ORIE 3310/5310/5311: Optimization II
Course Notes

David Shmoys and Adrian Lewis

1 Introduction

This course will cover methods to solve integer and linear programming problems, and present applications in which these methods are useful; whereas the emphasis of Optimization I was on linear programming, the emphasis of Optimization II will be on issues related to integer programming. There will be five major topics in the course: network optimization, dynamic programming, integer programming, non-linear programming, and modern linear programming techniques (in roughly this order).

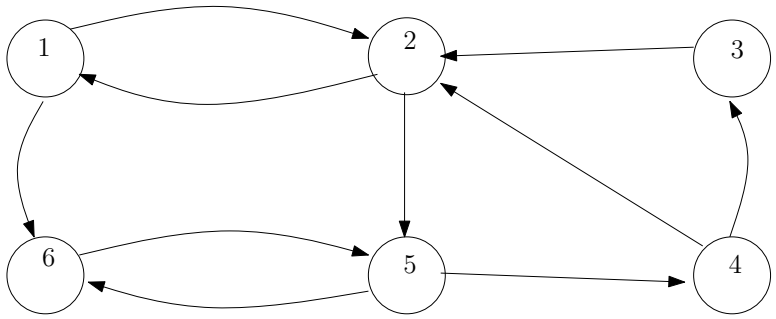
2 Network Optimization

Throughout Optimization I, the notion of a graph (or network) appeared repeatedly, starting with the very first lecture. Many network optimization models are an important intermediate step between linear programming and integer programming models: one is often interested in solving these models to get "integer answers", and solving them as linear programs is equivalent to solving them as integer program due to a very special propserty known as an *integrality property* that these special linear programs possess. This property makes these linear programs especially amenable to be solved, not only by the simplex method, but by specially designed procedures that take further advantage of their network structure.

One other reason that these network optimization models are important is that many applications (some of which do not appear to be a network optimization problem at all) can be formulated as one of these network optimization models, and hence solved by the methods developed for them. For example, one of the models that we will study is called the *minimum-cost network flow problem*. In Optimization I, we often "solved" a problem by setting it up as a linear programming problem (being careful that it really could be formulated with the requirement inherent in what it means to be a linear programming problem). Now we will also "solve" other problems by showing that they can be viewed as special cases of the minimum-cost network flow problem. This philosophy of first introducing a new optimization problem, developing methods to solve the new problem, and then showing several other applications that can be cast in this new model, is one of the themes of the first part of Optimization II.

2.1 The minimum-cost flow network problem

The *minimum-cost network flow problem* is also often called the *transshipment problem*, and is a specific optimization model that captures a setting of shipping a commodity between given locations at minimum total cost. (The two names for this problem are completely interchangeable.) To specify an input for this problem, we must first give a directed graph (or network), such as is shown below.



The circle elements are called *nodes* and the lines between nodes are called *directed edges*. (The fact that this is a directed graph is due to the fact that these lines have arrows on them that indicate, for example, a link *from* node 1 *to* node 6; that is, there is a direction associated with the line.) Nodes are often called *vertices* or *points*; directed edges are often called *arcs* or *links*. All of these terms will be used in this course.

Networks are simple to understand by looking at them, but how do we input them into a computer? They are specified completely, not by the picture, but stating the set of nodes N and the set of directed edges E. The graph depicted above is therefore the graph with node set $N = \{1, 2, 3, 4, 5, 6\}$ and directed edge set

$$E = \{(1,2), (2,1), (3,2), (5,4), (6,5), (5,6), (1,6), (2,5), (4,2), (4,3)\}.$$

The directed graph is just the first part of the input to the minimum-cost network flow problem. In addition, for each directed edge in the input graph, we are also given the per-unit cost of shipping a commodity across that edge. That is, for each $(i, j) \in E$, we are given a value c_{ij}. This value may be positive or negative (or even 0).

Next, for each directed edge in the input graph, we are also given an edge capacity, that corresponds to the maximum number of units that may be

shipped across that directed edge. That is, for each $(i, j) \in E$, we are given a value u_{ij}; in this case, we require that u_{ij} be positive (but it is allowed for u_{ij} to be infinite, to reflect that there is no upper bound on the amount of flow that can be shipped across that directed edge).

Finally, for each node in the input graph we first designate that node as either a *supply node*, a *demand node*, or as a *transit node*. We do this by specifying a "supply value" b_i for each node $i \in N$. If $b_i > 0$, then the node i is a supply node, and b_i indicates the amount of the commodity that is on hand to be shipped from node i. On the other hand, if $b_i < 0$ for a node i, this means that the node is a demand node, and requires that $|b_i| (= -b_i)$ units of the commodity be shipped to it. Finally, a transit node i is one for which the associated value b_i is equal to 0.

This completes the specification of the input to the minimum-cost network flow problem. We have implicitly indicated what the optimization problem is along the way, but we will make this much more precise next. But, roughly, the idea to find a plan for shipping the commodity through the network that meets the supply and demand constraints at minimum total cost, while obeying the capacity constraints of the network.

There is one further condition that our input must obey. If one thinks about the physical realities of a commodity, then if we are going to ship units of the commodity to meet the demand only from the supply points in the network, then the total amount of supply at those nodes must be at least the total demand of the demand nodes. The total supply can be written as

$$\sum_{i \in N : b_i > 0} b_i;$$

the total demand can be written as

$$\sum_{i \in N : b_i < 0} -b_i.$$

Thus, for our input to make sense, we require that

$$\sum_{i \in N : b_i > 0} b_i \geq \sum_{i \in N : b_i < 0} -b_i,$$

but this is the same thing as

$$\sum_{i \in N : b_i > 0} b_i - \sum_{i \in N : b_i < 0} -b_i \geq 0,$$

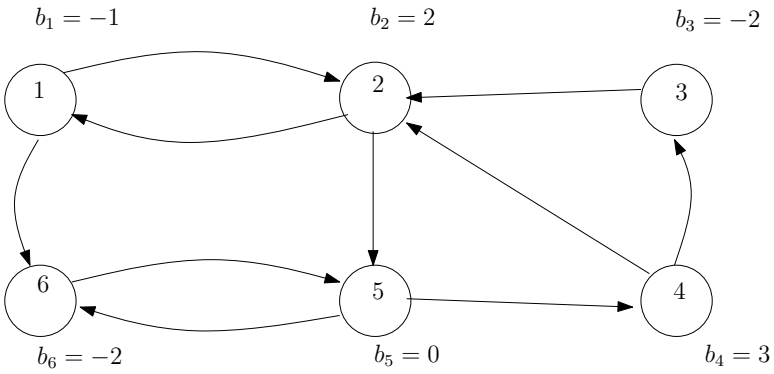
and this can be more compactly written as

$$\sum_{i \in N} b_i \geq 0.$$

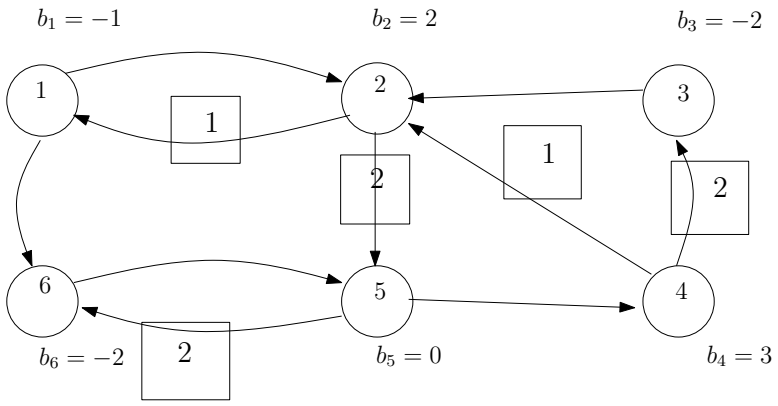
So we know that the total (signed) supply must be non-negative for our input to truly make sense. However, we will require something stronger for our inputs to the minimum-cost flow problem: we will require a so-called *balance condition*, that is,

$$\sum_{i \in N} b_i = 0,$$

or in other words, that the total supply is exactly equal to the total demand. (Otherwise, we do not have a legitimate input to the minimum-cost network flow problem.) In the picture below, we have added supply values to the nodes of the previous graph; these supply values satisfy the balance condition.



Furthermore, suppose that the capacity value for each edge in the graph is equal to 2, and each per-unit shipping cost is equal to 1. Then a feasible solution for this input is given in the picture below, by indicating the amount shipped along each directed edge by the boxed number; if there is no number next to an edge, then the amount shipped is equal to 0.



Observe that for each supply node, we exactly ship out the supply (in terms of the net amount shipped out of that node), and for each demand node, we exactly meet the required demand (again, in terms of the net amount shipped out). This is a consequence of the balance condition; since we have in total just enough supply to meet the total demand, every supply point must be completely exhausted after shipping, and no demand point can get any more than it requires.

In our informal discussion of the constraints of this problem, we have simply indicated that we are shipping units of a commodity over the edges of this network; we have not specified that the number of units needs to be an integer value. Hence, it will be possible to write this minimum-cost network flow problem as a linear program.

LP Formulation. There is a natural sequence of steps in setting up any linear programming formulation.

Step 1 Declare your decision variables.

In this case, we will introduce a variable x_{ij} for each directed edge $(i, j) \in E$, which indicates the number of units of the commodity that are shipped along this edge.

Step 2 Express the objective function in terms of these variables.

In this case, we want to minimize the total cost of the shipping plan; for each edge $(i, j) \in E$, we ship x_{ij} units at a per-unit shipping cost of c_{ij}, and so

the total cost is

$$\sum_{(i,j)\in E} c_{ij}x_{ij},$$

and our objective is to minimize this function.

Step 3 Express the constraints of the problem in terms of the decision variables.

In this case, we first indicate that the amount shipped is non-negative, and no more than the capacity, for each directed edge in the input: for each edge $(i, j) \in E$, $0 \leq x_{ij} \leq u_{ij}$.

Another constraint is that the commodity is just being shipped around this network, but not be either created nor consumed in the process. So, for example, at a transit node, the amount shipped out of the node must exactly equal the amount shipped into that node. So, in our example, for node 5, we must have that

$$x_{54} + x_{56} = x_{25} + x_{65},$$

or equivalently, that

$$[x_{54} + x_{56}] - [x_{25} + x_{65}] = 0.$$

This can be expressed more generally as: for each i such that $b_i = 0$, we must have

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = 0.$$

Next consider what happens at a supply node; for this type of node, then the *net* amount shipped out (total shipped out minus the total amount shipped in) must be exactly equal to its supply. For example, for node 2, we must have that

$$[x_{21} + x_{25}] - [x_{42} + x_{32} + x_{12}] = 2.$$

More generally, for each node with $b_i > 0$, we have that

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = b_i.$$

Finally, consider the demand nodes: here, the total shipped in minus the total shipped out should equal the demand. Since for each demand node i

we have $b_i < 0$ and $-b_i$ is the actual (positive) demand at that node, this constraint can ben written as:

$$\sum_{j:(j,i)\in E} x_{ji} - \sum_{j:(i,j)\in E} x_{ij} = -b_i.$$

But this constraint can be rewritten (by multiplying by -1) as:

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = b_i.$$

Look back at the 3 types of *flow conservation constraints* that we have written for the transit, supply, and demand nodes, respectively. We see now that there is only one form that they all take: for each node i, we simply require that

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = b_i;$$

these are what we will refer to as the flow conservation constraints for the minimum-cost network flow problem.

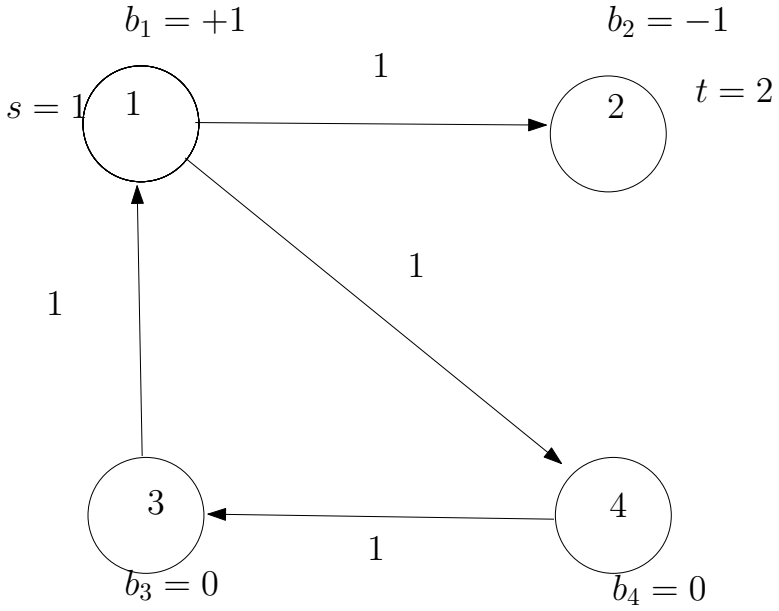
The importance of this particular optimization model is largely due to the number of applications that can be cast as minimum-cost network flow problems. One reason why this model has so many applications is due to the following theorem, which states that we can use the linear program to obtain integer solutions.

Theorem 2.1 (Integrality Property of Min-Cost Network Flow) *Consider any input to the minimum-cost network flow problem for which the capacity u_e for each directed edge e in the input graph is an integer, and the supply value b_i of each node i in the graph is an integer. Then there exists an optimal solution x^* to this input of the minimum-cost network problem for which the value x^*_e for each edge e is also an integer; furthermore, if $\hat{x}$ is a basic feasible solution, then $\hat{x}(e)$, for each $e \in E$, also is an integer.*

We will not prove this theorem in this course, but merely take it for granted that it is true. We will show that one important consequence of this theorem is that the minimum-cost network flow problem captures a number of applications that are at first glance unrelated.

Special Case # 1: The shortest path problem Suppose that we consider inputs to the minimum-cost network flow problem that satisfy the following restrictions. The input graph is still allowed to be any directed graph $G = (N, E)$ (that is, the directed graph with node set N and directed edge set E). However, for each edge $e \in E$, its capacity must equal 1; that is, $u_e = 1$ for each $e \in E$. Furthermore, the per-unit shipping cost for each directed edge e is now required to be non-negative; that is, $c_e \geq 0$ for each $e \in E$. Finally, the supply values have a very restricted structure: there is one node $s \in N$ (called the *source*) and another node $t \in N$ (called the *sink*), where $b_s = 1$, $b_t = -1$, and $b_i = 0$ for each other node in N. It is easy to see that these inputs do satisfy the required balance condition for the supply values.

The picture below gives a sample input of the required form.



The integrality theorem says that when we find the optimal solution, we can equally well be finding the optimal solution among all integer feasible solutions (since all capacities and supply values are integer).

But what does an integer feasible solution x look like? One way that we can construct a feasible integer solution is to take any path in the graph that starts at node s and ends at node t; let P denote the set of directed edges in this path. Suppose that we set $x_e = 1$ for each $e \in P$, and $x_f = 0$ for each directed edge $f \notin P$ (but in E). This is clearly an integer feasible solution

for our problem. Furthermore, the cost of this solution is

$$\sum_{e \in P} c_e * 1 + \sum_{f \notin P} c_f * 0 = \sum_{e \in P} c_e.$$

In other words, every path from s to t is a feasible solution to this input, and the cost of each such solution is the total cost of the edges in the path.

If these were the only feasible integer solutions, then finding the cheapest integer solution is the same thing as finding the cheapest (or as more oftenly called, shortest) path from s to t. (And since the integrality theorem implies that finding the optimal solution to the minimum-cost flow problem is the same as finding the optimal integer solution, we could say that solving the minimum-cost network flow problem of this special structure solves the shortest path problem. Unfortunately, life is not quite this simple.

Let's try to prove that the only feasible integer solutions correspond to paths, and see where things don't work right. Consider node s; since this has supply value equal to 1, there must be net flow out of node s equal to 1. Hence, there is some directed edge (s, i) leaving node s with a positive flow value x_{si}. But since we can assume that this value is an integer, and it must be no more than the capacity value equal to 1, there is only one such positive value. The flow value for this directed edge is equal to 1. Now there are two cases, either node i is the sink node t, or else it is a transit node. In the former case, we have (clearly) reached the sink node. In the latter case, we can now repeat the same sort of argument. The net flow for node i is 0, and we have just identified a directed edge (s, i) that is bringing 1 unit of flow into node i. Hence there is another directed edge (i, j) leaving node i that has positive flow (which again implies that $x_{ij} = 1$).

One natural conclusion to come to (which is almost correct) is that repeating this argument implies (eventually, or more formally, by induction) that any integer feasible solution x corresponds to a path from s to t in the input graph. There is one complication to this argument, however. Consider the feasible solution to the graph above when the flow on each edge is equal to 1. (Verify for yourself that this is a feasible solution!) But this doesn't seem like a path from node 1 to 2. If we apply the reasoning that we just gave, we might have first selected edge $(1, 4)$ to leave node 1. The complication is that by not being "picky" about which edge to choose, we might simply follow a cycle in the graph: a sequence of edges that cause us to start and end at the same node (in this case node 1). Of course, for any cycle, we are just shipping the commodity around the cycle without really making

any "progress", so for any such feasible solution x, if we identify a cycle of edges $(i_1, i_2), (i_2, i_3), \ldots, (i_{m-1}, i_m), (i_m, i_1)$ each with flow value =1, then we can generate a new integer feasible solution by changing these m values to 0. By repeating this enough times, then eventually the edges with flow value 1 will no longer have any cycles, and hence the original conclusion is correct: we will identify a path from s to t. We have argued the following fact about feasible solutions to our special case of minimum-cost network flow inputs:

Fact 2.2 *Any feasible integer solution can be viewed as a selection of directed edges made up of a number of cycles (with disjoint edges) and a path from the source node s to the sink node t.*

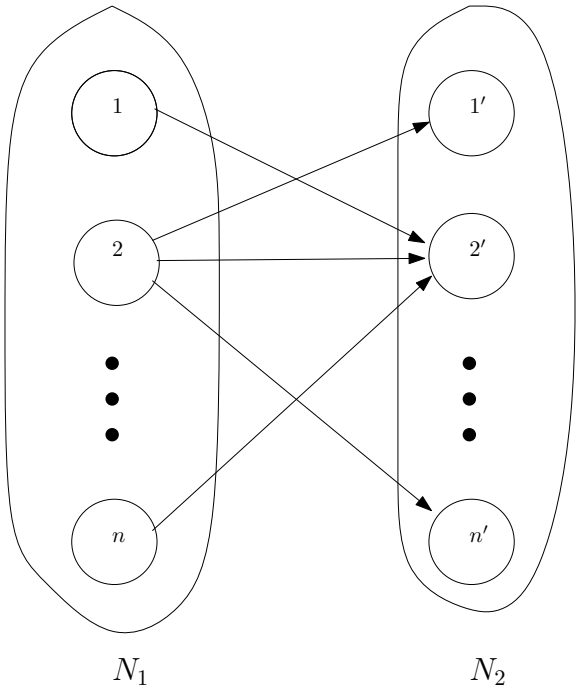
So now we know what feasible integer solutions correspond to; what can we say about the optimal integer solution? Does this still have anything to do with finding a shortest (or cheapest) path from node s to node t? Suppose that x^* is an optimal solution to the minimum-cost network flow problem of our special form that is integer (and whose existence is ensured by the integrality theorem). Suppose that there is a cycle in the set of edges e with $x^*_e = 1$. If we reset those flow values to 0, we get a new integer feasible solution x'. Since x^* is optimal, x' cannot cost less than x^*. But $c_e \geq 0$ for each $e \in E$ (by the format of our special case). Hence, for each edge e in the cycle, we must have that $c_e = 0$. But then x' is another optimal solution (and pushes flow across fewer edges). By repeating this "cycle deletion" argument, we can obtain a new feasible integer solution $\hat{x}$ that corresponds exactly to a path from s to t. Thus, although they might be "disguised" by additional cycles made up entirely of edges of cost 0, optimal solutions to the minimum-cost network flow problem correspond to solutions to the problem of finding the cheapest path from node s to node t.

So, if we need to compute the shortest path between two nodes in a graph, and we are given only software that solves the minimum-cost network flow problem, then we can still find the shortest path. Furthermore, the integrality theorem implies that by using just AMPL calling CPLEX as a *linear programming* solver for the minimum-cost network flow problem, we can still compute the shortest path between two nodes in a network. (However, this is far from the most efficient way to do this. Nonetheless, sometimes it might be the most expedient way to do so, if that is the solver that you have on hand.)

One final observation. Suppose that we didn't have the guarantee that $c_e \geq 0$ for each edge in the graph. For example, suppose that the cost of

each edge in the example above was equal to -1. Then the feasible solution shown (with the extra cycle) is the optimal solution, and it is cheaper than the simple single-edge path from s to t. So, we need the fact that costs are special. But do they need to be as special as requiring that each edge is non-negative? In fact, we do not need something that restrictive. If instead, we are given costs for the graph for which we have the promise that the total cost of each cycle is non-negative, we can use the exact same argument as we gave above to show that the shortest path from s to t must be an optimal solution to this special form of minimum-cost network flow inputs.

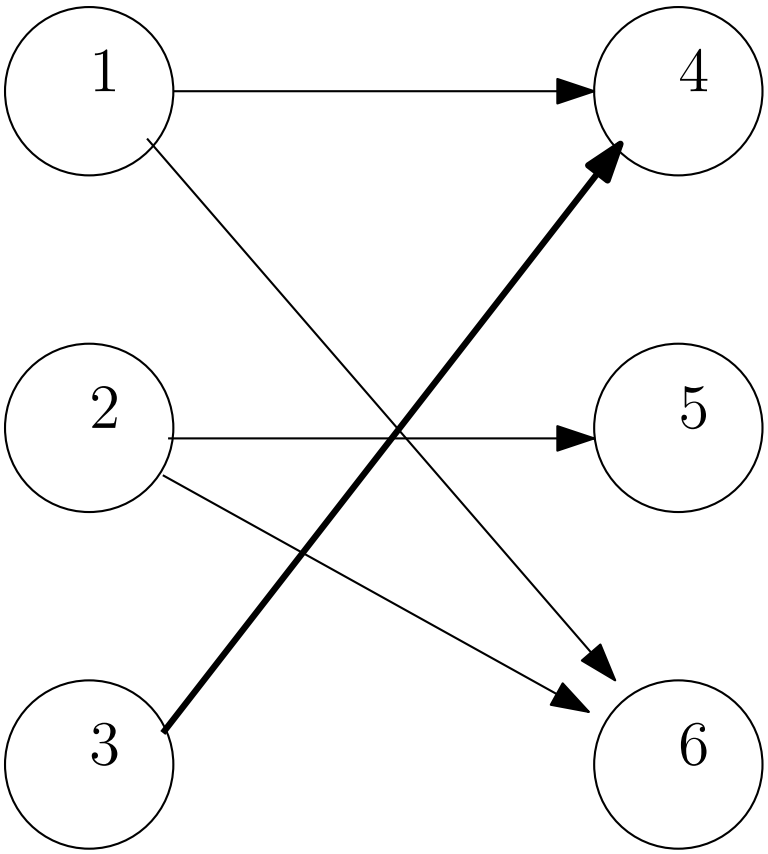
Special case #2: the assignment problem In this special case, we shall restrict attention to a special class of input graphs. In these graphs, we have two disjoint subsets N_1 and N_2, each with n nodes, and we have the additional property that each edge (i, j) has the requirement that $i \in N_1$, and $j \in N_2$. Thus, we have the following picture:



As in the previous special case, we will require that $u_{ij} = 1$ for each edge (i, j) in the graph. Finally, we will set

$$b_i = \begin{cases} 1 & \text{for each } i \in N_1 \\ -1 & \text{for each } i \in N_2 \end{cases}$$

Does the balance condition on the supply values hold? In adding up all of the supply values we add 1 for each node in N_1 and subtract 1 for each node in N_2. However, we indicated that there were n nodes in each of these two sets, so that implies that the balance condition holds. Consider a sample input as depicted below.



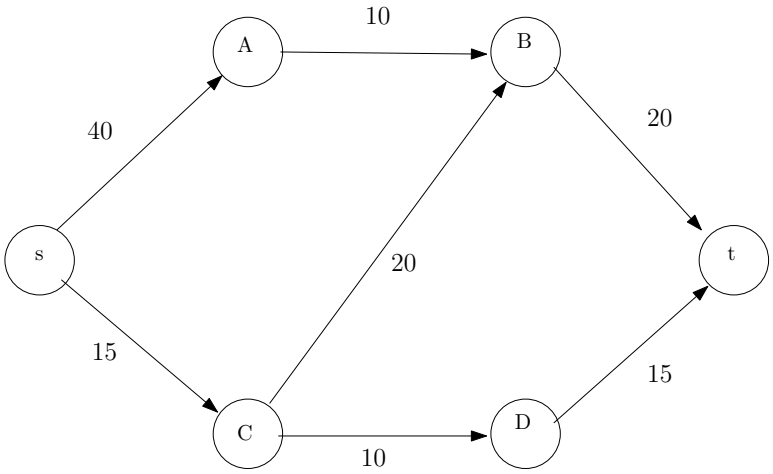
One feasible integer solution x for this input is to set $x_{16} = x_{25} = x_{34} = 1$ and $x_{14} = x_{26} = 0$. Observe that by focusing on the edges that have flow value equal to 1, we can view the flow as pairing up each node in N_1 with a distinct node in N_2. Unlike the special inputs for the shortest path problem, in this case it is clear that all feasible integer solutions have exactly this structure. If we let M denote the set of edges e for which $x_e = 1$, the cost of the solution is simply $\sum_{e \in M} c_e$. So, optimizing over all integer feasible solutions is the same thing as optimizing over all such pairings of all nodes in N_1 and all nodes in N_2.

Suppose that we think of the nodes in N_1 as corresponding to people, and the nodes in N_2 as corresponding to tasks; we have an edge from person $i \in N_1$ to task $j \in N_2$ whenever it is possible for that person to be assigned that task, and suppose that the "per-unit" cost c_{ij} indicates the total time that person i takes to do task j). Then the minimum-cost network flow problem solves the problem of pairing up people and tasks (one person per task) so that the total time taken is as small as possible. It is very important to note that we use the integrality property here in a crucial way; otherwise, it might be possible that in order to obtain an optimal solution, we might have to have people "share" tasks so that, for example, person 1 does half of job 4 and half of job 6. This type of application is why this special case is often referred to as the *assignment problem*.

So, just as in the previous special case, if we have an input to the assignment problem, then we can solve it using AMPL and the CPLEX *linear programming* solver, by setting up the problem as a minimum-cost network flow problem.

Special case #3: the maximum flow problem In the previous two examples, we started by saying how the input to the minimum-cost network flow problem was special, and then deduced how solving the minimum-cost network flow problem of that form leads to a problem that was interesting in its own right (and required the integrality property to make sense). In this example, we will start by introducing a new optimization problem, and then show how we can solve this new problem as a special case of the minimum-cost flow problem. This will be the more typical paradigm that we will use for all subsequent applications.

The new optimization problem is called the *maximum flow problem*. An input to the maximum flow problem consists of a directed graph $G = (N, E)$ with node set N and directed edge set E; there are two special nodes specified as part of the input: a source node $s \in N$ and a sink node $t \in N$. For each directed edge $e \in E$, there is a capacity u_e given (where we assume that $u_e > 0$). A sample input is given in the picture below.

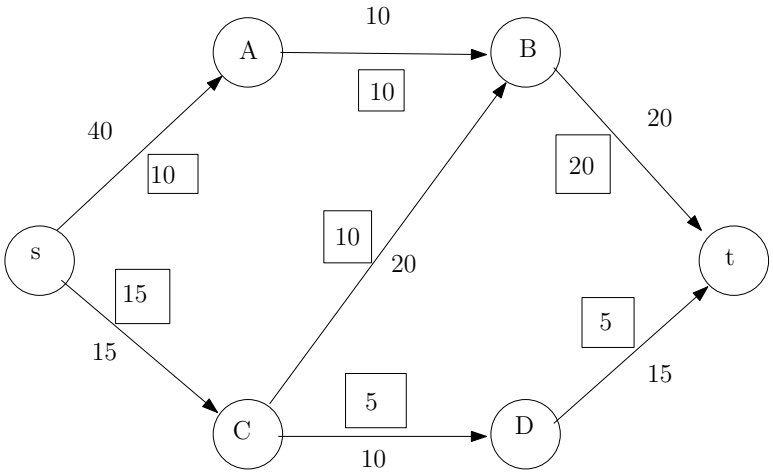


A feasible solution to the maximum flow problem consists of a flow value x_e for each edge $e \in E$ that satisfy the following conditions:

- (Capacity constraints) for each edge $e \in E$, $0 \leq x_e \leq u_e$;
- (Flow conservation constraints) for each node $i \neq s, t$,

$$\sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} = 0.$$

A feasible solution for the input above is given in the boxes next to each directed edge in the picture below.



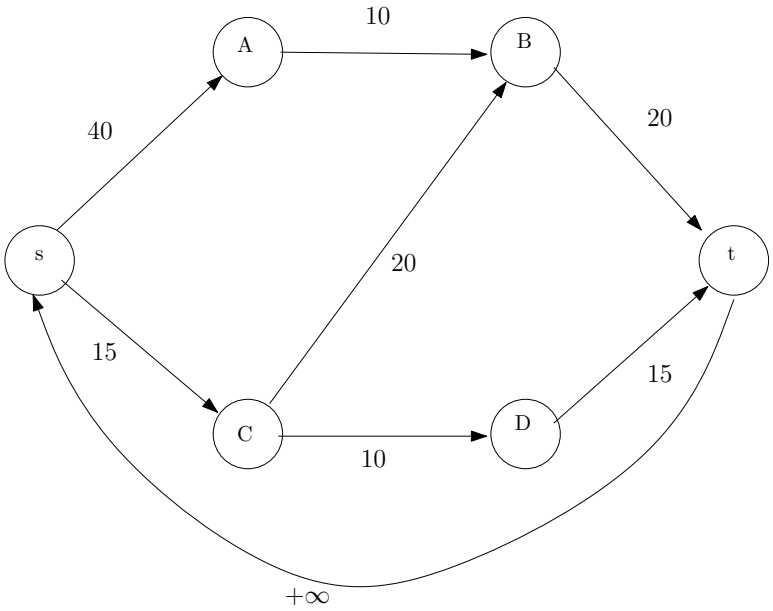
The value of a flow x that satisfies these constraints is equal to the net flow into the sink:

$$\sum_{j:(j,t)\in E} x_{jt} - \sum_{j:(t,j)\in E} x_{tj}.$$

The objective of the maximum flow problem is to compute a feasible flow of maximum value.

We will show that the maximum flow problem can be derived as a special case of the minimum-cost network flow problem. To do this, we will formulate the maximum flow problem as a special class of minimum-cost network flow problems. This process of formulating a problem of type A as a problem of type B has a few standard steps (just as the process of LP formulation has a few well-defined steps). We will illustrate this process by the concrete example of formulating the maximum flow problem as a minimum-cost network flow problem.

Step 1 Show how to take any input to the problem of type A and transform it into an input to the problem of type B. In this setting, we need to take an input to the maximum flow problem, and generate from it an input to the minimum-cost network flow problem. This is relatively straightforward. We have an input graph $G = (N, E)$ for the maximum flow problem. We add a directed edge from node t to node s, and get the new graph $G' = (N, E')$ where $E' = E \cup \{(t, s)\}$. For each edge $e \in E$, its capacity in G' is exactly the capacity u_e given as part of the maximum flow input. However, the capacity of the new arc (t, s), $u_{ts} = +\infty$. For each node $i \in N$, the supply values $b_i = 0$. For each directed edge $e \in E$, its cost $c_e = 0$, and for the new arc (t, s), $c_{ts} = -1$. This can be viewed as a "profit" for shipping along this new arc, which is sometimes called a "return arc" since it takes the flow that starts at the source and reaches the sink and returns it directly to the source. For the sample input to the maximum flow problem, the transformed input that is constructed for the minimum-cost network flow is depicted below.



Step 2 Show that, for any input to problem of type A, its feasible solutions correspond to the feasible solutions to the newly generated input for the problem of type B. Again, we illustrate this step by means of our example where A is the maximum flow problem and B is the minimum-cost flow problem. In this case, the correspondence is actually trivial (in the literal sense of the word). Any feasible solution x to an input for the maximum flow problem can be extended to a feasible solution to the new input G' to the minimum-cost flow problem by setting the flow value for each edge $e \in E$ to be x_e and setting the flow value for the new arc (t, s) to be

$$\sum_{j:(j,t)\in E} x_{jt} - \sum_{j:(t,j)\in E} x_{tj}.$$

This flow clearly satisfies the capacity constraints (since the only new constraint is for the arc (t, s) for which the capacity is $+\infty$). It requires only straightforward algebraic calculations to show that the resulting flow satisfies the flow conservation constraints implied by the fact that we have set all of the nodes to be transit nodes, that is, that each $b_i = 0$. Furthermore, given any feasible solution to the constructed input to the minimum-cost network flow problem, we can get a feasible solution to the starting input for the maximum flow problem, by simply ignoring the flow on the return arc. Note that if x_e denotes the flow on each of the arcs $e \in E$, then flow conservation

constraint for node t (that is, that the net flow through t is equal to 0 for the feasible solution to our minimum-cost network flow input) implies that the flow on the arc (t, s) is exactly equal to

$$\sum_{j:(j,t)\in E} x_{jt} - \sum_{j:(t,j)\in E} x_{tj}.$$

What have we accomplished by this argument? We have shown that the set of feasible solutions that we optimize over for the two inputs are exactly the same. Each feasible solution for the maximum flow input shows up as one minimum-cost flow input, and furthermore, no new solutions are considered, since for any feasible solution to the minimum-cost flow input, there is a feasible solution to the original maximum flow input that could have generated it. This is exactly what Step 2 requires. **Step 3** Show that the objective functions of the corresponding feasible solutions have an analogous correspondence as well.

What happens with the construction that we just gave. We start with a maximum flow feasible solution of value

$$\sum_{j:(j,t)\in E} x_{jt} - \sum_{j:(t,j)\in E} x_{tj}.$$

What is the cost of the feasible solution corresponding to it for the minimum-cost flow problem? The only edge with non-zero per-unit cost is the return arc (t, s), which has cost -1. We put a flow of value

$$\sum_{j:(j,t)\in E} x_{jt} - \sum_{j:(t,j)\in E} x_{tj}$$

on this arc, and so the cost of the solution is

$$-[\sum_{j:(j,t)\in E} x_{jt} - \sum_{j:(t,j)\in E} x_{tj}].$$

Our objective in the minimum-cost flow problem is to make this quantity as small as possible. But of course, if we make a function $-f$ as small as possible, then we make the function f as big as possible. That is, by solving this input of the minimum-cost network flow problem, we find a feasible solution to the maximum flow for which

$$\sum_{j:(j,t)\in E} x_{jt} - \sum_{j:(t,j)\in E} x_{tj}$$

is maximized; but that is exactly what the maximum flow problem requires.

Hence, we have shown that we can find an optimal solution to any input to the maximum flow problem by solving the corresponding input to the minimum-cost flow problem. Furthermore, since the maximum flow problem is thereby a special case of the minimum-cost network flow problem, we know that the integrality theorem for the min-cost flow problem yields an analogous result for the maximum flow problem.

Theorem 2.3 *For any input to the maximum flow problem for which all capacities are integer, there exists an optimal solution in which the flow value for each edge is also integral.*

3 The maximum flow problem

Although it is a special case of the minimum-cost network flow problem, the *maximum flow problem* is of great importance in its own right. This problem can be motivated by the following setting. Imagine that you have a network of pipes that are used to ship, for example, oil from its source, to where is it is refined. Each pipe in the network can maintain a certain capacity of flow (per second), which depends on its cross-section, and other less significant factors. At what rate can we deliver oil to the refinery?

The network of pipes corresponds to a directed graph $G = (N, E)$. Each directed edge $(i, j) \in E$ has a specified capacity u_{ij}. There is a specified source node $s \in N$ and a specified sink node $t \in N$. This is the entire input to the maximum flow problem.

To specify a solution for this input, we must give a flow value x_{ij} for each directed edge $(i.j) \in E$. Such a solution x is feasible if

1. $0 \leq x_{ij} \leq u_{ij}$ for each $(i.j) \in E$; and
2. $\sum_{j:(j,i)\in E} x_{ji} = \sum_{j:(i,j)\in E} x_{ij}$ for each node $i \in N - \{s, t\}$
   (i.e., each node that is neither the source nor the sink).

The first type of constraints, the inequalities, are called *capacity constraints* and the second type, the equations, are called *flow conservation constraints*. (These are exactly the constraints of the LP formulation that we gave in the previous section.)

The value of a flow x is the total net flow into t, which is equal to

$$\sum_{j:(j,t)\in E} x_{jt} - \sum_{k:(t,k)\in E} x_{tk}.$$

This is the objective function for the maximum flow problem; in other words, we wish to find a feasible flow of maximum value.

How big can the optimal flow value be? Partition the vertices N into a set S containing the source s and a set T containing the sink t. (The sets S and T form a partition of N if each node in N is in *exactly* one of S and T.) We shall call such a partition (S, T) a *cut*; note that the definition of a cut requires that $s \in S$ and $t \in T$. Observe that every unit of flow that goes from node s to node t must at some point pass along an edge $(i, j) \in E$ where

$i \in S$ and $j \in T$. However, there are only so many directed edges of this form, and each such edge (i, j) has an upper bound u_{ij} on the total flow that can use it. If we let $D(S, T)$ denote the set of directed edges (i, j) for which $i \in S$ and $j \in T$, then the *capacity* of the cut (S, T) is equal to $\sum_{(i,j) \in D(S,T)} u_{ij}$. Since each unit of flow from s to t "uses up" one unit of the capacity of the cut (S, T), the value of the maximum flow is at most the capacity of the cut. This claim is true for **any** cut (S, T). For the same input, some cuts may have large capacity, and some cuts may have small capacity. However, for each cut, its capacity places a limit on the value of the maximum flow. Hence, if we found a cut of minimum capacity, this would give us the best information about how big the maximum flow value might be.

This type of argument should be reminiscent of ideas that we used in the study of general linear programs. For any primal maximization linear program, there was a corresponding dual linear program. The objective function of any feasible solution to the dual gave an upper bound on the optimal value of the primal. By finding the feasible dual solution of minimum objective function value, we obtain the best such upper bound.

Even more importantly, there is the following verification property that is built into the notion of a feasible flow and a cut:

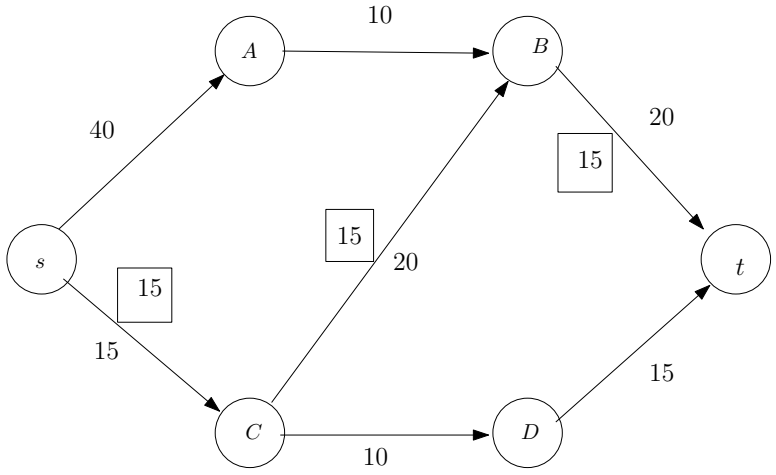
- If x^* is a feasible flow and (S, T) is a cut such that the value of x^* is equal to the capacity of (S, T), then x^* is a maximum flow.

Why is this? Let μ denote the value of the feasible flow x^* (and hence μ is also the capacity of the cut (S, T).) Suppose there were another feasible flow $\bar{f}$ of value greater than μ; say that its value is $\bar{\mu}$. But we know that the value of $\bar{f}$ is at most the capacity of the cut (S, T), which is μ. But this is a contradiction: $\bar{\mu}$ cannot both be bigger than μ and at most μ. Hence, there cannot be such a feasible flow, and x^* is a maximum flow. Again, this is exactly the same argument used for general linear programs: we can verify the optimality of a feasible solution for the primal LP by exhibiting a feasible solution to the dual of equal objective function value.

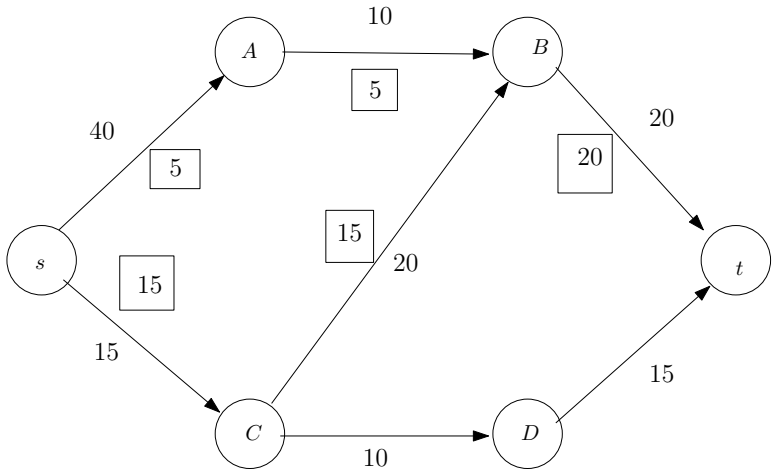
We will give an algorithm that generates a flow and cut simultaneously, such that the value of the flow is equal to the capacity of the cut; hence, the algorithm not only computes the maximum flow, but also computes something that verifies the optimality of this flow. Yet again, this should evoke the development of the simplex method and the strong duality theorem of linear programming: the simplex method produced both a feasible primal

solution and a feasible dual solution with equal objective function values, and hence these solutions were both optimal for their respective problems.

But now consider a closely related question for the maximum flow problem. Given a feasible flow, how can you tell (quickly) if the flow is optimal? To answer this question, first consider the following example (which shows a feasible solution of value 15 for the input we had considered previously, and hence we know in advance that this is not the optimal solution). In the figure below, the number next to each directed edge represents its capacity. The boxed number next to each arc specifies a flow value for that arc (if there is no box, then the flow value is 0).



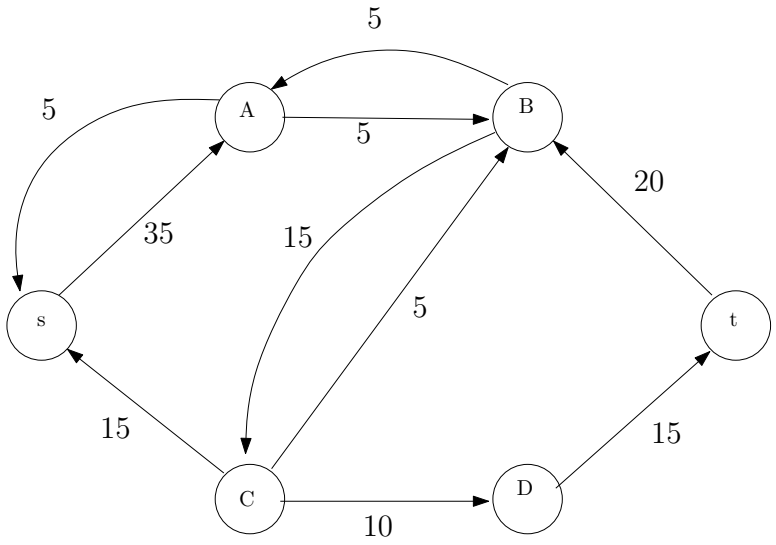
"Is the boxed flow optimal?" No. We can find a path from s to t, (s, A), (A, B), (B, t), such that each directed edge in this path is not used to its capacity, i.e., each has some *residual* capacity. In this example, the directed edges of the path have residual capacity equal to 40, 10,and 5, respectively. As a result, flow along this path can be increased by 5 units (that is, the minimum residual capacity on the path). But now we wish to know whether the new flow is optimal?



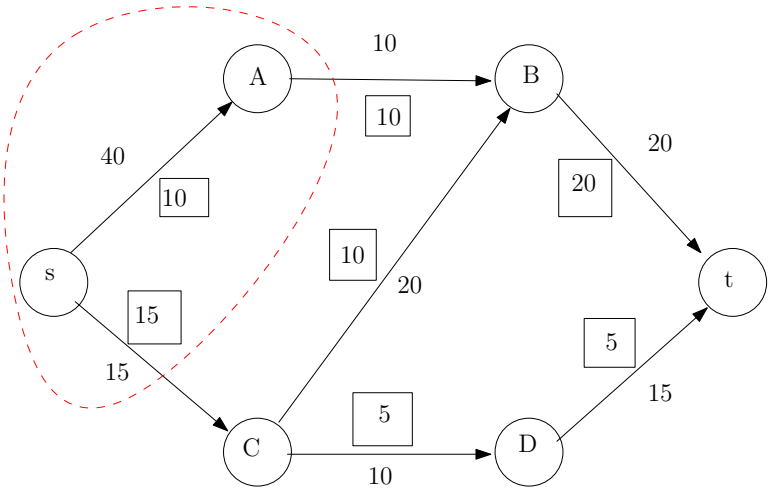
At first glance, one might be tempted to conclude that this solution is optimal. However, focus on the directed edge (C, B). A little reflection indicates that we have put too large of a flow across this edge, and that we can improve the overall solution by decreasing the flow on it. Such a decrease can be viewed as sending flow in the reverse direction, that is *from B to C*, like a "return to sender" action for unwanted mail. Thus, for each directed edge (i, j) for which the current flow value is less than its capacity, there is residual capacity on edge (i, j); furthermore, for each directed edge (i, j) for which there is positive flow, we can capture the effect of decreasing flow on this edge by introducing a directed edge (j, i) for which its residual capacity is equal to the current flow value on (i, j).

More generally, for any feasible flow x, we define the residual graph $G_x = (N, E_x)$, where the edge set E_x is defined as follows: for each directed edge $(i, j) \in E$ such that $x_{ij} < u_{ij}$ put (i, j) in E_x with residual capacity equal to $u_{ij} - x_{ij}$ (indicating how much the flow on (i, j) can be increased); and for each $(i, j) \in E$ with $x_{ij} > 0$ put $(j, i) \in E_x$ (not (i, j)!) with residual capacity equal to x_{ij} (indicating how much can flow on (i, j) can be decreased). The former set of edges are called *forward arcs* since they correspond to moving the flow forward on real arcs of the input; on the other hand, the latter set of edges are called *backward arcs* since flow moving on one of these corresponds to flow moving backwards on an arc of the input.

The residual graph for the previous feasible solution is the graph depicted below, where the value next to each edge reflects its residual capacity.



Consider the sequence of directed edges in this graph: (s, A), (A, B), (B, C), (C, D), (D, t). Each of these edges has residual capacity at least 5. Furthermore, we can use this path to indicate a set of changes to the previous feasible solution. For each forward arc (i, j) in the path, we increase the current solution by 5 (for example, for edge (s, A); for each backward arc (j, i), we decrease the flow on (the true) edge $(i, j) \in E$ by 5 (for example, for edge (C, B) we decrease the flow by 5). It is straightforward to check that this leads to a new feasible solution: we maintain the flow conservation constraints, as well as the capacity constraints. The resulting feasible solution is given below (again, in the boxed figures); this is the feasible solution that was introduced in the previous section.



Is this an optimal flow? We have also indicated a cut (S, T), where $S = \{s, A\}$ and $T = \{B, C, D, t\}$. The capacity of (S, T) is 25, and since the current flow has value equal to 25, the current flow is optimal. The current flow x also has other useful properties.

For each arc $(i, j) \in D(S, T)$, we have that

$$x_{ij} = u_{ij};$$

furthermore, there does not exist any directed edge $(j, i) \in E$ with $j \in T$ and $i \in S$ such that

$$x_{ji} > 0.$$

This will always be true at optimality!

We will now outline an algorithm that essentially just follows the basic ideas that we have just implemented on this example. This algorithm will take a given feasible flow x, and either finds a new feasible flow of greater value, or else proves that the current flow is optimal. The key element of this algorithm is the notion of a *residual graph*, which is constructed anew for each current feasible flow x.

If there is a path from s to t in G_x, then x is not optimal; we will indicate below how this path can be used to generate a better feasible flow.

If there is no path from s to t in G_x, let S be the set of nodes in N for which there is a path from s (and T corresponds to the rest of the nodes, $N - S$). Then we claim that (S, T) is a cut of capacity equal to the value of

current flow x. We next indicate why this is true. First, we know that for each arc $(i, j) \in D(S, T)$, it must be the case that $x_{ij} = u_{ij}$. The reason for this easy; if x_{ij} were less than u_{ij}, then (i, j) would be an arc in the residual graph; if $i \in S$, then there is a path from s to i in G_x, and so there must be a path from s to j in G_x too, and hence $j \in S$. But this is a contradiction, and hence $x_{ij} = u_{ij}$. Similarly, it must be the case that for each arc (j, i) such that $j \in T$ and $i \in S$, $x_{ji} = 0$. Otherwise, there would be a backwards arc (i, j) in G_x, and since $i \in S$, then j must be in S too, which is a contradiction. The capacity of the cut is therefore equal to the total flow from nodes in S to nodes in T, minus the total flow from nodes in T to nodes in S. The only place that flow "can end up" is the sink, and hence we can argue that the latter quantity is just the value of this flow x. The capacity of the cut (S, T) is equal to the value of the flow x.

Now we can give the steps of the algorithm that either improves a given feasible flow x, or else proves that it is optimal.

(1) Build the residual graph G_x.

(2) Compute the set of nodes S that are reachable from s by a path in G_x.

(3) If $t \in S$ then we have found a path from s to t in G_x; compute the minimum of the residual capacity of arcs in this path; call this value δ.

   (a) For each forward arc (i, j) in path, increase flow on (i, j) by δ.

   (b) For each backward arc (j, i) in path, decrease flow on (i, j) by δ.

If $t \notin S$, then $(S, N - S)$ is a cut that proves the optimality of f.

There are two remaining questions.

(1) How is step (2) done?

   We will provide a so-called labeling algorithm, that computes the set of nodes in a graph that are reachable from a specified source node s. (This is a standard procedure that is generally known as *breadth-first search.*) In the course of its execution, the algorithm marks certain nodes with a $\surd$. Furthermore, the algorithm maintains a list of checked nodes that it still needs to process.

   Labeling algorithm - Mark the source s with a $\surd$. Let the list initially contain just node s.

Until the list is empty:

> Take any node (e.g., the first one) off of the list. Suppose that i is the name of this node. For each arc leaving i, that is, each $(i, j) \in E_x$, if j is unchecked, check it and add it to the list, highlighting the edge (i, j).

The nodes that have been checked are those nodes that are reachable from s by a path in G_x. Furthermore, by following the highlighted edges, one can trace backwards from each reachable node to the source, thereby identifying the corresponding path.

This can be easily implemented. Try to see what happens when you run this algorithm on the example given above.

(2) How do we compute an optimal flow from scratch?

Up until now, we have focused on the problem, given a feasible flow x, decide if it is optimal, and if not, find a better feasible flow. But now, how do we find an optimal flow without any given feasible flow? There is an easy fix for this; the flow in which each arc has flow value equal to 0 is feasible, and so we can always start with this one. Then we can apply the algorithm given above until we have a feasible flow that can not be improved, and this is a maximum flow. This algorithm was discovered by Ford & Fulkerson, and is usually called the Ford-Fulkerson algorithm.

The Ford-Fulkerson algorithm computes both the maximum flow **and** a cut of minimum capacity. While the application that we started with was formulated as a maximum flow problem, there are also many applications in which the primary aim is computing a so-called minimum cut instead. The fact that the capacity of the minimum cut is always equal to the value of the maximum flow is called the *max-flow min-cut theorem*.

There is one final observation that should be made about the Ford-Fulkerson algorithm. Suppose that the input has capacities that are all integer; i.e., u_{ij} is an integer, for each $(i, j) \in E$. Then it is easy to see that the algorithm always produces an optimal solution in which each flow value x_{ij} is also integer. Hence, we have shown that maximum flow problem has the *integrality property* as claimed in the previous section.

The following theorem summarizes the main result of this section.

Theorem 3.1 (Max-flow-min-cut theorem) *For any input to the maximum flow problem, the value of the maximum flow is equal to the total capacity of the minimum cut, and the "augmenting path" algorithm of Ford and Fulkerson finds both. Furthermore, if the input capacities are all integers, then algorithm finds an optimal flow in which each arc is assigned an integer flow value.*

4 The project selection problem

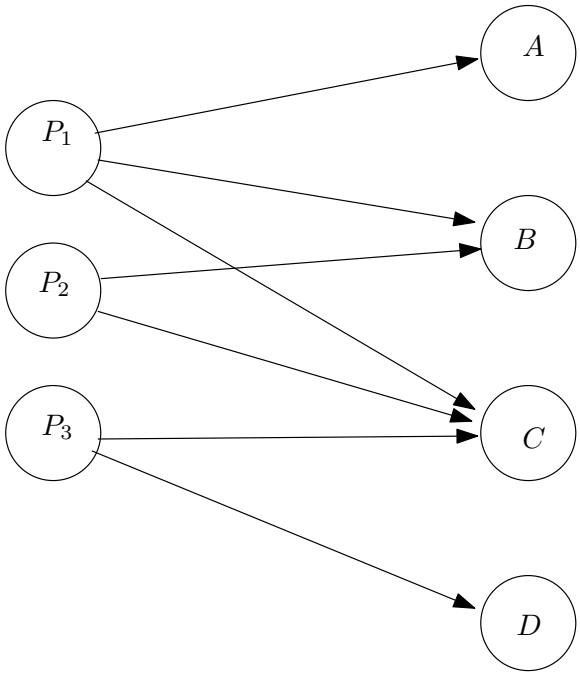
One of the surprising aspects of the maximum-flow minimum-cut theorem is that while we started thinking about solving one optimization problem, we ended up solving two problems for the price of one. We now also have the possibility of, given a directed graph $G = (N, E)$, with a specified source node s and sink node t, where each directed edge has an associated capacity, to find the cut of minimum capacity. We next give a new optimization problem that can be formulated as a minimum-cut problem.

The project selection problem is as follows. There is a collection of projects that one might take on: $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$. Each project P_i has an associated benefit b_i, $i = 1, \ldots, m$. There is also a collection of tools $\mathcal{T} = \{1, \ldots, n\}$ that are needed for doing these projects. Each tool j has an associated cost c_j, $j = 1, \ldots, n$. (One might imagine that the projects are clients that a management consulting firm might take on, and the tools are software licenses that are needed for some of these projects.) For each project P_i, there is a subset of tools $T_i \subseteq \mathcal{T}$ that are required to do that project; once the tool is purchased, it may be used for as many different projects as required. The aim is to select a subset of the project, and purchase the requisite tools so as to maximize the net profit associated with these activities (where the net profit is the total benefit accrued minus the total cost incurred).

We will show how to formulate the project selection problem as a minimum cut problem. A priori, this seems extremely surprising since there is no graph as part of the description of the problem, no apparent cuts, and even we trying to maximize something on the one hand, whereas the min-cut problem is a minimization problem. We will do this formulation by the same 3-step process we have repeated for other problems. First, we will show how to map a project selection input into an input for the minimum cut problem. Then we will demonstrate that feasible solutions to the project selection problem have a precise correspondence to feasible solutions for the resulting minimum cut problem. Finally, we will show that optimizing the objective of the minimum cut problem exactly corresponds to finding the best selection of projects. An example of an input to the project selection problem is as follows. Suppose that the set of tools is $\{A, B, C, D\}$, and suppose that the set of projects is $\{P_1, P_2, P_3\}$. The requirements for the projects are as follows: the first requires tools A, B, and C; the second requires B and C; and the last requires C and D. The tool costs are, respectively, 5, 10, 5, and

15; the project benefits are, respectively, 15, 20, and 10.

At first, it seems that there is no graph structure here at all. However, it is extremely natural to represent the requirements as a graph, in a way reminiscent of what we did for the assignment problem earlier. Suppose that we construct a graph with two sets of nodes: one set corresponding to the projects, and one set corresponding to the tools. We can draw the first set on the left, and the second on the right, and include an edge from a particular project node to a tool node whenever that tool is needed for that project. So, in the example that we gave above, we might build the following graph, which we call the *tool incidence graph*.



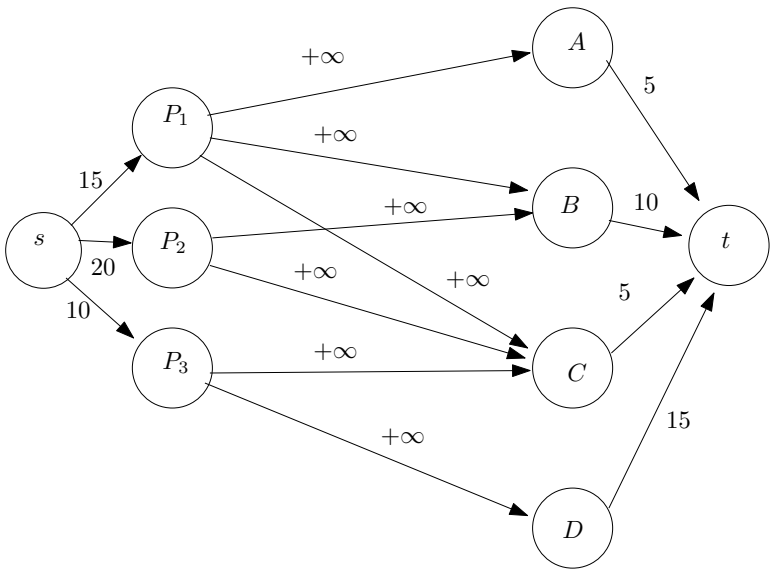
What do the constraints tell us? If we select a project node P_i, then we must also select all of the tools in T_i, that is, all tools required for this project. What might this have to do with a minimum cut problem? We can view the min-cut problem as "selecting" those nodes that will be on the source side of the cut. Thus, we might want a way to enforce the constraint that if we select a project node P_i, then for each node j in T_i, we must also select j. If we give the edge (P_i, j) an infinite capacity, then (assuming that we

need only concern ourselves with cuts of finite capacity) we have enforced the condition that if node P_i is on the source side, then node j is on the source side of any (reasonable) cut. Furthermore, for each project P_i not selected, one loses the opportunity to gain its benefit b_i, and for each tool j selected, one pays the cost c_j of using it. We now follow our 3-step plan to show that we can formulate the project selection problem as a minimum cut problem, relying on the previous discussion for motivation.

Step 1 For a project selection input, build an input to the minimum cut problem by starting with the tool incidence graph, and then adding a source node s and a sink node t, as well as the edge (s, P_i) for each project P_i, and the edge (j, t) for each tool j.

- For each edge (P_i, j) in the tool incidence graph, we give the corresponding edge in our min-cut input graph the capacity $+\infty$.
- For each edge (s, P_i) in our input graph, we assign it capacity b_i.
- For each edge (j, t) in our input graph, we assign it capacity c_j.

This completes the construction of the input to the maximum flow problem. Let $G = (N, E)$ denote the input graph that we have defined. The figure below shows the graph constructed for the sample input to the project selection problem given previously.



Step 2 We want to argue that the feasible solutions to the project selection problem correspond to cuts in our input to the minimum cut problem. This needs a bit of care. We will show that there is a one-to-one correspondence between feasible solutions to the project selection problem and feasible solutions *of finite capacity* to our minimum-cut input.

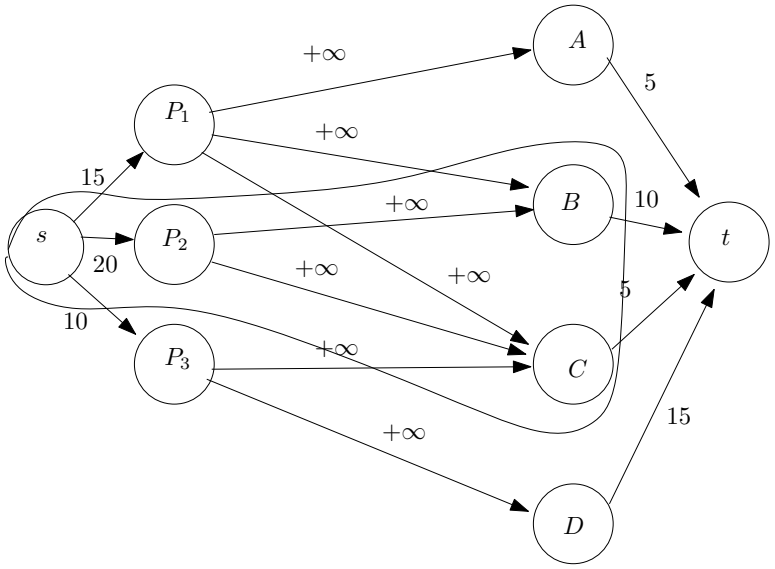
Consider any feasible solution to the project selection problem. That is, we select a subset $P' \subset \mathcal{P}$ of projects and a set of tools T' such that $T' \supseteq \cup_{i:P_i \in P'} T_i$ (that is, T' contains all of the needed tools for these projects). Then we shall think of the cut $(\{s\} \cup P' \cup T', N - \{s\} - P' - T')$ as the corresponding cut. So, for example, if we decide not to do any projects, and buy no tools, then this corresponds to the cut $(\{s\}, \mathcal{P} \cup \mathcal{T} \cup \{t\})$; observe that this cut always has finite capacity (equal to the total benefit of all of the projects, and so the optimal value is going to be finite).

We first prove that the cut $(\{s\} \cup P' \cup T', N - \{s\} - P' - T')$ has finite capacity: suppose that it doesn't (and we will reach a contradiction); then an edge of the form (P_i, j) must cross the cut, but the only such edges of that form that exist are for tools j required by the project P_i; if $P_i \in P'$, then $j \in T'$, and so the edge does not cross the cut, which is a contradiction.

Now consider any cut $(S, N - S)$ for our min-cut input of finite capacity. Let P' be $S \cap \mathcal{P}$ and let T' be $S \cap \mathcal{T}$. Suppose that we select the projects in P' and the tools in T'. Is this a feasible tool selection; that is, do we have all of the tools needed to do the projects? Suppose not: then there is a project $P_i \in P'$ that requires a tool j that is not in T'. But then the edge (P_i, j) crosses the cut $(S, N - S)$ and since it has infinite capacity, so does that cut, which is a contradiction.

We have just argued that we can list all feasible project selections, and all finite capacity cuts, in such a way that they exactly pair up in a so-called 1-1 correspondence. This completes the second required step, and shows that if we consider all of the finite capacity cuts, that is the same thing as considering all of the feasible project selections).

The figure below shows the cut corresponding to the selection of project P_2 and its required tools B and C for the example above: note that the edges crossing the cut are (B, t) and (C, t), whose capacities of 10 and 5, respectively correspond to the cost of buying the tools B and C, as well edges (s, P_1) and (s, P_3), whose capacities of 15 and 10, respectively, correspond the "opportunitiy cost" of not taking on projects P_1 and P_3. This suggests that we are capturing the right objective in solving this minimum cut problem, but now we must make this more rigorous.



Step 3 By showing the correspondence of the feasible solutions between the two inputs, we know that optimizing over the feasible solutions to one problem is a rephrasing of optimizing over the feasible solutions for the other. But does the "best" feasible solution for one correspond to the "best" feasible solution for the other? For that, we must compare what happens with the objective function values of the corresponding solutions that we have paired up. Suppose that we consider the feasible solution to the project selection problem with projects P' and tools T'; it has objective function value

$$\sum_{i:P_i \in P'} b_i - \sum_{j \in T'} c_j.$$

Consider the corresponding cut $(\{s\} \cup P' \cup T', N - \{s\} - P' - T')$. What is its capacity? The only edges that cross the cut are the edges (s, P_i) for each $P_i \notin P'$ and the edges (j, t) for each $j \in T'$. So the capacity of the cut is equal to

$$\sum_{i:P_i \notin P'} b_i + \sum_{j \in T'} c_j.$$

Let $B = \sum_{i:P_i \in \mathcal{P}} b_i$; that is, B is the total benefit available from all projects. We can view the capacity just computed as

$$B - \sum_{i:P_i \in P'} b_i + \sum_{j \in T'} c_j.$$

But choosing the cut of finite capacity for which that is smallest, is the same as choosing the finite capacity cut for which

$$-[B - \sum_{i:P_i \in P'} b_i + \sum_{j \in T'} c_j] = \sum_{i:P_i \in P'} b_i - \sum_{j \in T'} c_j - B$$

is largest. But this is equivalent to our project selection objective, since it merely subtracts B from the value of each feasible solution. So maximizing the net profit of any feasible project selection is equivalent to finding the minimum capacity cut in our input graph.

We have thereby shown that the formulation is correct; we can find the optimal solution for any input to the project selection problem by setting up and solving the corresponding (maximum-flow) minimum-cut input. Suppose that you wanted to solve such an input (as the one above) by hand, using the Ford-Fulkerson algorithm. That algorithm did not explicitly discuss the option of having infinite capacity arcs. In fact, these do not pose any problems, once one realizes that is simply means that the forward residual capacity of such an arc remains infinite in spite any positive flow already pushed across it. Alternatively, one can simply substitute a sufficiently large number for infinity. The role of infinity in the proof above is simply that any cut that has just one infinite capacity arc crossing it must be worse than any "finite" capacity cut. So, if we define $W = \sum_{i:P_i \in \mathcal{P}} b_i + \sum_{j \in \mathcal{T}} c_j + 1$, then we can use W in place of $+\infty$ for the capacity of each arc (P_i, j) in G. (In fact, since the "trivial" cut in which the source is the only node on the source side has capacity B, it suffices to set $W = B + 1$; we know that no "infinite capacity" edge could be used in the optimal cut.)

5 The Assignment Problem

In the *assignment problem*, there is a set of workers to be assigned to a set of tasks. There are the same number of workers as tasks. Let n denote the number of workers (and the number of tasks). We will think of the n tasks as task 1, task 2, through task n. We will think of the n workers as worker 1, worker 2, through worker n. When we presented this optimization problem earlier, we indicated each worker is capable of performing only a subset of tasks. However, it will be simpler to assume that for *each* worker j, and for *each* task i we are given a specified time t_{ij} (which might be infinite,

or equivalently, a VERY large number) that it takes for that task to be accomplished by that worker. Thus the data for this problem is the array:

$$\begin{array}{|cccccc|}\hline t_{11} & t_{12} & \dots & t_{1j} & \dots & t_{1n} \\ t_{21} & t_{22} & \dots & t_{2j} & \dots & t_{2n} \\ & & & \vdots & & \\ t_{i1} & t_{i2} & \dots & t_{ij} & \dots & t_{in} \\ & & & \vdots & & \\ t_{n1} & t_{n2} & \dots & t_{nj} & \dots & t_{nn} \\ \hline \end{array},$$

where the entries in the ith row are times for the ith task, the entries in the jth column are times for the jth worker, and the entry t_{ij} is thus the amount of time required for the ith task to be done by the jth worker.

An assignment specifies which worker is assigned to which task, where each task is assigned to exactly one worker, and each worker is assigned exactly one task. The assignment problem is to find an assignment in which the total time spent on the assigned tasks is as small as possible.

We can formulate the assignment problem as an integer programming problem. We use the variable x_{ij} to signify whether task i is assigned to worker j, for any $i = 1, \dots, n$, $j = 1, \dots, n$. Each of these n^2 variables will take the value 0 or 1. If $x_{ij} = 1$, then we interpret this to mean that task i is assigned to worker j. If $x_{ij} = 0$, then we interpret this to mean that task i is not assigned to worker j. Bearing this in mind, we can write constraints that exactly capture whether the values of

$$x_{11}, x_{12}, \dots, x_{1n}, x_{21}, x_{22}, \dots, x_{2n}, \dots, x_{n1}, x_{n2}, \dots, x_{nn}$$

correspond to a feasible assignment. First, we must constrain each x_{ij} to be either 0 or 1:

$$0 \leq x_{ij} \leq 1, \text{ integer, for each } i = 1, \dots, n, \ j = 1, \dots, n. \tag{5.1}$$

Second, we must constrain that each task is assigned to exactly one worker:

$$\sum_{j=1}^{n} x_{ij} = 1, \text{ for each } i = 1, \dots, n. \tag{5.2}$$

Finally, we must constrain that each worker is assigned to exactly one task:

$$\sum_{i=1}^{n} x_{ij} = 1, \text{ for each } j = 1, \dots, n. \tag{5.3}$$

In other words, any way to set the variables x_{ij}, for all $i = 1, \ldots, n$, $j = 1, \ldots, n$, corresponds to an assignment exactly when it satisfies the conditions (5.1), (5.2), and (5.3). The total time required for an assignment is

$$\sum_{i=1}^{n} \sum_{j=1}^{n} t_{ij} x_{ij}, \tag{5.4}$$

and so our objective is to find an assignment x that minimizes (5.4).

For the time being, assume that each $t_{ij} \geq 0$. (We will see that this assumption is unimportant.) In this case, how can we be convinced that a particular assignment is an optimal one? Here is one special case in which it is easy to see that we have an optimal solution. Suppose that we have an assignment of value 0. Since each t_{ij} is non-negative, the value of any assignment is non-negative. Hence, an assignment of value 0 must be optimal. Notice, of course, that if the value of the assignment x is 0, then if x calls for task i to be assigned to worker j, then $t_{ij} = 0$. In other words, each entry used for the assignment is 0; we shall call such an assignment an *all-0 assignment*. Of course, this is a very special case, but we shall see that it is not so special after all.

Suppose that we changed each entry in row 1 of the input by subtracting 17. How would the problem change? For any feasible assignment, exactly one entry from row 1 is included, and so the value of any feasible assignment would go down by 17. However, since the value of each assignment goes down by the same amount, the best assignment before the change is still the best assignment after the change. We have derived an equivalent problem. Of course, the same thing is true if we subtracted the same value from each entry in a particular column. Consider the following example:

$$\begin{array}{|ccc|} \hline 6 & 4^* & 7 \\ 2 & 5 & 1^* \\ 3^* & 4 & 2 \\ \hline \end{array}.$$

We will show that the assignment indicated by the *'s is optimal. Subtract 4 from each entry in row 1:

$$\begin{array}{|ccc|} \hline 2 & 0 & 3 \\ 2 & 5 & 1 \\ 3 & 4 & 2 \\ \hline \end{array}.$$

and then 1 from each entry in row 2:

$$\begin{array}{|ccc|}\hline 2 & 0 & 3 \\ 1 & 4 & 0 \\ 3 & 4 & 2 \\ \hline\end{array}.$$

and then 2 from each entry in row 3:

$$\begin{array}{|ccc|}\hline 2 & 0 & 3 \\ 1 & 4 & 0 \\ 1 & 2 & 0 \\ \hline\end{array}.$$

If we now subtract 1 from each entry in column 1, we get

$$\begin{array}{|ccc|}\hline 1 & 0^* & 3 \\ 0 & 4 & 0^* \\ 0^* & 2 & 0 \\ \hline\end{array}.$$

where we see that the assignment indicated above has each starred entry equal to 0, i.e., it is an all-0 assignment. Also, we have done this transformation in a way that maintained that the *adjusted entries* are all non-negative. (This is crucial so that we can still prove that any all-0 assignment is optimal.) Since we have found the best assignment for the last array, we have also found the best assignment for the original data.

Unfortunately, life is not always so simple. Consider the following input, and try the same thing as before. Subtract as much as possible from each row (while maintaining that the new values are non-negative) and then as much as possible from each column.

$$\begin{array}{|ccc|}\hline 8 & 10 & 7 \\ 10 & 11 & 8 \\ 5 & 6 & 7 \\ \hline\end{array} \quad \begin{array}{c}\longrightarrow \\ \text{subtract 7} \\ \text{from row 1}\end{array} \quad \begin{array}{|ccc|}\hline 1 & 3 & 0 \\ 10 & 11 & 8 \\ 5 & 6 & 7 \\ \hline\end{array} \quad \begin{array}{c}\longrightarrow \\ \text{subtract 8} \\ \text{from row 2}\end{array} \quad \begin{array}{|ccc|}\hline 1 & 3 & 0 \\ 2 & 3 & 0 \\ 5 & 6 & 7 \\ \hline\end{array}$$

$$\longrightarrow$$
subtract 5
from row 3

$$\begin{array}{|ccc|}\hline 1 & 3 & 0 \\ 2 & 3 & 0 \\ 0 & 1 & 2 \\ \hline\end{array} \quad \begin{array}{c}\longrightarrow \\ \text{subtract 1} \\ \text{from column 2}\end{array} \quad \begin{array}{|ccc|}\hline 1 & 2 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 2 \\ \hline\end{array}$$

Now we have an input where we cannot subtract anything from any row or column, and yet there is no all-0 assignment (you can check this by trying all 3! assignments).

What do we do now? Now we turn our attention to something called a 0-cover. A *line* is a column or a row. Any line *covers* all of the elements contained in it. For example, in the array below, the first row is a line that covers the elements 1, 2, and 0. A *0-cover* is a collection of lines such that every 0 is in some line; that is, each 0 is covered by a line that has been selected. So, for the example below, we can form a 0-cover consisting of 2 lines, the 3rd row and the 3rd column:

$$\begin{array}{cc} & \Downarrow \\ & \boxed{\begin{array}{ccc} 1 & 2 & 0 \\ 2 & 2 & 0 \\ 0 & 0 & 2 \end{array}} \\ \Rightarrow & \end{array}$$

It turns out that whenever there is no all-0 assignment, there always is a 0-cover that uses fewer than n lines. (Remember that n is the number of rows or columns, and so there always is a 0-cover with n lines, e.g., take all n rows.) Here is how we can use a 0-cover to update our problem in a more sophisticated way. Let k be the value of the smallest entry not covered by a line. (In our example, it is the entry in the first row, first column, which has value 1, and so $\delta = 1$.) First subtract δ from each and every entry; then, for each row in the 0-cover, add δ to each entry in that row; finally, for each column in the 0-cover, add δ to each entry in that column. (Observe that this decreases by δ each entry not covered by a line, does not change each entry covered by exactly one line, and increases by δ each entry covered by two lines; hence, no entry becomes negative by making these changes.) For our example, we subtract 1 from each entry in rows 1 and 2, and add 1 to each entry in column 3, thereby yielding

$$\boxed{\begin{array}{ccc} 0^* & 1 & 0 \\ 1 & 1 & 0^* \\ 0 & 0^* & 3 \end{array}};$$

an all-0 assignment (which is optimal) is indicated by the stars. Once again, this is also an optimal assignment for the original input, since we have only changed the data by repeatedly subtracting (or adding) the same value to each entry in a particular row or column.

We now have a complete algorithm, which is usually called the *Hungarian algorithm*. In each iteration, we look where the 0 entries are in the current equivalent version of our input. Either we can find an assignment that uses only the 0 entries (and is clearly optimal) or we can find a 0-cover with fewer than n lines. In the latter case, we find the smallest uncovered entry and make the changes as indicated above.

One thing should not be obvious: that the algorithm does finish, and in fact finishes quickly. One reason for this is that in each iteration the cost of each feasible assignment goes down by a positive amount. Suppose that there are $\ell < n$ lines in the cover. We first decrease the cost of any feasible assignment by δn (since there are n entries in each assignment decreased by δ), and then we increase the cost by $\delta\ell$. Hence, the total change is $\delta\ell - \delta n = \delta(\ell - n)$, which is less than 0. Since we stop when there is a 0-cost assignment, we cannot continue to decrease indefinitely, and hence the algorithm must terminate. Eventually, there will be an all-0 assignment, and at that point, the algorithm will stop.

One step that we omitted is to prove the claim that whenever there is no all-0 assignment, there always is a 0-cover that uses fewer than n lines. This can be done by setting up a maximum flow input with a source, a sink, and nodes corresponding to the rows and columns; there is an arc of capacity 1 from the source to each row node, an arc of capacity 1 from each column node to the sink, and an infinite capacity arc from the ith row node to the jth column node whenever the $(i, j)$the entry is equal to 0. It is easy to see that the maximum flow value for this input is equal to n if and only if there is an all-0 assignment.

Hence, if there is no all-0 assignment, there must be a cut in this input with capacity less than n. This cut is of finite capacity, and hence all of the arcs that cross the cut are of finite capacity: either from the source to a row node (on the sink side of the cut), or from a column node (on the source side of the cut) to the sink. Consider the set of lines formed by taking exactly those row nodes on the sink side of this cut, together with the column nodes on the source side of this cut. The number of lines selected is exactly equal to the capacity of this cut, and so is less than n. But is it a 0-cover? Suppose that it isn't. Any uncovered 0 must be in an unselected row (on the source side of the cut) and in an unselected column (on the sink side of the cut). But then there is an infinite arc starting at a node on the source side, and ending at a node on the sink side! This is clearly impossible, and hence we have selected a 0-cover. We have shown that whenever there is no all-0

assignment, there is a "small" 0-cover of size strictly less than n.

The Hungarian algorithm actually solves the problem of finding an optimal solution to the following linear programming problem:

$$\text{minimize } \sum_{i=1}^{n}\sum_{j=1}^{n} t_{ij}x_{ij}$$

subject to

$$\sum_{j=1}^{n} x_{ij} = 1, \text{ for each } i = 1, \ldots, n,$$

$$\sum_{i=1}^{n} x_{ij} = 1, \text{ for each } j = 1, \ldots, n,$$

and

$$0 \leq x_{ij}, \quad \text{for each } i = 1, \ldots, n, \ j = 1, \ldots, n.$$

In other words, we are solving the assignment problem without the constraint that the x_{ij} need to be integers. In other words, a task may be half assigned to one worker, a third assigned to another, and a final sixth assigned to yet another worker. Why does the Hungarian algorithm solve this problem, in which the solutions are not constrained to be integer? There are only three assumptions on which the correctness of this algorithm is based:

1. If all of the entries are non-negative, and there exists an all-0 assignment, then that all-0 assignment is optimal; even if fractional assignments are allowed, this claim remains true.

2. If u_i is subtracted from each entry in row i, then the value of each assignment decreases by u_i. This also remains true if we allow fractional assignments. Before making these changes, the contribution of row i to the objective function is $\sum_{j=1}^{n} t_{ij}x_{ij}$. If we subtract u_i from each entry in row i, then it becomes

$$\sum_{j=1}^{n}(t_{ij}-u_i)x_{ij} = \sum_{j=1}^{n} t_{ij}x_{ij} - \sum_{j=1}^{n} u_i x_{ij} = \sum_{j=1}^{n} t_{ij}x_{ij} - u_i \sum_{j=1}^{n} x_{ij} = \sum_{j=1}^{n} t_{ij}x_{ij} - u_i,$$

   where the last equation follows from the constraints (5.2). But this means that the value of each fractional assignment decreases by u_i.

3. If v_j is subtracted from each entry in column j, then the value of each assignment decreases by v_j. An argument nearly identical to the previous one shows that the value of each fractional assignment decreases by v_j in this case too.

Consequently, the Hungarian algorithm guarantees that we find an optimal solution x in which each x_{ij} is an integer, even when we let fractional solutions be feasible. This is an *integrality property* just like the one we saw for the maximum flow problem (and claimed for the minimum-cost network flow problem), and it too has many applications.

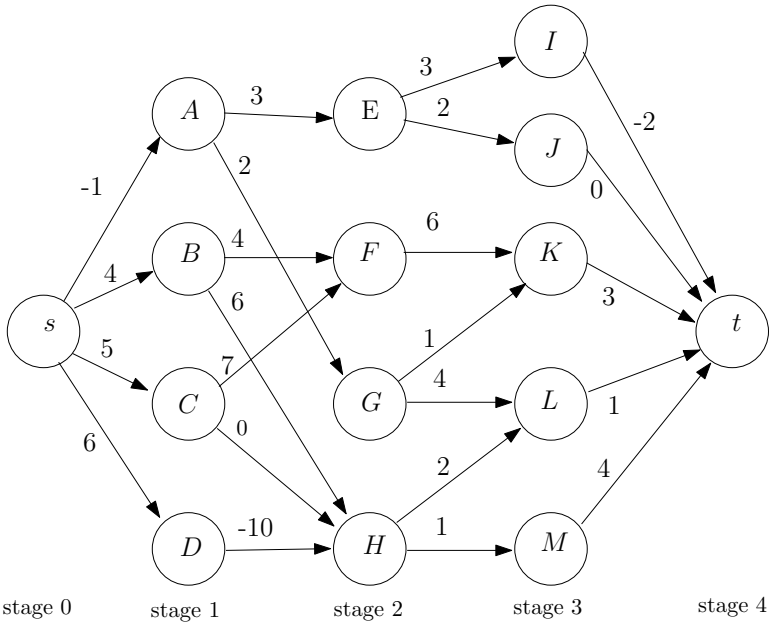
Finally, we started with the assumption that the original data for the assignment problem were all non-negative. Observe that this is completely unimportant: if $-\tau$ is the smallest entry in the original data, and we add τ to each entry, then we get an equivalent problem in which all entries are non-negative.

6 Dynamic Programming

The next main topic is a general algorithmic technique for solving optimization problems; this is in contrast to previous topics in this course and in the fall, where we have focused on a class of optimization models, such as the minimum-cost network flow problem, or linear programming.

6.1 Example 1: shortest paths in layered graphs

Suppose that we want to compute the shortest path from node s to node t in the graph below. Observe that this input graph has a special structure: we can partition the nodes into "layers", first s, then A, B, C, D, then E, F, G, H, then I, J, K, L, M, and a final layer with just node t. Each directed edge in the graph goes from a node in one layer, to a node in the next one. Such a graph is often called a *layered graph*. As indicated in the figure, each edge has an associated cost, and the aim is to select a path from s to t of minimum total cost.



Of course, we already know how to solve this problem, even if the costs are allowed to take on negative values. We can formulate this problem as a

minimum-cost flow problem: since there are no cycles in this graph at all, there are no cycles of total cost that is negative, even if every cost in the input is negative.

For every problem that we attack by dynamic programming, we will view the selection of an optimal solution as a process of making a series of decisions; each decision corresponds to a *stage* of the dynamic programming procedure. For our shortest path problem, it is clear that selecting a path corresponds to selecting, for each node encountered, which directed edge will be used to leave that node. By making a series of such decisions, we select a path from s to t. In the language of dynamic programming, we refer to the current situation after making some number (possibly 0) of decisions, as the current *state*. In this way, the nodes of each layer correspond to the possible states for each stage of the process. One central feature of dynamic programming is that we define a related optimization problem for each possible state encountered throughout the process. In this way, we are solving one optimization input that we care about by solving a large number of related problems along the way.

In this case, we shall define

$$f_i^*(r)$$

to be the minimum total cost remaining to go from state r in stage i to completion (at node t in stage 4). As an example of this, we see that $f_4^*(t) = 0$ and $f_3^*(M) = 4$. In this notation, the $*$ is used to denote that we are referring to the optimal value, the subscript i refers to the relevant stage of the process, and the value r refers to the highlighted state. Let $c(p, q)$ denote the cost of the edge from a generic node p to another node q.

For any state in stage i, we can evaluate all of our current choices by considering: (1) the cost of the next step, and (2) the optimal remaining cost of completing that path. That is, we have that

$$f_i^*(r) = \min_{\{\text{edges leaving } r \text{ to a state } p \text{ in stage } i+1\}} c(r, p) + f_{i+1}^*(p).$$

This is a recurrence relation: it expresses one optimal value in terms of other optimal values (that are hopefully easier to compute). The fact that we have this relation is often called the "principle of optimality". This allows us to work backwards in computing the one optimal value that we want to compute (from s to t). Initially, these are literally trivial to compute: we have already remarked that $f_4^*(t) = 0$. For the stage 3 nodes, we have that $f_3^*(M) = 4$, $f_3^*(L) = 1$, $f_3^*(K) = 3$, $f_3^*(J) = 0$, and $f_3^*(I) = -2$; for each of these, there is

a unique path to t consisting of the solitary (direct) edge. To compute the f^* values for the stage 2 nodes requires a bit more work.

$$\begin{aligned} f_2^*(E) &= \min\{3 + f_3^*(I), 2 + f_3^*(J)\} \\ &= \min\{3 + f_3^*(I), 2 + f_3^*(J)\} \\ &= \min\{3 + -2, 2 + 0\} \\ &= 1. \end{aligned}$$

$$\begin{aligned} f_2^*(F) &= \min\{6 + f_3^*(K)\} \\ &= \min\{6 + 3\} \\ &= 9. \end{aligned}$$

$$\begin{aligned} f_2^*(G) &= \min\{1 + f_3^*(K), 4 + f_3^*(L)\} \\ &= \min\{4, 5\} \\ &= 4 \end{aligned}$$

$$\begin{aligned} f_2^*(H) &= \min\{2 + f_3^*(L), 1 + f_3^*(M)\} \\ &= \min\{3, 5\} \\ &= 3 \end{aligned}$$

Continuing on with the stage 1 decisions, we see that

$$\begin{aligned} f_1^*(A) &= \min\{3 + f_2^*(E), 2 + f_2^*(G)\} \\ &= \min\{4, 6\} \\ &= 4. \end{aligned}$$

$$\begin{aligned} f_1^*(B) &= \min\{4 + f_2^*(F), 6 + f_2^*(H)\} \\ &= \min\{13, 9\} \\ &= 9. \end{aligned}$$

$$\begin{aligned} f_1^*(C) &= \min\{7 + f_2^*(F), 0 + f_2^*(H)\} \\ &= \min\{16, 3\} \\ &= 3 \end{aligned}$$

$$\begin{aligned} f_1^*(D) &= \min\{-10 + f_2^*(H)\} \\ &= -7 \end{aligned}$$

And now completing the process with the starting node s:

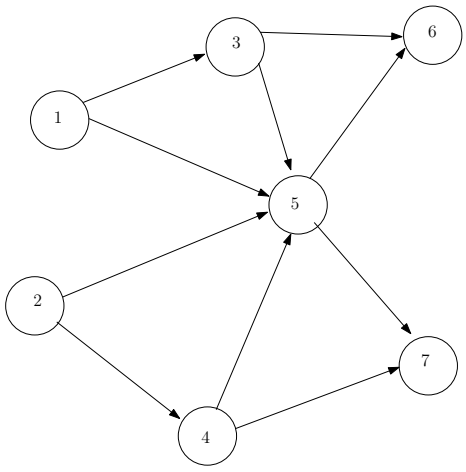
$$\begin{aligned} f_0^*(s) &= \min\{-1 + f_1^*(A), 4 + f_1^*(B), 5 + f_1^*(C), 6 + f_1^*(D)\} \\ &= \min\{3, 13, 8, -1\} \\ &= -1. \end{aligned}$$

This shows us that the length of the shortest path from s to t is equal to -1. Can we also compute the shortest path itself (and not just its total cost)? To do this we work forwards. At stage 0, what was the optimal decision? It was a choice corresponding to the value that gave us the minimum. The minimum value -1, was generated from $6 + f_1^*(D)$. In other words, it reflected the decision to go next to node D.

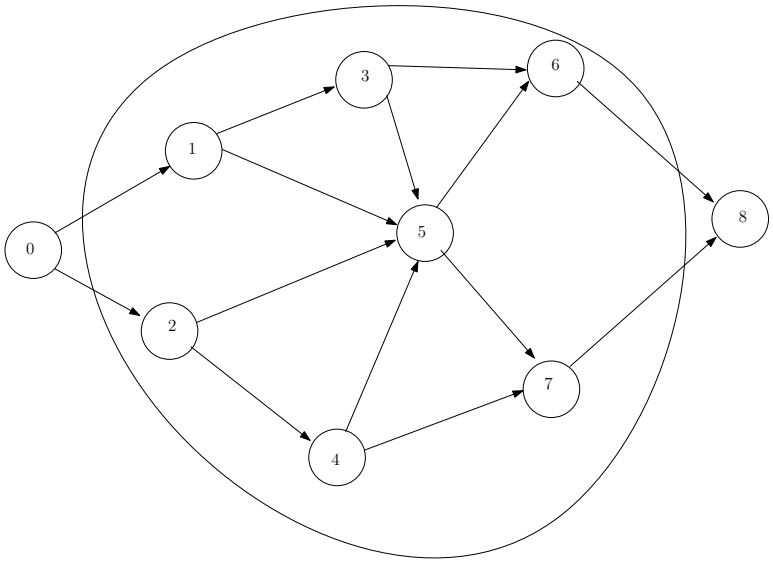
But now, how do we get from D to the node t? Its minimum value of -7 was generate by the recursion from $-10 + f_2^*(H)$, and so the next stop in the shortest path is node H. From H, we selected the minimum value of 3 from $2 + f_3^*(L)$, and so we next go to node L. Finally, from node L there is only the one decision possible to go to t. Thus, the shortest path from s to t consists of the directed edges $(s, D), (D, H), (H, L), (L, t)$, and has total cost equal to -1. This procedure can clearly be applied to any layered network to compute the shortest path between any pair of nodes.

6.2 Example 2: Critical path analysis

Suppose that you are trying to manage a project that has many components, handled by a variety of subcontractors, but is working towards completing one unified product. There may be a large collection of subtasks within this project, and for some of these subtasks there may be what we call precedence constraints: subtask i must be completed before subtask j may be started. We can model the project by a graph, such as the one below, where each node corresponds to a subtask, and each directed edge in the graph reflects a precedence constraint: for example, in the graph below subtask 4 must precede subtask 6. Each subtask has an associated time that will be required to complete it. Hence, for each node we have an associated (non-negative) value.



As part of managing this project, you want to understand what is the critical sequence of subtasks that is limiting the completion of the project. This corresponds to computing the *longest* path from some node in the graph to another; if we add "dummy" nodes reflecting the "start" and "end" of the project (as we have done below) we now merely need to compute the longest path between these two nodes; however, note that the length of the longest path is with respect to the node values – edges do not have any cost or length in this set-up.



This graph does not appear to have the layered structure that we exploited in the previous example. However, this graph is still *acyclic*; there are no directed cycles in this graph. Acyclic graphs always have a special ordering of their nodes known as a *topological ordering*: we can always number the nodes $0, 1, \ldots, n$ such that each arc in the graph (i, j) has $i < j$. (Observe that the numbering of the nodes in our example already has this property.)

Why does any acyclic graph have a topological ordering? We show this by giving an algorithm to find a good ordering of the nodes. First, the fact that the graph is acyclic implies that there must be some node with no arcs leaving it (otherwise if we blindly follow a path taking any edge that leaves the current node, at some point we must exhaust all of the nodes and come to some node that we have already been at, contradicting the fact that the graph was acyclic). Choose such a node without outgoing arcs, and call it node n. Now delete that node and all arcs entering it (to yield an acyclic graph with one fewer node). Now we can repeat this basic step until all nodes have been numbered.

Why is this topological ordering of the nodes useful? Now we can repeat the approach used in the layered graph example, by having each node alone in a stage, and using the ordering to specify the stages. The only other difference is that now we are computing maxima, and not minima. All other features of our dynamic programming solution carry over to this new problem without any adjustment needed.

6.3 Example 3: Inventory planning

Dynamic programming is a central tool used in the area of inventory and production planning. We will consider a number of examples of this sort, starting with a very simple model, and gradually adding more realistic features.

Suppose that you are trying to plan production to meet demand for some substantial product (such as an Airbus A380). You have been given the demand for the next N years. Furthermore, for each of these years, you are given the per-unit cost (in millions of dollars) to produce these planes. However, some years you might not produce any planes, and in other years you will produce planes; for each year in which you produce a positive number of planes, you incur a set-up cost. Furthermore, in each year there is a production capacity that limits the number of planes that can be produced in that year. Furthermore, there is a limit of the on-hand inventory of planes at the

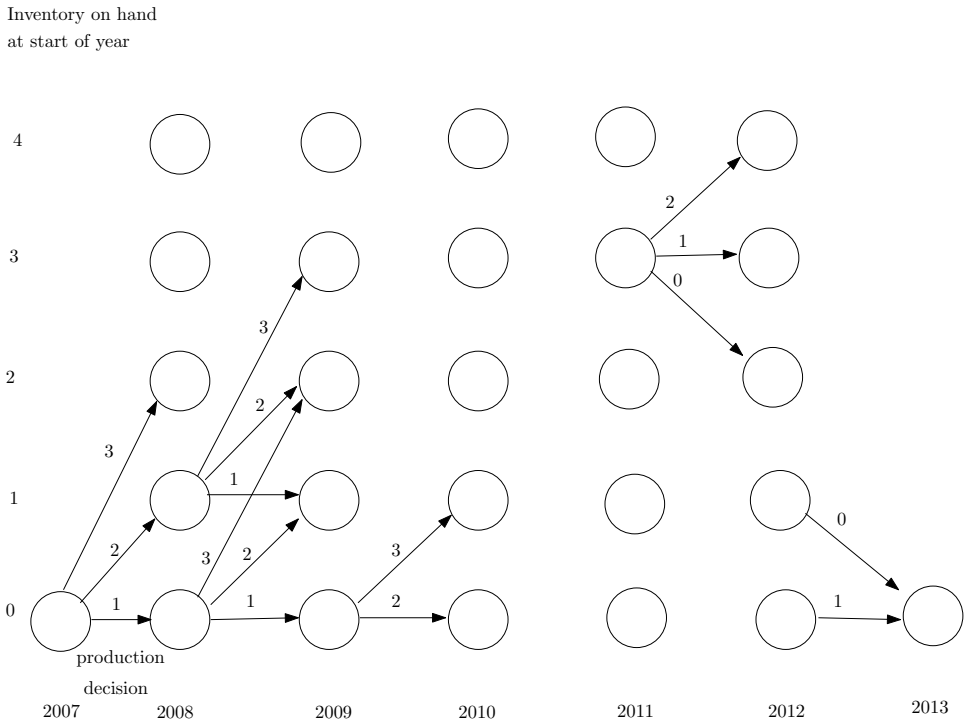
end of a year that are completed but not yet required by customers. Planes produced in a given year i can meet the same year-end demand requirement of thE customers. The goal is to determine a production schedule for the next N years so as to minimize the total cost incurred.

A typical input for this problem might be as follows: the planning horizon consists of 6 years from 2007 to 2012; for each of these years, the setup production cost for producing > 0 planes is \$15M; at most 3 planes can be produced in each year, and at most 4 planes may be held in inventory at the end of each year. The demands and unit-production costs are given in the table below.

Year	2007	2008	2009	2010	2011	2012
Demand	1	1	2	2	1	1
Unit-Production Cost	54	56	58	57	50	48

If we think about what constitutes a feasible solution to this problem, then we must specify the number of units produced in each year, and this specification must meet certain requirements: first we never produce more than the capacity in each year; second, we must never have more than the inventory limit left on hand at the end of a year; finally, we must meet the demand for each year.

The first question in approaching an optimization problem using dynamic programming is to understand what constitutes the stages and states of the decision process that you are attempting to capture. What decision is made at each stage? In this case, one natural way to approach the problem is to have the stages correspond to the years of production and to have the decision reflect the number of planes produced in that year. But what makes sense for the "current state" of the system? In fact, it is also natural to let the state correspond to the size of the on-hand inventory at the start of the production year. In fact, we can view the entire process as being modeled by a layered graph, much like the one that appeared in the first dynamic programming example.



For each state in a given stage, there is a set of allowed actions, each of which corresponds to an arc in the layered graph. In the graph above (or at least the part that we have completed) the number next to each arc gives the number of units produced for the corresponding decision. For example, the arc from the 0 level node in 2007 to the 0 level node in 2008 is labeled with a 1, since there is one unit of demand in 2007, and so if we produce only 1 unit in 2007, then we will start 2008 with no inventory on hand. Each path from the node at the 0 level for 2007 to the node at the 0 level in 2013 corresponds to a sequence of production decisions for the years 2007 to 2012. We could, in principle, also keep nodes corresponding to having remaining inventory on hand at the end of the production horizon, but it is clear that the most cost-effective plan would involve ending the horizon with no inventory left. We can also associate a cost with each directed edge in the graph corresponding to that production decision. For each arc labeled 0 (to produce nothing) we know that no cost is incurred. But, for a directed edge labeled, for example, 2, then we incur both the setup cost, plus the per-unit cost for the two planes produced. Having done all of this, we see that we have set up the inventory

control problem as a shortest path problem in this graph. Hence, we know that we can apply the methods of dynamic programming to solve it, in just the manner that we solved the first shortest path example.

However, the graph in this figure is extremely well-structured, and this suggests that we really didn't need the graph given explicitly in the first place. This is the idea behind how we will capture the dynamic program needed to solve it. We will represent all of the crucial elements of this graph, but we will do so in a compact way that means that it is no longer necessary to write out the graph in this detail. We know that we have stages for each year of the production horizon, and that in each year, 2007 through 2013, we can have states corresponding to the possible incoming inventory levels, 0 through 4. (This also includes nodes that we know are irrelevant, but one of the recurring themes in these dynamic programming formulations is that, although these extra states are not relevant, keeping them around might simplify the description, and doesn't hurt either.)

Having defined what the stages and the states are, we can define the residual optimization problem associated with state s in stage n. That is, we can define $f_n^*(s)$ to be the minimum total remaining cost to meet demand from year n until the end of the planning horizon if we enter year n with an on-hand inventory of s units. So, for example, $f_{2010}^*(3)$ would refer to the cheapest path of decisions starting at the beginning of 2010 with an on-hand inventory of 3 planes to reach the target of having no planes on hand at the start of 2013 (or in graphical terms, the cheapest path length from the level 3 node in 2010 to reach the target node).

The value for $f_{2013}^*(0)$ is easy; that is equal to 0. Also, if we want to be a bit sloppy, we could simply allow that $f_{2013}^*(s) = 0$, for $s = 0, 1, 2, 3, 4$. This means that we allow for the possibility of disposing of excess on-hand inventory at the end of the horizon (at no cost). Of course, building planes is so expensive that one would never overproduce, so merely focusing on $s = 0$ would have sufficed.

Given the statement of the dynamic programming function by means of an English language description, we need to translate this into a recurrence relation that expresses the optimal value for a given state in a particular stage in terms of optimal values for a number of states in subsequent stages. We have already given *boundary* conditions for this recurrence, by specifying the values of this function for the year 2013. (These conditions are usually called initial conditions, since they help to initialize the computation to compute all other values. However, since for the context in which we will use this,

these "initial" conditions tend to correspond (as they do here) to the end of the process, we adopt this somewhat less standard terminology to avoid confusion.)

As an example of the recurrence relation, suppose that we want to compute $f^*_{2008}(0)$. Given that we currently have 0 units on hand, and have a demand of 1 in 2008, it follows that we can produce 1, 2, or 3 planes in 2008 (more is impossible due the annual production limit). The set-up cost is 15 and the per-unit cost this year is 56, so the total cost for each of these three options is 71, 127, and 183, respectively. Thus, we know that

$$f^*_{2008}(0) = \min\{71 + f^*_{2009}(0), 127 + f^*_{2009}(1), 183 + f^*_{2009}(2)\}.$$

More generally, if we are in state s in stage n, then we need to produce at least $\max\{\texttt{demand[n]} - s, 0\}$ units this year. If we produce k, then $s + k - \texttt{demand[n]}$ is carried over to next year, and so $s + k - \texttt{demand[n]} \leq \texttt{invLimit}$, where `invLimit` denotes the limit of on-hand inventory at the start of any year; in other words,

$$k \leq \texttt{invLimit + demand[n]} - s.$$

If we let `prodLimit` denote the annual capacity on production, and define

$$A(n, s) = \{k : \max\{\texttt{demand[n]} - s, 0\} \leq\ k \leq \min\{\texttt{prodLimit}, \texttt{invLimit + demand[n]} - s\}\},$$

then we obtain the recurrenence relation:

$$f^*_n(s) = \min_{k \in A(n,s)} \left\{\ \texttt{cost[n,k]} \quad + f^*_{n+1}(s + k - \texttt{demand[n]})\right\},$$

where `cost[n,k]` denotes the total cost of producing k planes in year n. From this recurrence relation, we immediately know how to compute the optimal solution. We start with the boundary conditions, and work backwards in time computing, one layer at a time, the next group of optimal values, keeping track of the decisions corresponding to each minimum computed. Then, once these values are computed, we can march forward through the computation, by following those decisions that yielded the optimal decision at each state.

Thus, the basic steps of setting up a problem as a dynamic program consists of the following basic steps:

1. specify the stages;

2. specify the states for each stage;

3. specify the allowed actions for each state (in each stage);

4. give an English-language description of the optimization function that will be solved for each state and each stage;

5. specify boundary conditions for this function that can be used as the base cases for the recurrence relation;

6. specify the recurrence relation;

7. compute backwards via the recurrence relation, finding the optimal values for the entire state space, while also keeping track of the decision corresponding to the optimal decision for each state in each stage;

8. compute forwards starting at the initial decision, following the sequence of optimal decisions to the target destination.

This is the essence of solving a problem by dynamic programming.

6.4 Distribution of effort problems

One type of problem that can be solved via dynamic programming are so-called "distribution of effort problems". This type of problem occurs when you have some resource that is to be shared among a number of competing entities, and you want to do this so as to maximize the collective benefit.

A concrete example of this is the *non-linear knapsack problem*. In this problem, you have a "knapsack" of a given capacity (where as far as a real-life application, the knapsack can be thought of as a truck to be packed with shipments). We have a collection of items, and for each item type, there is a given number of available items of this type, and a specified weight. So, for example, the knapsack might have weight capacity 50, and we have three types of items (1, 2, and 3), with the following data:

item type	1	2	3
number available	4	4	4
item weight	10	9	11

In general, we will let N denote the number of types, and let `weight[n]` denote the weight of items of type n, $(n = 1, \ldots, N)$, and let `itemLimit[n]` denote the number of items of type n that are available $(n = 1, \ldots, N)$; the knapsack capacity will be denoted `weightLimit`. Then, for each item type, there is an associated benefit that results from packing a given number of those items in the knapsack. So, in our example, this might be:

Number of item type packed	0	1	2	3	4
type 1	0	46	70	90	105
type 2	0	20	45	75	110
type 3	0	50	75	80	100

In general, we will let `benefit[n, j]` denote the benefit of packing j items of type n into the knapsack. The goal is to select those items that feasibly fit within the knapsack (that is, have total weight at most the capacity) of maximum total benefit.

In solving this problem by dynamic programming, we again proceed through the 8 steps we outlined previously.

1. What are the stages? A feasible solution is viewed as being selected through a series of decisions, each of which will correspond to a stage of our dynamic program. Here, it is natural to view each item type as

a stage, and to view that the decision made in that stage is to select the number of items of that type that are packed in the knapsack.

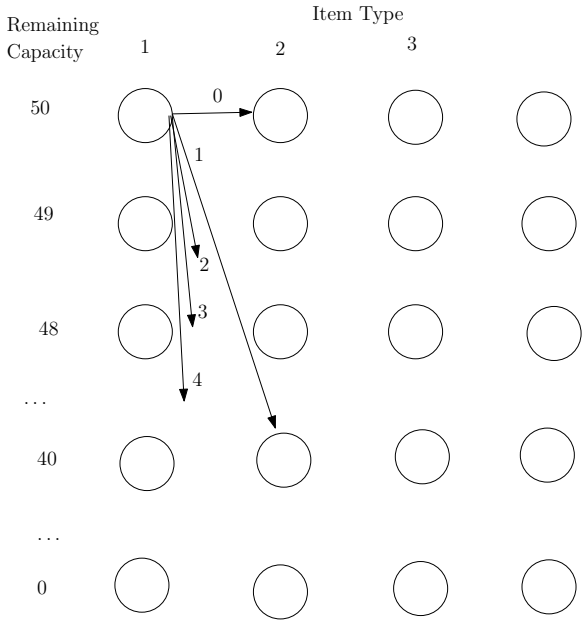
2. What are the states for each stage? The state information should capture the aspects of the decisions already made that might have an impact on all future decisions, and describe the current situation after having made all of the decisions of the earlier stages. Here, one natural approach is to focus on the capacity remaining after having made decisions about the item types considered earlier.

3. The next thing is to figure out what actions may be taken at each state in each stage. Suppose that we are in stage n, and that we have remaining capacity s; then clearly we can pack

$$j = 0, 1, \ldots, \min\{\texttt{itemLimit[n]}, \lfloor s/\texttt{weight[n]} \rfloor\}$$

items of type n given this current point in the decision process.

We could also depict this process graphically in the following way (where the figure below depicts only a fragment of the entire graph). Stated another (equivalent) way, for each stage state s in stage n, it follows that j is a possible action if $j * \texttt{weight[n]} \leq s$ and $j \leq \texttt{itemLimit[n]}$.



For each such action j, we can have an arc that goes from the state s node in stage n to the state $n - j * \texttt{weight[n]}$ in stage $n + 1$. We have added a layer $N + 1$ in this case to show the ending point after making a decision about items of type N. For each arc in the graph there is a corresponding benefit, `benefit[n,j]`. In this graph, we are trying to find a maximum benefit path from the node in stage 1 of state `weightLimit` to the $N + 1$st layer.

4. We are now ready to give the English-language description of the optimization function for each state s in stage n: we let $f_n^*(s)$ be the value of the maximum benefit possible with items of type n or greater using total weight at most s. (And in terms of the graph, it is the most beneficial path from each node to some node in layer $N + 1$.)

5. We can specify the boundary conditions easily enough. For each node in the final layer, there are no decisions left to be made, and no more benefit to accrue: $f_{N+1}^*(s) = 0$ for each $s = 0, \ldots, \texttt{weightLimit}$.

6. The recurrence relation makes use of the "principle of optimality" in the usual way: for each remaining weight s for the items $n, \ldots, N$, we will gain benefit by making an allowed decision of some j items of type n to pack in the knapsack, and then use the remaining capacity of $s - j * \texttt{weight[n]}$ optimally among the items $n + 1, \ldots, N$. Thus, $f_n^*(s)$ is equal to the maximum over all choice of `j`,

   $$\texttt{j= 0}, \ldots, \texttt{min\{itemLimit[n]}, \lfloor \texttt{weightLimit/weight[n]} \rfloor\}$$

   of the value of

   $$\texttt{benefit[n,j]} + f_{n+1}^*(s - j * \texttt{weight[n]}).$$

7. We can now compute the optimal value to the given input by filling in a table of the following form:

Unused cap. s	$f_1^*(s)$	Type 1 opt	$f_2^*(s)$	Type 2 opt	$f_3^*(s)$	Type 3 opt	$f_4^*(s)$
50							
49							
48							
...							
1							
0							

We can use the boundary conditions to fill in the last column with all zeroes. Then, we can apply the recurrence relation to fill in the two columns to the left of that: for example,

$$\begin{aligned} f_3^*(50) &= \max\{50 + f_4^*(39), 75 + f_4^*(28), 80 + f_4^*(17), 100 + f_4^*(6)\} \\ &= \max\{50, 75, 80, 100\} \\ &= 100, \end{aligned}$$

and hence the corresponding optimal item 3 decision is to pack *4* items of type 3. And similarly, $f_3^*(40) = \max\{50, 75, 80\}$, with an optimal corresponding decision of packing *3* items of type 3.

8. In this way, we can complete the entire table. If one does this, then one would find out that $f_1^*(50) = 170$; that is, the optimal value is 170. To figure out the optimal solution, we would look at the optimal decision corresponding to the entry of 170; that entry in the table is 1. That means that we should pack 1 item of type 1. That means that there is a total capacity of 40 remaining for the items of types 2 and 3. So we check out the entry for the optimal decision in stage 2 with capacity 40, and find out that that decision is 3. So we pack 3 items of type 2. This leaves capacity 13, and we can check that entry of the table to see that the optimal value for that is 1. So we pack 1 item of type 3. This gives the entire optimal solution.

This completes the process of finding the optimal solution by dynamic programming.

6.5 Using AMPL for dynamic programming

We show next how to use AMPL to model and solve problems by dynamic programming. We will use the dynamic programming formulation of the non-linear knapsack problem as a first example. We set up a data file in exactly the same way that we would have for using AMPL to solve an integer programming problem, or a linear programming problem. The data file for the input above might be:

```
param N=3;  #number of items
param weightLimit:=50; #specify weight capacity of knapsack
```

```
param : itemLimit weight  :=
1          4        10
2          4         9
3          4        11    ;

param benefit (tr)
:   1   2   3 :=  #item types 1, 2 and 3
0   0   0   0
1  45  20  50
2  70  45  70
3  90  75  80
4 105 110 100; # benefit j copies of item n
```

The model file is exactly analogous to the description we have provided in the more traditional mathematical notation.

```
param N;   # number of items
param weightLimit;  # maximum weight that we can take
param itemLimit {n in 1..N};
         # limit on number of copies of item n available
param benefit {n in 1..N, i in 0..itemLimit[n]};
         # benefit of i copies of type n items
param weight {n in 1..N}>=0 integer; # weight of type n item


set options {n in 1..N, s in 0..weightLimit} :=
    # number of copies of item type n that fits
    {i in 0..itemLimit[n]: i*weight[n] <= s };

param f {n in 1..N+1, s in 0..weightLimit} :=
    # max benefit from items n..N using weight at most s
   if n=N+1 then 0   # no more items left to consider
      else
         max {i in options[n,s]}
                 (benefit[n,i] + f[n+1,s-i*weight[n]]);

param DPvalue := f[1,weightLimit]; # compute the optimal value
```

```
set opt {n in 1..N, s in 0..weightLimit} :=
     # optimal decisions
  {i in options[n,s]: f[n,s]=benefit[n,i]+ f[n+1,s-i*weight[n]]};
```

If we trace through this code, we see that the model first inputs all of the data. The first new thing is the set `options[n,s]`. This gives the set of allowed decisions if we are in state `s` in stage `n`. That is, for each (n, s), we maintain a separate set that includes all of the possible actions. The parameter `f` is the optimization function; we see that its definition includes both the boundary cases, and the recurrence relation. The parameter `DPvalue` simply isolates that part of the information that we actually need to compute the optimal solution. By displaying `DPvalue` in AMPL, we make sure that the computation is actually performed. Finally, the set `opt` corresponds to those columns of the table above that list the optimal decisions. The only difference, and the reason that it is a set, rather than a parameter, is that we store all of the decisions that achieve the maximum, and hence also record all optimal solutions. However, the main point about this AMPL model is that it looks exactly like what we have already done.

For these files to compute the optimal solution, we use the scrolling window access to AMPL, as before. However, there are no variables in this model, so there is no need to "solve" it.

```
sw: ampl
ampl: model knapdp.mod;
ampl: data knapdp.dat;
ampl: display DPvalue;
DPvalue = 170
```

As we have done here, you should always start by displaying the DPvalue, since this ensures that the computation is performed (that is, this command causes step (7) of our general procedure to be carried out (or, more precisely, enough of it so as to be able to determine that entry's value). You can then do the step (8) of the process by hand, with the help of AMPL.

```
ampl: display weightLimit;
weightLimit = 50
```

```
ampl: display opt[1,50];
set  opt[1,50] := 1;

ampl: display weight[1];
weight[1] = 10

ampl: display opt[2,40];
set  opt[2,40] := 3;

ampl: display weight[2];
weight[2] = 9

ampl: display opt[3,13];
set  opt[3,13] := 1;
```

What have we just done? We did exactly what we did when we traced through the hand-computed table. Later, we will discuss more automatic ways to generate this output.

We can create an analogous AMPL setup for solving the inventory production model covered earlier. First, we have a data file as follows:

```
param setUp:=15;  #set up cost
param start:=1;      # starting year
param end:=6;        # ending year

param invLimit:= 4;    #limit on allowed inventory level
param prodLimit:=3;    #limit on maximum prodiction per year
param unitCost:=
1  54
2  56
3  58
4  57
5  50
6  48;  #unit cost of production beyond set up cost
param dem:=
1   1
2   1
3   2
```

```
4   2
5   1
6   1;  #demand in year n
```

The corresponding model files follows exactly the same pattern as the one that we used for the knapsack problem.

```
param setUp >=0;  #set up cost
param start;      # first year of demand
param end;        # last year of demand
param invLimit >=0;    #limit on allowed inventory level
param prodLimit >=0;    #limit on maximum prodiction per year
param unitCost {n in start..end} >=0;  #unit production cost in year n
param dem {n in start..end} >=0;    #demand in year n

param cost{n in start..end, i in 0..prodLimit}:=
           #cost of producing i in period n
     if i=0 then 0 else (setUp + i*unitCost[n]);


set options {n in start..end, s in 0..invLimit}:=
           #the set of allowed production levels
            {i in 0..prodLimit: i+s >= dem[n] and i+s-dem[n] <= invLimit};

param f{n in start..end+1, s in 0..invLimit}:=
                     #future cost starting in stage n state s
        if n = end+1 then 0
                      else min {i in options[n,s]} (cost[n,i]+f[n+1,i+s-dem[n]]);

param DPvalue:=f[start,0];

set opt{n in start..end, s in 0..invLimit}  :=
         # each table entry gives the set of optimal decisions
         {i in options[n,s] : f[n,s] = cost[n,i]+f[n+1,i+s-dem[n]] };
```

This general setup can be repeated for *any* dynamic program.

6.6 The curse of dimensionality

Consider again the non-linear knapsack problem. Suppose that we were interested in giving an integer programming formulation of it (or more generally, some mathematical programming formulation of it). As usual, we start by introducing decision variables. Let x_n denote the number of items of type n that we decide to pack into the knapsack. Then we can write the constraints on feasible solutions as

$$\sum_{n=1}^{N} \texttt{weight[n]} * x_n \leq \texttt{weightLimit},$$

and $0 \leq x_n \leq$ `itemLimit[n]` for each $n = 1, \ldots, N$. These are all valid constraints of an integer linear program. However, the most natural way to capture the objective function is to maximize

$$\sum_{n=1}^{N} \texttt{benefit[n,}x_n\texttt{]}.$$

This is not a linear function of the decision variables and hence this is not a correct integer linear programming formulation of this problem. However, it is a correct formulation, just not as an integer linear program. We will return to integer programming formulations of this model during the segment of the course on integer programming.

Suppose that we wanted to generalize this knapsack problem, by making our trucking application somewhat more realistic. In addition to having a weight constraint on the items that can be packed on the truck, we might also have a volume constraint. That is, we are also given, for each type $n = 1, \ldots, N$, the `volume[n]` for each item of that type, along with a capacity `volumeLimit` for the total volume of the items packed. The aim, as before, it to pack the maximum benefit collection of items that satisfy both the weight and the volume constraint. For the mathematical programming formulation above, we need only add the second cumulative constraint:

$$\sum_{n=1}^{N} \texttt{volume[n]} * x_n \leq \texttt{volumeLimit}.$$

How do we adapt the dynamic programming solution? We still select a feasible solution across a series of stages, where in stage n we decide how

many items of type n to pack. However, if we have already made the decisions for items of types $1, \ldots, n-1$, we need to know more than the previous state information of the remaining weight capacity that can be used for items of types $n, \ldots, N$. In fact, we need to know both the remaining weight capacity s, and the remaining volume capacity v. Thus, we can represent a state in stage n by an ordered pair (s, v).

The decisions possible for stage n in state (s, v) are analogous to what happened for the simpler version: we may pack any number of items j, for which $j = 1, \ldots,$ `itemLimit[n]`, where j`*weight[n]` $\leq$ `weightLimit` and j`*volume[n]` $\leq$ `volumeLimit`. The optimization function $f_n^*(s, v)$ is the maximum benefit possible when selecting from items of types $n, \ldots, N$, to fit in weight capacity s and volume capacity v. We add a "dummy" stage $N + 1$ to initialize the process, and set $f_{N+1}^*(s, v) = 0$ for each $s = 0, \ldots,$ `weightLimit`, $v = 0, \ldots,$ `volumeLimit`.

The recurrence relation is also quite similar. We know that $f_n^*(s, v)$ is equal to the maximum, over all possible allowed decisions to pack j items of type n, of the value

$$\texttt{benefit[n,j]} + f_{n+1}^*(s - j\texttt{*weight[n]}, v - j\texttt{*volume[n]}).$$

The process of computing the optimal value is identical to what we have done previously. The boundary conditions allow us to compute $f_{N+1}^*(\cdot, \cdot)$. Given all of the values f_{n+1}^*, we can compute all of the f_n^*; working backwards starting with N+1, we can use this to compute the value of f_1^*(`weightLimit, volumeLimit`), which is the optimal value.

How much work does it take to compute the optimal value? We need to compute the optimization function for each state in the entire process: there are $O(N$`*weightLimit*volumeLimit`$)$ states in total. In comparison, in the original knapsack problem, there were only $O(N$`*weightLimit`$)$ states. We have gone from a "1-dimensional" state space (to represent that there is 1 non-trivial constraint, to a 2-dimensional state space (to represent that there are 2 non-trivial constraints). And if we consider what happens if we consider a version with d constraints, we end up with a d-dimensional state space, and a number of states that is of the form, N`*Limit`^{d}. We say, in this case, that the number of states grows *exponentially* in the dimension d (since the number is an exponential function of d). As a result, even with a relatively small number of constraint, this approach becomes computationally infeasible; this phenomenon is commonly known as the *curse of dimensionality*.

Finally, we can also extend the AMPL formulations to this more general setting, and we will see this amounts to only a minor modification of the .mod and .dat files.

First, the data file is as follows:

```
param N=3;  #number of items
param weightLimit:=50;
param volumeLimit:=60;

param : itemLimit weight volume :=
1          4        10     18
2          4         9     12
3          4        11      8  ;

param benefit (tr)
:   1   2   3 :=  #items 1, 2 and 3
0   0   0   0
1  45  20  50.
2  70  45  70
3  90  75  80
4 105 110 100; # benefit i copies of item n
```

And then the augmented model file is:

```
param N;   # number of items
param weightLimit;  # maximum weight that we can take
param volumeLimit;  # maximum volume that we can take
param itemLimit {n in 1..N};
         # limit on number of copies of item n available
param benefit {n in 1..N, i in 0..itemLimit[n]};
         # benefit of i copies of type n items
param weight {n in 1..N}>=0 integer; # weight of type n item
param volume {n in 1..N}>=0 integer; # volume of type n item


set options {n in 1..N, s in 0..weightLimit, v in 0..volumeLimit} :=
    # number of copies of item type n that fits
    {i in 0..itemLimit[n]: i*weight[n] <= s and i*volume[n]<=v};
```

```
param f {n in 1..N+1, s in 0..weightLimit, v in 0..volumeLimit} :=
    # max benefit from items n..N using weight at most s
   if n=N+1 then 0   # no more items left to consider
      else
         max {i in options[n,s,v]}
                     (benefit[n,i] + f[n+1,s-i*weight[n],v-i*volume[n]]);

param DPvalue := f[1,weightLimit,volumeLimit]; # compute the optimal value

set opt {n in 1..N, s in 0..weightLimit, v in 0..volumeLimit} :=
     # optimal decisions
{i in options[n,s,v]:
        f[n,s,v]=benefit[n,i]+ f[n+1,s-i*weight[n],v-i*volume[n]]};
```

If we also start to run this model in the usual way, we see the following:

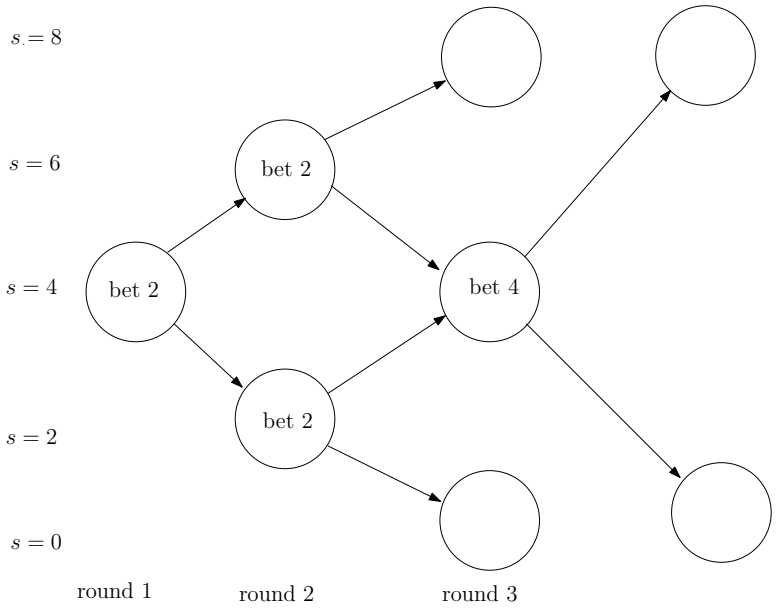
```
ampl: model knapdp2.mod;
ampl: data knapdp2.dat;
ampl: display DPvalue;
DPvalue = 160

ampl: display opt[1,weightLimit,volumeLimit];
set  opt[1,weightLimit,volumeLimit] := 0 1;
```

Notice that in the last command, we see that the optimal solution is not unique; one can either choose 0 or 1 items of type 1, and still extend that choice to an optimal solution.

6.7 Stochastic dynamic programming

Dynamic programming techniques can also be applied to settings in which there is a probabilistic element to the problem specification. As a first "toy" example, consider the following betting game. Suppose that you have 4 chips, and you wish to end up with 8 chips in order to win. In each round, you may make a decision to bet any of the chips that you have, and either lose them, or win back double your bet. That is, if you start with s chips, and bet k, you end up with either $s - k$, or $s - k + (2k) = s + k$ chips. So, one simple strategy is to bet all of your chips at once: you bet all 4 chips, and either win (with probability .6) or lose (with probability .4). The outcomes of different rounds of bets are independent events. This means that the probability of certain events happening in different rounds is equal to the product of the probabilities of them happening round by round. The goal is to design a strategy that maximizes your chance of winning. (The simple one gets us .6, but is that the best possible?)

To see how one might do better, consider the approach indicated in the following figure:



We next compute the probability of winning with this strategy. The probability that we will win after two rounds of betting is (.6)(.6)=.36, whereas

the probability of being broke after two rounds is (.4)(.4)=.16. This means that the probability of having 4 chips after 2 rounds of betting is equal to $1 - .16 - .36 = .48$. Thus, the probability of winning by having 4 after two rounds, and then winning in the 3rd is equal to (.48)(.6)=.288. So, in total, the probability of winning with this strategy is equal to .648 (better than the .6 of the simple one).

Furthermore, it is clear that we can improve the probability by playing longer and longer games. Observe that when we got back to 4 chips after round 2, then we played the simple strategy in round 3. But we could further improve things, by instead playing the entire 3-round strategy at this point. (The combined strategy would then take 5 rounds.) And then we could further improve things by using this 5-round strategy to make a 7-round one, and so forth.

But now, suppose we ask the question, what is the best 3-round strategy? Can we determine that? This is exactly suited for dynamic programming. We can follow the same 8-step program to construct a dynamic programming formulation here too. Let N denote the number of rounds in which we will bet.

What are the stages of decision-making? Each stage corresponds to a round of betting, and we decide in that round how many chips to bet.

What are the states for each stage? We can make the simplifying assumption that the best strategy will never allow us to go above 8 chips, or below 0 as a possible outcome (since any strategy that does this can be converted to another strategy, with at least as large a success probability, in which we always have between 0 and 8 chips). After understanding this, then it is clear that we let the number of chips held at the start of the round correspond to the state, and for each round the states range from 0 to 8.

What options do we have in each state in each stage? If we have s chips (and hence are in state s) then we can bet k such that $0 \leq k \leq \min\{s, 8-s\}$ chips.

What is the optimization function? We define $f_n^*(s)$ to be the maximum probability (over all possible $(N - n + 1)$-round strategies of playing the remaining rounds) of winning starting in the nth round of N with s chips.

What are the boundary conditions? As we have done so far, we introduce a dummy stage $N + 1$ to indicate that all of the rounds have been completed. In this case, we know that $f_{N+1}^*(8) = 1$, whereas $f_{N+1}^*(s) = 0$ for each $s = 0, \ldots, 7$.

What is the recurrence relation? Suppose that we decide to bet k chips starting in stage n in state s. Then with probability .6 we have $s+k$ chips, to win in the remaining rounds, whereas with probability .4, we have $s-k$ chips for the remaining rounds. So in total the probability of winning (given that we complete this bet optimally in the remaining rounds) is

$$(.6)f^*_{n+1}(s+k) + (.4)f^*_{n+1}(s-k).$$

Hence, we choose k to maximize this, and get that

$$f^*_n(s) = \max_{k=0,\ldots,\min\{s,8-s\}} (.6)f^*_{n+1}(s+k) + (.4)f^*_{n+1}(s-k).$$

This means that we can compute the optimal values, working backwards from round $N+1$, exactly as we have done for our deterministic examples. This yields the optimal value as before; but what exactly is the optimal solution? We don't know in which state we will be, for any particular stage (since this is a probabilistic process). So, to specify an optimal strategy, we must indicate the number of chips to be bet in each possible state in each stage.

We can use AMPL to compute this as well. Here are the AMPL .dat and .mod files.

```
=========================================================
chips.dat
=========================================================
param p_win:=0.6;
param p_lose:=0.4;
param rounds:=3;
param chips_have:=4;
param chips_want:=8;
=========================================================
chips.mod
=========================================================
param p_win >=0; # probability of winning
param p_lose >=0; # probability of losing
param chips_have;   # number of chips in possession
param chips_want;   # number of chips needed to win
param rounds; # max number of rounds allowed
```

```
set options {s in 0..chips_want}:=
             # number of chips we can bet when having s chips
{i in 0..s: i <= chips_want-s};

param f{r in 1..rounds+1, s in 0..chips_want}:=
            #probability of winning given that we are in round r with s chips
      if r = rounds+1 then
if s=chips_want then 1
       else 0
                   else
        max {i in options[s]} (p_win*f[r+1,s+i] + p_lose*f[r+1,s-i]);

param DPvalue:=f[1,chips_have]; # maximum probability

set opt{r in 1..rounds, s in 0..chips_want}  :=
         # each table entry gives the set of optimal decisions
         {i in options[s]: f[r,s] = p_win*f[r+1,s+i] + p_lose*f[r+1,s-i]};
```

Furthermore, we can run this model to get the following results.

```
ampl: model bet.mod;
ampl: data bet.dat;
ampl: display DPvalue;
DPvalue = 0.648

ampl: display f;
f [*,*] (tr)
:     1      2      3    4    :=
0   0       0      0     0
1   0.216   0      0     0
2   0.36    0.36   0     0
3   0.504   0.36   0     0
4   0.648   0.6    0.6   0
5   0.744   0.6    0.6   0
6   0.84    0.84   0.6   0
7   0.936   0.84   0.6   0
8   1       1      1     1
;
```

```
ampl: display opt;
set opt[1,0] := 0;
set opt[1,1] := 1;
set opt[1,2] := 0 2;
set opt[1,3] := 1 3;
set opt[1,4] := 2;
set opt[1,5] := 1 3;
set opt[1,6] := 0;
set opt[1,7] := 1;
set opt[1,8] := 0;
set opt[2,0] := 0;
set opt[2,1] := 0 1;
set opt[2,2] := 2;
set opt[2,3] := 1 2 3;
set opt[2,4] := 0 4;
set opt[2,5] := 0 1 3;
set opt[2,6] := 2;
set opt[2,7] := 1;
set opt[2,8] := 0;
set opt[3,0] := 0;
set opt[3,1] := 0 1;
set opt[3,2] := 0 1 2;
set opt[3,3] := 0 1 2 3;
set opt[3,4] := 4;
set opt[3,5] := 3;
set opt[3,6] := 2;
set opt[3,7] := 1;
set opt[3,8] := 0;
```

To look at the answers above, one might initially suspect that there are multiple optimal strategies. While in some sense this is true, for all practical purposes it is not. In the true initial state (where there are 4 chips), the optimal strategy is to bet 2. If you bet 2, then in the next round you have either 2 or 6. In each of these two states, there is the optimal decision of betting 2. Then you have either 0, 4, or 8, and in each of these states (in round 3) the answer is unique. The strategy plotted above is the unique optimal solution.

Now we turn to a better motivated example of stochastic dynamic programming. We will consider a version of the inventory control model in which the demand is not known in advance. In this case, we will assume that the demand in year `n` is a random variable, with value between 0 and some parameter `maxdem`. We shall assume that the demands in distinct years are independent random variables.

In considering such a model, it is necessary to specify what the so-called system dynamics are. In each year, given on-hand inventory level, the probability distribution for both this year and all years in the future, we decide on the production level. In our previous setting, we always had to meet the demand for that year. Suppose that this were the case in this probabilistic generalization. This would mean that no matter what demand turns out to be realized, we must have produced enough to satisfy that demand. We will consider instead a more flexible setting, in which we can take fuller advantage of the randomness in the demand.

Suppose that we no longer need to meet all of the demand in year `n`, but for each unit that we underproduce, we incur a so-called *lost sales penalty* of `penalty`. (So, for example, if we decide to produce 1 planes the first year, and the demand is 3, we must pay `2*penalty`.) Furthermore, we assume that if we overproduce, and end up with more planes than the inventory limit at the end of the year, then we can dispose of those extra units at no cost (but also no benefit).

As before, we wish to minimize the cost, but now it is the expected cost (with respect to the random demand) that we wish to minimize. In this case, we have three types of costs - set-up, unit production, and lost sales penalties.

A sample .dat file for an AMPL formulation might look as follows. Note that we also now explicitly state a maximum annual demand.

```
param setUp:=15;  #set up cost
param start:=1;      # starting year
param end:=6;        # ending year

param invLimit:= 4;    #limit on allowed inventory level
param prodLimit:=3;    #limit on maximum prodiction per year
param unitCost:=
1  54
2  56
```

```
3  58
4  57
5  50
6  48;  #unit cost of production beyond set up cost

param maxdem:=3;       # maximum demand in any year
param penalty:=72;     # penalty for each lost sale
param demProb (tr):
        1 2 3 4 5 6  :=
0 .3 .2 .1 .1  0  0
1 .3 .3 .2 .2 .2 .2
2 .4 .4 .4 .4 .3 .3
3  0 .1 .3 .3 .5 .5 ;
# probability of given demands in each year
```

The .mod file begins much as before, by reading the input and formulating a combined `cost` parameter that blends in both the set-up cost, and the unit production cost.

```
param setUp >=0;  # set up cost
param start;      # first year of demand
param end;        # last year of demand
param invLimit >=0;    # limit on allowed inventory level
                       # inventory above this level will be discarded
param prodLimit >=0;   # limit on maximum prodiction per year
param unitCost {n in start..end} >=0; # unit production cost in year n
param maxdem >=0;                     # maximum possible demand
param penalty>=0;                     # cost of not satisfying demand
param demProb {n in start..end, d in 0..maxdem} >=0;
                       # probability of demand d in year n
check {n in start..end}: sum {d in 0..maxdem} demProb[n,d]=1;

param cost{n in start..end, i in 0..prodLimit}:=
           #cost of producing i in period n
     if i=0 then 0 else (setUp + i*unitCost[n]);
```

Now, for this model, the options are actually simpler. Since we may now both exceed the inventory limit, and also not meet the demand, we are left with the following, for the options in each stage and state.

```
set options {n in start..end, s in 0..invLimit}:=
                {i in 0..prodLimit};
          #the set of allowed production levels
```

The meaning of the optimization function is not very different from the deterministic version: for each stage `n` and each state `s`, we wish to compute the minimum total expected cost incurred (over all possible strategies) for the given (probabilistic demand) starting in year `n` through year `end` starting with inventory level `s`. We can rely on the fact that expectation is a linear function: the expected total cost is the sum of the expectation of the costs incurred in the current year plus the expected total cost incurred in the remaining years. This allows us to apply the standard principle of optimality and derive the same sort of recurrence relation that we have used throughout this unit.

Let us consider what the boundary conditions are. As before, we introduce a dummy year denoted `end+1`, beyond the last year in which there is production, and we set the boundary conditions for that year. Here we shall assume that excess inventory can disposed of at no cost, and so `fend+1, s in 0..invLimit:= 0`.

As a warmup for the recurrence, let us compute the function values `f[end, s in 0..invLimit]`. Suppose that we decide to produce `i` planes in year `end` with onhand inventory `s`. First, we must pay `cost[end,i]` for this. What is our expected lost sales cost? To compute the expected value of some cost `L` (with respect to a random variable `D`), we add up, over all possible outcomes of the variable `D`, the product of the probability that `D` has that realization, times the corresponding realized value of `L`. More concretely, we have the sum

$$\sum_{d=0}^{\texttt{maxdem}} \text{(Probability that the demand is } \texttt{d}) \cdot (\text{ lost sales cost when demand is } \texttt{d}).$$

Now, when `d` less than or equal to `s+i`, then there are no sales lost, and so this sum only has non-zero terms when the demand varies from `s+i+1`

to maxdem. In general, we know that demProb[end,d] gives the probability that the demand is d for this year. Furthermore, provided, that d is indeed at least s+i, then the lost sales penalty incurred is penalty*(d-(s+i)). So we have that

```
param f{end, s in 0..invLimit}:=
   min {i in options[n,s]} (cost[n,i]  +
         sum {d in i+s+1..maxdem} demProb[n,d]*(d-i-s)*penalty );
```

To be careful, we must also note that this takes into account that the expected future cost is equal to 0, no matter how many planes we are left with for year end+1. In general, we see that things will be more complicated for an arbitrary year n, since in considering the future expected cost, one needs to know the on-hand inventory for the next year n+1, and that will depend on the realized demand. In order to simplify the notation here, we introduce a parameter leftover[s,i,d], that gives the number of items on-hand at the start of the next year, if we start in the current year with s on hand, produce i, and the realized demand is d.

```
param leftover{s in 0..invLimit, i in 0..prodLimit,
                                    d in 0..maxdem} :=
   if s+i < d then 0
              else if s+i-d > invLimit then invLimit
       else s+i-d;
```

Having done this, it is easy to see that the full version of the AMPL statement needed to capture both the boundary conditions and the recurrence relation is then

```
param f{n in start..end+1, s in 0..invLimit}:=
            #future cost starting in stage n state s
 if n = end+1 then 0
   else
         min {i in options[n,s]}

  (cost[n,i]
       # production cost

       +sum {d in i+s+1..maxdem} demProb[n,d]*(d-i-s)*penalty
```

```
       # expected lost sales cost

       +sum {d in 0..maxdem} f[n+1,leftover[s,i,d]]*
                           demProb[n,d]);
       # expected future cost (conditioned on d)
```

As before, we complete the .mod file by computing the optimal value, and the sets that give the possible optimal decisions. Finally, one can also use AMPL to compute a realization of the random demand.

```
param DPvalue:=f[start,0];

set opt{n in start..end, s in 0..invLimit}  :=
    # each table entry gives the set of optimal decisions
    {i in options[n,s] : f[n,s] =
    cost[n,i]
 +sum {d in i+s+1..maxdem} demProb[n,d]*(d-i-s)*penalty
 +sum {d in 0..maxdem} f[n+1,leftover[s,i,d]]*demProb[n,d]};


# calculate one realization of the random demand

param random {n in start..end} := Uniform(0,1);
param cumdemProb {n in start..end, d in 0..maxdem} :=
           sum {i in 0..d} demProb[n,i] ;
set excess {n in start..end} :=
   {d in 0..maxdem : random[n] <= cumdemProb[n,d]};
param realdem {n in start..end} :=
          min {d in excess[n]} (d);
```

7 Integer Programming

Integer programming methods are among the most powerful optimization tools available, and are used in a wide range of applications. Whereas 30 years ago, most real-world problems were unsolvable by this approach, today integer programs with thousands of variables can often be solved with the stroke of the "enter" key. However, one of the most frustrating aspects of integer programming is that one never knows for sure, in advance, whether a given model will be solved effiicently enough, or whether one will have stumbled upon a basically unsolvable input. Nonetheless, modern integer programming techniques have advanced the field to the point where applications get solved routinely.

7.1 Modeling the non-linear knapsack problem as an IP

We start by returning to the non-linear knapsack problem for which we had a nearly linear integer programming formulation earlier. Recall that for this problem, we have a collection of items, and for each of `N` item types, there is a given number `itemLimit[n]` of available items of type `n`, and a specified weight `weight[n]`. There is a knapsack of capacity `weightLimit`, into which we will pack the items. For each item type `n`, there is an associated benefit `benefit[n,i]` of packing `i` items. In our earlier formulation, the difficulty was in expressing the objective, since that is where the non-linearity of the problem was present.

The key idea is to introduce a new type of decision variable `y[n,i]`, which is a binary variable that indicates whether exactly `i` items of type `n` are packed (by setting this variable to 1).

This leads to the following AMPL model for an integer programming formulation.

```
param N;  #number of items
param weightLimit;  # maximum weight that we can take
param itemLimit {n in 1..N};  # limit on number of copies of item n available
param benefit {n in 1..N, i in 0..itemLimit[n]};
                  # benefit of i copies of item  type n
param weight {n in 1..N}>=0;  # weight of item type n
```

```
var x {n in 1..N} integer, >=0, <=itemLimit[n];
                # number of copies of item type n to take
var y {n in 1..N, i in 0..itemLimit[n]} binary;
        # "meaning" y[n,i]=1 if x[n]=i

maximize Benefit:
    sum {n in 1..N, i in 0..itemLimit[n]} y[n,i]*benefit[n,i];

subject to Weight: sum {n in 1..N} weight[n]*x[n] <=weightLimit;

subject to Select {n in 1..N}: sum {i in 0..itemLimit[n]} y[n,i]=1;
                    #select one y[n,i] for each n
subject to Meaning {n in 1..N}:
               x[n] = sum {i in 0..itemLimit[n]} y[n,i]*i;
```

The "Select" constraints ensure that for each type, there is exactly one value selected for the number of items packed. A first attempt at a formulation might have omitted the "Meaning" constraints, but observe that otherwise there is no linkage between the `x` and `y` variables, and the former are needed to ensure that the weight constraint is obeyed. Finally, it is important to recognize that the "Meaning" constraints are linear, in spite of the fact that we multiply the decision variables `y[n,i]` by `i`, rather than an explicit constant. Of course, the reason is that this is just a mathematical shorthand; the constraint is actually

```
x[n] = y[n,1]+2*y[n,2]+3*y[n,3]+... +itemLimit[n]*y[n,itemLimit[n]];
```

7.2 Some simple integer programming constraints

Suppose that we have an integer program in which each variable x_i corresponds to an activity i that is be considered for selection. Suppose that there are n activities, and therefore n corresponding decision variables x_i, $i = 1, \ldots, n$, where $x_i = 1$ means that we have selected activity i (and $x_i = 0$ otherwise). If we want to select exactly one of the activities, then we use the constraint

$$x_1 + x_2 + \cdots + x_n = 1.$$

If we want to select at least two of the activities, then we use the constraint

$$x_1 + x_2 + \cdots + x_n \geq 2.$$

Suppose that you want a constraint to express that activity 3 is possible only if both activities 1 and 2 are also selected. This can be done by adding the pair of constraints

$$x_3 \leq x_1, \quad \text{and} \quad x_3 \leq x_2.$$

Finally, suppose that you can only select activity 3 if either activity 1 or activity 2 are selected (or both). This can be enforced by the constraint

$$x_3 \leq x_1 + x_2.$$

7.3 The traveling salesman problem

One of the most famous (or even infamous) optimization problems is the so-called *traveling salesman problem* (or TSP). In its original statement, there are n cities that a salesman needs to visit (including his home city) and he wishes to select a tour that visits each of them, starting and ending with his home city, of the minimum total distance. The input given is the distance c_{ij} between each pair of cities i and j (and so $c_{ij} = c_{ji}$ for each pair of cities i, j).

One way to write down a feasible solution is to simply list the order in which the cities $1, 2, \ldots, n$ are visited. If one thinks of the home city as city 1, then there are $(n-1)!$ permutations of the remaining cities to specify the ordering (and that, in some sense, double counts each tour twice, since, for example, there is no difference from the tour that starts city 1, then goes to 2, then 3, et cetera until city n, and then returns to city 1, versus the tour that starts at 1, goes to n, then $n-1$, et cetera, until it reaches city 2, and then returns to 1). In fact, it is useful to note that the home city plays no special role in the problem whatsoever, since we just need a (cyclic) tour that visits each city once, and then it doesn't matter at which city one views the tour as starting and ending.

In fact, there are a number of settings in which the TSP arises in (more) practical settings. For example, in the production of printed circuit boards, there is a drilling action needed at a number of specified places on the board. These correspond to the cities. The holes are produced by a device, typically a laser), that is positioned over each of the places requiring a hole, in some

order. That ordering can be viewed as a tour through all of those points. Between any two drilling actions, the laser must be precisely repositioned over the next position. Thus, the time that elapses from the start of drilling one hole, to the start of drilling the next, consists of the time needed to produce the hole, plus the time required to reposition the laser (which is actually the dominant time in the calculation). Thus, we can use this time as the "distance" between the two positions, and then the time needed to produce all of the holes on one board is therefore just the total length of the tour. (Note that we want to return to where we started so that we are in the right place to start drilling the next board.) A sample input of this variety can be viewed at

`http://www.tsp.gatech.edu/rl5915/rl5915.html`

and an optimal tour for this input can be viewed at

`http://www.tsp.gatech.edu/rl5915/rl5915_sol.html`

This is an input with 5915 "cities". That means that there are roughly 5914! different tours to consider. How many is that? Using some standard estimation techniques, one finds that this number is roughly $5.92 * 10^{19749}$. To put this in perspective (if you can call it that), the number of atoms in the universe is generally believed to be less than a googol (i.e., 10^{100}) and so here we have a number that is something like a googol raised to the 200^{th} power. And yet of all of those tours, the tour at the URL above is *the very best one*. How could one possibly know for sure that this is the best one? Such is the power of modern integer programming methods.

If we are going to apply integer programming methods to solve the TSP, then we are going to need an integer programming formulation of the problem. We start by introducing some additional notation. Another way to view the input to the TSP is as a complete undirected graph $G = (N, E)$; that is, a graph with node set $\{1, 2, \ldots, n\}$ where the edge set E contains the edge $\{i, j\}$ for *every* possible pair of (different) nodes i and j. (We use the notation $\{i, j\}$ to denote an edge, rather than (i, j). This reflects the fact that we have an undirected edge between the two nodes. The "ordered pair" notation that uses the ordinary parentheses indicates that i is first and j is second, whereas the braces, as used for a set, does not indicate any particular ordering of the two elements.) Thus, a generic element of E might be denoted e (some edge), or $\{i, j\}$, if we wish to specified its endpoints.

We start, as usual, by defining the decision variables needed for the integer programming formulation. For each edge $e \in E$, we let

$$x_e = \begin{cases} 1 & \text{edge } e \text{ is used in the selected tour} \\ 0 & \text{edge } e \text{ is not used in the selected tour.} \end{cases}$$

The objective function is quite easy to express; we wish to minimize

$$\sum_{e \in E} c_e x_e.$$

What constraints do all feasible solutions need to satisfy? Clearly, one must have that

$$0 \leq x_e \leq 1, \quad \text{integer, for each } e \in E.$$

If one thinks of what a feasible solution looks like, then for each node i, there are two edges used with i as one of its endpoints. For example, if $n = 6$, then we know that exactly 2 of the set of edges

$$T = \{\{1,2\},\{1,3\},\{1,4\},\{1,5\},\{1,6\}\}$$

are selected in *any* feasible tour. This can be expressed by the so-called *degree constraint* that

$$x_{\{1,2\}} + x_{\{1,3\}} + x_{\{1,4\}} + x_{\{1,5\}} + x_{\{1,6\}} = 2.$$

Since this is the kind of constraint to which we will often need to refer, we introduce some notation to be able to write this more compactly. We will use the notation $\delta(1)$ to refer to the set T, or more generally, for each node $j \in N$, we let

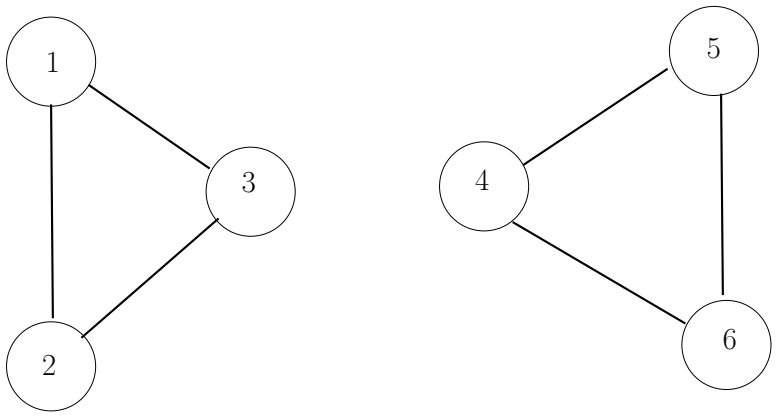
$$\delta(j) = \{\{i,j\} : \{i,j\} \in E\}.$$

Then we can write the degree contraint for node j as

$$\sum_{e \in \delta(j)} x_e = 2,$$

where we will have one such constraint for each node $j \in N$.

Suppose that we have an integer solution that satisfies all of the degree constraints. Does this correspond precisely to a feasible solution to the TSP? Consider the following figure for a 6-node input, where $x_{\{1,2\}}$, $x_{\{1,3\}}$, $x_{\{2,3\}}$, $x_{\{4,5\}}$, $x_{\{4,6\}}$, and $x_{\{5,6\}}$ are all set to 1, and all other decision variables are set to 0.



Clearly, the degree constraints are satisfied, but we have not selected a tour. (Each of these two cycles is often called a *subtour*.) We need to ensure in some way that the edges selected connect all of the nodes. One way to think about why this happened here is to consider following the set of edges:

$$\{\{1,4\},\{1,5\},\{1,6\},\{2,4\},\{2,5\},\{2,6\},\{3,4\},\{3,5\},\{3,6\}\}.$$

None of these edges has been selected, and it is clear that any solution that leaves all of them out cannot be feasible: there is no way to get from the group of nodes on the left to the group of nodes on the right. So, we could add a constraint

$$x_{\{1,4\}} + x_{\{1,5\}} + x_{\{1,6\}} + x_{\{2,4\}} + x_{\{2,5\}} + x_{\{2,6\}} + x_{\{3,4\}} + x_{\{3,5\}} + x_{\{3,6\}} \geq 1.$$

Given that we specify a feasible solution by indicating the edges used, this doesn't specify a direction in which the tour is traversed (like either clockwise or counterclockwise). However, no matter which direction one does traverse any tour, there must be some edge used to "leave" the set {1,2,3} and a different edge used to "enter" it again later. (And there might be several times that this set is left and entered again.) So, in fact, any feasible solution must satisfy the following constraint:

$$x_{\{1,4\}} + x_{\{1,5\}} + x_{\{1,6\}} + x_{\{2,4\}} + x_{\{2,5\}} + x_{\{2,6\}} + x_{\{3,4\}} + x_{\{3,5\}} + x_{\{3,6\}} \geq 2.$$

Again, this is somewhat awkward to write down, and so we will want to introduce some notation to make it more compact. For any subset $S \subseteq N$, let $\delta(S)$ denote the set of edges in the cut in which we partition the node set

into S and $N - S$; that is, $\delta(S)$ is the set of edges that have one endpoint in S, and one endpoint in $N - S$:

$$\delta(S) = \{\{i, j\} \in E : i \in S, j \in N - S\}.$$

Thus, the set of edges listed just above is exactly $\delta(\{1, 2, 3\})$. In this way, the constraint just indicated above can be written more compactly as

$$\sum_{e \in \delta(\{1,2,3\})} x_e \geq 2.$$

But with this one additional constraint, do we guarantee that any feasible solution to the integer program given thus far corresponds to a feasible tour? Of course not. The same problem we illustrated by giving two subtours, one on nodes 1,2,3 and the other on nodes 4,5,6 can be duplicated with any partition of the nodes into two sets with some 3 nodes on one side, and the remaining 3 nodes on the other. Thus, we need to have a family of *cut constraints*, where there is one such constraint for each set $S \subseteq N$, where S has between 1 and $n - 1$ nodes (and so there are also between 1 and $n - 1$ nodes in $N - S$):

$$\sum_{e \in \delta(S)} x_e \geq 2, \qquad \text{for each } S \subseteq N, S \neq \emptyset,\ S \neq N.$$

But with this family of constraints, then we do have an exact correspondence between feasible solutions to the integer program, and feasible solutions to the TSP. The degree constraints ensure that the edges selected constitute a collection of cyles on disjoint sets of nodes, and the cut constraints ensure that in fact that there must only be one cycle, since otherwise a cut constraint would be violated. Thus, we have a correct integer programming formulation of the traveling salesman problem.

Now we have an integer programming formulation; so now we can solve the TSP by using the IP methods discussed last semester. Or can we? We solved IPs by branch-and-bound, where even the initial step was to solve the LP relaxation. Can we solve the LP relaxation of the formulation that we just constructed? How big is this formulation? We have one constraint for each possible proper, non-empty subset S. If there are n nodes in the input, there $2^n - 2$ such subsets! So if there are (as in the figure at the start of this section) 5915 nodes, there are roughly $3.91 * 10^{1780}$ such constraints. While that is still alot less than the number of tours, it is still far too large to consider even storing the LP on a computer, let alone solving it.

7.4 Solving large-scale LPs by constraint generation

We turn now to a method for solving the LP relaxation that works very quickly in practice, even though the LP being solved is enormous. It is based on the following simple idea: if you find the optimal solution to the LP but with only a small subset of the constraints, and the resulting optimal solution (to the smaller problem) satisfies all of the constraints that you have ignored, then this optimal solution is also optimal to the original LP.

First, let us make this statement more precise, and then carefully step through a detailed proof of its correctness.

Relaxation principle. Consider any optimization problem, in which we are (for example) minimizing an objective function $f(x)$ subject to x satisfying two sets of constraints, $x \in P_1$, *and* $x \in P_2$. If x^* is an optimal solution to the problem of minimizing $f(x)$ subject to $x \in P_1$, and it turns out that $x^* \in P_2$, then x^* is an optimal solution to the original optimization probem.

We shall repeatedly refer to two different optimization problems: the *original* optimization problem: minimize $f(x)$ subject to $x \in P_1$ and $x \in P_2$; and the *relaxed* optimization problem: minimize $f(x)$ subject to $x \in P_1$. Let $\mathcal{O}$ and $\mathcal{R}$ denote the optimal *values* to these two optimization problems.

Observation 1. Let $\bar{x}$ be an optimal solution to the original optimization problem. Then $\bar{x}$ is a feasible solution to the relaxed optimization problem.
Observation 2. The optimal solution to an optimization problem always has value at least as good as the value of one given feasible solution.
Observation 3. Since x^* is an optimal solution to the relaxed optimization problem, $f(x^*) \leq f(\bar{x})$.
Observation 4. Applying our notation, we have that $\mathcal{R} \leq \mathcal{O}$. (That is, a relaxation to a minimization problem always provides a lower bound.)
Observation 5. We are told that x^* also satisfies $x^* \in P_2$. Hence x^* is a feasible solution to the original optimization problem.
Observation 5. Applying Observation 2 again, we get that $\mathcal{O} \leq f(x^*)$.
Observation 6. But then $\mathcal{O} \leq \mathcal{R}$.
Observation 7. But then $\mathcal{O} = \mathcal{R}$.
Observation 8. But then x^* is a feasible solution to the original optimization problem of value $\mathcal{O}$; it is an optimal solution to the original optimization problem!

This principle is something that we will repeatedly use throughout this unit on integer programming. However, here we are using it in the context of

solving linear programs. In particular, we want to solve the LP relaxation of our integer programming formulation of the TSP. Call that linear program, the TSP-LP.

Suppose that we have a subroutine for the following: given a solution x^* that satifies the degree constraints of the TSP-LP, we either find a subset S (for which $S \neq \emptyset$ and $S \neq N$) for which the corresponding cut constraint is not satisfied by x^*, that is,

$$\sum_{e \in \delta(S)} x_e^* < 2;$$

otherwise, we correctly conclude that all cut constraints are satisfied, or in other words, x^* is a feasible solution to the TSP-LP.

We show next that such a subroutine is all that we need to solve the TSP-LP. First, compute the optimal solution x^* to the LP to minimize $\sum_{e \in E} c_e x_e$ subject to all of the degree constraints. After this, in each iteration, we take the current solution x^* and apply the subroutine. If no violated constraints are found, the current solution is optimal for the TSP-LP. If a violated constraint is found, we add that constraint to the current LP, and the use CPLEX (or any other LP solver) to recompute an optimal solution x^* (using all of the constraints added thus far, but no more).

First observe that if this procedure ever stops, then the resulting solution x^* must be an optimal solution to the TSP-LP. This is a direct consequence of the Relaxation Principle: P_1 consists of the degree constraints and all of the cut constraints that have been added in the iterations until that point, and P_2 consists of all remaining cut constraints. We never need to add the same constraint twice, since once it is added to the current LP, it stays as part of the constraints until the end. And furthermore, eventually we will add all of the cut constraints, and hence we will trivially satisfy all of the cut constraints that have not been included (since none were omitted). So, eventually, the algorithm must stop (with an optimal solution to the TSP-LP).

It is something of a miracle that this approach works in practice. It is possible that one would just need to resolve LP after LP, and to have, in the end, this procedure take longer than just solving the original LP right away. But this is not what actually happens. We typically end up needing only a few iterations. Furthermore, note that the dual simplex method can use the previous optimal tableau as the starting tableau when one additional constraint has been added. This means that it is not so inefficient to add these

constraints one at a time, where after each addition we need to reoptimize.

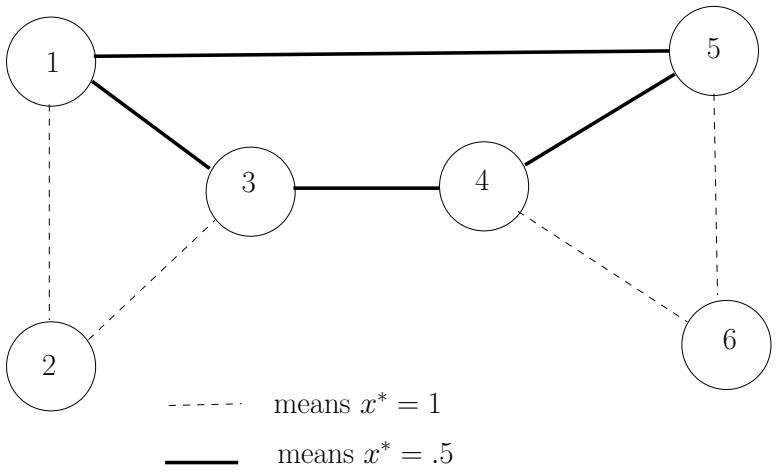
But if we are to solve the TSP-LP, we will need the subroutine described above. We need to identify a partition of the nodes into two sets S and $N - S$, such that the total weight of x^* associated with edges "going across the cut" is less than 2 (if such a "cut" exists). This sounds suspiciously like a minimum-cut problem, and indeed, we will be able to use a standard minimum-cut algorithm to construct the desired subroutine.

There are two apparent differences between the minimum-cut problem (as we presented it earlier in the course) and the kind of input we have at hand for our subroutine. The minimum-cut problem discussed earlier took a directed graph as an input; we now have an undirected graph. The minimum-cut problem that we discussed eawrlier requires a specified source node and sink node; we have no such nodes now. We will show that these are easy differences that can be overcome with simple modifications.

Here is the complete description of the required subroutine. Take the complete graph $G = (N, E)$, and build a directed graph G_D as follows: for each undirected edge $\{i, j\}$ include two directed edges (i, j) and (j, i) and give each of these two edges the capacity $x^*_{\{i,j\}}$. Let node 1 be the source node. Then, for each node $t = 2, 3, \ldots, n$, let t be the sink node, and find the minimum cut separating 1 and t in G_D (using, for example, the Ford-Fulkerson algorithm); if the resulting minimum cut $(S, N - S)$ (where $1 \in S$ and $t \in N - S$) has capacity less than 2, then S gives a cut constraint that is violated by x^*. If the minimum cut for *every* $t = 2, 3, \ldots, n$ has capacity at least 2, then *all* cut constraints are satisfied. It is a straightforward exercise to prove that this algorithm correctly serves its purpose as the necessary subroutine.

Such a subroutine is often called a *separation procedure*, and this overall approach is called a *constraint generation* method for solving large-scale linear programs.

As an example, consider the solution x^* illustrated in the figure below. (The only edges shown are those for which the corresponding x^* value is positive; $x^*(e) = 0$ for each other edge $e \in E$.)

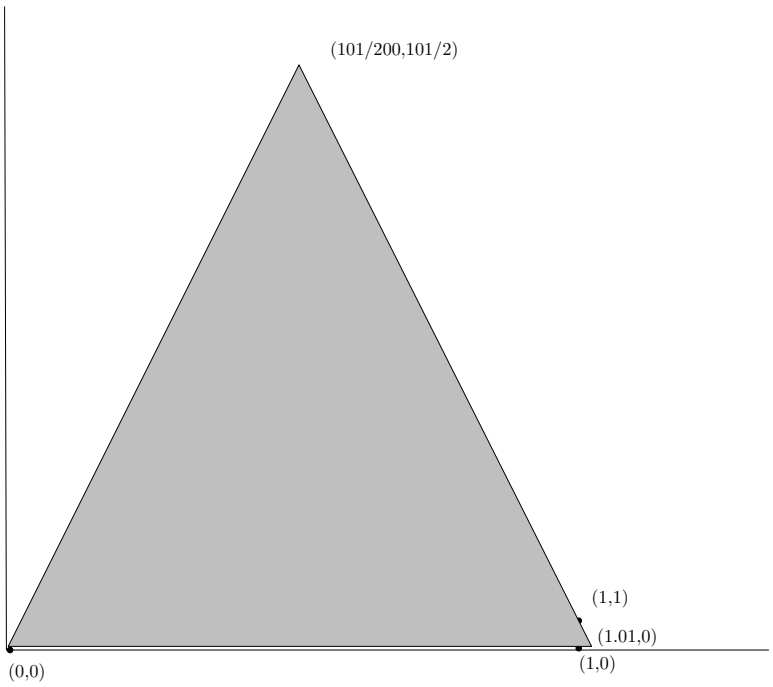


This is a fractional solution that satsfies all of the degree constraints, but not all of the cut constraints. In particular, the cut corresponding to $S = \{1, 2, 3\}$ only has weight 1 crossing it, which is too small. So, in this case, the subroutine will find this cut when it chooses node 4 to be the sink (but the minimum cut using either node 2 or node 3 as the sink will have capacity 2).

7.5 Cutting Planes

Integer programming problems are typically hard to solve. Nonetheless, the state of the art today enables IP models to be solved sufficently well, so that it is a key element of the optimization toolkit.

Consider the following integer program: maximize $x + y$ subject to $y \leq 100x$, $y + 100x \leq 101$, $x, y \geq 0$, integer. The main technique to solve integer programs is to use branch-and-bound, which starts by solving the linear programming relaxation. What does that linear programming relaxation look like in this case? Since there are only two variables, we can draw a graph of the feasible region, as shown below.

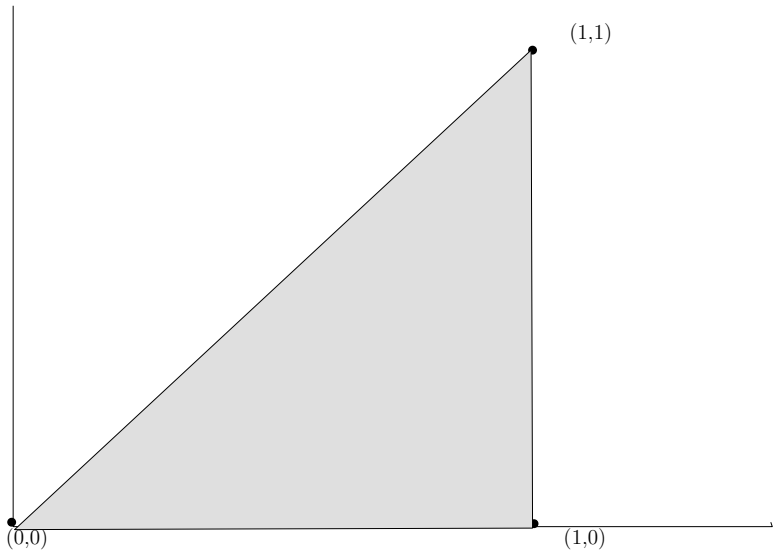


The optimal solution to this linear program is $(101/200, 101/2)$, which has objective function value equal to 51.005. This is not integer, so we have not solved the IP yet. We do know, however, that the LP optimal value is an upper bound on the IP optimal value.

By considering the graph above, we also know quite a bit about the integer program. Feasible solutions are the points in shaded region that take

on integer values for both variables; they are the points on the so-called integer lattice. In this case, there are only 3 feasible integer solutions: (0,0), (1,0), and (1,1). Hence, the optimal value is 2, and the optimal solution is (1,1). Thus, we see that the upper bound found by solving the LP relaxation is *horrible*! While this is a contrived example, it remains true that many natural IP formulations of real-world optimization problems also have such a large *integrality gap*.

Of course, there are other ways to write down an integer program that has the same set of feasible integer solutions. Suppose that we want the linear programming relaxation to be as "good" as possible (whatever good means, for now). We know that the feasible region of any linear program has to be convex: that is, if two points are feasible, then the entire line segment connecting them must also be feasible. So, if the integer program feasible region contains the 3 points indicated above, then we know that all of the points in the region below must be feasible for any linear programming relaxation: this region is often called the *convex hull* of all feasible integer solutions.



Of course, this feasible region can also be described as those points (x, y) satisfying $y \leq x$, $x \leq 1$, and $x, y \geq 0$. If we add the constraint x and y are integer to this description, then we have obtained an equivalent integer

programming formulation to our original one. However, this new formulation is *much* nicer than the original one, in that its linear programming relaxation has the property that each basic feasible solution is an integer solution. That is, its linear programming relaxation has the *integrality property*.

If some sense, the holy grail of integer programming is to devise some algorithmic method to start with a bad formulation, such as the first one given above, and produce from it the good one with the integrality property. However, we know that this is really too demanding, since there are examples of small integer programs where just writing down all of the inequalities that make up the boundary convex hull would be well beyond any reasonable means. However, the success of modern integer programming methods is based on the fact that for a given objective function, it suffices to know (or even get a good approximation) to a description of the part of the convex hull "near to" the optimal integer solution.

If we think about the two linear relaxations above, we know that what is wrong with the first is that it extends much too far in "upwards". So, if we start with such a bad formulation, it would be good to have a way to "chop off" pieces of the LP feasible region that do not contain any feasible integer solutions. That way, we still have an LP relaxation of our original IP; the set of integer points that are contained within the remaining feasible region are exactly the same 3 points. Of course, the way that we "chop off" such a piece of the feasible region is to add an additional linear inequality that must also be satisfied; such an inequality is called a *cutting plane*.

More formally, given an integer programming formulation of the form $Ax \leq b$, $x \geq 0$, integer, then an inequality $\alpha^T x \leq \beta$ is a *cutting plane* if (1) each feasible integer solution x^* satisfying $Ax^* \leq b$, $x^* \geq 0$ also satisfies the inequality $\alpha^T x^* \leq \beta$, and (2) there exists some $\hat{x}$ such that $A\hat{x} \leq b$, $\hat{x} \geq 0$, and yet $\alpha^T \hat{x} > \beta$. (The first part states that all integer feasible solutions remain feasible when we add the cutting plane to the formulation; the second states that at least one solution was "cut off" by adding the inequality to the formulation.) As an example, if we consider the original IP formulation at the start of this section, then the inequality $y \leq 2$ is a cutting plane (as is $y \leq 1$).

The concept of cutting planes is not so useful in and of itself. It only becomes useful when we have techniques to generate cutting planes automatically. Ideally, what would want would work as follows: we solve an initial LP relaxation to get a solution x^*; then we generate a cutting plane $\alpha^T x \leq \beta$ for which $\alpha^T x^* > \beta$; then we resolve the strengthened LP relaxation

(with the new cutting plane added) to get a new optimal solution (since x^* is no longer feasible), and so this process repeats, until hopefully, the resulting optimal solution to the LP relaxation is integer. By the Relaxation Principle, we know that this is an optimal solution to our original integer program. Of course, in practice, we might not be able to generate any additional cutting planes for the current LP optimum. The good news here is that the final LP optimal value still provides a bound on the IP optimal value (for example, an upper bound if we have a maximization problem) and it is a better bound than the bound given by the original LP relaxation. So this improved bound can be used within the context of branch-and-bound. More precisely, at every node of the branch-and-bound tree, rather than just solve current LP relaxation, we repeatedly try to add cutting planes to strengthen the LP relaxation. This combination of cutting plane generation and branch-and-bound is often called *branch-and-cut*.

In order to gain some intuition about finding cutting planes (and because it will turn out to be much more useful in general as well), consider now the (linear) knapsack problem, which is as follows: the input consists of n types of items, where for each item type i we are given a weight w_i & a value v_i, as well as a knapsack capacity W. We may select any integer number of items of each type, provided that their total weight is no more than the knapsack's capacity. The aim is to make the selection that maximizes the total value of the items selected, where packing ℓ items of type i has value $\ell \cdot v_i$ (and hence it is called the linear knapsack problem).

Thus, we can formulate the knapsack problem as the following integer programming problem, where x_i, $i = 1, \ldots, n$, is a decision variable that represents the number of items of type i that are selected: maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leq W$, $x \geq 0$, integer.

Consider the following input to the knapsack problem, with capacity $W = 50$.

type	1	2	3	4	5
weight	21	11	51	21	30
value	40	12	500	50	41

How do the IP optimum and the optimal solution to its LP relaxation compare? By inspection, it is not hard to convince oneself that the optimal integer solution x^* has $x_4^* = 2$ and each other $x_i^* = 0$; the optimal value is 100. On the other hand, for the LP relaxation, one can also see that what matters is getting the most "bang per buck" out of the knapsack capacity,

and the third type gets nearly a value of 10 for each unit of knapsack capacity used, and this far exceeds any of the other types. So, for the LP relaxation optimum, we set $\bar{x}_3 = 50/51$, and each other $\bar{x}_i = 0$; the LP optimal value is roughly 490. This is, of course, an upper bound on the IP optimal value, but clearly, it is a pretty bad one.

On the other hand, something truly stupid is happening here. The LP is getting alot of profit out of an item type that is clearly unusable. The weight of one item of type 3 exceeds the entire knapsack capacity. Thus, we know as far as the integer program is concerned, we must set $x_3 = 0$. So, this means that the inequality $x_3 \leq 0$ is a cutting plane for our current formulation. And of course, in general, we have that $x_i \leq 0$ is a cutting plane for each item type i such that $w_i > W$.

Suppose that we resolve our LP relaxation *strengthened* by adding this additional inequality ($x_3 \leq 0$). In this case, we get a new LP optimal solution $\tilde{x}$, where $\tilde{x}_4 = 2\frac{8}{21}$, and each other $\tilde{x}_i = 0$. Again, we have obtained a "fractional" solution; that is, the resulting solution is not an integer one. And again, there is something simple that is going wrong here. We know that x_4 could never be as large as 3 for any feasible *integer* solution. Hence, if we add the constraint $x_4 \leq 2$ to our current formulation, then we do not change the set of feasible integer solutions. This is, again, one specific instance of a general means to construct a cutting plane for the knapsack problem:

$$x_i \leq \lfloor W/w_i \rfloor, \text{ for each } i = 1, \ldots, n, \tag{7.5}$$

where this is a cutting plane only when the "floor" function has to round the fraction W/w_i down to the next integer (that is, the fraction does not happen to be integer). For our example, this generate the inequalities, $x_1 \leq 2$, $x_2 \leq 4$, $x_3 \leq 0$ (so this is just a generalization of our first construction), $x_4 \leq 2$, and $x_5 \leq 1$.

Let us resolve the new LP relaxation with all of these cutting planes (one last time). In this case, we get that $x_4 = 2$, $x_1 = 8/21$, Now we know that both types 1 and 4 weight 21, so we know that we can use at most two of these two types collectively. This can be expressed as the constraint $x_1 + x_4 \leq 2$. It should now be clear that this process of generating cutting planes can become quite involved, and we will restart with a new example to gain some additional insight.

Now consider the following input for which the capacity $W = 100$.

type	1	2	3	4	5
weight	52	53	51	10	21
value	102	99	99	2	4

From the constraints (7.5), we get the cutting planes $x_1 \leq 1$, $x_2 \leq 1$, $x_3 \leq 1$, and $x_5 \leq 4$. (Note that we do not include an inequality for x_4. Why?) If we solve the resulting LP relaxation, we get the optimal solution $x_1^* = 1$, $x_3^* = 48/51$, and all other x^* components are equal to 0. In the previous example, we could exploit the fact that multiple types had the same weight, but this is not true here, although all of the types 1, 2, and 3 have similar weights. How we take into account all three types of items? The following simple idea generalizes what we have done so far. Let H denote the set of all types for which their item weight is more than half the knapsack capacity. Clearly, no feasible packing of the knapsack can include more than 1 item of weight more than half its capacity; that is,

$$H = \{i : w_i > W/2\}.$$

Hence, we can add the constraint

$$\sum_{i \in H} x_i \leq 1.$$

For the input above, we generate the cutting plane $x_1 + x_2 + x_3 \leq 1$.

Consider another input, with capacity $W = 100$.

type	1	2	3	4	5
weight	34	35	36	10	9
value	100	101	201	8	3

What do all of the cutting planes generated so far yield for this input? From (7.5), we get $x_1 \leq 2$, $x_2 \leq 2$, $x_3 \leq 2$, and $x_5 \leq 11$. Unfortunately, the set H for this input is empty, and so we do not generate a new curring plane from that idea. But the data suggests that something similar is helpful. Let S_3 denote all of the types for which their weight is more than one third of the total knapsack capacity; that is $S_3 = \{i : w_i > W/3\}$. There can be at most 2 items packed that weigh more than one third of the capacity, and so

$$\sum_{i \in S_3} x_i \leq 2;$$

so in this example, we get the constraint $x_1 + x_2 + x_3 \leq 2$. More generally, of course, we can define $S_k = \{i : w_i > W/k\}$, and obtain a new constraint

$$\sum_{i \in S_k} x_i \leq k - 1.$$

Integer programming problems with binary variables are particularly important, and there are stronger techniques for generating cutting planes for them. For example, consider the variant of the knapsack problem in which there is only one item of each type. This is the model that we used in solving the deterministic NCAA betting pool. Consider the following input for that model, where W is again 100:

type	1	2	3	4	5
weight	51	21	31	11	42
value	90	50	70	20	70

Now the LP optimum is $x_2^* = 1$, $x_3^* = 1$, $x_4^* = 1$, and $x_1^* = 37/51$. Consider any subset of types S for which $\sum_{i \in S} w_i > W$. We know that we cannot pack one of each of those types, and so

$$\sum_{i \in S} x_i \leq |S| - 1,$$

where $|\cdot|$ refers to the size of S, that is, the number of elements in S.

But why pay so much attention to the knapsack problem? Suppose that you are solving a 0-1 integer program to maximize $c^T x$ subject to constraints $Ax \leq b$, $0 \leq x \leq 1$, integer; suppose that $\alpha^T x \leq \beta$ is one of the rows of this system of inequalities. If each $\alpha_j \geq 0$ (and so then must $\beta \geq 0$ if there exists a feasible solution) then one can view the optimization problem, maximize $c^T x$ subject to $\alpha^T x \leq \beta$ as an input to the knapsack problem. The reasoning behind generating cutting planes ensures that all integer solutions that satsify this one constraint remain satisfied when we add the new constraints. So, we can take this one row of $Ax \leq b$, and use it to generate additional constraints for an arbitrary binary integer program. And so we can generate constraints for each such inequality in the systems of constraints.

However, it might be that all constraints have coefficients that are both positive and negative. Suppose that we have a constraint

$$4x_1 - 2x_2 + \text{other terms} \leq 2.$$

If we introduce a new variable $y_2 = 1 - x_2$, then we can rewrite this constraint as $4x_1 - 2(1 - y_2) + \text{other terms} \leq 2$, and now the coefficient has become nonnegative. We can then apply the cutting plane techniques for the knapsack problem to this constraint, and when this is done, substitute back $1 - x_2$ for y_2 to express the new constraints in terms of the original decision variables. Thus, cutting planes for knapsack problems are one of the most fundamental tools in devising cutting planes for general integer programs.

In Sections 7.4 and 7.5, we have discussed similar approaches to optimization problems. Both techniques, constraint generation and cutting plane methods, involved solving a series of linear programs, adding additional inequalities as we proceed. However, when we refer to constraint generation, then this typically is a technique for solving linear programs; these linear programs have constraints that are defined implicitly and so might have an exponential number of constraints. Nonetheless, the linear program being solved is fully specified in advance, and at termination, the optimal solution is found. In contrast, cutting plane methods are used to solve an integer program, but might terminate without solving the IP at hand. The ultimate linear program that is solved is not known in advance. But the two methods have a common underlying philosophy of attempting to solve optimization problems by including all of the constraints that are needed, while still trying to include as few as possible. And both are ultimately based on the relaxation principle.

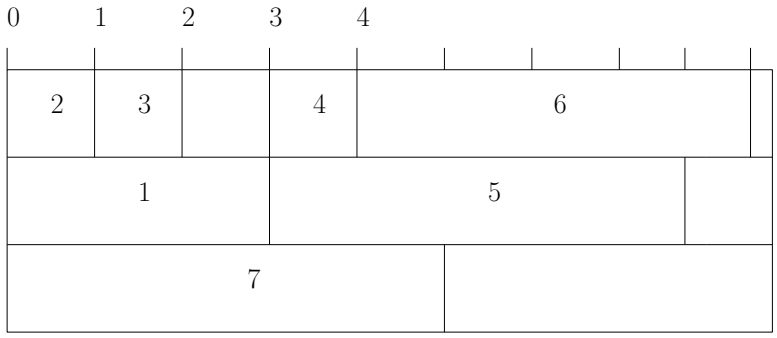
7.6 A useful IP formulation for a scheduling problem

Consider the following scheduling problem: there are m identical machines, and n jobs on which they must be scheduled; each job must be scheduled on exactly one machine, and must be processed to completion once started; each machine may process at most one job at a time; for each job j, we are given an integer processing time p_j that specifies the number of time units for which the job must be processed; for each job j, we are given a set S_j of jobs whose processing must be completed before job j may be started.

A sample input for this problem is as follows: let $m = 3$, $n = 7$, and let

j	p_j	S_j
1	3	$\emptyset$
2	1	$\emptyset$
3	1	$\emptyset$
4	1	{1,3}
5	5	{1,2}
6	5	{4}
7	5	$\emptyset$

A feasible schedule can be depicted as follows:



The time axis runs from left to right, and each row of the diagram reflects the jobs that are scheduled on one of the three identical machines.

There are many possible criteria that we might wish to optimize, but for this example, we want to find a schedule that minimizes the time by which all jobs have been completed. This *length* of the schedule is often also called its *makespan*.

We wish to construct an integer programming formulation of this problem. As usual, the first step in this process is to consider possible decision variables. Here, there are a number of possibilties.

Approach #1 We might try to use binary variables x_{ij} that indicate whether job j is assigned to be processed on machine i. This doesn't tell us when a given job is processed, and hence it might be hard to express our precedence constraints.

Approach #2 We might try to use variables C_j to indicate the time at which job j completes. This is useful to enforce the precedence constraints, but then it seems hard to ensure that only one job is being processed by each machine.

Approach #3 We can introduce binary variables x_{ijt} that captures the information in both of the previous two approaches by indicating (by setting the variable to 1) that job j starts on machine i at time t. The most notable drawback of this approach is that there are lots of variables. Note that if the data are integer, then we may assume without loss of generality that each job begins processing at an integer time.

There are many other possible classes of decision variables, but we will give a formulation that uses the last two types of variables. Let T denote an upper bound on the length of the schedule. One trivial bound is $T = \sum_{j=1}^{n} p_j$. As shown in lab, it is often useful to have a better bound to limit the number of variables that are needed. This can be done by applying a simple heuristic procedure to first build one feasible schedule, and then using its makespan to set T.

We have variables x_{ijt} that are 0-1:

$$0 \leq x_{ijt} \leq 1, \text{ integer, for each } i = 1, \ldots, m, \ j = 1, \ldots, n, \ t = 0, \ldots, T.$$

We need to ensure that each job is scheduled exactly once:

$$\sum_{i=1}^{m} \sum_{t=0}^{T-p_j} x_{ijt} = 1.$$

Most importantly, we need to enforce that at most one job at a time is being processed by each machine i. Suppose we focus on whether a job j is being processed during the time interval $[t, t+1)$. Clearly, if we start the job at time t and its processing time is at least 1, then it will be active for this entire interval. Similarly, if we start the job at time $t-1$ and its processing

time is at least 2, then it will be active for this entire interval. In general, any starting time that is at least $t + 1 - p_j$ will cause this job j to be active throughout this interval. In other words, if we consider the sum

$$\sum_{s=\max\{0,t+1-p_j\}}^{t} x_{ijt}$$

then this sum is either 0 or 1, where it is 1 exactly when job j is active throughout the interval $[t, t + 1)$. But then we can write the constraint that at most one job is active throughout this interval by requiring that:

$$\sum_{j=1}^{n} \sum_{s=\max\{0,t+1-p_j\}}^{t} x_{ijt} \leq 1, \quad \text{for each } i = 1, \ldots, m, \ t = 0, \ldots, T.$$

Now we need to ensure that no job starts prior to the time that each of its predecessors completes. If we introduce a decision variable C_j, to indicate the time at which job j is completed, then we can set

$$C_j = p_j + \sum_{t=0}^{T-p_j} t \cdot x_{ijt}, \quad \text{for each } j = 1, \ldots, n.$$

Now, to enforce the constraint implied by $j \in S_k$ that job j must be completed by the time that job k is started, we merely enforce that

$$C_j \leq C_k - p_k, \quad \text{for each } j \in S_k, \ k = 1, \ldots, n.$$

These constraints ensure that all feasible solutions correspond to feasible schedules and vice versa. Finally, we must express the objective function in terms of these variables. In fact, the simplest approach is to introduce one additional variable, $C_{\max}$. If we require that

$$C_j \leq C_{\max}, \quad \text{for each } j = 1, \ldots, n,$$

and then wish to minimize $C_{\max}$, then we will automatically ensure that $C_{\max}$ is, at optimality, always set equal to $\max_{j=1,\ldots,n} C_j$, which is precisely what we wish to attain. This "trick" of modeling the minimization of a maximum of several quantities by adding this extra variable is a quite useful one, and occurs in all sorts of applications.

Having written out this formulation in traditional mathematical notation, it is almost automatic to convert it into AMPL. We mentioned at the start of this section that this approach requires quite a large number of variables. However, this formulation has a linear programming relaxation that always (from empirical experience) provides such a good lower bound, so that branch-and-bound methods perform extremely well.

```
param N;  #number of jobs
param M; #number of machines
param proc {j in 1..N}; #processing times of jobs;
set pred {j in 1..N}; #predecessor sets

param T; #one option is :=sum {j in 1..N} proc[j];

var x {i in 1..M, j in 1..N, t in 0..T} binary;
    #x[i,j,t]=1 if job j is assigned to machine i at time t

var done {j in 1..N}; #finishing time of job j
var ms; #makespan

minimize Makespan: ms;

subject to Assign {j in 1..N}:
       sum {i in 1..M, t in 0..T-proc[j]} x[i,j,t]=1;
       #each job must be assigned

subject to time {j in 1..N}:
      done[j]=proc[j] + sum {i in 1..M, t in 0..T-proc[j]} t*x[i,j,t];

subject to makespan {j in 1..N}:
      ms >= done[j];

subject to conflict { i in 1..M, t in 0..T}:
   sum {j in 1..N, s in 0..T: s <=t and s+proc[j]>t} x[i,j,s] <=1;

subject to predecessor {k in 1..N, j in pred[k]}:
      done[j]<=done[k]-proc[k];
```

7.7 IP formulations of the roommate pairing problem

Consider the following problem: suppose that there are $2n$ individuals to be paired up as roommates in n rooms. If individuals i and j get paired up, then a benefit of b_{ij} is accrued. We will have the redundant input representation that $b_{ij} = b_{ji}$. The goal is to find a pairing of all $2n$ individuals into n pairs so as to maximize the net total benefit. We are going to study this problem from the perspective of integer programming formulations.

Once again, we start by introducing the decision variables. We will also have redundant decision variables; that is, we will have two variables x_{ij} and x_{ji} mean precisely the same thing, that person i is paired with person j, for each $i \neq j$; for notational simplicity, we will also introduce a variable x_{ii}, for each person i, but this variable will be set to 0, and hence plays no significant role. Given this, it is straightforward to devise the integer programming formulation. We wish to maximize

$$\frac{1}{2} \sum_{i=1}^{2n} \sum_{j=1}^{2n} b_{ij} x_{ij}$$

subject to

$$\sum_{i=1}^{2n} x_{ij} = 1, \text{ for each } j = 1, \ldots, 2n;$$

$$x_{ij} = x_{ji}, \text{ for each } i, j = 1, \ldots, 2n, \ i \neq j;$$

$$x_{ii} = 0, \text{ for each } i = 1, \ldots, 2n;$$

and

$$x_{ij} \geq 0, \text{ integer, for each } i, j = 1, \ldots, 2n.$$

There are two obvious questions that should immediately come to mind. First, why the $1/2$ in the objective function? That is because each benefit b_{ij}, gained by selecting the pair i and j, is counted twice in the summation, once for b_{ij} and again for b_{ji}; dividing by 2 simply ensures that we count each benefit exactly once. The second thing is that it seems tempting to also add the constraints that

$$\sum_{j=1}^{2n} x_{ij} = 1, \text{ for each } i = 1, \ldots, 2n.$$

However, these are already implied by the existing constraints: if we think about arranging the variables in an n by n matrix, where x_{ij} is the entry in the ith row and jth column, then the original constraints say that for each column, the sum of the x's in that column is exactly equal to 1. However, we also constrain that this is a symmetric matrix, and so for each row i, the sum of the x's in row i must also sum to 1. But this is precisely what the "new" constraints require.

This formulation closely resembles the integer programming formulation for the assignment problem. The assignment problem formulation had the so-called integrality property: if we consider its linear programming relaxation, then every basic feasible solution is an integer solution. Does this new formulation also have the integrality property? Unfortunately not. This can be seen by considering the following example:

	A	B	C	D	E	F
A	-	10	10	1	1	1
B	10	-	10	1	1	1
C	10	10	-	1	1	1
D	1	1	1	-	10	10
E	1	1	1	10	-	10
F	1	1	1	10	10	-

It is easy to argue that an optimal solution to this input is to pair {A,B}, {C,D}, and {E,F} for a total benfit of 21. (There are many optimal solutions.) However, the optimal solution to the linear programming relaxation of the formulation above is given by the following matrix $X = (x_{ij})$.

	A	B	C	D	E	F
A	0	0.5	0.5	0	0	0
B	0.5	0	0.5	0	0	0
C	0.5	0.5	0	0	0	0
D	0	0	0	0	0.5	0.5
E	0	0	0	0.5	0	0.5
F	0	0	0	0.5	0.5	0

.

This fractional solution has objective function value equal to 30.

What went wrong? We have 1.5 "pairings" among the set of 3 people, {A,B,C}. There can be at most 1 (real) pairing that uses only a subset of

these 3 people. Someone from the set {A,B,C} must be paired with someone from the set {D,E,F}.

This can be expressed as a constraint in terms of our decision variables in the following way:

$$x_{AD} + x_{AE} + x_{AF} + x_{BD} + x_{BE} + x_{BF} + x_{CD} + x_{CE} + x_{CF} \geq 1.$$

Observe that this constraint is a cutting plane with respect to the previous formulation for our input. This idea can be generalized in the following way: suppose that we split our original set of $2n$ people into two sets, S and $\bar{S}$, where S and $\bar{S}$ each have an odd number of people (but together they have all $2n$). Then, it is clear that in any pairing of all $2n$ people, at least one person from S must be paired with a person from $\bar{S}$. That is, we have the following "odd-man out" constraints:

$$\sum_{i \in S} \sum_{j \in \bar{S}} x_{ij} \geq 1, \text{ for each subset } S \subseteq \{1, 2, \ldots, 2n\} \text{ such that } |S| \text{ is odd.}$$

While it is natural to view these as cutting planes, one can also think about the strengthened integer programming formulation that includes each of these constraints. This latter perspective is the one that we wish to take: we have an (improved) integer programming formulation of the roommate pairing problem with the degree constraints and all odd-man out constraints. One important consideration to bear in mind is that solving larger formulations might be easier to solve (in spite of being larger) if they provide better bounds for the branch-and-bound procedure. This formulation is a striking example of what "better" can mean.

From the discussion of the traveling salesman problem, we know that it is possible to solve the linear programming relaxation of this stronger formulation if we are given a "black box" to decide, given a fractional solution $\bar{x}$, whether all of these odd-man out constraints are satisfied by $\bar{x}$, or else find an subset S for which the corresponding constraint is violated by $\bar{x}$. Indeed, as in the case of the TSP, one can use network flow techniques to solve this problem, this so-called separation problem for these inequalities, but the details of this "black box" are beyond the scope of this course. Nonetheless, it is doable. In other words, this stronger formulation has a linear programming relaxation that can be solved by constraint generation. (Recall that this means that we can first solve the relaxation with just the original set of constraints (where the sum of the x's for each column is equal to 1); in each

iteration, we use the black box to test if all odd-man out constraints are satisfied, and either prove the optimality of the current LP solution (if all of these constraints are satisfied), or else identify a new constraint to add to the LP (if they are not all satisfied).)

However, this stronger formulation has an even more remarkable property. It has the integrality property; every basic feasible solution for its linear programming relaxation is, in fact, an integer solution! Hence, if we want to solve the roommate pairing problem, we can do this by solving this (rather large) linear program (by using constraint generation).

7.8 Gomory cuts

Suppose that we are solving the following integer program: maximize $2x_2 - x_1$ subject to

$$\begin{aligned} -4x_1 + 6x_2 &\leq 9, \\ x_1 + x_2 &\leq 4, \\ x_1,\ x_2 &\geq 0, \quad \text{integer.} \end{aligned}$$

If we consider its linear programming relaxation in standard form (as we solved them in the previous semester), then we get the following linear program: maximize $2x_2 - x_1$ subject to

$$\begin{aligned} -4x_1 + 6x_2 + x_3 &= 9, \\ x_1 + x_2 + x_4 &= 4, \\ x_1,\ x_2,\ x_3,\ x_4 &\geq 0. \end{aligned}$$

Since the right-hand sides are integer, and the coefficients of x_1 and x_2 are also integer, then we know that for every integer feasible solution (x_1, x_2), the corresponding values for x_3 and x_4 are also integer.

The new linear programming relaxation has optimal solution $x = (3/2, 5/2, 0, 0)$. Recall that our final simplex tableau that yields the optimal solution has done a series of elementary row operations on the system of equations to yield the equivalent form

$$\left[I | \bar{A}_N\right] x = \left[\bar{b}\right],$$

where components of the vector x have been rearranged so that the first components correspond to the current basic variables, and the rest are non-basic (with the matrix coefficients corresponding to I and $\bar{A}_N$, respectively).

Suppose that when we solve this linear programming relaxation, as we have done here, the resulting optimal solution is not integer. This means that there exists some row for which the component of $\bar{b}$ is not integer. For example, in our example, we have the equation:

$$x_2 + (1/10)x_3 + (1/10)x_4 = 5/2. \tag{7.6}$$

We can use this to generate a cutting plane in the following way: for each non-basic variable x_j, use the fact that we constrain each variable to be non-negative to lower bound the left-hand side. For instance, in our example, we have that

$$x_2 + 0x_3 + 0x_4 \leq x_2 + (1/10)x_3 + (1/10)x_4 = 5/2.$$

Since (7.6) is satisfied by *every* feasible solution to the linear program, it is also satisfied by every integer solution. Hence $x_2 \leq 5/2$ is also satisfied by every integer solution. However, every integer solution that satisfies that constraint, also satisfies the stronger constraint that $x_2 \leq 2$. This is a cutting plane! By adding an extra slack variable, we can add this cutting plane to our LP relaxation by adding the equation $x_2 + x_5 = 2$, where once again $x_5 \geq 0$. Once again, we know that every feasible integer solution (x_1, x_2) has a corresponding x_5 that is integer. (Observe that it is also straightforward to include x_5 in the new basis, and we can restart the (primal) simplex method from the point where we left off with the previous optimal solution.)

But now we can repeat this idea. We solve the new LP relaxation. The new optimal solution has $(x_1, x_2) = (3/4, 2)$. (Of course, we do not care if the slack variables, either the original ones, or the newly added ones are integer; but we know that if one of the variables is fractional, one of the original variables must be fractional as well.) For any fractional variable from our original formulation, that variable must be basic for this optimal solution. (Since 0 is an integer!) Hence, there is a row of the optimal tableau that corresponds to this fractional variable (from our original formulation). In our case, that variable is x_1, and the equation from the optimal tableau is:

$$x_1 + (-1/4)x_3 + (6/4)x_5 = 3/4.$$

Again, we use the non-negativity of the variables to round down each of the coefficients to the nearest integer and get the fact that

$$x_1 + (-1)x_3 + (1)x_5 \leq x_1 + (-1/4)x_3 + (6/4)x_5 = 3/4.$$

All feasible fractional solutions to our previous system satisfy this inequality. Hence, all feasible solutions with x_1 and x_2 integer satisfy this constraint, and hence all feasible integer solutions for $(x_1, x_2, x_3, x_4, x_5)$ satisfy this constraint:

$$x_1 - x_3 + x_5 \leq 0.$$

This constraint is violated by the basic feasible solution that is optimal for the previous LP, and hence this is a cutting plane. But now we add yet another slack variable, and repeat. When we solve the new linear program, we get that in the optimal solution we have that $(x_1, x_2) = (1, 2)$, and hence this is the optimal integer solution to our original integer program, by the relaxation principle.

The procedure that we have just followed for this example can be summarized as follows:

1. Start with an IP formulation in standard form with integer coefficients and integer right-hand sides in which all variables are constrained to be integer.

2. Solve the current LP relaxation.

3. Identify a basic variable that is not integer in the current LP optimal solution.

4. Identify a row of the optimal tableau corresponding to this fractional variable: let this row in the optimal tableau be denoted

$$x_i + \sum_{j \in N} \bar{a}_j x_j = \bar{b}_i,$$

   where $\bar{b}_i$ is not an integer, and N denotes the set of non-basic variables.

5. Add the cutting plane

$$x_i + \sum_{j \in N} \lfloor \bar{a}_j \rfloor x_j \leq \lfloor \bar{b}_i \rfloor$$

   to the current LP by introducing a new slack variable.

6. Repeat steps 2-5 until the resulting optimal LP solution is integer; this solution is an optimal solution to the original integer program.

Around 1960, Gomory proved that this procedure (with some care) is guaranteed to terminate with an optimal solution. When this result was first obtained, it was greeted with the anticipation that "integer programming was a solved problem". It is important to understand why this result is non-trivial to prove, even though one cuts off a piece of the feasible region of the LP relaxation with every iteration. The subtlety is that one could be cutting off vanishingly small volumes of solutions, so that one might never converge to the optimal solution. However, the beauty of Gomory's proof is that one can show that the algorithm does terminate within a specified (finite) number of iterations.

Finite is not the same thing as fast, however. And within a few years after Gomory originally published his method, cutting planes were widely discredited as an approach to solving integer programs. It was more than a decade later, in the mid-70's, that cutting planes reemerged as a viable computational method for solving integer programs. The new ideas in vogue then were to focus on so-called deepest cuts, the cuts corresponding to facets of the convex hull, so as to avoid the slow convergence properties of Gomory cuts. The irony is that when progress from this new cutting plane approach slowed in the 90's, then researchers once again returned to Gomory cuts, and showed that, with the right perspective, the underlying ideas could be a valuable part of modern integer programming code (as indeed they are in CPLEX 10.1).

8 Nonlinear optimization

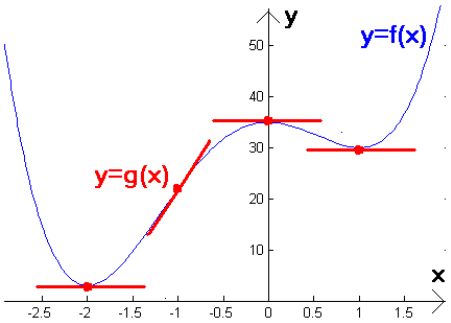
Linear programs are relatively easy to solve. By contrast, as we have seen, integer programs may be much harder. To solve an integer program, we try to capitalize on our ability to solve linear programs quickly, studying linear programming relaxations that in some sense approximate the underlying problem.

Even without integrality constraints, optimization problems may be much harder than linear programs, due to *nonlinearity*. Since we understand linear programs so well, we might naturally study nonlinear optimization problems first by approximating them using linear programs. Approximating nonlinear functions by linear functions is the central idea of differential calculus, so we begin by reviewing some ideas from calculus.

Consider the following nonlinear function of one variable, x:

$$f(x) = 35 - 12x^2 + 4x^3 + 3x^4.$$

We plot the graph of this function below.



The derivative of this function is

$$f'(x) = -24x + 12x^2 + 12x^3,$$

so using the values $f(-1) = 22$ and $f'(-1) = 24$, we arrive at the "linear approximation" to f at -1:

$$f(x) \approx f(-1) + f'(-1)(x - (-1)) = 22 + 24(x + 1) = g(x) \text{ for all } x \text{ near } -1.$$

Calculus provides a crucial tool to help us maximize or minimize nonlinear functions: at any maximizing or minimizing point $\bar{x}$, the graph of the linear

approximation should be flat. In our example, we deduce that the derivative $f'(\bar{x})$ should be zero, which implies $\bar{x} = -2$, 0, or 1. From the graph we can see that $\bar{x} = -2$ is a *minimizer*: in other words, $f(x) \geq f(-2)$ for all x. On the other hand, $\bar{x} = 1$ is a *local minimizer*, meaning $f(x) \geq f(1)$ for all x near 1. Sometimes, to emphasize the difference, we refer to minimizers as *global minimizers*.

AMPL allows us to model nonlinear functions, and the solver MINOS includes some capability to solve nonlinear optimization problems. This flexibility comes with some pitfalls, however. To illustrate, consider the following example.

```
ampl: var x;
ampl: minimize objective: 35 - 12*x^2 + 4*x^3 + 3*x^4;
ampl: solve;
MINOS 5.5: optimal solution found.
0 iterations, objective 35
Nonlin evals: obj = 3, grad = 2.
ampl: display x;
x = 0
```

As we can see from the graph, despite what AMPL tells us, $x = 0$ is certainly not a solution of our problem: in fact it is a local *maximizer*!

We can encourage AMPL to rethink by restarting at a different initial point.

```
ampl: let x:=10;
ampl: solve;
MINOS 5.5: optimal solution found.
4 iterations, objective 3
Nonlin evals: obj = 12, grad = 11.
ampl: display x;
x = -2
```

This time, AMPL has indeed solved the problem. But different initial points can produce different output.

```
ampl: let x:=0.5;
ampl: solve;
MINOS 5.5: optimal solution found.
```

```
3 iterations, objective 30
Nonlin evals: obj = 9, grad = 8.
ampl: display x;
x = 1
```

This time AMPL produced a local, rather than global, minimizer.

Nonlinear solvers like MINOS generate a sequence of iterates, like the simplex method. Just like the revised simplex method, at each iteration the solver tests for optimality, and if the test fails, the solver generates an improved iterate. As we mentioned, we know from calculus that at a minimizer of a function of one variable, the linear approximation should be flat. The solver's test for optimality also involves linear approximations, and since these approximations are typically only accurate near the current iterate, the most we can usually hope for is a *local* minimizer or maximizer.

As in linear programming, our real interest is in optimizing functions of several variables $x_1, x_2, \ldots, x_n$. For a function $f(x_1, x_2, \ldots, x_n)$, we know from calculus that the linear approximation at a point $\bar{x}$ is

$$f(x) \approx f(\bar{x}) + \nabla f(\bar{x})^T (x - \bar{x}) \quad \text{for } x \text{ near } \bar{x},$$

where $\nabla f(\bar{x})$ is the *gradient vector*:

$$\nabla f = \Big(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \Big)^T.$$

For example, if

$$f(x_1, x_2) = x_1^2 - x_2,$$

then

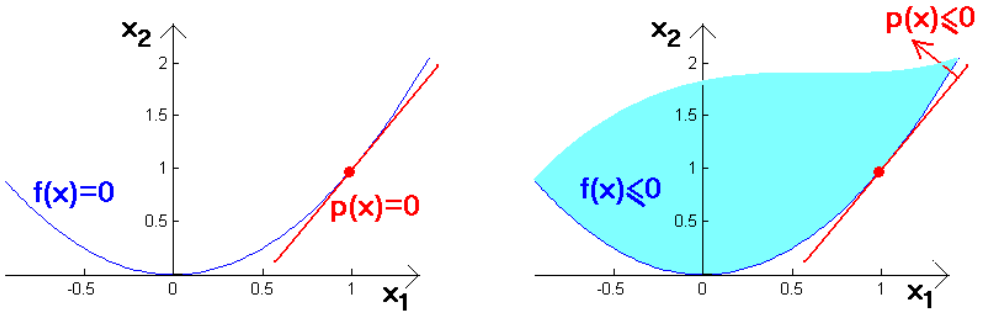
$$\nabla f = \begin{bmatrix} 2x_1 \\ -1 \end{bmatrix},$$

and so for points x near $\bar{x} = [1, 1]^T$ we have the linear approximation

$$f(x) \approx 0 + \begin{bmatrix} 2 \\ -1 \end{bmatrix}^T \left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) = p(x).$$

As the graph below illustrates, we can approximate sets defined by nonlinear constraints using the linear approximation: near $\bar{x}$ we know

$$\begin{aligned} \{x : f(x) = 0\} &\approx \{x : p(x) = 0\} \\ \{x : f(x) \leq 0\} &\approx \{x : p(x) \leq 0\}. \end{aligned}$$



A general nonlinear optimization involves many variables $x_1, x_2, \ldots, x_n$, a nonlinear objective function of the vector x, that we wish to maximize or minimize, and many nonlinear constraints involving x, that might be equations or inequalities. Concentrating on maximization, we could write our general *nonlinear program* as

$$\text{(NLP)} \qquad \left\{ \begin{array}{lrcll} \text{maximize} & f(x) & & & \\ \text{subject to} & g_i(x) & \leq & 0 & (i = 1, 2, \ldots, p) \\ & h_j(x) & = & 0 & (j = 1, 2, \ldots, q). \end{array} \right.$$

Consider a feasible solution $\bar{x}$. If we replace each function in the nonlinear program by its linear approximation, we arrive at a *linear* program:

$$\text{(ALP)} \qquad \left\{ \begin{array}{lrcll} \text{maximize} & f(\bar{x}) + \nabla f(\bar{x})^T(x - \bar{x}) & & & \\ \text{subject to} & g_i(\bar{x}) + \nabla g_i(\bar{x})^T(x - \bar{x}) & \leq & 0 & (i = 1, 2, \ldots, p) \\ & h_j(\bar{x}) + \nabla h_j(\bar{x})^T(x - \bar{x}) & = & 0 & (j = 1, 2, \ldots, q). \end{array} \right.$$

Usually (subject to some technical details that we do not discuss here), if $\bar{x}$ is an optimal solution for the nonlinear program (NLP), then it is also an optimal solution for the approximating linear program (ALP). But in that case, by the Strong Duality Theorem, there exists a complementary slack dual solution. Arguing in this way, we arrive at the following optimality conditions.

Optimality conditions for nonlinear programs *If $\bar{x}$ is an optimal solution for the nonlinear program* (NLP)*, then there exist* **Lagrange multipliers**

$$y_i \geq 0 \quad (i = 1, 2, \ldots, p) \quad \textit{and} \quad z_j \quad (j = 1, 2, \ldots, q)$$

satisfying the **complementary slackness** *conditions*

$$y_i = 0 \textit{ whenever } g_i(\bar{x}) < 0 \quad (i = 1, 2, \ldots, p),$$

and the **stationarity** *condition*

$$\nabla f(\bar{x}) = \sum_{i=1}^{p} y_i \nabla g_i(\bar{x}) + \sum_{j=1}^{q} z_j \nabla h_j(\bar{x}).$$

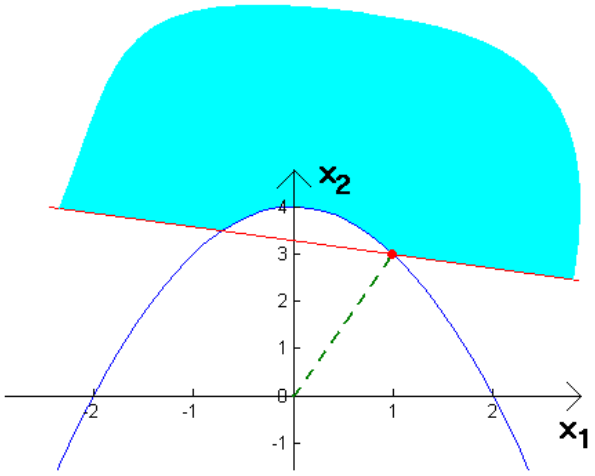
Continuing the analogy with linear programming, the Lagrange multipliers provide important sensitivity information. The number y_i is the rate of change of the optimal value of (NLP) with respect to small changes in the right-hand side of the constraint $g_i(x) \leq 0$, and z_j has an analogous interpretation. As with linear programs, AMPL will display Lagrange multipliers as "shadow prices", using the command

```
ampl: display <name of constraint>;
```

As an example, consider the nonlinear program

$$\begin{array}{lrcl} \text{minimize} & x_1^2 + x_2^2 & & \\ \text{subject to} & x_1^2 + x_2 & \geq & 4 \\ & x_1 + 4x_2 & \geq & 13. \end{array}$$

Geometrically, our problem is to find a point $[x_1, x_2]^T$ in the plane satisfying the two constraints and as close to the origin as possible. Consulting the picture below, we can convince ourselves that the point $\bar{x} = [1, 3]^T$ is the optimal solution, giving an optimal value of 10.



Let us check the optimality conditions.

In the standard form we described above, our problem is

$$\begin{array}{lrcrcl} \text{maximize} & f(x) & = & -x_1^2 - x_2^2 & & \\ \text{subject to} & g_1(x) & = & 4 - x_1^2 - x_2 & \leq & 0 \\ & g_2(x) & = & 13 - x_1 - 4x_2 & \leq & 0. \end{array}$$

Calculating the gradients gives

$$\begin{aligned} \nabla f &= \begin{bmatrix} -2x_1 \\ -2x_2 \end{bmatrix} = \begin{bmatrix} -2 \\ -6 \end{bmatrix} \text{ at } \bar{x} \\ \nabla g_1 &= \begin{bmatrix} -2x_1 \\ -1 \end{bmatrix} = \begin{bmatrix} -2 \\ -1 \end{bmatrix} \text{ at } \bar{x} \\ \nabla g_2 &= \begin{bmatrix} -1 \\ -4 \end{bmatrix} \text{ at } \bar{x}. \end{aligned}$$

Since $g_1(\bar{x}) = 0 = g_2(\bar{x})$, we learn nothing from the complementary slackness conditions. The Lagrange multipliers $y_1, y_2 \geq 0$ must satisfy the equation

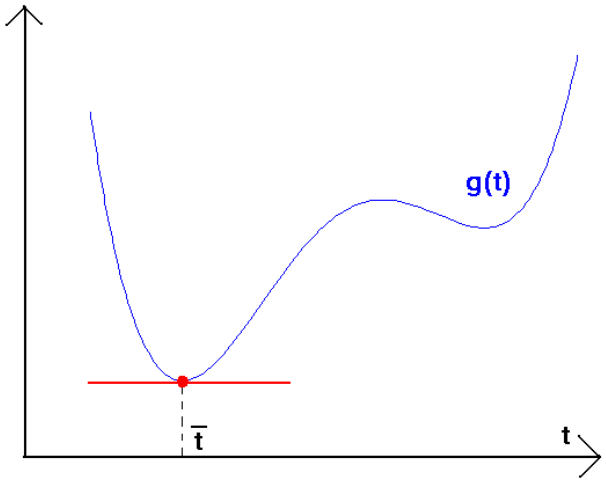
$$\begin{bmatrix} -2 \\ -6 \end{bmatrix} = y_1 \begin{bmatrix} -2 \\ -1 \end{bmatrix} + y_2 \begin{bmatrix} -1 \\ -4 \end{bmatrix},$$

from which we deduce $y_1 = \frac{2}{7}$ and $y_2 = \frac{10}{7}$.

If we change the number 4 in the original problem to $4+\epsilon$, for some small number ϵ, then the right-hand side in the problem rewritten in standard form changes from 0 to $-\epsilon$. Hence the the optimal value of this problem increases by approximately $-y_1\epsilon$, so the optimal value of the original problem decreases by approximately $\frac{2}{7}\epsilon$.

9 Minimizing nonlinear functions

As we recalled in the last section, we know from calculus that if a differentiable function of one variable $g(t)$ is minimized by setting $t = \bar{t}$ then the derivative must vanish there: $g'(\bar{t}) = 0$.



Similarly, if a differentiable function of several variables $f(x_1, x_2, \ldots, x_n)$ is minimized by setting $x = \bar{x}$, then the gradient vector there must be zero: $\nabla f(\bar{x}) = 0$. To see this, consider any vector d and define a function of one variable $g(t) = f(\bar{x} + td)$. Then $t = 0$ minimizes g, so using the chain rule we deduce

$$0 = g'(0) = \nabla f(\bar{x})^T d.$$

If we set $d = \nabla f(\bar{x})$, we deduce $\nabla f(\bar{x}) = 0$.

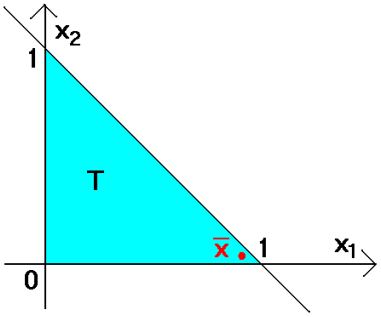
As an example, consider the function

$$(9.1)\quad f(x_1, x_2) \;=\; -192x_1 - 97x_2 - 194\Big(\log(100 - x_1 - x_2) + \log x_1 + \log x_2\Big),$$

defined on the interior of the triangle

$$T \;=\; \Big\{(x_1, x_2) : x_1, x_2 \geq 0,\; x_1 + x_2 \leq 100\Big\}.$$

We'll see later that this function is minimized at the point $\bar{x} = [97, 2]^T$.



For the moment, we can at least check that $\nabla f(\bar{x}) = 0$, by substituting $\bar{x}$ into the formula

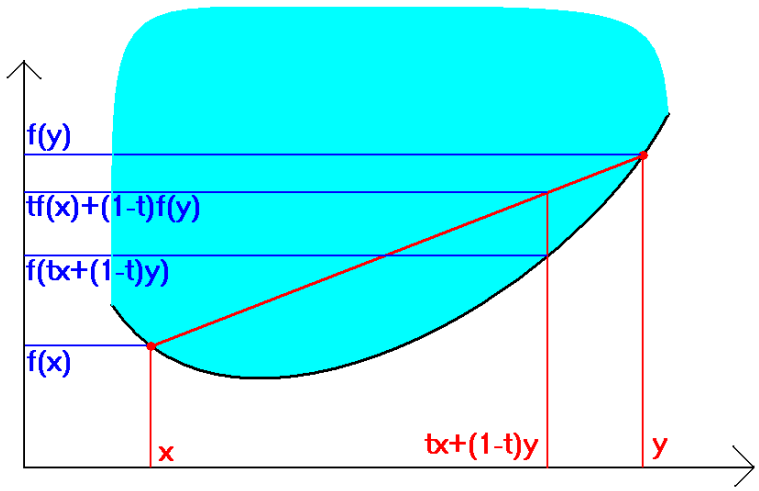
$$\nabla f = \begin{bmatrix} -192 + \dfrac{194}{100 - x_1 - x_2} - \dfrac{194}{x_1} \\ \\ -97 + \dfrac{194}{100 - x_1 - x_2} - \dfrac{194}{x_2} \end{bmatrix}.$$

From the first picture in this section, we can easily see that $g'(\bar{t}) = 0$ does not guarantee that $\bar{t}$ minimizes the function g. Similarly, for general functions f, knowing $\nabla f(\bar{x}) = 0$ does not guarantee that $\bar{x}$ minimizes f. However, for one very special class of functions, the points where the gradient vanishes are *exactly* the minimizing points. The property we need is convexity.

We call a function f *convex* if it satisfies the inequality

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y) \tag{9.2}$$

for any points x and y and any number t in the interval $[0, 1]$. We came across the idea of a "convex set" earlier in our study of linear programming: a set is convex if, whenever two points lie in the set, so must the line segment joining the two points. The complicated-looking property of f above simply amounts to the region above the graph of f being a convex set.



Checking property (9.2) looks complicated, but fortunately we can use calculus to help. We know that a function of one variable $g(t)$ is convex

providing the second derivative g'' is always nonnegative. For example, the function $g(t) = -\log t$ is convex on the interval $(0, \infty)$ because $g''(t) = \frac{1}{t^2} > 0$ for all $t > 0$. By a similar calculation, for any constant $\mu \geq 0$, any function of the form

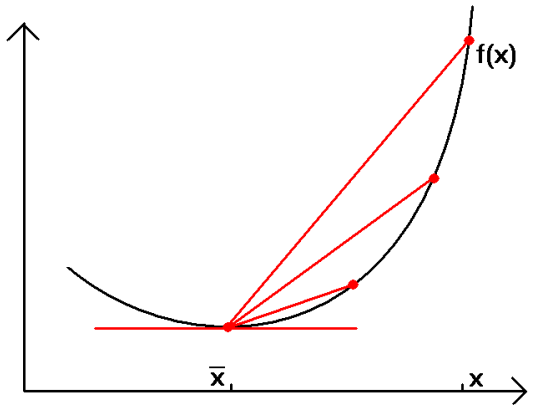
$$g(t) = at + b - \mu \sum_i \log(c_i t + d_i)$$

is convex on the interval of numbers t satisfying $c_i t + d_i > 0$ for all i. Consequently, the function f defined by equation (9.1) is convex, because along any line segment it behaves like the function g above.

As we noticed, for a general function f, knowing $\nabla f(\bar{x}) = 0$ does not guarantee that the point $\bar{x}$ minimizes f. However, for *convex* functions, it does!

Theorem 9.3 (convex sufficient optimality condition) *Any point where the gradient of a convex function vanishes is a global minimizer.*

Proof We sketch the idea of the proof in the following picture. Suppose $\nabla f(\bar{x}) = 0$, and consider any other point x. As we move along the line segment $[\bar{x}, x]$ from x back towards $\bar{x}$, the slope of the line joining the corresponding points on the graph of f decreases, because f is convex, and must approach zero, because $\nabla f(\bar{x}) = 0$, so the slopes must be nonnegative. □



So, to minimize a differentiable convex function $f(x_1, x_2, \ldots, x_n)$, we want to find a vector x satisfying $\nabla f(x) = 0$, a problem that amounts to solving n nonlinear equations in n unknowns:

$$h_i(x_1, x_2, \ldots, x_n) = 0 \quad (i = 1, 2, \ldots, n), \tag{9.4}$$

where $h_i = \frac{\partial f}{\partial x_i}$. How do we go about this computationally?

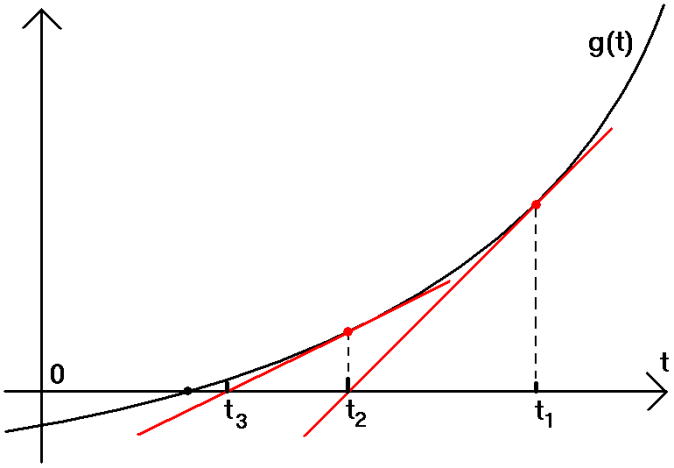
A traditional approach to solving a single equation in one variable, $g(t) = 0$, is *Newton's method*: starting from an estimate t_1 of a solution, we replace g by its linear approximation and solve the resulting equation to obtain a new estimate t_2. Thus t_2 solves the equation

$$g(t_1) + g'(t_1)(t - t_1) = 0$$

so

$$t_2 = t_1 - \frac{g(t_1)}{g'(t_1)}.$$

Repeating this process generates a sequence of iterates that, under favorable conditions, converges very quickly to a solution of the original equation.



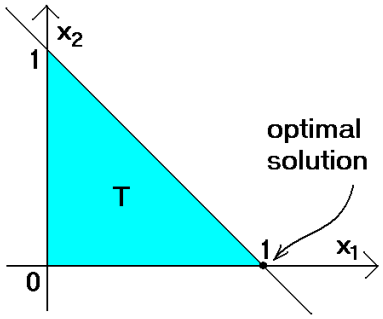
Newton's method for solving the system of equations (9.4) is analogous. Starting from an estimate x^1 of a solution, we replace each function h_i by its linear approximation and solve the resulting system of linear equations

$$h_i(x^1) + \nabla h_i(x^1)^T(x - x^1) = 0 \quad (i = 1, 2, \ldots, n),$$

obtaining a new estimate x^2. We then repeat the process.

We chose our example (9.1) with a purpose in mind. We might imagine that the problem of minimizing the function f is somehow related to the problem of solving the linear program

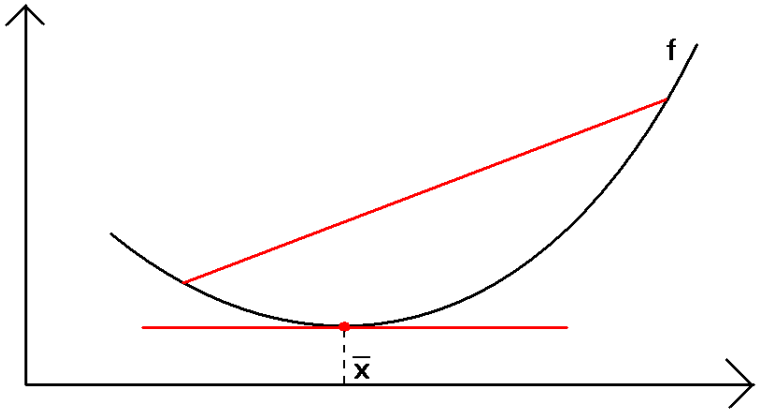
$$\begin{array}{lrcrcl} \text{minimize} & -192x_1 & - & 97x_2 & & \\ \text{subject to} & x_1 & + & x_2 & \leq & 100 \\ & x_1 & , & x_2 & \geq & 0 \end{array}$$



In the linear program, the constraints restrict our choice of points to lie inside the triangle T. On the other hand, the function f that we are trying to minimize has two parts: the first part consists of the objective functions from the linear program, while the second part consists of logarithmic terms discouraging us from choosing points too close to the boundary of the triangle. As we can see, the optimal solutions of the two problems are close to each other, suggesting that we could have found an approximate solution to the linear program by applying Newton's method to minimize f. We pursue this idea in the next section.

10 Interior point methods

To minimize a function $f(x_1, x_2, \ldots, x_n)$, we look for *critical points* (points $\bar{x}$ at which the gradient $\nabla f(\bar{x})$ is zero): as we saw in the last section, providing f is convex, any critical point must minimize f.



To solve the equation $\nabla f(x) = 0$, we can try Newton's method: given a current estimate of a solution x^k, we calculate our next estimate x^{k+1} by

replacing each component of the function ∇f by its linear approximation at x_k, and then solving the resulting linear system of equations. If all goes well, Newton's method converges to a solution very quickly.

So much for unconstrained minimization, but how should we proceed if our optimization problem has constraints? In the last section we saw an example of a convex function that we derived from a linear program. We started with the objective function of the linear program, and then added a logarithmic "barrier" term corresponding to each constraint, with the intent of discouraging the choice of points close to the boundary of the feasible region. In this section we pursue this idea more carefully, linking it to linear programming duality theory and introducing a new "interior-point" approach that has revolutionized the theory and practice of linear programming in recent years.

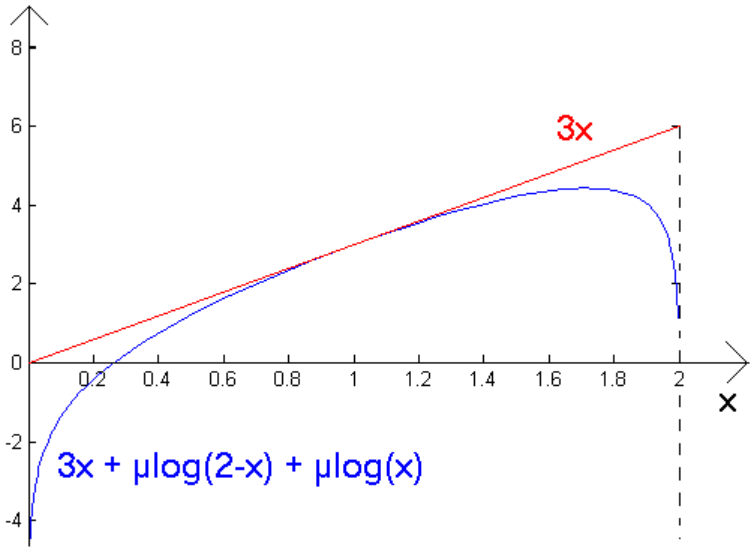
Consider the simple linear program in just one variable,

$$\begin{array}{lrcl} \text{maximize} & 3x & & \\ \text{subject to} & x & \leq & 2 \\ & x & \geq & 0, \end{array}$$

with optimal solution $x = 2$. Following the same strategy, consider the related problem

$$\text{maximize} \;\; g(x) \;=\; 3x + \mu \log(2 - x) + \mu \log x,$$

for some number $\mu > 0$, called the *barrier parameter*. The function g is only defined on the interval $(0, 2)$, so the constraints of the original problem are still present, at least implicitly. However, as the graph of g below illustrates, unlike in the original linear program, as we move the point x around in the feasible region $(0, 2)$, if x approaches a boundary point, the value of g approaches $-\infty$. The logarithmic terms we have added discourage choosing points x too close to the boundary points 0 or 2, thereby enforcing the constraints.



Of course, by switching our attention to the function g, as well as enforcing the constraints in a different manner, we have also altered our objective function. However, if we choose a small value for the barrier parameter μ, then the function $g(x)$ does not differ much from the original objective function $3x$, except close to the boundary of the feasible region. We might then suspect that maximizing g will give us a good approximation to the optimal solution of the linear program.

We check this guess. First, notice that the function $-g$ is convex: we can verify this by calculating the second derivative g''. Hence any critical point of g must be a maximizer. We therefore seek to solve the equation

$$0 = g'(x) = 3 - \frac{\mu}{2-x} + \frac{\mu}{x},$$

leading to the quadratic equation

$$3x^2 + (2\mu - 6)x - 2\mu = 0.$$

The solutions of this equation are

$$x = 1 - \frac{\mu}{3} \pm \sqrt{1 + \frac{\mu^2}{9}}.$$

The square-root lies in the interval $(1-\frac{\mu}{3}, 1+\frac{\mu}{3})$. Since the solution x that we seek must lie in the interval $(0, 2)$, we deduce that the unique critical point

is

$$x \;=\; 1 - \frac{\mu}{3} + \sqrt{1 + \frac{\mu^2}{9}} \;\in\; \Big(2 - \frac{2\mu}{3}, 2\Big).$$

As we guessed, as the barrier parameter μ decreases to zero, the critical point converges to the optimal solution of the linear program, namely $x = 2$.

Returning to our earlier linear programming example,

$$(LP1) \qquad \left\{ \begin{array}{lrcrcl} \text{minimize} & -192x_1 & - & 97x_2 & & \\ \text{subject to} & x_1 & + & x_2 & \leq & 100 \\ & x_1 & , & x_2 & \geq & 0, \end{array} \right.$$

following the same strategy leads us to seek to maximize the *barrier function*

$$f(x) \;=\; 192x_1 + 97x_2 + \mu\Big(\log(100 - x_1 - x_2) + \log x_1 + \log x_2\Big).$$

Earlier, we considered the special case when $\mu = 194$. In this general case, to find a critical point, we differentiate with respect to the variables x_1 and x_2 and arrive at the two equations

$$\begin{aligned} 192 - \frac{\mu}{100 - x_1 - x_2} + \frac{\mu}{x_1} &= 0 \\ 97 - \frac{\mu}{100 - x_1 - x_2} + \frac{\mu}{x_2} &= 0, \end{aligned}$$

or in other words,

$$\frac{\mu}{100 - x_1 - x_2} - \frac{\mu}{x_1} = 192 \tag{10.1}$$

and

$$\frac{\mu}{100 - x_1 - x_2} - \frac{\mu}{x_2} = 97. \tag{10.2}$$

These equations are reminiscent of the dual problem for our linear program,

$$\begin{array}{lrcr} \text{minimize} & 100y & & \\ \text{subject to} & y & \geq & 192 \\ & y & \geq & 97 \\ & y & \geq & 0, \end{array}$$

and even more so if we introduce slack variables:

$$\begin{array}{lrcrcrcr} \text{minimize} & 100y & & & & & & \\ \text{subject to} & y & - & t_1 & & & = & 192 \\ & y & & & - & t_2 & = & 97 \\ & y & , & t_1 & , & t_2 & \geq & 0, \end{array}$$

When $\mu = 194$, we verified earlier that the vector $x = [97, 2]^T$ solved the critical point equations (10.1) and (10.2), and so, using convexity, must maximize the barrier function f. Our discussion at the beginning of this section leads us to hope that, when the barrier parameter μ is small, critical points of f approximate the optimal solution to the linear program $(LP1)$. In this example, even when μ takes the relatively large value 194, the critical point $[97, 2]^T$ clearly approximates the linear program's optimal solution $[100, 0]^T$.

As we know from elementary linear programming duality theory, we can quantify the quality of a feasible solution of a linear program, with a certain primal objective value, by producing a dual feasible solution whose dual objective value is close to the primal objective value: the true optimal value must then be sandwiched between the two values. We can use this strategy to judge how close to optimal critical points of the barrier function f are.

Consider a critical point $\bar{x}$ for the barrier function f. Since $\bar{x}$ solves the equations (10.1) and (10.2), we can easily produce a dual feasible solution:

$$\bar{y} = \frac{\mu}{100 - \bar{x}_1 - \bar{x}_2}\,, \quad \bar{t_1} = \frac{\mu}{\bar{x}_1}\,, \quad \bar{t_2} = \frac{\mu}{\bar{x}_2}.$$

The *duality gap* between the primal objective value of $\bar{x}$ and the dual objective value of $(\bar{y}, \bar{t_1}, \bar{t_2})$ is then

$$\begin{aligned} 100\bar{y} - (192\bar{x}_1 + 97\bar{x}_2) &= 100\bar{y} - \bar{x}_1(\bar{y} - \bar{t_1}) - \bar{x}_2(\bar{y} - \bar{t_2}) \\ &= \bar{y}(100 - \bar{x}_1 - \bar{x}_2) + \bar{x}_1\bar{t_1} + \bar{x}_2\bar{t_2} \\ &= 3\mu. \end{aligned}$$

As we hoped, when μ is small, the duality gap is small, so the corresponding critical point of f is nearly optimal for our linear program.

A little algebra shows a similar result in general. For a linear program in standard inequality form, with n variables and m constraints, the linear program's objective value at a critical point of the corresponding barrier function with barrier parameter μ is within an amount $\mu(m + n)$ of the optimal value.

This suggests an *interior point* strategy for solving linear programs in standard inequality form, so-called because, unlike the the simplex method, the iterates $x^1, x^2, x^3 \ldots$ that we generate lie in the interior of the feasible region. Starting from an iterate x^k, we use Newton's method to find an approximation x^{k+1} to a critical point of the barrier function. We then reduce the barrier parameter μ, and repeat the process. By forcing the duality gap to shrink to zero, we guarantee convergence to optimality. This interior-point philosophy has revolutionized linear programming in recent years, producing computational schemes that are superior to the simplex method in theory (being, in particular, "polynomial-time"), and that are highly competitive or dominant in practice. We outline the ideas in more detail in the next sections.

11 The central path

In the previous section we worked through a particular example of a linear program in standard inequality form, suggesting a solution scheme based on approximately minimizing the corresponding barrier function. We now return to the general case.

We consider the general linear program in standard inequality form,

$$(P) \qquad \begin{cases} \text{maximize} & c^T x & & \\ \text{subject to} & Ax & \leq & b \\ & x & \geq & 0, \end{cases}$$

where we assume the matrix A is m-by-n. For everything that follows, we assume the following condition.

Assumption The primal linear program (P) has a *strictly feasible solution*, by which we mean a vector $\hat{x} > 0$ satisfying $A\hat{x} < b$.

(Inequalities written between vectors mean they hold for each component.) As usual, we can introduce slack variables, converting the problem into standard equality form:

$$\begin{array}{lrcccl} \text{maximize} & c^T x & & & & \\ \text{subject to} & Ax & + & s & = & b \\ & x & , & s & \geq & 0. \end{array}$$

The dual problem corresponding to the linear program (P) is

$$(D) \qquad \begin{cases} \text{minimize} & b^T y & & \\ \text{subject to} & A^T y & \geq & c \\ & y & \geq & 0. \end{cases}$$

Analogously to the primal problem, we assume the following condition.

Assumption The dual linear program (D) has a strictly feasible solution: there exists a vector $\hat{y} > 0$ satisfying $A^T \hat{y} > c$.

By the Strong Duality Theorem, and complementary slackness, for vectors $x \geq 0$, $s \geq 0$, $y \geq 0$, and $t \geq 0$, we know (x, s) is optimal for the primal problem (P) and (y, t) is optimal for the dual problem (D) if and only the following conditions hold:

$$(CS) \qquad \begin{cases} Ax + s &= b & \\ A^T y - t &= c & \\ x_j t_j &= 0 & (j = 1, 2, \ldots, n) \\ y_i s_i &= 0 & (i = 1, 2, \ldots, m). \end{cases}$$

Following the pattern of our example in the last section, we next introduce the barrier problem corresponding to a barrier parameter $\mu > 0$:

$$(PB_\mu) \qquad \text{maximize } f_\mu(x),$$

where

$$f_\mu(x) \;=\; c^T x + \mu \sum_{i=1}^{m} \log \Big(b_i - \sum_{j=1}^{n} a_{ij} x_j \Big) + \mu \sum_{j=1}^{n} \log x_j.$$

This function is well-defined at the strictly primal-feasible point $\hat{x}$, for example, and it turns out that the existence of the strictly dual-feasible point $\hat{y}$ guarantees that the set

$$\{x : f_\mu(x) \geq f_\mu(\hat{x})\}$$

is bounded. Since f_μ is clearly a continuous function on this set, it must therefore have a maximizer $x(\mu)$. Just as we argued for our example in the

previous section, the function $-f_\mu$ is convex and differentiable, so $x(\mu)$ must be a critical point. By differentiating with respect to x_j, we deduce

$$c_j - \mu \sum_i \frac{a_{ij}}{b_i - \sum_k a_{ik} x_k(\mu)} + \frac{\mu}{x_j(\mu)} \;=\; 0,$$

for each j. A little more thought shows that $x(\mu)$ must in fact be the unique maximizer for $-f_\mu$.

Again following the pattern of our example from the last section, we can now construct a dual feasible solution that allows us to quantify how close to primal optimality the vector $x(\mu)$ is. Specifically, for all i and j we define

$$\begin{aligned} s_i(\mu) &= b_i - \sum_j a_{ij} x_j(\mu) \\ y_i(\mu) &= \frac{\mu}{s_i(\mu)} \\ t_j(\mu) &= \frac{\mu}{x_j(\mu)}. \end{aligned}$$

We have now sketched a proof of the following fundamental result.

Theorem 11.1 (existence of the central path) *Given a linear program in standard inequality form, suppose there exist strictly feasible primal and dual solutions. Then for all values of the barrier parameter $\mu > 0$, the equations*

$$(CP_\mu) \qquad \left\{ \begin{array}{rcll} Ax + s &=& b & \\ A^T y - t &=& c & \\ x_j t_j &=& \mu & (j = 1, 2, \ldots, n) \\ y_i s_i &=& \mu & (i = 1, 2, \ldots, m) \end{array} \right.$$

have a unique solution $\Big(x(\mu), y(\mu), s(\mu), t(\mu)\Big) > 0$.

The curve

$$\Big\{ \Big(x(\mu), y(\mu), s(\mu), t(\mu)\Big) : \mu > 0 \Big\}$$

is called the *central path*.

As we observed, the vector $x(\mu)$ is strictly feasible for the primal linear program (P) and solves the primal barrier problem (PB_μ). The curve traced

by this vector as the barrier parameter μ shrinks to zero is called the *primal central path*. Similarly, the vector $y(\mu)$ is strictly feasible for the dual linear program (D), and solves a dual barrier problem analogous to the primal version. The curve traced out by $y(\mu)$ is called the *dual central path*.

Notice that if we were to set $\mu = 0$, the central path equations (CP_μ) become exactly the equations (CS) describing primal and dual feasibility and complementary slackness. Thus we can think of the central path equations as describing primal and dual feasibility and *perturbed* complementary slackness.

In the last section, we worked through two simple examples where the critical point $x(\mu)$ for the primal barrier function approached primal optimality as the barrier parameter μ shrunk towards zero, and we claimed that this was true generally. We can now justify our claim. If we set

$$(x, y, s, t) \;=\; \Big(x(\mu), y(\mu), s(\mu), t(\mu)\Big) \;>\; 0,$$

then the vector x is feasible for the primal problem (P), the vector y is feasible for the dual problem (D), and the duality gap between the corresponding primal and dual objective values is

$$\begin{aligned} b^T y - c^T x &= (Ax + s)^T y - c^T c \\ &= s^T y + (A^T y - c)^T x \\ &= s^T y + t^T x \\ &= (m + n)\mu, \end{aligned}$$

and this quantity shrinks to zero as μ shrinks to zero. Hence $x(\mu)$ approaches primal optimality and $y(\mu)$ approaches dual optimality.

The central path theorem suggests a natural iterative scheme for solving linear programs. Starting at a current iterate x^k, we use Newton's method to find an approximate solution x^{k+1} of the central path equations (CP_μ). We then reduce the barrier parameter μ, and repeat the process. We examine this process more carefully in the next section.

12 A primal-dual interior point method

Summarizing the framework of the last section, we consider a primal linear program in standard equality form

$$(P) \qquad \left\{ \begin{array}{lrcl} \text{maximize} & c^T x & & \\ \text{subject to} & Ax & \leq & b \\ & x & \geq & 0, \end{array} \right.$$

where the matrix A is m-by-n. We introduce a vector of slack variables s. The corresponding dual problem is

$$(D) \qquad \left\{ \begin{array}{lrcl} \text{minimize} & b^T y & & \\ \text{subject to} & A^T y & \geq & c \\ & y & \geq & 0, \end{array} \right.$$

and for this problem we introduce a vector of slack variables t. We assume the primal and dual problems both have strictly feasible solutions. Then according to the Central Path Theorem, for any barrier parameter $\mu > 0$, the central path equations

$$(CP_\mu) \qquad \left\{ \begin{array}{rcll} Ax + s & = & b & \\ A^T y - t & = & c & \\ x_j t_j & = & \mu & (j = 1, 2, \ldots, n) \\ y_i s_i & = & \mu & (i = 1, 2, \ldots, m) \end{array} \right.$$

have a unique solution $\Big(x(\mu), y(\mu), s(\mu), t(\mu)\Big) > 0$. Furthermore, the duality gap between the primal and dual objective values at the corresponding feasible solutions is

$$b^T y(\mu) - c^T x(\mu) = (m + n)\mu.$$

Hence, when μ is small, the primal-feasible vector $x(\mu)$ is nearly optimal for (P) and the dual-feasible vector $y(\mu)$ is nearly optimal for (D).

If we could trace the central path $\Big(x(\mu), y(\mu), s(\mu), t(\mu)\Big)$ as the barrier parameter μ shrinks to zero, then we might hope in the limit to find optimal solutions to the problems (P) and (D). In practice, we cannot trace this nonlinear curve exactly, but we *can* hope to approximate it: this idea is the foundation of modern interior-point methods for solving linear programs. In this section, we outline the essential idea.

In their simplest form, unlike the simplex method, interior point methods do not aim to solve linear programs exactly. Instead, we fix some *tolerance* $\epsilon > 0$, and stop the algorithm when we have found primal and dual feasible solutions with duality gap less than ϵ. Schematically, the *primal-dual interior point algorithm* proceeds as follows.

Primal-dual algorithm
Given:

- feasible $x > 0$ for (P), with slack $s > 0$
- feasible $y > 0$ for (D), with slack $t > 0$
- barrier parameter $\mu > 0$,

suppose (x, y, s, t) approximately solves (CP_μ). Then:

while $(m + n)\mu > \epsilon$,
    $\mu = \frac{\mu}{10}$;
    while some $|y_i s_i - \mu|$ or $|x_j t_j - \mu| > \frac{\mu}{10}$,
        use Newton's method to improve solution of (CP_μ);
    end;
end;

The key step in this conceptual algorithm is the Newton iteration: how do we use Newton's method to improve our current approximate solution to the central path equations (CP_μ)? Following the philosophy of Newton's method, we simply replace the central path equations with their linear approximations at the current approximate solution, and solve this linear system of equations to find our new approximate solution.

If the current approximate solution is $z = (x, y, s, t)$, we can write the new approximate solution in the form

$$(x, y, s, t) + (\Delta x, \Delta y, \Delta s, \Delta t) = z + \Delta z,$$

where the increment Δz is small. This new solution should satisfy the linear approximation to the central path equations (CP_μ), namely:

$$\begin{aligned} A\Delta x + \Delta s &= 0 \\ A^T \Delta y - \Delta t &= 0 \\ x_j t_j + x_j \Delta t_j + t_j \Delta x_j &= \mu \quad (j = 1, 2, \ldots, n) \\ y_i s_i + y_i \Delta s_i + s_i \Delta y_i &= \mu \quad (i = 1, 2, \ldots, m). \end{aligned}$$

(As usual when forming a linear approximation, we derive these equations simply by substituting the new solution into the original equations (CP_μ), and throwing away any term involving a product of two components of Δz.) We can now solve this linear system for Δz. Newton's method simply repeats this iteration until we find a sufficiently accurate solution to (CP_μ).

One difficulty remains. The solution z that we seek for the central path equations (CP_μ) must also be strictly positive. Unfortunately, even if our current approximate solution satisfies $z > 0$, taking the *full Newton step* from z to the new approximation $z + \Delta z$ may destroy this property. In that case, we take a less aggressive step. We first calculate what proportion of the full Newton step would take us to the boundary of the feasible region:

$$\max\{\beta : z + \beta \Delta z \geq 0\}.$$

In order to stay in the interior of the feasible region, instead of the full Newton step, we move a fixed fraction of the distance to the boundary:

$$z = z + \min\left\{\frac{9}{10}\beta, 1\right\}\Delta z.$$

Practical interior point codes refine this basic outline somewhat, but retain the fundamental structure we have described. We illustrate one iteration on a simple example. Consider the linear program

$$\begin{array}{lrcrcl} \text{maximize} & 3x_1 & + & 2x_2 & & \\ \text{subject to} & 2x_1 & + & x_2 & \leq & 4 \\ & x_1 & , & x_2 & \geq & 0, \end{array}$$

with dual problem

$$\begin{array}{lrcl} \text{minimize} & 4y & & \\ \text{subject to} & 2y & \geq & 3 \\ & y & \geq & 2 \\ & y & \geq & 0. \end{array}$$

The primal optimal solution is $x^* = [0, 4]^T$, and the dual optimal solution is $y^* = 2$.

After introducing slack variables $s \geq 0$ for the primal and $t_1, t_2 \geq 0$ for

the dual, we can write the central path equations

$$\begin{aligned} 2x_1 + x_2 + s &= 4 \\ 2y - t_1 &= 3 \\ y - t_2 &= 2 \\ x_1 t_1 &= \mu \\ x_2 t_2 &= \mu \\ ys &= \mu \end{aligned}$$

Suppose we start with barrier parameter $\mu = 1$ and the approximate solution

$$x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad s = 1, \quad y = 3, \quad t = \begin{bmatrix} 3 \\ 1 \end{bmatrix}.$$

Forming the linear approximation to the central path equations at this point results in the linear system

$$\begin{aligned} 2\Delta x_1 + \Delta x_2 + \Delta s &= 0 \\ 2\Delta y - \Delta t_1 &= 0 \\ \Delta y - \Delta t_2 &= 0 \\ 3 + 3\Delta x_1 + \Delta t_1 &= 1 \\ 1 + \Delta x_2 + \Delta t_2 &= 1 \\ 3 + \Delta y + 3\Delta s &= 1. \end{aligned}$$

Solving this system gives

$$\Delta x = \begin{bmatrix} -\frac{1}{6} \\ \\ \frac{3}{4} \end{bmatrix}, \quad \Delta s = -\frac{5}{12}, \quad \Delta y = -\frac{3}{4}, \quad \Delta t = \begin{bmatrix} -\frac{3}{2} \\ \\ -\frac{3}{4} \end{bmatrix}.$$

Hence we find $\beta = \frac{4}{3}$: since $\frac{9}{10}\beta > 1$, we take a full Newton step, resulting in the new approximate solution

$$x = \begin{bmatrix} \frac{5}{6} \\ \\ \frac{7}{4} \end{bmatrix}, \quad s = \frac{7}{12}, \quad y = \frac{9}{4}, \quad t = \begin{bmatrix} \frac{3}{2} \\ \\ \frac{1}{4} \end{bmatrix}.$$

The primal-dual interior point method that we have described is very compact to implement. We include a MATLAB implementation at the end of this section, along with some sample output.

We find an interesting contrast when we compare the primal-dual interior point method with the simplex method. We know from the Strong Duality Theorem that three conditions must hold at optimality:

- primal feasibility
- dual feasibility
- complementary slackness.

Consider the revised simplex method. At each iteration, we maintain a current primal-feasible solution, and construct a complementary-slack dual vector. If this vector is dual-feasible, our current solution is optimal; otherwise, we make an improvement. The dual simplex method is analogous: at each iteration, we maintain a current *dual*-feasible solution, and construct a complementary-slack primal vector to check optimality. By contrast, at each iteration of the primal-dual method, we maintain both a primal-feasible and a dual-feasible solution, and we make an improvement as long as complementary slackness is not satisfied sufficiently accurately. Each of the three methods constructs iterates satisfying two out of the three conditions for optimality, and uses the third to verify optimality or improve.

A primal-dual interior point algorithm

```
function [x,y] = pdalgo(A,b,c,x0,y0,mu0,alpha)
%
% Todd's primal-dual algorithm for
% max c'x : Ax <= b, x >= 0,  with dual
% min b'y : A'y >= c, y >= 0.
% Needs strictly feasible initial x0, y0,
% and initial positive mu0.
% Takes step sizes up to proportion alpha
% (in (0,1)) of way to boundary.
% Stops when duality gap <= 10^-6.
%
[m,n] = size(A);
mumin = 10^(-6) / (m+n);
x = x0; y = y0; mu = mu0;
totalsteps = 0;
while (mu > mumin),
    mu = mu / 10;
    k = 0;
    s = b - A*x; t = A'*y - c;
    error = max(max(abs(x.*t - mu)),max(abs(y.*s - mu)));
    while (error > mu/10),
        k = k+1;
        [xnew,ynew] = iter(A,b,c,x,y,mu,alpha);
        x = xnew; y = ynew;
        s = b - A*x; t = A'*y - c;
        error = max(max(abs(x.*t - mu)),max(abs(y.*s - mu)));
    end;
    fprintf(' number of steps = %2.0f\n', k);
    fprintf(' current mu = %3.2e\n', mu);
    totalsteps = totalsteps + k;
end;
totalsteps
```

The Newton iteration

```
function [xnew, ynew] = iter(A,b,c,x,y,mu,alpha)
%
% take iteration of Todd's primal-dual ip algorithm
%

[m,n] = size(A);
s = b - A*x; t = A'*y - c;
if ((min(x) <= 0) | (min(y) <= 0) |
        (min(s) <= 0) | (min(t) <= 0)),
    fprintf('infeasible');
    return;
end;
invt = 1./t; invy = 1./y;
Mat = A * diag(invt.*x) * A' + diag(invy.*s);
rhs = mu * (A * invt + invy) - b;
dy = Mat \ rhs;
dt = A' * dy;
dx = mu * invt - invt .* x .* dt - x;
ds = -A * dx;
minr = min([dx;ds;dy;dt] ./ [x;s;y;t]);
stepmax = alpha * (-1/minr);
step = min(stepmax, 1);
xnew = x + step * dx; ynew = y + step * dy;
snew = s + step * ds; tnew = t + step * dt;
```

Sample interior point output

```
>> A = [1 0; 0 2; 3 2];
>> b = [4; 12; 18];
>> c = [3; 5];
>> x0 = [1;1];
>> y0 = [2;2;2];
>> mu0 = 10;
>> alpha = .9;
>> [x,y] = pdalgo(A,b,c,x0,y0,mu0,alpha)
 number of steps =  4
 current mu = 1.00e+000
 number of steps =  2
 current mu = 1.00e-001
 number of steps =  1
 current mu = 1.00e-002
 number of steps =  1
 current mu = 1.00e-003
 number of steps =  1
 current mu = 1.00e-004
 number of steps =  1
 current mu = 1.00e-005
 number of steps =  1
 current mu = 1.00e-006
 number of steps =  1
 current mu = 1.00e-007

totalsteps = 12

x =
    2.0000
    6.0000

y =
    0.0000
    1.5000
    1.0000
```

13 Column generation methods

We return to the question of solving large-scale integer programs and their linear programming relaxations. We have already studied a technique for solving large-scale (but still compactly represented) linear programs by constraint generation methods. In constraint generation methods, we ignored many of the explicitly given constraints, solved the remaining LP, and then checked if the resulting optimal solution satisfied all of the constraints that were ignored; if so, the solution is optimal for the entire LP (by the relaxation principle); if not, then there is some constraint that is violated by the current optimum which can be added to the LP, and then we resolve.

Now we will focus on an approach dual to constraint generation, known as column generation. We will consider the cutting stock problem and the "pattern" formulation that was introduced in recitation section earlier. This problem was concerned with the efficient production of specified demand for given widths of paper rolls (called *finals*), so as to minimize the number of original longer rolls (called *raws*) used. The input consists of a number of raws of fixed width (for example, 100 inches), and there is is a demand for a given number of different widths of finals: for example, 97 rolls of width 45 inches, 610 rolls of width 36 inches, 395 rolls of width 31 inches, and 211 rolls of width 14 inches. The goal is to cut all of these finals from as few raws as possible. (This example is from the textbook by Chvatal on linear programming.)

If one thinks about cutting one raw, then we specify the number of finals of width 45, the number of width 36, the number of width 31, and the number of width 14. If those four numbers are given by the vector (f_1, f_2, f_3, f_4), then for this pattern to be feasible, given the raw is of width 100, we have that

$$45f_1 + 36f_2 + 31f_3 + 14f_4 \leq 100.$$

So, for example, possible vectors include

$$(2,0,0,0),\ (0,2,0,0),\ (0,0,3,0),\ (0,0,0,7),\ (1,1,0,1),\ (0,2,0,2),\ldots$$

In total, there are exactly 37 ways to specify a (non-trivial) pattern for cutting a raw. Some might seem not so useful. At first, it might seem wasteful to use (0,0,0,5), for example, since one might instead cut 7 finals of width 14, but this lesser pattern might still get used to complete a packing nonetheless (if, as one example, there were only 5 finals required of width 14). We can

build a table of these patterns as a matrix $A = (a_{ij})$, where (the transpose of) each feasible vector is a column of A: the jth pattern corresponds to the jth column of A, where (f_1, f_2, f_3, f_4) is the jth of those 37 patterns, and

$$a_{1j} = f_1,\ a_{2j} = f_2,\ a_{3j} = f_3,\ \text{ and } a_{4j} = f_4.$$

That is, a_{ij} is the number of finals of the ith width produced by using the jth pattern to cut a raw.

Let N be the number of distinct patterns. (We have $N = 37$.) Let b_i denote the demand for finals of the ith width, w_i. We can construct an integer programming formulation for our optimization model as follows. Let x_j be a decision variable that specifies the number of raws that are to cut according to the jth pattern. Then, using as few raws as possible, while meeting the specified demand can be expressed as: minimize $\sum_{j=1}^{N} x_j$ subject to

$$\sum_{j=1}^{N} a_{ij} x_j = b_i,$$

for each final type i, where we also have that $x_j \geq 0$ and integer, $j = 1, \ldots, N$.

We shall focus on the question of solving the linear relaxation of this integer program. This is a very powerful relaxation, in that it produces an exceptionally good lower bound on the optimal value for the integer program. For example, for our input, the LP optimum is to cut 48.5 raws by the pattern (2,0,0,0), 105.5 raws by the pattern (0,2,0,2), 100.75 raws by the pattern (0,2,0,0), and 197.5 raws by the pattern (0,1,2,0), for a value of 452.25. How close is this to the integer optimal value? If we simply round up each of these fractions obtained for the patterns to the next integer, then we get a collection of patterns that clearly meets (and even exceeds) the demand, and has objective function value 454. We can do even better by rounding down (thereby using 48+105+100+197=450 raws), and then seeing how many finals of each width remain to be cut. In this case, we would still need to produce 1 final of width 45, 3 finals of width 36, 1 final of width 31, and 1 final of width 14. The patterns (1,1,0,1), (0,2,0,0), and (0,0,1,0) suffice to complete a solution using 453 raws. Since the LP relaxation has optimal value greater than 452, we know that the IP optimal value is at least 453. Hence, the solution we constructed using 453 raws is optimal!

So we know that computing the optimal solution to the linear programming relaxation is particularly useful, either to construct a good (and possibly

optimal) solution from it, or as a component of branch-and-bound to solve the IP (as we did in recitation section). However, with just 4 widths of finals, we had 37 distinct patterns. What if there were 100 different widths required for some input? The number of patterns, and hence the number of variables in the LP would become enormous. However, the number of rows in this LP is small: just the number of different widths of finals. The basic tool for solving such LPs is one from the previous semester: the revised simplex method.

Here is a brief recap of the revised simplex method. We are interested in solving the linear program: minimize $z = c^T x$ subject to $Ax = b$, $x \geq 0$. (In our case, each component of c is 1.) In each iteration, we partition our variables into the basic set of variables B, and the nonbasic variables N. This allows us to rewrite our constraints as $A_B x_B + A_N x_N - b = 0$. If we multiply these equations by dual variables y^T, we can re-express the objective function z as

$$z = c^T x - y^T(A_B x_B + A_N x_N - b) = (c_B^T - y^T A_B)x_B + (c_N^T - y^T A_N)x_N + y^T b.$$

The steps of each iteration of the revised simplex can be summarized as follows:

1. Let $\bar{x}$ denote the current basic feasible solution. (We will have to initialize this somehow, but this is simply $\bar{x}_B = A_B^{-1} b$ and $\bar{x}_N = 0$.)

2. Compute the current values for the dual variables. We set them so as to ensure that the objective function is expressed solely in terms of the non-basic variables; that is, the coefficient for each basic variables is equal to 0. So we must solve $c_B^T - y^T A_B = 0$ for y. (Alternatively, one can think of this as ensuring the complementary slackness conditions are satisfied.) Let $\bar{y}$ denote the resulting solution.

3. Check if there is a variable with a negative reduced cost. That is, check if there exists some k such that $c_k - \bar{y}^T A_k < 0$ where A_k is the kth column of A (and x_k is clearly a nonbasic variable).

4. Compute an update for column k: solve the equation $A_B d = A_k$ for the vector d.

5. Use the vector d to compute the ratio test: let

$$t = \min\{\frac{\bar{x}_i}{d_i} : \ d_i > 0, \ i \in B\}.$$

6. Compute the update $\tilde{x}_k = t$, $\tilde{x}_j = 0$, $j \in N$, $j \neq k$, and $\tilde{x}_B = \bar{x}_B + td$.

If in Step 2, there does not exist a column with a negative reduced cost, then the current solution is optimal.

How is this method affected by an enormous number of columns? In Steps 1, 3, 4, and 5, all calculations deal with subsystems of size equal to the size of the basis. That is equal to the number of distinct widths that we have for our input, since that is the number of linearly independent constraints that we have. So it is only in Step 2 that we must check something for each column. The straightforward way is to compute $c_k - \bar{y}^T A_k$ for each column k, and check if each is nonnegative. However, suppose that we had a "black box" to find a column k with negative (if one exists) by solving an auxiliary optimization problem, exactly analogous (and dual) to the kind of black box we used to check constraint violation in our constraint generation methods. This could take care of Step 2 without writing all of the columns explicitly.

Let us consider the cutting stock LP relaxation, and explore the specifics of the computation that is required in Step 2. We wish to determine whether there exists a column k such that $c_k - \bar{y}^T A_k < 0$. But what is one such column A_k? It is (the transpose of) a pattern f for cutting one raw in a feasible manner. That is, we want to decide if there is some feasible pattern $f = (f_1, f_2, f_3, f_4)$ such that

$$\bar{y}_1 f_1 + \bar{y}_2 f_2 + \bar{y}_3 f_3 + \bar{y}_4 f_4 > 1,$$

where the 1 is the coefficient of each x_j in our original LP. In words, if we get profit $\bar{y}_i$ for each of the f_i finals of width w_i we cut from one raw, can we make more than one unit of profit for the finals produced from one raw?

This is a knapsack problem in which the dual variables $\bar{y}_i$ are the values, and the widths w_i are the weights, and the length of each raw is the weightlimit of the knapsack! Thus, to complete Step 2, we do not need to check each column one at a time, but rather we need only solve one input to the knapsack problem. If the optimal solution to the knapsack problem input has value greater than 1, then the optimal pattern corresponds to a variable of negative reduced cost that should be brought into the basis. If the optimal solution to the knapsack problem input has value at most 1, then the current solution is optimal (for the LP).

One final point: we need a starting basic feasible solution. This particularly simple to obtain here. Nothing like phase I is required. Consider the patterns that produce only one width of final, producing as many as possible

per raw. For our input, that corresponds to the patterns: (2,0,0,0), (0,2,0,0), (0,0,3,0), and (0,0,0,7), which has the corresponding basic feasible solution with these variables, respectively, set to 48.5, 305, 131 2/3, and 30 1/7.

14 One final optimization model

Suppose that a network of hospitals has obtained funding to add burn centers at k of its hospitals. Each burn center will serve all of the patients at the hospital at which it is located, as well as all other hospitals within the network for which it is the closest burn center. Associated with each burn center is a helicopter used to transport patients from the other hospitals that it serves. As this is an emergency care facility, what is most important is that each hospital is as close as possible to its burn center; in fact, any selection of hospitals to be burn centers is evaluated in terms of the maximum time needed to transport someone from a hospital to its assigned burn center. Of course, we wish to select where to install the burn centers so as to minimize this quantity.

First, we shall give an integer programming formulation of this problem. There are n hospitals in the network, and the input data for the problem consists of the times t_{ij} needed to transport someone at hospital j to a burn center at hospital i. (We may assume that $t_{ij} = t_{ji}$ for each pair of hospitals i and j.) We introduce a decision variable

$$y_i = \begin{cases} 1 & \text{open a burn center at hospital } i \\ 0 & \text{don't open a burn center at hospital } i \end{cases}$$

for each $i = 1, \ldots, n$, and also decision variables

$$x_{ij} = \begin{cases} 1 & \text{if burn center at hospital } i \text{ serves hospital } j \\ 0 & \text{if burn center at hospital } i \text{ doesn't serve hospital } j \end{cases}$$

for each $i, j = 1, \ldots, n$. Finally, we have a variable z that will capture the objective function (for our standard min-max trick). We need that each hospital is assigned to be served by some burn center (possibly at itself):

$$\sum_{i=1}^{n} x_{ij} = 1, \text{ for each } j = 1, \ldots, n.$$

We need that exactly k burn centers are opened:

$$\sum_{i=1}^{n} y_i = k.$$

We need that if a hospital j is assigned to be served by a burn center at hospital i, then indeed a burn center was opened at hospital i:

$$x_{ij} \leq y_i, \quad \text{for each } i = 1, \ldots, n, \ j = 1, \ldots, n.$$

And we need that the variables are binary:

$$0 \leq x_{ij}, y_i \leq 1, \ \text{integer, for each } i = 1, \ldots, n, \ j = 1, \ldots, n.$$

But now to capture the objective function, we need to minimize the maximum of all travel times that are used: those values are $t_{ij}x_{ij}$, and so if we constrain

$$t_{ij}x_{ij} \leq z, \quad \text{for each } i = 1, \ldots, n, \ j = 1, \ldots, n,$$

and then wish to minimize z, we have an integer programming formulation of our problem.

One important element in the solution of real-world integer programming solving is the ability to get good feasible solutions. What might one do for this problem to get a good solution? There are a number of different approaches that one might try.

- Greedy algorithms: gradually build a feasible solution, adding one element at a time, and for each selection, make the best possible selection. There are often many possible implementations of greedy strategies for the same optimization problem. For our model, we could keep track of the hospital that is currently worst off (that is, furthest from its closest burn center) and install a burn center at that location. Repeat until k burn centers have been opened.

- Local improvement algorithms: devise a notion of slightly changing the current solution (for example, for the burn center problem, we can add a burn center at one new hospital while removing one from another), and then repeatedly try to find a "neighboring" feasible solution that costs less than the current one. When no "local" change improves the solution, then we have obtained what is called a "local optimum"

(which need not be the optimal solution) that is often a pretty good approximation to the optimum.

There are occasions when the optimization problem is "special" enough that one can prove that such a local improvement strategy finds an optimal solution. In fact, the previous semester was largely about such a problem, and the corresponding algorithm! The Simplex method is a local improvement procedure, in which the local change is making one non-basic variable basic, while making one basic variable non-basic. Or more geometrically, we are moving to a new corner point of the feasible region by traversing an "edge" of the boundary of feasible region. The proof of correctness of the algorithm in effect proves that a locally optimal solution is the so-called "global" optimum solution.

- Linear programming + rounding: another common approach to obtaining good solutions to an integer program is one that we have seen several times in this course; solve the linear programming relaxation, and then cleverly round the fractional solution to a nearby integer one that is "nearly" as good as the fractional optimum.

- Randomization: one surprisingly useful tool to use in computing good solutions is to allow the algorithm doing this to "toss coins". This important algorithmic paradigm can be used with each of the previous three approaches. For example, with a greedy algorithm, rather than just computing the best next addition, one might compute the top 3 best next additions, and then randomly select among the 3 as the one to try. Why does this make a greedy algorithm better? If one runs the procedure once, it probably only makes it worse. However, greedy methods typically run very fast, so we can run the entire procedure 100 times, or 1000 times, and then take the best solution found among all of those attempts.

  A similar modification can be applied to local improvement algorithms. One can view a local improvement algorithm as following a path in an (ENORMOUS) graph of all feasible solutions, where each node corresponds to one feasible solution, and two nodes are adjacent if the corresponding feasible solutions are "neighboring" solutions for our local improvement algorithm. A local improvement algorithm follows a path in this graph in which the solutions are always improving in objective function value (or perhaps we move to the neighbor with the most

improvement). A randomized version of this merely chooses a neighbor at random and then makes the decision whether to move to this new solution or to try another random neighbor, based on a coin flip. Exactly how this decision works leads to many different algorithms. One approach, known as *simulated annealing*, always moves to the new solution if the random neighbor has a better objective function value, but moves to it with probability decreasing exponentially fast depending on how much worse the new solution is (and then the base of the exponent gradually increased so eventually one is just doing a local improvement algorithm that insists on improvement at each step).

Finally, we can use randomization to round the fractional solution to a linear programming relaxation to help generate an integer solution. Consider the integer programming formulation for the burn center problem. Suppose that $(\bar{x}, \bar{y})$ is an optimal solution to its linear programming relaxation. Each value $\bar{y}_i$ is between 0 and 1. Suppose we interpret this as a probability, and for each i, we independently flip a biased coin with a probability of $\bar{y}_i$ of heads, and $1 - \bar{y}_i$ of tails; we open a burn center if the outcome is a heads. What is the expected number of centers opened? A simple calculation shows that it is $\sum_{i=1}^{n} \bar{y}_i$, which is equal to k. Of course, we might open more centers, or fewer centers, from any set of coin flips, but then we can use a greedy-like strategy to either add or delete centers as needed. This general approach is called *randomized rounding*.

One aspect of Operations Research, as opposed to say, pure mathematics, is that the frontier of research is much closer to the subjects that you learn as part of your undergraduate curriculum. Not only are subjects taught that were research papers only a few years ago (such interior point methods for linear programming, or modern cutting plane methods for integer programming), but one is in a position to understand issues that are the subject of current research; and one can even start to think about them without nearly the same depth of background needed in more classically based areas. For example, last class we discussed how to use the optimal solution for the linear programming relaxation to the cutting stock problem as a means for obtaining a good integer solution. It is the subject of speculation that it might be possible to design an algorithm that *provably is guaranteed* to produce an integer solution that uses at most a couple more raws than the lower bound implied by LP relaxation. If one could design such an algorithm, that

would be a result that, for example, would make a spectacularly brilliant PhD thesis.

Cornell is a research university. One implication of this fact is that the faculty that teach courses are not simply lecturers, but are also working to help advance the state of the art. So, in this last piece of the course, I thought that I would give an example of the sort of research that I (D.S.) have been involved with that ties directly in to the optimization model covered today, the burn center model.

Suppose that we consider the natural class of inputs in which the travel times satisfy the so-called *triangle inequality*. That is, for any triple of hospitals i, and j, and k, the time to go directly from hospital i to k, is at most the travel time to go from i to j, and then to go from j to k: $t_{ik} \leq t_{ij} + t_{jk}$. These inputs are called metric inputs. Consider the specific version of the greedy algorithm discussed earlier. That is, start by opening any one burn center (say, hospital number 1). Then, in each iteration, we identify the hospital that is currently worst off, in that it is currently furthest from its nearest burn center, and open the next burn center at that hospital. One can prove the following guarantee:

For any metric input to the burn center problem, the greedy algorithm always finds a solution which has objective function value at most twice the optimal value.

(This was a result in my PhD thesis.) The reasoning behind this guarantee is not so hard, so we'll give it now.

Let $S^* = \{j_1, \ldots, j_k\}$ denote the set of hospitals selected in the optimal solution, and let r^* denote the corresponding objective function value, that is, the optimal value. We know that each hospital is within a distance of r^* of some selected hospital in S^*. This solution partitions the hospitals into clusters $V_1, \ldots, V_k$, where each hospital j is placed in V_i if it is closest to j_i among all of the points in S^* (and ties are broken arbitrarily). Each pair of points j and j' in the same cluster V_i are at most $2r^*$ apart: by the triangle inequality, the distance $t_{jj'}$ between them is at most the sum of t_{j,j_i}, the distance from j to j_i, plus $t_{j_i,j'}$, the distance from j_i to j'; since t_{j,j_i} and t_{j',j_i} are each at most r^*, we see that $t_{jj'}$ is at most $2r^*$.

Now consider the set of hospitals S selected by the greedy algorithm. If one center in S is selected from each cluster of the optimal solution S^*, then every hospital is clearly within $2r^*$ of some selected burn center in S. However, suppose that the algorithm selects two points within the same

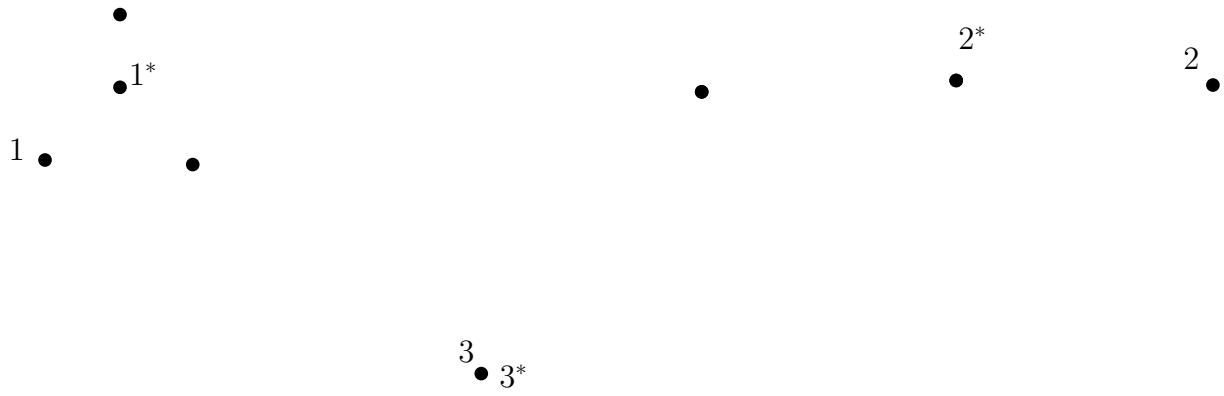


Figure 1: An input to the burn center problem where $k = 3$ and the distances are given by the Euclidean distances between points. The execution of the greedy algorithm is shown; the nodes 1,2,3 are the nodes selected by the greedy algorithm, whereas the nodes $1^*, 2^*, 3^*$ are the three nodes in the optimal solution.

cluster. That is, in some iteration, the algorithm selects a hospital $j \in V_i$, even though the algorithm had already selected another hospital $j' \in V_i$ in an earlier iteration. Again, the distance between these two points is at most $2r^*$. The algorithm selects j in this iteration because it is currently the furthest from the points already in S. Hence, all points are within a distance of at most $2r^*$ of some burn center already selected for S. Clearly, this remains true as the algorithm adds more burn centers in subsequent iterations, and we have proved the guarantee.

Observe that the input in Figure 1 shows that this analysis is tight. In fact, there is substantial evidence (taught in CS 482) that one can prove that no efficient algorithm has a better guarantee for this problem.