-interested* agents.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

Auction theory thus belongs to the realm of game theory. In this chapter we also speak about auctions, but the discussion has little to do with game theory. In the spirit of the contract-net paradigm, in our auctions agents will engage in a series of bids for resources, and at the end of the auction the assignment of the resources to the "winners" of the auction will constitute an optimal (or near optimal, in some cases) solution. However, in the standard treatment of auctions (and thus in Chapter 11) the bidders are assumed to bid in a way that maximizes their personal payoff. Here there is no question of the agents deviating from the prescribed bidding protocol for personal gain. For this reason, despite the surface similarity, the discussion of these auction-like methods makes no reference to game theory or mechanism design. In particular, while these methods have some nice properties—for example, they are intuitive, provably correct, naturally parallelizable, appropriate for deployment in distributed systems settings, and tend to be robust to slight perturbations of the problem specification—no claim is made about their usefulness in adversarial situations. For this reason it is indeed something of a red herring, in this cooperative setting, to focus on questions such as whether a given contract is profitable for a given agent. In noncooperative settings, where contract nets are also sometimes pressed into service, the situation is of course different.

In the next two sections we will be looking at two classical optimization problems, one representable as a linear program (LP) and one only as an integer program (IP) (for a brief review of LPs and IPs, see Appendix B). There exists a vast literature on how to solve LPs and IPs, and it is not our aim in this chapter (or in the appendix) to capture this broad literature. Our more limited aim here is to look at the auction-style solutions for them. First we will look at an LP problem—the problem of *weighted matching in a bipartite graph*, also known as the *assignment problem*. We will then look at a more complex, IP problem—that of *scheduling*. As we shall see, since the LP problem is relatively easy (specifically, solvable in polynomial time), it admits an auction-like procedure with tight guarantees. The IP problem is NP-complete, and so it is not surprising that the auction-like procedure does not come with such guarantees.

2.3.2 *The assignment problem and linear programming*

The problem and its LP formulation

weighted matching

assignment problem

The problem of *weighted matching in a bipartite graph*, otherwise known as the *assignment problem*, is defined as follows.

Definition 2.3.1 (Assignment problem)
A (symmetric) assignment problem *consists of:*

- *a set N of n agents;*
- *a set X of n objects;*
- *a set $M \subseteq N \times X$ of possible assignment pairs; and*
- *a function $v : M \mapsto \mathbb{R}$ giving the value of each assignment pair.*

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

feasible assignment

An assignment is a set of pairs $S \subseteq M$ such that each agent $i \in N$ and each object $j \in X$ is in at most one pair in S. A feasible assignment *is one in which all agents are assigned an object. A feasible assignment S is* optimal *if it maximizes* $\sum_{(i,j)\in S} v(i, j)$.

An example of an assignment problem is the following (in this example, $X = \{x_1, x_2, x_3\}$ and $N = \{1, 2, 3\}$).

i	$v(i, x_1)$	$v(i, x_2)$	$v(i, x_3)$
1	2	4	0
2	1	5	0
3	1	3	2

In this small example it is not hard to see that $(1, x_1), (2, x_2), (3, x_3)$ is an optimal assignment. In larger problems, however, the solution is not obvious, and the question is how to compute it algorithmically.

We first note that an assignment problem can be encoded as a linear program. Given a general assignment problem as defined earlier, we introduce the indicator matrix $\mathbf{x}$; $x_{i,j} = 1$ indicates that the pair (i, j) is selected, and $x_{i,j} = 0$ otherwise. Then we express the linear program as follows.

$$\begin{aligned} \text{maximize} \quad & \sum_{(i,j)\in M} v(i, j)x_{i,j} & \\ \text{subject to} \quad & \sum_{j|(i,j)\in M} x_{i,j} \leq 1 & \forall i \in N \\ & \sum_{i|(i,j)\in M} x_{i,j} \leq 1 & \forall j \in X \end{aligned}$$

On the face of it the LP formulation is inappropriate since it allows for fractional matches (i.e., for $0 < x_{i,j} < 1$). But as it turns out this LP has integral solutions.

Lemma 2.3.2 *The LP encoding of the assignment problem has a solution such that for every i, j it is the case that $x_{i,j} = 0$ or $x_{i,j} = 1$. Furthermore, any optimal fractional solution can be converted in polynomial time to an optimal integral solution.*

Since any LP can be solved in polynomial time, we have the following.

Corollary 2.3.3 *The assignment problem can be solved in polynomial time.*

This corollary might suggest that we are done. However, there are a number of reasons to not stop there. First, the polynomial-time solution to the LP problem is of complexity roughly $O(n^3)$, which may be too high in some cases. Furthermore,

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

the solution is not obviously parallelizable, and is not particularly robust to changes in the problem specification (if one of the input parameters changes, the program must essentially be solved from scratch). One solution that suffers less from these shortcomings is based on the economic notion of competitive equilibrium, which we explore next.

The assignment problem and competitive equilibrium

Imagine that each of the objects in X has an associated price; the price vector is $p = (p_1, \ldots, p_n)$, where p_j is the price of object j. Given an assignment $S \subseteq M$ and a price vector p, define the "utility" from an assignment j to agent i as $u(i, j) = v(i, j) - p_j$. An assignment and a set of prices are in *competitive equilibrium* when each agent is assigned the object that maximizes his utility given the current prices. More formally, we have the following.

competitive equilibrium

Definition 2.3.4 (Competitive equilibrium) *A feasible assignment S and a price vector p are in* competitive equilibrium *when for every pairing $(i, j) \in S$ it is the case that $\forall k,\ u(i, j) \geq u(i, k)$.*

It might seem strange to drag an economic notion into a discussion of combinatorial optimization, but as the following theorem shows there are good reasons for doing so.

Theorem 2.3.5 *If a feasible assignment S and a price vector p satisfy the competitive equilibrium condition then S is an optimal assignment. Furthermore, for any optimal solution S, there exists a price vector p such that p and S satisfy the competitive equilibrium condition.*

For example, in the previous example, it is not hard to see that the optimal assignment $(1, x_1), (2, x_2), (3, x_3)$ is a competitive equilibrium given the price vector $(2, 4, 1)$; the "utilities" of the agents are 0, 1, and 1, respectively, and none of them can increase their profit by bidding for one of the other objects at the current prices. We outline the proof of a more general form of the theorem in the next section.

This last theorem means that one way to search for solutions of the LP is to search the space of competitive equilibria. And a natural way to search that space involves auction-like procedures, in which the individual agents "bid" for the different resources in a prespecified way. We will look at open outcry, ascending auction-like procedures, resembling the English auction discussed in Chapter 11. Before that, however, we take a slightly closer look at the connection between optimization problems and competitive equilibrium.

Competitive equilibrium and primal-dual problems

Theorem 2.3.5 may seem at first almost magical; why would an economic notion prove relevant to an optimization problem? However, a slightly closer look

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

removes some of the mystery. Rather than looking at the specific LP corresponding to the assignment problem, consider the general ("primal") form of an LP.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} c_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} a_{ij} x_i \leq b_j && \forall j \in \{1, \ldots, m\} \\ & x_i \geq 0 && \forall i \in \{1, \ldots, n\} \end{aligned}$$

Note that this formulation makes reverse use the $\leq$ and $\geq$ signs as compared to the formulation in Appendix B. As we remark there, this is simply a matter of the signs of the constants used.

production economy

The primal problem has a natural economic interpretation, regardless of its actual origin. Imagine a *production economy*, in which you have a set of resources and a set of products. Each product consumes a certain amount of each resource, and each product is sold at a certain price. Interpret x_i as the amount of product i produced, and c_i as the price of product i. Then the optimization problem can be interpreted as profit maximization. Of course, this must be done within the constraints of available resources. If we interpret b_j as the available amount of resource j and a_{ij} as the amount of resource j needed to produce a unit of product i, then the constraint $\sum_i a_{ij} x_i \leq b_j$ appropriately captures the limitation on resource j.

Now consider the dual problem.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} b_i y_i \\ \text{subject to} \quad & \sum_{i=1}^{m} a_{ij} y_i \geq c_j && \forall j \in \{1, \ldots, n\} \\ & y_i \geq 0 && \forall i \in \{1, \ldots, m\} \end{aligned}$$

shadow price

It turns out that y_i can also be given a meaningful economic interpretation, namely, as the *marginal value* of resource i, also known as its *shadow price*. The shadow price captures the sensitivity of the optimal solution to a small change in the availability of that particular resource, holding everything else constant. A high shadow price means that increasing its availability would have a large impact on the optimal solution, and vice versa.[3]

This helps explain why the economic perspective on optimization, at least in the context of linear programming, is not that odd. Indeed, armed with these intuitions, one can look at traditional algorithms such as the Simplex method and give them an economic interpretation. In the next section we look at a specific auction-like algorithm, which is overtly economic in nature.

3. To be precise, the shadow price is the value of the Lagrange multiplier at the optimal solution.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

A naive auction algorithm

We start with a naive auction-like procedure which is "almost" right; it contains the main ideas, but has a major flaw. In the next section we will fix that flaw. The naive procedure begins with no objects allocated, and terminates once it has found a feasible solution. We define the naive auction algorithm formally as follows.

Naive Auction Algorithm

// Initialization:
$S \leftarrow \emptyset$
forall $j \in X$ **do**
  $p_j \leftarrow 0$
repeat
  // Bidding Step:
  let $i \in N$ be an unassigned agent
  // Find an object $j \in X$ that offers i maximal value at current prices:
  $j \in \arg\max_{k|(i,k)\in M}(v(i,k) - p_k)$
  // Compute i's bid increment for j:
  $b_i \leftarrow (v(i,j) - p_j) - \max_{k|(i,k)\in M; k\neq j}(v(i,k) - p_k)$
  // which is the difference between the value to i of the best and second-best objects at current prices (note that i's bid will be the current price plus this bid increment).
  // Assignment Step:
  add the pair (i, j) to the assignment S
  if *there is another pair* (i', j) **then**
    remove it from the assignment S
  increase the price p_j by the increment b_i
until S *is feasible* // that is, it contains an assignment for all $i \in N$

It is not hard to verify that the following is true of the algorithm.

Theorem 2.3.6 *The naive algorithm terminates only at a competitive equilibrium.*

Here, for example, is a possible execution of the algorithm on our current example. The following table shows each round of bidding. In this execution we pick the unassigned agents in order, round-robin style.

round	p_1	p_2	p_3	bidder	preferred object	bid incr.	current assignment
0	0	0	0	1	x_2	2	$(1, x_2)$
1	0	2	0	2	x_2	2	$(2, x_2)$
2	0	4	0	3	x_3	1	$(2, x_2), (3, x_3)$
3	0	4	1	1	x_1	2	$(2, x_2), (3, x_3), (1, x_1)$

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

At first agents 1 and 2 compete for x_2, but quickly x_2 becomes too expensive for agent 1, who opts for x_1. By the time agent 3 gets to bid he is priced out of his preferred item, x_2, and settles for x_3.

Thus when the procedure terminates we have our solution. The problem, though, is that it may not terminate. This can occur when more than one object offers maximal value for a given agent; in this case the agent's bid increment will be zero. If these two items also happen to be the best items for another agent, they will enter into an infinite bidding war in which the price never rises. Consider a modification of our previous example, in which the value function is given by the following table.

i	$\mathbf{v(i, x_1)}$	$\mathbf{v(i, x_2)}$	$\mathbf{v(i, x_3)}$
1	1	1	0
2	1	1	0
3	1	1	0

The naive auction protocol would proceed as follows.

round	$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$	**bidder**	**preferred object**	**bid incr.**	**current assignment**
0	0	0	0	1	x_1	0	$(1, x_1)$
1	0	0	0	2	x_2	0	$(1, x_1), (2, x_2)$
2	0	0	0	3	x_1	0	$(3, x_1), (2, x_2)$
3	0	0	0	1	x_2	0	$(3, x_1), (1, x_2)$
4	0	0	0	2	x_1	0	$(2, x_1), (1, x_2)$
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Clearly, in this example the naive algorithm will have the three agents forever fight over the two desired objects.

A terminating auction algorithm

To remedy the flaw exposed previously, we must ensure that prices continue to increase when objects are contested by a group of agents. The extension is quite straightforward: we add a small amount to the bidding increment. Thus we calculate the bid increment of agent $i \in N$ as follows.

$$b_i = u(i, j) - \max_{k|(i,k)\in M; k\neq j} u(i, k) + \epsilon$$

Otherwise, the algorithm is as stated earlier.

Consider again the problematic assignment problem on which the naive algorithm did not terminate. The terminating auction protocol would proceed as follows.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

round	p_1	p_2	p_3	bidder	preferred object	bid incr.	current assignment
0	ϵ	0	0	1	x_1	ϵ	$(1, x_1)$
1	ϵ	2ϵ	0	2	x_2	2ϵ	$(1, x_1), (2, x_2)$
2	3ϵ	2ϵ	0	3	x_1	2ϵ	$(3, x_1), (2, x_2)$
3	3ϵ	4ϵ	0	1	x_2	2ϵ	$(3, x_1), (1, x_2)$
4	5ϵ	4ϵ	0	2	x_1	2ϵ	$(2, x_1), (1, x_2)$

Note that at each iteration, the price for the preferred item increases by at least ϵ. This gives us some hope that we will avoid nontermination. We must first though make sure that, if we terminate, we terminate with the "right" results.

First, because the prices must increase by at least ϵ at every round, the competitive equilibrium property is no longer preserved over the iteration. Agents may "overbid" on some objects. For this reason we will need to define a notion of *ϵ-competitive equilibrium*.

ϵ-competitive equilibrium

Definition 2.3.7 (ϵ-competitive equilibrium) *S and p satisfy ϵ-competitive equilibrium when for each $i \in N$, if there exists a pair $(i, j) \in S$ then $\forall k,\ u(i, j) + \epsilon \geq u(i, k)$.*

In other words, in an ϵ-equilibrium no agent can profit more than ϵ by bidding for an object other than his assigned one, given current prices.

Theorem 2.3.8 *A feasible assignment S with n goods that forms an ϵ-competitive equilibrium with some price vector is within $n\epsilon$ of optimal.*

Corollary 2.3.9 *Consider a feasible assignment problem with an integer valuation function $v : M \mapsto \mathbb{Z}$. If $\epsilon < \frac{1}{n}$ then any feasible assignment found by the terminating auction algorithm will be optimal.*

This leaves the question of whether the algorithm indeed terminates, and if so, how quickly. To see why the algorithm must terminate, note that if an object receives a bid in k iterations, its price must exceed its initial price by at least $k\epsilon$. Thus, for sufficiently large k, the object will become expensive enough to be judged inferior to some object that has not received a bid so far. The total number of iterations in which an object receives a bid must be no more than

$$\frac{\max_{(i,j)} v(i, j) - \min_{(i,j)} v(i, j)}{\epsilon}.$$

Once all objects receive at least one bid, the auction terminates (do you see why?). If each iteration involves a bid by a single agent, the total number of iterations is no more than n times the preceding quantity. Thus, since each bid requires $O(n)$ operations, the running time of the algorithm is $O(n^2 \max_{(i,j)} \frac{|v(i,j)|}{\epsilon})$. Observe that if $\epsilon = O(1/n)$ (as discussed in Corollary 2.3.9), the algorithm's running time is $O(n^3 k)$, where k is a constant that does not depend on n, yielding worst-case performance similar to linear programming.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

2.3.3 *The scheduling problem and integer programming*

The problem and its integer program

scheduling problem

The *scheduling problem* involves a set of time slots and a set of agents. Each agent requires some number of time slots and has a deadline. Intuitively, the agents each have a task that requires the use of a shared resource, and that task lasts a certain number of hours and has a certain deadline. Each agent also has some value for completing the task by the deadline. Formally, we have the following definition.

Definition 2.3.10 (Scheduling problem) *A scheduling problem consists of a tuple $C = (N, X, q, v)$, where:*

- *N is a set of n agents;*
- *X is a set of m discrete and consecutive time slots;*
- *$q = (q_1, \ldots, q_m)$ is a reserve price vector, where q_j is a reserve value for time slot x_j; q can be thought of as the value for the slot of the owner of the resource, the value he could get for it by allocating it other than to one of the n agents; and*
- *$v = (v_1, \ldots, v_n)$, where v_i, the valuation function of agent i, is a function over possible allocations of time slots that is parameterized by two arguments: d_i, the deadlines of agent i, and λ_i, the required number of time slots required by agent i. Thus for an allocation $F_i \subseteq 2^X$, we have that:*

$$v_i(F_i) = \begin{cases} w_i & \text{if } F_i \text{ includes } \lambda_i \text{ hours before } d_i; \\ 0 & \text{otherwise.} \end{cases}$$

A solution to a scheduling problem is a vector $F = (F_\emptyset, F_1, \ldots, F_n)$, where F_i is the set of time slots assigned to agent i, and $F_\emptyset$ is the time slots that are not assigned. The value of a solution is defined as

$$V(F) = \sum_{j | x_j \in F_\emptyset} q_j + \sum_{i \in N} v_i(F_i).$$

A solution is optimal if no other solution has a higher value.

Here is an example, involving scheduling jobs on a busy processor. The processor has several discrete time slots for the day—specifically, eight one-hour time slots from 9:00 A.M. to 5:00 P.M. Its operating costs force it to have a reserve price of \$3 per hour. There are four jobs, each with its own length, deadline, and worth. They are shown in the following table.

job	length (λ)	deadline (d)	worth (w)
1	2 hours	1:00 P.M.	\$10.00
2	2 hours	12:00 P.M.	\$16.00
3	1 hours	12:00 P.M.	\$6.00
4	4 hours	5:00 P.M.	\$14.50

Even in this small example it takes a moment to see that an optimal solution is to allocate the machines as follows.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

time slot	agent
9:00 A.M.	2
10:00 A.M.	2
11:00 A.M.	1
12:00 P.M.	1
13:00 P.M.	4
14:00 P.M.	4
15:00 P.M.	4
16:00 P.M.	4

complementarity

substitutes

set packing problem

The question is again how to find the optimal schedule algorithmically. The scheduling problem is inherently more complex than the assignment problem. The reason is that the dependence of agents' valuation functions on the job length and deadline exhibits both *complementarity* and *substitutability*. For example, for agent 1 any two blocks of two hours prior to 1:00 are perfect substitutes. On the other hand, any two single time slots before the deadline are strongly complementary; alone they are worth nothing, but together they are worth the full $10. This makes for a more complex search space than in the case of the assignment problem, and whereas the assignment problem is polynomial, the scheduling problem is NP-complete. Indeed, the scheduling application is merely an instance of the general *set packing problem*.[4]

The complex nature of the scheduling problem has many ramifications. Among other things, this means that we cannot hope to find a polynomial LP encoding of the problem (since linear programming has a polynomial-time solution). We can, however, encode it as an integer program. In the following, for every subset $S \subseteq X$, the boolean variable $x_{i,S}$ will represent the fact that agent i was allocated the bundle S, and $v_i(S)$ his valuation for that bundle.

$$\begin{aligned}
\text{maximize} \quad & \sum_{S \subseteq X, i \in N} v_i(S) x_{i,S} \\
\text{subject to} \quad & \sum_{S \subseteq X} x_{i,S} \leq 1 && \forall i \in N \\
& \sum_{S \subseteq X : j \in S, i \in N} x_{i,S} \leq 1 && \forall j \in X \\
& x_{i,S} \in \{0, 1\} && \forall S \subseteq X, i \in N
\end{aligned}$$

In general, the length of the optimized quantity is exponential in the size of X. In practice, many of the terms can be assumed to be zero, and thus dropped. However, even when the IP is small, our problems are not over. IPs are not in general solvable in polynomial time, so we cannot hope for easy answers. However, it turns out that a generalization of the auction-like procedure can be

4. Even the scheduling problem can be defined much more broadly. It could involve earliest start times as well as deadlines, could require contiguous blocks of time for a given agent (this turns out that this requirement does not matter in our current formulation), could involve more than one resource, and so on. But the current problem formulation is rich enough for our purposes.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

applied in this case too. The price we will pay for the higher complexity of the problem is that the generalized algorithm will not come with the same guarantees that we had in the case of the assignment problem.

A more general form of competitive equilibrium

competitive equilibrium

We start by revisiting the notion of *competitive equilibrium*. The definition really does not change, but rather is generalized to apply to assignments of bundles of time slots rather than single objects.

Definition 2.3.11 (Competitive equilibrium, generalized form) *Given a scheduling problem, a solution F is in* competitive equilibrium *at prices p if and only if:*

- *For all $i \in N$, it is the case that $F_i = \arg\max_{T \subseteq X}(v_i(T) - \sum_{j|x_j \in T} p_j)$ (the set of time slots allocated to agent i maximizes his surplus at prices p);*
- *For all j such that $x_j \in F_\emptyset$, it is the case that $p_j = q_j$ (the price of all unallocated time slots is the reserve price); and*
- *For all j such that $x_j \notin F_\emptyset$ it is the case that $p_j \geq q_j$ (the price of all allocated time slots is greater than the reserve price).*

As was the case in the assignment problem, a solution that is in competitive equilibrium is guaranteed to be optimal.

Theorem 2.3.12 *If a solution F to a scheduling problem C is in equilibrium at prices p, then F is also optimal for C.*

We give an informal proof to facilitate understanding of the theorem. Assume that F is in equilibrium at prices p; we would like to show that the total value of F is higher than the total value of any other solution F'. Starting with the definition of the total value of the solution F, the following equations show this inequality for an arbitrary F'.

$$\begin{aligned} V(F) &= \sum_{j|x_j \in F_\emptyset} q_j + \sum_{i \in N} v_i(F_i) \\ &= \sum_{j|x_j \in F_\emptyset} p_j + \sum_{i \in N} v_i(F_i) \\ &= \sum_{j|x_j \in X} p_j + \sum_{i \in N} \left[v_i(F_i) - \sum_{j|x_j \in F_i} p_j \right] \\ &\geq \sum_{j|x_j \in X} p_j + \sum_{i \in N} \left[v_i(F_i') - \sum_{j|x_j \in F_i'} p_j \right] = V(F') \end{aligned}$$

The last line comes from the definition of a competitive equilibrium, for each agent i, there does not exist another allocation F_i' that would yield a larger profit at the current prices (formally, $\forall i, F_i'\ v_i(F_i) - \sum_{j|x_j \in F_i} p_j \geq v_i(F_i') - \sum_{j|x_j \in F_i'} p_j$). Applying this condition to all agents, it follows that there exists no alternative allocation F' with a higher total value.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

Consider our sample scheduling problem. A competitive equilibrium for that problem is shown in the following table.

time slot	agent	price
9:00 A.M.	2	$6.25
10:00 A.M.	2	$6.25
11:00 A.M.	1	$6.25
12:00 P.M.	1	$3.25
13:00 P.M.	4	$3.25
14:00 P.M.	4	$3.25
15:00 P.M.	4	$3.25
16:00 P.M.	4	$3.25

Note that the price of all allocated time slots is higher than the reserve prices of $3.00. Also note that the allocation of time slots to each agent maximizes his surplus at the prices p. Finally, also notice that the solution is stable, in that no agent can profit by making an offer for an alternative bundle at the current prices.

Even before we ask how we might find such a competitive equilibrium, we should note that one does not always exist. Consider a modified version of our scheduling example, in which the processor has two one-hour time slots, at 9:00 A.M. and at 10:00 A.M., and there are two jobs as in Table 2.1. The reserve price is $3 per hour. We show that no competitive equilibrium exists by case analysis. Clearly, if agent 1 is allocated a slot he must be allocated both slots. But then their combined price cannot exceed $10, and thus for at least one of those hours the price must not exceed $5. However, agent 2 is willing to pay as much as $6 for that hour, and thus we are out of equilibrium. Similarly, if agent 2 is allocated at least one of the two slots, their combined price cannot exceed $6, his value. But then agent 1 would happily pay more and get both slots. Finally, we cannot have both slots unallocated, since in this case their combined price would be $6, the sum of the reserve prices, in which case both agents would have the incentive to buy.

This instability arises from the fact that the agents' utility functions are superadditive (or, equivalently, that there are complementary goods). This suggest some restrictive conditions under which we are guaranteed the existence of a competitive equilibrium solution. The first theorem captures the essential connection to linear programming.

Theorem 2.3.13 *A scheduling problem has a competitive equilibrium solution if and only if the LP relaxation of the associated integer program has a integer solution.*

job	length (λ)	deadline (d)	worth (w)
1	2 hours	11:00 A.M.	$10.00
2	1 hour	11:00 A.M.	$6.00

Table 2.1 A problematic scheduling example.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

The following theorem captures weaker sufficient conditions for the existence of a competitive equilibrium solution.

Theorem 2.3.14 *A scheduling problem has a competitive equilibrium solution if any one of the following conditions hold:*

- *For all agents $i \in N$, there exists a time slot $x \in X$ such that for all $T \subseteq X$, $v_i(T) = v_i(\{x\})$. (Each agent desires only a single time slot, which must be the first one in the current formulation.)*
- *For all agents $i \in N$, and for all $R, T \subseteq X$, such that $R \cap T = \emptyset$, $v_i(R \cup T) = v_i(R) + v_i(T)$. (The utility functions are additive.)*
- *Time slots are gross substitutes; demand for one time slot does not decrease if the price of another time slot increases.*

An auction algorithm

ascending-auction algorithm

Perhaps the best-known distributed protocol for finding a competitive equilibrium is the so-called *ascending-auction algorithm*. In this protocol, the center advertises an *ask price*, and the agents bid the ask price for bundles of time slots that maximize their surplus at the given ask prices. This process repeats until there is no change.

Let $b = (b_1, \ldots, b_m)$ be the bid price vector, where b_j is the highest bid so far for time slot $x_j \in X$. Let $F = (F_1, \ldots, F_n)$ be the set of allocated slots for each agent. Finally, let ϵ be the price increment. The ascending-auction algorithm is given in Figure 2.7.

The ascending-auction algorithm is very similar to the assignment problem auction presented in the previous section, with one notable difference. Instead of calculating a bid increment from the difference between the surplus gained from the best and second-best objects, the bid increment here is always constant.

Let us consider a possible execution of the algorithm to the sample scheduling problem discussed earlier. We use an increment of \$0.25 for this execution of the algorithm.

round	bidder	slots bid on	$\mathbf{F} = (\mathbf{F_1}, \mathbf{F_2}, \mathbf{F_3}, \mathbf{F_4})$	b
0	1	(9,10)	({9, 10}, {∅}, {∅}, {∅})	(3.25,3.25,3,3,3,3,3,3)
1	2	(10,11)	({9}, {10, 11}, {∅}, {∅})	(3.25,3.5,3.25,3,3,3,3,3)
2	3	(9)	({∅}, {10, 11}, {9}, {∅})	(3.5,3.5,3.25,3,3,3,3,3)
⋮	⋮	⋮	⋮	⋮
24	1	∅	({11, 12}, {9, 10}, {∅}, {12, 13, 14, 15})	(6.25,6.25,6.25,3.25, 3.25,3.25,3.25,3.25)

At this point, no agent has a profitable bid, and the algorithm terminates. However, this convergence depended on our choice of the increment. Let us consider what happens if we select an increment of \$1.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

foreach *slot* x_j **do**
  $b_j \leftarrow q_j$
  // Set the initial bids to be the reserve price
foreach *agent* i **do**
  $F_i \leftarrow \emptyset$
repeat
  foreach *agent* $i = 1$ *to* N **do**
    foreach *slot* x_j **do**
      if $x_j \in F_j$ **then**
        $p_j \leftarrow b_j$
      else
        $p_j \leftarrow b_j + \epsilon$
      // Agents assume that they will get slots they are currently the high bidder on at that price, while they must increment the bid by ϵ to get any other slot.
    $S^* \leftarrow \arg\max_{S \subseteq X | S \supseteq F_i} (v_i(S) - \sum_{j \in S} p_j)$
    // Find the best subset of slots, given your current outstanding bids
    // Agent i becomes the high bidder for all slots in $S^* \setminus F_i$.
    foreach *slot* $x_j \in S^* \setminus F_i$ **do**
      $b_j \leftarrow b_j + \epsilon$
      if *there exists an agent* $k \neq i$ *such that* $x_j \in F_k$ **then**
        set $F_k \leftarrow F_k \setminus \{x_j\}$
      // Update the bidding price and current allocations of the other bidders.
    $F_i \leftarrow S^*$
until F *does not change*

Figure 2.7 The ascending-auction algorithm.

round	bidder	slots bid on	$\mathbf{F} = (\mathbf{F_1}, \mathbf{F_2}, \mathbf{F_3}, \mathbf{F_4})$	b
0	1	(9,10)	({9, 10}, {∅}, {∅}, {∅})	(4,4,3,3,3,3,3,3)
1	2	(10,11)	({9}, {10, 11}, {∅}, {∅})	(4,5,4,3,3,3,3,3)
2	3	(9)	({∅}, {10, 11}, {9}, {∅})	(5,5,4,3,3,3,3,3)
3	4	(12,13,14,15)	({∅}, {10, 11}, {9}, {12, 13, 14, 15})	(5,5,4,4,4,4,4,3)
4	1	(11,12)	({11, 12}, {10}, {9}, {13, 14, 15})	(5,5,5,5,4,4,4,3)
5	2	(9,10)	({11, 12}, {9, 10}, {∅}, {13, 14, 15})	(6,6,5,5,4,4,4,3)
6	3	(11)	({12}, {9, 10}, {11}, {13, 14, 15})	(6,6,6,5,4,4,4,3)
7	4	∅	({12}, {9, 10}, {11}, {13, 14, 15})	(6,6,6,5,4,4,4,3)
8	1	∅	({12}, {9, 10}, {11}, {13, 14, 15})	(6,6,6,5,4,4,4,3)

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

Unfortunately, this bidding process does not reach the competitive equilibrium because the bidding increment is not small enough.

It is also possible for the ascending-auction algorithm algorithm to not converge to an equilibrium independently of how small the increment is. Consider another problem of scheduling jobs on a busy processor. The processor has three one-hour time slots, at 9:00 A.M., 10:00 A.M., and 11:00 A.M., and there are three jobs as shown in the following table. The reserve price is $0 per hour.

job	length (λ)	deadline (d)	worth (w)
1	1 hour	11:00 A.M.	$2.00
2	2 hours	12:00 P.M.	$20.00
3	2 hours	12:00 P.M.	$8.00

Here an equilibrium exists, but the ascending auction can miss it, if agent 2 bids up the 11:00 A.M. slot.

Despite a lack of a guarantee of convergence, we might still like to be able to claim that if we do converge then we converge to an optimal solution. Unfortunately, not only can we not do that, we cannot even bound how far the solution is from optimal. Consider the following problem. The processor has two one-hour time slots, at 9:00 A.M. and 10:00 A.M. (with reserve prices of $1 and $9, respectively), and there are two jobs as shown in the following table.

job	length (λ)	deadline (d)	worth (w)
1	1 hour	10:00 A.M.	$3.00
2	2 hours	11:00 A.M.	$11.00

The ascending-auction algorithm will stop with the first slot allocated to agent 1 and the second to agent 2. By adjusting the value to agent 2 and the reserve price of the 11:00 A.M. time slot, we can create examples in which the allocation is arbitrarily far from optimal.

One property we can guarantee, however, is termination. We show this by contradiction. Assume that the algorithm does not converge. It must be the case that at each round at least one agent bids on at least one time slot, causing the price of that slot to increase. After some finite number of bids on bundles that include a particular time slot, it must be the case that the price on this slot is so high that every agent prefers the empty bundle to all bundles that include this slot. Eventually, this condition will hold for all time slots, and thus no agent will bid on a nonempty bundle, contradicting the assumption that the algorithm does not converge. In the worst case, in each iteration only one of the n agents bids, and this bid is on a single slot. Once the sum of the prices exceeds the maximum total value for the agents, the algorithm must terminate, giving us the worst-case running time $O(n \max_{F_i} \frac{\sum_{i \in N} v_i(F_i)}{\epsilon})$.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

2.4 Social laws and conventions

social law

Consider the task of a city transportation official who wishes to optimize traffic flow in the city. While he cannot redesign cars or create new roads, he can impose *traffic rules*. A traffic rule is a form of a *social law*: a restriction on the given strategies of the agents. A typical traffic rule prohibits people from driving on the left side of the road or through red lights. For a given agent, a social law presents a tradeoff; he suffers from loss of freedom, but can benefit from the fact that others lose some freedom. A good social law is designed to benefit all agents.

One natural formal view of social laws is from the perspective of game theory. We discuss game theory in detail starting in Chapter 3, but here we need very little of that material. For our purposes here, suffice it to say that in a game each agent has a number of possible strategies (in our traffic example, driving plans), and depending on the strategies selected by each agent, each agent receives a certain payoff. In general, agents are free to choose their own strategies, which they will do based on their guesses about the strategies of other agents. Sometimes the interests of the agents are at odds with each other, but sometimes they are not. In the extreme case the interests are perfectly aligned, and the only problem is that of coordination among the agents. Again, traffic presents the perfect example; agents are equally happy driving on the left or on the right, provided everyone does the same.

social convention

A social law simply eliminates from a given game certain strategies for each of the agents, and thus induces a subgame. When the subgame consists of a single strategy for each agent, we call it a *social convention*. In many cases the setting is naturally symmetric (the game is symmetric, as are the restrictions), but it need not be that way. A social law is good if the induced subgame is "preferred" to the original one. There can be different notions of preference here; we will discuss this further after we discuss the notion of *solution concepts* in Chapter 3. For now we leave the notion of preference at the intuitive level; intuitively, a world where everyone (say) drives on the right and stops at red lights is preferable to one in which drivers cannot rely on such laws and must constantly coordinate with each other.

This leaves the question of how one might find such a good social law or social convention. In Chapter 7 we adopt a democratic perspective; we look at how conventions can emerge dynamically as a result of a learning process within the population. Here we adopt a more autocratic perspective, and imagine a social planner imposing a good social law (or even a single convention). The question is how such a benign dictator arrives at such a good social law. In general the problem is hard; specifically, when formally defined, the general problem of finding a good social law (under an appropriate notion of "good") can be shown to be NP-hard. However, the news is not all bad. First, there exist restrictions that render the problem polynomial. Furthermore, in specific situations, one can simply hand craft good social laws.

Indeed, traffic rules provide an excellent example. Consider a set of k robots $\{0, 1, \ldots, k-1\}$ belonging to Deliverobot, Inc., who must navigate a road system connecting seven locations as depicted in Figure 2.8.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

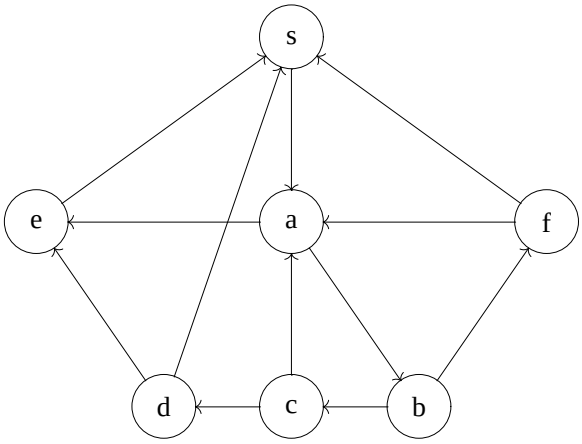

Figure 2.8 Seven locations in a transportation domain.

Assume these k robots are the only vehicles using the road, and their main challenge is to avoid collisions among themselves. Assume further that they all start at point s, the company's depot, at the start of the day. We assume a discrete model of time, and that each robot requires one unit of time to traverse any given edge, though the robots can also travel more slowly if they wish. At each of the first k time steps one robot is assigned initial tasks and sent on its way, with robot i sent at time i ($i = 0, 1, \ldots, k-1$). Thereafter they are in continuous motion; as soon as they arrive at their current destination they are assigned a new task, and off they go. A collision is defined as two robots occupying the same location at the same time. How can collisions be avoided without the company constantly planning routes for the robots, and without the robots constantly having to negotiate with each other? The tools they have at their disposal are the speed with which they traverse each edge and the common clock they implicitly share with the other robots.

Here is one simple solution: Each robot drives so that traversing each link takes exactly k time units. In this case, at any time t the only robot who will arrive at a node—any node—is $i \equiv t \mod k$. This is an example of a simple social convention that is useful, but that comes at a price. Each robot is free to travel along the shortest path, but will traverse this path k times more slowly than he would without this particular social law.

Here is a more efficient convention. Assign each vertex an arbitrary label between 0 and $k-1$, and define the time to traverse an edge between vertices labeled x and y to be $(y-x) \mod k$ if $(y-x) \mod k > 0$, and k otherwise. Observe that the difference in this expression will sometimes be negative; this is not a problem because the modulo nevertheless returns a nonnegative value. To consider an example, if agent i follows the sequence of nodes labeled s, x_1, x_2, x_3 then its travel times are $(x_1 - s) \mod k$, $(x_2 - x_1) \mod k$, $(x_3 - x_2) \mod k$, presuming that none of these expressions evaluated to zero. Adding these travel times to the start time we see that i reaches node x_3 at time $t \equiv i + x_1 + (x_2 - x_1) + (x_3 - x_2) \equiv x_3 + i \mod k$. In general, we have that at time t agent i will

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

always either be on an edge or waiting at a node labeled $(t - i) \mod k$, and thus there will be no collisions.

A final comment is in order. In the discussion so far we have assumed that once a social law is imposed (or agreed upon) it is adhered to. This is of course a tenuous assumption when applied to fallible and self-interested agents. In Chapter 10 (and specifically in Section 10.7) we return to this topic.

2.5 History and references

Distributed dynamic programming is discussed in detail in Bertsekas [1982]. LRTA* is introduced in Korf [1990], and our section follows that material, as well as Yokoo and Ishida [1999].

Distributed solutions to Markov Decision Problems are discussed in detail in Guestrin [2003]; the discussion there goes far beyond the specific problem of joint action selection covered here. Additional discussion specifically on the issue of problem selection in distributed MDPs can be found in Vlassis et al. [2004].

Contract nets were introduced in Smith [1980], and Davis and Smith [1983] is perhaps the most influential publication on the topic. The marginal-cost interpretation of contract nets was introduced in Sandholm [1993], and the discussion of the capabilities and limitations of the various contract types (O, C, S, and M) followed in Sandholm [1998]. Auction algorithms for linear programming are discussed broadly in Bertsekas [1991]. The specific algorithm for the matching problem is taken from Bertsekas [1992]. Its extension to the combinatorial setting is discussed in Parkes and Ungar [2000]. Auction algorithms for combinatorial problems in general are introduced in Wellman [1993], and the specific auction algorithms for the scheduling problem appear in Wellman et al. [2001].

Social laws and conventions, and the example of traffic laws, were introduced in Shoham and Tennenholtz [1995]. The treatment there includes many additional tweaks on the basic traffic grid discussed here, as well as an algorithmic analysis of the problem in general.

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003

Downloaded from https://www.cambridge.org/core. University of Warwick, on 10 Jan 2019 at 08:28:45, subject to the Cambridge Core terms of use, available at https://www.cambridge.org/core/terms. https://doi.org/10.1017/CBO9780511811654.003