Lecture notes 2: Applications

Vincent Conitzer

In this set of notes, we will discuss a number of problems of interest to computer scientists where linear/integer programming can be fruitfully applied.

1 Maximum flow

In the maximum flow problem, we are given a directed graph with a distinguished *source node* and a distinguished *sink node*. In addition, each edge has a *capacity*, which is the largest amount of flow that can go over this edge. Our goal is to route as much flow from the source to the sink as we can, along the edges of the graph. For example, consider the directed graph with capacities in Figure 1, where S is the source and T is the sink.

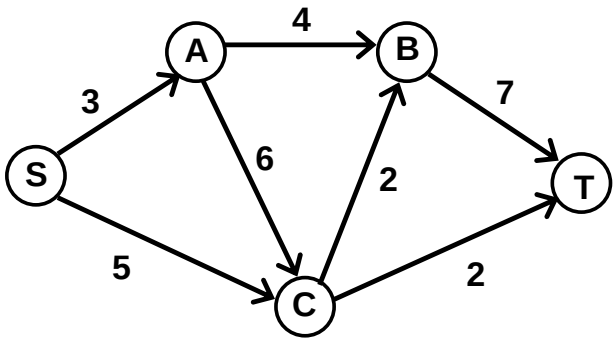

Figure 1: An instance of the maximum flow problem.

The maximum flow that can be sent from S to T is 7, by sending 3 units from S to A; 4 from S to C; 3 from A to B; 0 from A to C; 2 from C to B; 5 from B to T; and 2 from C to T. Note that every node other than S and T receives exactly as much flow as it sends (since these nodes cannot "produce" flow).

We can model the maximum flow problem as a linear program as follows. For any nodes v and w, let f_{vw} (a variable) be the flow from v to w. We must have $f_{vw} \leq c_{vw}$, where c_{vw} (a parameter) is the capacity of edge (v, w). (For pairs v, w without an edge from v to w, we can either not have a flow variable f_{vw} at all, or we can say $c_{vw} = 0$.) In addition, we have a constraint that for any node $v \notin \{s, t\}$ (where s and t are the source and sink), $\sum_u f_{uv} = \sum_w f_{vw}$—that is, the flow into v equals the flow out of v. Putting it all together, letting V be the set of vertices (nodes) and E the set of (directed) edges, we get:

maximize $\sum_{v \in V:(v,t) \in E} f_{vt}$
subject to
$(\forall (v, w) \in E)\ f_{vw} \leq c_{vw}$
$(\forall v \in V - \{s, t\})\ \sum_{u \in V:(u,v) \in E} f_{uv} - \sum_{w \in V:(v,w) \in E} f_{vw} = 0$
$(\forall (v, w) \in E)\ f_{vw} \geq 0$

As you can see, the way this linear program is written looks somewhat different from before: we are using $\forall$s and $\sum$s, and we are not using x_j for the variables. Still, it is a linear program, and for any instance it can be converted to the standard way of writing linear programs that we saw before. For example, for the instance in Figure 1, the program becomes:

maximize $f_{BT} + f_{CT}$
subject to
$f_{SA} \leq 3$
$f_{SC} \leq 5$
$f_{AC} \leq 6$
$f_{AB} \leq 4$
$f_{CB} \leq 2$
$f_{BT} \leq 7$
$f_{CT} \leq 2$
$f_{SA} - f_{AC} - f_{AB} = 0$
$f_{SC} + f_{AC} - f_{CB} - f_{CT} = 0$
$f_{AB} + f_{CB} - f_{BT} = 0$
$f_{SA} \geq 0$, $f_{SC} \geq 0$, $f_{AC} \geq 0$, $f_{AB} \geq 0$, $f_{CB} \geq 0$, $f_{BT} \geq 0$, $f_{CT} \geq 0$

It is generally helpful to allow some flexibility in how we write linear programs, especially when we write them in an abstract form; but it is important to stay aware of the distinction between variables and parameters.

The fact that we can model the maximum flow problem as a (polynomial-size) linear program immediately implies that the maximum flow problem can be solved in polynomial time, because linear programs can be solved in polynomial time. There are specialized algorithms for the maximum flow problem that are more efficient than solving it as a linear program. Nevertheless, modeling it as a linear program is quick and simple. It is also a very flexible approach that is easily modified to variants of the problem. Finally, we will see later that we can prove some basic properties of the maximum flow problem using this linear program formulation.

2 Combinatorial auctions

Imagine that you are in the following situation. On an online auction site, you have found someone selling a desktop computer that you like. Unfortunately, it does not come with a monitor. However, you have found another seller, on the same website, who is selling a monitor that you like. If you were to win both items, this would be worth $500 to you. However, if you win only the desktop computer, or only the monitor, this would be worth nothing to you at all. (For simplicity, let us assume that you cannot buy a desktop or monitor anywhere else.) Now, you are about to go on vacation, so for each auction, you must specify how high you are willing to bid. (Auction websites often ask you to specify this, and will then automatically bid up to the price that you stated.) How high should you be willing to bid in each auction? If you say you are willing to go up to, say, $250 in each auction, you run the risk of winning one of them but not the other, resulting in a loss of up to $250 in total for you. In fact, regardless of what you say, you may end up with a loss—unless you say $0 on both items. What you would like to do is to specify that you are willing to pay up to $500 on the *bundle* of both the desktop and the monitor. This is what a *combinatorial auction* would allow you to do. Such auctions are becoming more common.

In a combinatorial auction, there is a set of items I which are simultaneously for sale, and bidders place bids on bundles. For example, if $I = \{A, B, C, D\}$, a bid might be $(\{A, C, D\}, 5)$, indicating that this bidder is willing to pay 5 for receiving *all* of the items A, C, and D. In a *sealed-bid* combinatorial auction, all of these bids are submitted simultaneously. Once we have received all the bids, we are faced with the *winner determination problem* of deciding which of these bids win. We can award each item at most once, so we cannot accept two bids if they have one or more items in common. Under this constraint, generally, the goal is to accept bids in a way that maximizes the sum of the values of the accepted bids.

For example, suppose that there are 4 items, A, B, C, D, and that we receive the following bids: $(\{A, B\}, 4), (\{B, C\}, 5), (\{A, C\}, 4), (\{A, B, D\}, 7), (\{D\}, 1)$. It is easy to see that of the first 4 bids, we can accept at most 1, because every pair of the first 4 bids overlaps. Hence, the optimal solution is to just accept $(\{A, B, D\}, 7)$, and let C go unallocated, for a total value of 7. The second-best feasible solution is to accept $(\{B, C\}, 5)$ and $(\{D\}, 1)$, for a total value of 6.

We can model the winner determination problem as an integer program as follows. For every bid b, let there be a variable x_b that takes values in $\{0, 1\}$ (a *binary variable*); setting this variable to 1 means that the bid is accepted, and setting it to 0 means that it is rejected. For each bid b and item i, let there be a parameter a_{ib} which is 1 if item i is included in bid b, and 0 if it is not. Finally, let v_b be the value of bid b. Then we can write the winner determination problem as follows:

maximize $\sum_b v_b x_b$
subject to
$(\forall i \in I)\ \sum_b a_{ib} x_b \leq 1$
$(\forall b)\ x_b \in \{0, 1\}$

(Effectively, $x_b \in \{0, 1\}$ is shorthand for $x_b \geq 0$, $x_b \leq 1$, and x_b is an integer.)

The winner determination problem is NP-hard, so we should not expect to find a linear program formulation (without integrality constraints) for it. However, there is a variant of the winner determination problem that corresponds to a linear program. Namely, suppose that bids are *partially acceptable*. For example, given the bid $(\{A, B\}, 6)$, we can accept 1/3 of this bid, meaning that we still have 2/3 of item A and 2/3 of item B left, and we have obtained a value of $6/3 = 2$ by doing this. In the above example, we can obtain a better solution if bids are partially acceptable: accept 1/2 of each of the bids $(\{B, C\}, 5), (\{A, C\}, 4), (\{A, B, D\}, 7), (\{D\}, 1)$. Each item occurs twice in these bids, so each item is allocated completely. The total value generated is 8.5, more than the 7 that we could obtain without partial acceptability. In fact, this is the optimal solution.

We can model the winner determination problem with partially acceptable bids as a linear program, simply by replacing the requirement $x_b \in \{0,1\}$ by $x_b \in [0,1]$, that is,

maximize $\sum_b v_b x_b$
subject to
$(\forall i \in I)\ \sum_b a_{ib} x_b \leq 1$
$(\forall b)\ x_b \in [0,1]$

(Effectively, $x_b \in [0,1]$ is shorthand for $x_b \geq 0$ and $x_b \leq 1$.) All that we have done is drop the integrality constraints. A linear program that is obtained from a (mixed) integer program by dropping the integrality constraints is known as the *linear program relaxation* of that (mixed) integer program.

Above, we saw an example where accepting bids partially increases the optimal value of the objective. Given that we showed how to model the winner determination problem with partially acceptable bids as a linear program, and given that the winner determination problem without partially acceptable bids is NP-hard, it should come as no surprise that such examples exist. This is because if there were no such examples, then we could solve any winner determination problem instance without partially acceptable bids using the linear program formulation for partially acceptable bids (since in the end, the optimal solution returned for this linear program would not make any use of the partial acceptability anyway). Because linear programs can be solved in polynomial time, this would prove P=NP.

As a computational problem, the combinatorial auction winner determination problem is also known as the weighted maximum set packing problem.

3 Combinatorial reverse auctions

An important variant of combinatorial auctions is combinatorial *reverse* auctions. In a combinatorial reverse auction, there is a single buyer who needs to procure a set of items I, and there are multiple sellers who are bidding to sell subsets of these items to the buyer. For example, if $I = \{A, B, C, D\}$, a bid might be $(\{A, C, D\}, 5)$, indicating that the corresponding seller is willing to sell all of A, C, D for a price of 5. It should be noted that unlike in combinatorial (forward) auctions, now, lower bids are better. The buyer must accept a subset of the bids that includes at least one copy of each item (duplicates can be thrown away). We can model this as an integer program as follows:

minimize $\sum_b v_b x_b$
subject to
$(\forall i \in I)\ \sum_b a_{ib} x_b \geq 1$
$(\forall b)\ x_b \in \{0,1\}$

where again, a_{ib} is a parameter indicating whether item i occurs in bid b.

As a computational problem, the combinatorial reverse auction winner determination problem is also known as the weighted minimum cover problem. It is NP-hard. Again, we can consider the variant where bids are partially acceptable, which corresponds to the linear program relaxation of this integer program and can therefore be solved in polynomial time. (And, again, examples can be constructed where partial acceptability allows us to find better solutions—unsurprisingly, since otherwise we would have P=NP.)

4 Game theory

Game theory studies settings in which there are multiple players, each of whom has to take one or more actions in some domain. Each player has her own utility function, and her utility potentially depends on the actions of all players. We will study the special case of a *two-player zero-sum normal-form game*. In such a game, there are only two players, and their interests are diametrically opposed: player 2's utility is the negative of player 1's utility. The players simultaneously choose their *strategies*, and based on this, the utilities are determined. For example, the game of rock-paper-scissors falls within this framework, and we can represent this game as follows:

	rock	**paper**	**scissors**
rock	0	-1	1
paper	1	0	-1
scissors	-1	1	0

The rows correspond to player 1's strategies in the game, and the columns to player 2's strategies. Each entry of the matrix gives player 1's utility for that particular combination of strategies (a win gives a utility of 1, a loss a utility of -1, and a draw a utility of 0). This implicitly tells us the utility of player 2, which is the negative of player 1's utility. (In *general-sum* games, there is no such relationship between the players' utilities, so that typically, two utilities are listed in each entry, one for each player.)

How should we play in a two-player zero-sum normal-form game? It may seem like the answer is that it all depends. Playing rock makes sense if the other player plays scissors, but playing scissors makes sense if the other player plays paper. But of course, strategies are chosen simultaneously, so we do not know what the other player will do (unless she is very transparent). One possibility is to play randomly, placing a probability on each strategy. Such a probability distribution is called a *mixed strategy* (and the original strategies are called *pure strategies* to distinguish them). Now, we can take the following paranoid approach: we can suppose that the other player will find out our mixed strategy (but not our random draw) before choosing her strategy, and will choose the best possible strategy for herself (that is, the worst possible strategy for us) in response. We wish to do as well as possible in spite of this pessimistic view.

To make things concrete, let us suppose that we are player 2, so that we are trying to minimize the utility of player 1. We will choose probabilities p_j on the n columns (where j indicates a particular column). Our pessimistic view is that no matter how we choose these probabilities, the row player will manage to play a row $i \in \arg\max_i \sum_{j=1}^{n} p_j u_1(i,j)$. (Here, $u_1(i,j)$ is the utility for player 1 when 1 plays i and 2 plays j.) Still, we can set our probabilities to minimize the maximum utility that the row player can get, holding her to a utility of only $\min_{(p_1,\ldots,p_n)} \max_i \sum_{j=1}^{n} p_j u_1(i,j)$. Such an optimal setting of the p_j is known as a *minimax* strategy. For rock-paper-scissors, the (unique) minimax strategy is to put probability 1/3 on each pure strategy. Then, regardless of the pure strategy that player 1 chooses, her expected utility will be $(1/3)\cdot 1 + (1/3)\cdot 0 + (1/3)\cdot(-1) = 0$. Hence, this mixed strategy guarentees an expected utility of at most 0 for player 1.

Now let us consider a variant of rock-paper-scissors where losing with scissors to rock is considered especially humiliating, resulting in a utility of -2:

	rock	**paper**	**scissors**
rock	0	-1	2
paper	1	0	-1
scissors	-2	1	0

For this game, the minimax strategy for the column player is to play rock with probability 0.25, paper with probability 0.5, and scissors with probability 0.25: it is easy to check that against this mixed strategy, player 1 will always receive an expected utility of 0.

Here is a linear program for finding a minimax strategy:

minimize u
subject to
$(\forall i)\ u - \sum_j p_j u_1(i,j) \geq 0$
$\sum_j p_j = 1$
$(\forall j)\ p_j \geq 0$

The variables of this linear program are the p_j and u, which is the utility that player 1 can obtain by acting optimally. There is an interesting trick in this linear program. We would like to write $u = \max_i \sum_{j=1}^n p_j u_1(i,j)$, but this is not linear because of the max operator. So, instead, the linear program requires that u is *at least* $\sum_{j=1}^n p_j u_1(i,j)$ for every i, thereby forcing it to be at least $\max_i \sum_{j=1}^n p_j u_1(i,j)$. Then, in the optimal solution, u will be set exactly to $\max_i \sum_{j=1}^n p_j u_1(i,j)$, because we try to minimize u. In addition to these constraints, there are constraints that ensure that the p_j actually constitute a probability distribution.

We should ask whether a minimax strategy is the right way to play in a two-player zero-sum game. After all, it seems to be a very pessimistic model to think that the opponent will always play a best response to our mixed strategy. Hence, while it is clear that we can get at least the expected utility for ourselves that this model suggests, we may well wonder if, under a less pessimistic model, we would do better. In particular, what if the roles were reversed, and we were able to see the opponent's mixed strategy first, and best-respond to it? This is a very optimistic model, and certainly we would do at least as well under this model as under the pessimistic model. But a remarkable result known as the *Minimax Theorem* shows that in fact, we will do no better. Formally, $\min_{(p_1,\ldots,p_n)} \max_i \sum_{j=1}^n p_j u_1(i,j) = \max_{(q_1,\ldots,q_m)} \min_j \sum_{i=1}^m q_i u_1(i,j)$. Here, the q_i are the probabilities that player 1 places on the m rows to maximize the minimum expected utility that player 2 can give her (a maximin strategy). This result strongly justifies the use of minimax (or maximin) strategies (it is worthwhile thinking a little about why this is so). We will prove this result when we discuss duality.

Incidentally, game theory extends far beyond two-player zero-sum normal-form games. For one, the utilities do not need to sum to zero. In this more general setting, we typically solve for a *Nash equilibrium* (which can be done using mixed integer programming). Minimax strategies are a special case of Nash equilibrium. There can also be more players. Finally, there can be a temporal structure to the game, with players acting in sequence and seeing (some of) their opponents' moves. Such games are known as *extensive-form* games. There are some very nice techniques for extending the above techniques to this more general setting.

5 Markov decision processes

Consider an environment that can be in any one of a set of *states*. In each period, an actor (who can see the current state) takes an action, and based on this action and the current state, the environment *transitions* to another state. The transition is not necessarily deterministic, though. Given any two states s and s', and an action a, $P(s,a,s')$ gives the probability that, given that the current state is s and the actor takes action a, the next state is s'. That is, if S_t is the state at time t and A_t is the action at time t, then $P(s,a,s') = P(S_{t+1} = s' | S_t = s, A_t = a)$, for any t. Note that the current state and the current action tell us everything that can help us in predicting the next state: $P(S_{t+1}|S_1,\ldots,S_t,A_1,\ldots,A_t) = P(S_{t+1}|S_t,A_t)$. That is, the next state is conditionally independent of the past states and the past actions, given the current state and the current action. This property is known as the *Markov* property. Also, for each state and action pair (s,a), the actor receives some immediate (expected) reward $R(s,a)$. The whole system together is called a *Markov*

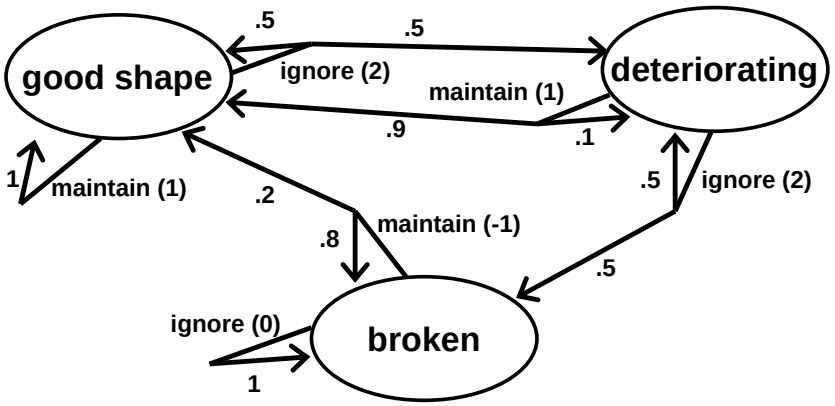

Figure 2: A Markov decision process.

decision process (MDP).

Figure 2 gives an example of an MDP. In this MDP, we (the actor) have a machine, which can be in three different states: in good shape, deteriorating, or broken. At each stage, we know the current state of the machine, and must choose between two actions: maintain the machine, or ignore it. For each state and action, there are arrows that are labeled with the probabilities of going to the other states, given that state-action pair. (If there is no arrow to a state from a particular state-action pair, that means that the probability of going to that state is 0.) Generally, maintaining the machine will tend to put the machine in a better state, whereas ignoring it will tend to put it in a worse state. However, maintenance is costly: it costs 1 to perform a maintenance action. On the other hand, if the machine is currently operational (in good shape or deteriorating), it will generate a revenue of 2 for us. Each state-action pair has, in parentheses, the total reward that we get (revenue minus any maintenance cost).

In principle, the Markov decision process continues forever, so the actor will collect infinitely many rewards. Usually, however, these rewards are *discounted*: the further in the future the reward, the less it is worth. Specifically, there is some $\gamma \in [0, 1)$ such that the reward in period t is multiplied by γ^t to obtain its (present) value. There is a variety of motivations for discounting. One is that the rewards are measured in some currency, and there is an interest rate r, so that a reward t periods from now must be divided by $(1 + r)^t$ to obtain its present value; this corresponds to setting $\gamma = 1/(1 + r)$. An alternative interpretation is that each period, there is a probability of $1 - \gamma$ that the Markov decision process terminates. Thus, the probability that t periods from now, the Markov decision process is still going, is γ^t. In any case, the actor's total utility is $\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t)$.

Now, let us take the above machine example, and consider how we should act for $\gamma = 0.9$. Specifically, we are interested in finding the optimal *policy*. A policy maps states to actions. For example, one policy is to always maintain the machine; another is to only maintain the machine if it is broken, and ignore it otherwise. Which policy will maximize our total expected utility? Of course, our total expected utility depends on which state we are in in the beginning. Given the optimal policy, for every state s, there is some *value* $v^*(s)$, which is the expected total utility given that we start in that state and continue with the optimal policy. Because the policy is optimal, the

following *Bellman optimality equation* must hold for every state s:

$$v^*(s) = \max_a R(s,a) + \gamma \sum_{s'} P(s,a,s') v^*(s')$$

The idea is that taking an action has two effects on our overall utility: first, we receive an immediate reward; and second, we transition to a different state, which will give us a sequence of rewards in the future. The expected utility of that sequence of rewards is the value of that state—with the caveat that all of those rewards will occur one step further into the future, hence we must multiply the value of that state by γ. Under the optimal policy, an action that maximizes the sum of these two components must be chosen.

We will develop a linear program that will give us $v^*(s)$ for every s. Once we have these, it is easy to find the optimal policy: for every state, choose the optimal action as in the Bellman optimality equation. In fact, we will also use the Bellman optimality equation in our linear program. As in the game theory example above, we are confronted with the difficulty that we cannot use the max operator in a linear program. The solution is similar. We use the following linear program:

minimize $\sum_s v^*_s$
subject to
$(\forall s, a)\ v^*_s \geq R(s,a) + \gamma \sum_{s'} P(s,a,s') v^*_{s'}$

The constraints force each v^*_s to be at least $\max_a R(s,a) + \gamma \sum_{s'} P(s,a,s') v^*_{s'}$; because the objective is to minimize these values, in the optimal solution, they will be set exactly equal to $\max_a R(s,a) + \gamma \sum_{s'} P(s,a,s') v^*_{s'}$.

For the machine example above, the optimal solution to this linear program sets $v^*(\text{good shape}) = 16.69$, $v^*(\text{deteriorating}) = 15.96$, and $v^*(\text{broken}) = 7.19$. From this, it is easy to derive that the optimal policy is to ignore the machine when it is in good shape, and maintain it otherwise.

6 Rank aggregation (the Kemeny rule)

Suppose that we have some set of elements I, and several rankings of these elements. For example, we may have $I = \{A, B, C, D\}$, and the following three rankings: $A \succ_1 B \succ_1 D \succ_1 C$, $C \succ_2 A \succ_2 B \succ_2 D$, and $D \succ_3 B \succ_3 C \succ_3 A$. Here, $a \succ_j b$ indicates that ranking j ranks a higher than b. We wish to obtain a single aggregate ranking $\succ$ of I from these rankings.

There are many motivations for this problem. A group of people may have to make a joint choice from a set of alternatives I; the ranking $\succ_j$ corresponds to the jth person's preferences over the alternatives, and the aggregate ranking would correspond to the group's aggregate preferences. (This type of setting is typically referred to as a *social choice* or *voting* setting.) Alternatively, perhaps we have entered the same query in several different search engines; each engine j returns its own ranking $\succ_j$ of the relevant pages, and we wish to come up with an aggregate ranking of these pages.

There are *many* different ways to determine an aggregate ranking, and which one is optimal has been a topic of intense debate in the social choice community at least since the 18th century. Here, we will study one nice rule for determining an aggregate ranking, the *Kemeny rule*.

The Kemeny rule works as follows. Given a potential aggregate ranking $\succ$, one of the input rankings $\succ_j$, and $a, b \in I$, let $\delta(\succ, \succ_j, a, b) = 1$ if $a \succ b$ and $b \succ_j a$, and let it be 0 otherwise. Then, the *Kemeny score* of ranking $\succ$ is $\sum_j \sum_{a \neq b} \delta(\succ, \succ_j, a, b)$. That is, the Kemeny score of a potential aggregate ranking is the total number of *disagreements*, where a disagreement occurs whenever there is some input ranking and some pair of elements such that the aggregate ranking and the input ranking disagree on that pair of elements. The Kemeny rule chooses an aggregate

ranking with the lowest Kemeny score (if there are multiple such rankings, the Kemeny rule does not specify a tiebreaker).

For the example above, the unique Kemeny ranking is $A \succ B \succ D \succ C$. The score of this ranking is 7 (for every pair of alternatives, there is one disagreement, except for the pair A, C, for which there are two disagreements).

It turns out that the problem of determining a Kemeny ranking is NP-hard (it is closely related to a problem known as the minimum feedback arc set problem), so there is little hope of finding a linear program formulation for it. Instead, we will settle for an integer program formulation. For every ordered pair of alternatives a, b, let x_{ab} be a binary variable that is 1 if a is ranked ahead of b in the aggregate ranking, and 0 otherwise. We need $x_{ab} + x_{ba} = 1$ (exactly one of a and b must be ahead of the other). We must also have $x_{ab} + x_{bc} + x_{ca} \leq 2$, because if all of these variables are equal to 1, then that means that a is ranked ahead of b, b is ranked ahead of c, and c is ranked ahead of a, which is impossible. Conversely, if the variables are set in such a way that these constraints hold, then this will in fact correspond to a ranking. Now, if we let n_{ab} (a parameter) be the number of input rankings that rank a ahead of b, then we seek to minimize $\sum_{a \neq b} n_{ba} x_{ab}$. Putting it all together, we obtain:

minimize $\sum_{a \neq b} n_{ba} x_{ab}$
subject to
$(\forall a, b : a \neq b)$ $x_{ab} + x_{ba} = 1$
$(\forall a, b, c : a \neq b, b \neq c, c \neq a)$ $x_{ab} + x_{bc} + x_{ca} \leq 2$
$(\forall a, b : a \neq b)$ $x_{ab} \in \{0, 1\}$

7 Sudoku

Our last application is a little more unusual: we will consider how to solve Sudoku puzzles using integer programming. (Incidentally, Sudoku was invented in the United States under the less inspiring name "Number Place" but became popular in Japan.) A Sudoku puzzle consists of a 9×9 grid, subdivided into nine 3×3 blocks. Each of the entries must be filled in with a number from $\{1, \ldots, 9\}$, and some of the entries have already been filled in. The constraints are that every number should occur exactly once in each row, each column, and each block.

In this problem, the only goal is to find a feasible solution, so we will not have any objective. What should our variables be? It is tempting to say that for each entry, we should have an integer variable that takes values in $\{1, \ldots, 9\}$. However, some reflection reveals that the numerical values of the numbers have no meaning in this puzzle; we may as well have used letters, or anything else. For example, in "celebrity Sudoku" the entries must be filled in with pictures of celebrities. Because of this, it seems to make more sense to define binary variables $x_{ijk} \in \{0, 1\}$, where $x_{ijk} = 1$ if in the ith row, in the jth column, there is a k. We can formulate the Sudoku problem as follows:

subject to
$(\forall i, j)$ $\sum_k x_{ijk} = 1$
$(\forall i, k)$ $\sum_j x_{ijk} = 1$
$(\forall j, k)$ $\sum_i x_{ijk} = 1$
$(\forall k,$ block $b)$ $\sum_{(i,j) \in b} x_{ijk} = 1$

The first constraint says that each entry should have one number in it; the second, third, and fourth constraints say that each number should occur once in each row, column, and block, respectively. The first three constraints have an interesting symmetry among i, j, and k, but this symmetry is ruined by the fourth constraint. Note that the integer program is easily applied to, say, 16×16 Sudoku as well, and in fact it is easy to modify it to solve other variants of Sudoku as well.

Readers are encouraged to try out this formulation on their favorite Sudoku puzzles to see just how fast this approach is.

8 Conclusion

The applications above were chosen to illustrate the variety of problems of interest to computer scientists that can be solved using linear or integer programming. However, the list is nowhere close to exhaustive. The variety of problems that can be solved using (mixed) integer programming is especially large.

Transforming a problem to a linear or integer program and then using a standard solver does not always result in the fastest known algorithm to solve a problem (though sometimes it does). However, often it is fast enough, and it is relatively easy to do. Moreover, it is generally relatively easy to extend the linear or integer program to variants of the original problem. There are other benefits as well: for a problem that can be modeled as a (mixed) integer program, it is often useful to have its linear program relaxation to obtain an upper bound; and the *dual* of a linear program (which is a related linear program) often gives useful insight into the problem, as we will see shortly.