Chapter 10: Network Flow Programming

Linear programming, that amazingly useful technique, is about to resurface: many network problems are actually just special forms of linear programs! This includes, for example:

- the transportation problem,
- the transshipment problem,
- the assignment problem,
- the maximum flow and minimum cut problem,
- the shortest route problem,
- the shortest route tree problem,
- etc.

We will see in this chapter how these problems can be cast as linear programs, and how the solutions to the original problems can be recovered. We will also see that there are specialized algorithms that can solve network linear programs many times faster than if they are solved by the general-purpose simplex method. Formulating and solving network problems via linear programming is called *network flow programming.*

Any network flow problem can be cast as a *minimum-cost network flow program*. A min-cost network flow program has the following characteristics.

Variables. The unknown flows in the arcs, the x_i, are the variables.

Flow conservation at the nodes. The total flow into a node equals the total flow out of a node, as shown in Figure 10.1(a) for example. It makes things easier later if we follow the convention of writing the flow conservation equation at a node as:

$$\sum_{\text{outflows}} x_j - \sum_{\text{inflows}} x_j = 0$$

Source and sink nodes. Some nodes are connections to the environment surrounding the network. At these entryway nodes, there may be a net gain of flow into the network (source node), or a net loss of flow out of the network (sink node). To emphasize that flow conservation still holds at source and sink nodes, a dashed "phantom" arc can be shown on the network diagram. The phantom arc will be an inflow for a source node, and an outflow for a sink node. See the examples in Figures 10.1(b) and 10.1(c).

Now we use a similar convention for writing the flow conservation equation at the source or sink node:

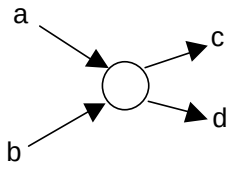
$$\sum_{\text{outflows}} x_j - \sum_{\text{inflows}} x_j = b_i$$

When written this way, b_i is a positive constant for a source node, and a negative constant for a sink node. The magnitude of b_i is the amount of flow in the phantom source or sink arc. Note also that the relationship may not be an equality relationship: inequalities are common for sources and sinks. For example, the flow of water exiting a supply network must be at least 100 liters per minute, or the flow of oil entering a refinery network must not exceed 10,000 barrels per day.

When all of the flow conservation equations (or inequalities) are written following the *outflows – inflows* convention, it is easy to see what type of node is associated with each relationship by looking at the value of the right hand side constant. The constant will be zero for a simple flow conserving node, positive for a source node, and negative for a sink node.

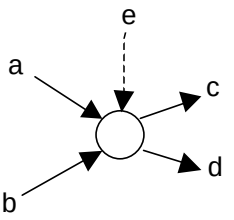
Bounds on the arc flows. There may be upper and lower bounds on the flows in the arcs (i.e. the variables in the model. $x_j \geq b_j$ is a lower bound on an arc flow, and $x_j \leq b_j$ is an upper bound on an arc flow. For example, the maximum flow of water through a particular pipe is limited because of the pipe diameter and interior roughness, so an upper bound is applied: $x_j \leq 25$ *l/sec*.

Upper bounds are easy to understand, but why might there be a nonzero lower bound? This might represent the minimum required production rate at a factory (e.g. 250 vehicles per day required at the output arc of an automobile plant) or a minimum flow rate through steam piping to prevent condensation.



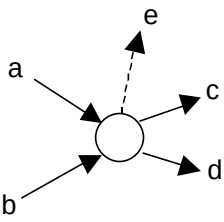
c + d - a - b = 0

(a) flow conserving node



c + d - a - b = e

(b) source node



c + d - a - b = -e

(c) sink node

Figure 10.1: Flow conservation at nodes.

The default arc flow bounds are a lower bound of zero and no upper bound. It is not unusual for almost all of the arcs in a network model to have the default bounds, with only a few arcs having specified upper or lower bounds, typically those at the "edges" of the network, representing, say, upper limits on the rates of raw materials flowing into the network and lower limits on the production the rate at the main outflow from the network.

Some modeling systems will allow a negative lower bound on an arc flow. My strong advice is *don't do this!* The meaning of a negative flow is a *backwards* flow in the arc. This immediately destroys the best feature of a network model, the intuitive understanding of the system that you gain by looking at the network diagram. You expect the flows to follow the directions indicated by the arrowheads, but the arrowheads may lie if you permit negative flows. There are other ways to accommodate two-way flow, if necessary, such as pairing oppositely oriented arcs between two nodes.

Cost per unit of flow. There is a cost per unit of flow, c_j, associated with each arc. In many network models, the cost per unit of flow is zero for most of the arcs, with costs being typically associated with arcs at the "edges" of the network. The default value of c_j is zero.

Objective function. In a minimum cost network flow problem, the objective is to find the values of the variables (the x_j) that minimize the total cost of the flows over the network:

$$\text{minimize} \sum_{arcs} c_j x_j$$

Of course, the solution must respect all of the constraints: flow conservation at the nodes, and the upper and lower flow bounds on the arcs.

As we will see in the remainder of this chapter, an astonishing array of network problems can be cast as minimum cost network flow programs.

From Network Diagram to Linear Program

A huge attraction of network models is the immediate intuitive understanding provided by the diagram. In fact, given a properly labeled diagram, the conversion to a minimum cost network flow linear program is automatic. There are some commercial modeling systems that support this direct conversion.

There are three parameters associated with each arc: the lower flow bound, the upper flow bound, and the cost per unit of flow. The arc labeling convention that we will use shows a triple of numbers in square brackets, [*l, u, c*]. *l* is the lower bound on the flow in the arc, with a default value of zero if not explicitly specified; *u* is the upper bound on the flow in the arc, with a default value of infinity if not explicitly specified; *c* is the cost per unit of flow in the arc, with a default value of zero if not explicitly specified. For example, an arc having a lower flow bound of zero, and upper flow bound of 25, and a cost per unit of flow of $6 would be labeled [0, 25, 6].

Source and sink node behavior is controlled by the label on the phantom arc associated with the node. If the upper and lower flow bounds on the phantom arc are identical, then the node relationship is an equation, but if the upper and flow bounds on the arc differ, then the node relationship is an inequality.

Consider the network diagram in Figure 10.2 for example. The phantom arcs on the 3 source and sink nodes are fully labeled. Node A is a source of up to 12 units of flow at a cost of $5 per unit of flow. Node C is a sink of up to 4 units of flow at an *income* of $6 per unit of flow – the negative cost per unit of flow means income. Node D is a sink of exactly 8 units of flow, but with no cost or income associated with that flow. The remaining arcs are also labeled following the convention. Note that arc 4 has a positive lower bound.

Given the fully labeled diagram, writing the associated linear program is a mechanical process: write the minimum cost objective function, then the node conservation equations, the arc flow bounds, and the nonnegativity constraints. The linear program constraints associated with Figure 10.2 are:

node A:	$x_1 + x_2 + x_3 \leq 12$
node B:	$x_4 - x_1 = 0$
node C lower bound:	$x_5 - x_2 \geq -4$
node C upper bound:	$x_5 - x_2 \leq 0$
node D:	$-x_3 - x_4 - x_5 = -8$
flow bound arc 2:	$x_2 \leq 6$
flow bound arc 3:	$x_3 \leq 3$
flow bound arc 4:	$x_4 \geq 4$
nonnegativity:	$x_1, x_2, x_3, x_4, x_5 \geq 0$

The minimum cost objective function can be written as:

$$\text{minimize } 5A - 6C + 2.5x_3 + 3.7x_4 + 0.5x_5$$

where A and C represent the nonnegative flows in the phantom arcs associated with nodes A and C. But those two variables are not needed because the phantom arc flows can be rewritten in terms of the other flows incident on (i.e. touching) the associated nodes, as follows:

$$A = x_1 + x_2 + x_3 \text{ and } C = x_2 - x_5$$

These relationships are substituted into the objective function to remove the variables A and C from the model entirely. The final version of the objective function is then:

$$\text{minimize } 5x_1 - x_2 + 7.5x_3 + 3.7x_4 + 6.5x_5$$

As we have seen, the network diagram contains all the information needed to derive an associated linear programming model via a straightforward mechanical writing of the constraints and objective function. For this reason, we will assume from here onward that a properly labeled network diagram *is* the formulation.

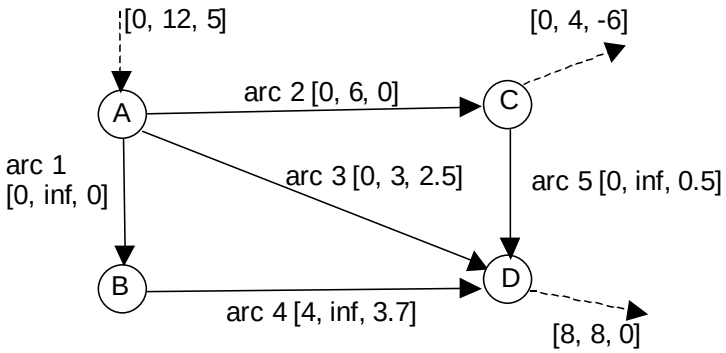


Figure 10.2: A fully labeled network diagram.

The linear programming version of the network model has some very interesting properties. Look at the left hand sides of all of the constraints. What do you notice? All of the coefficients there are either 0, +1, or –1. This is because all of the node relationships are simple summations of flows, and the remainder of the constraints are simple bounds. But this fact has some very important consequences.

The most important consequence is that all of the pivot steps during the linear programming solution consist of simple addition and subtraction: there is no need for multiplication. Now this doesn't seem very dramatic, but at the very least it eliminates floating-point computer operations in favor of much faster arithmetic operations.

But it gets better: those simple addition/subtraction operations can actually be replaced by *logical* operations in a different solution algorithm known as the *network simplex method*, which can be hundreds of times faster than the ordinary simplex method when applied to a network problem. In fact, the solution times for networks are so much faster than regular linear programs that some sophisticated linear programming solvers will scan the LP for network portions. If it finds that parts of the LP are in network form, it has a way of solving those portions separately using the fast network simplex method, and then tying the solution back into the rest of the linear program in an iterative manner. Overall solution time is reduced in this way.

The second important consequence is this: if all of the constraint right hand side values are integers (as in our example), and if all of the pivot operations are simple additions and subtractions, then we can guarantee that the solution values of the variables at the optimum will also all be integers. This is known as the *unimodularity* property. This is extremely useful in solving certain types of integer programming problems, such as assignment problems. Most integer programming problems must be solved using much slower solution algorithms, so it is very fortunate that a fast technique such as linear programming can be used on some problems.

The Transportation Problem

The transportation problem is simple in form, but surprisingly useful in practice. It consists of a set of sources of some product (e.g. factories producing canned vegetables), which are directly linked to sinks of the product (e.g. markets in various cities which buy the canned vegetables). Each link has an associated cost per unit of flow (e.g. cost per delivered truckload in this case).

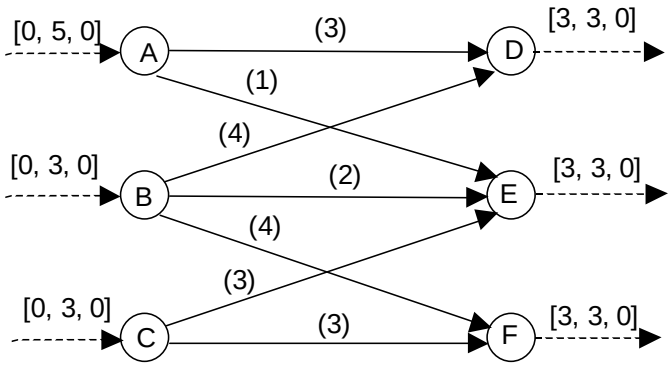


Figure 10.3: A simple transportation model. The transportation arcs are labeled with the cost per unit of flow.

Consider the example in Figure 10.3, which has three factories (A, B, and C) shipping to three markets (D, E, and F). The "transportation arcs" are the arcs which directly connect the sources (factories) to the sinks (markets); these are labeled only with the cost per unit of flow because the lower flow bounds are all at the default of zero, and the upper flow bounds are all at the default of infinity.

The question in this case is: how many truckloads per day should be produced at each factory and shipped to each sink to meet the market demands at minimum total cost? After solution, the flows in the transportation arcs will be known; hence the number of truckloads to ship from each factory to each market will be known.

It's always good to give a network model of this type a simple "idiot test" at first glance. In Figure 10.3 we see that the factories can produce up to a total of 11 units of flow while the markets demand exactly nine units of flow. This model passes the idiot test: there is sufficient supply to meet the demand. Of course, the model may fail for other reasons, e.g. the demand at a particular market cannot be met from the supply available to it. The linear programming solver will detect any of these problems, and appropriate infeasibility analysis routines can be brought into play.

The Assignment Problem

The assignment problem is a classic that also appears in the integer programming literature. In the usual form of the problem, you need to assign a set of people to a set of tasks. Each person takes a certain number of minutes to do a certain task, or cannot do a particular task at all, and each person can be assigned to exactly one task. How should the people be assigned to the tasks to minimize the total time taken to do all of the tasks?

The data for an assignment problem is often collected in a table, as shown below for example. The number in each cell indicates the number of minutes required for a particular person to do a particular task. The notation "n.a." in a cell indicates that the associated person cannot do the task associated with the cell. It's not obvious how to assign the people to the tasks by simple inspection of the table. For example, you may try looking at each task and simply choosing the best person for that task. But as you can see, person A is the best for tasks 2 and 4 and second-best for task 3. How should the tie be broken? Other ad hoc procedures also soon run into trouble. A more organized approach is needed.

	task 1	**task 2**	**task 3**	**task 4**
person A	n.a.	9	7	13
person B	16	13	8	n.a.
person C	10	n.a.	6	15
person D	11	n.a.	13	17

Surprisingly, the assignment problem can be cast as a transportation problem! Each person is modeled as a source node which introduces exactly one unit of flow into the network, and each task is modeled as a sink node which removes exactly one unit of flow from the network, as shown in Figure 10.4. Each arc has the default upper and lower flow bounds, but the cost per unit of flow is set equal to the number of minutes for the person to do the job. To avoid diagram clutter, each arc is labeled only with the cost per unit of flow.

After the solution of the resulting network linear program, the flows in the arcs (i.e. the values of the variables in the linear program) will be known. The flow in any arc will be exactly zero or exactly one. Why? Isn't it possible to send fractional units of flow and still satisfy all of the source and sink relationships? It's because (i) the unimodularity property restricts the arc flows

to integer values because all of the node equations have integer constants, and (ii) the sources and sink nodes in the model all have inflows or outflows of exactly one unit of flow. Given this, the optimal set of assignments is shown by the arcs that have a positive flow. Each positive-flow arc indicates a person-to-task assignment that should be made. The objective function value gives the minimum total time associated with this assignment.

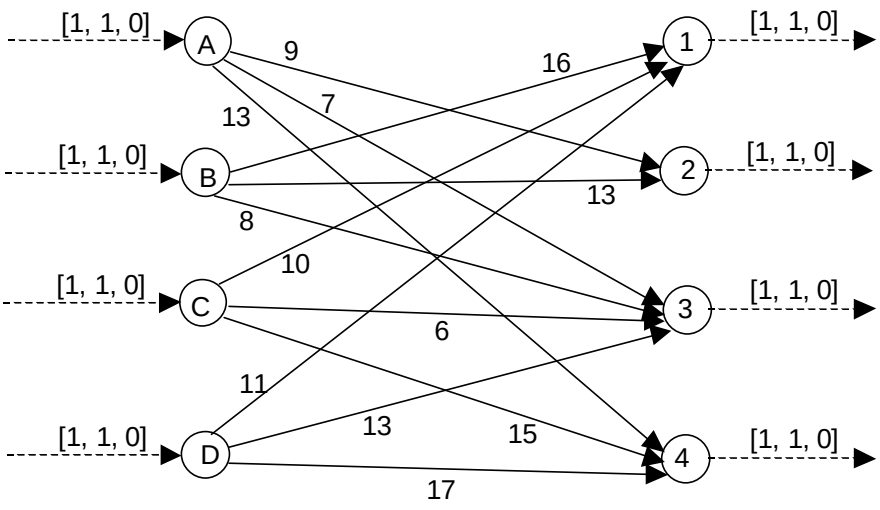


Figure 10.4: The assignment problem cast as a transportation network.

A straightforward variation of the assignment problem is the case in which there are more people than jobs. This is easy to handle simply by making each person the source of *up to* one unit of flow, by labeling the phantom arc associated with each person as [0,1,0].

The Transshipment Problem

This is a variation on the transportation problem in which shipping via intermediate nodes is allowed. In other words, not all of the nodes in the model are sources or sinks; some are simple flow conserving nodes. In addition, the sources and sinks may also transship flow. As in the transportation model, the nodes are assumed to have the default flow bounds. A simple, partially labeled transshipment network is illustrated in Figure 10.5. Node B is a source node that also acts as a transshipment node, node G is a sink node that also acts as a transshipment node, nodes D and E are pure transshipment nodes.

The Shortest Route Problem

Believe it or not, the shortest route problem, previously solved via Dijkstra's algorithm, can also be cast as a minimum cost network flow program, and therefore solved by extremely fast network flow programming codes. It's just a matter of properly labeling the nodes and arcs and interpreting the LP solution. Here are the main points of the problem setup:

- Create the network diagram,
- Label each arc with a lower flow bound of zero, an upper flow bound of infinity, and a cost per unit flow equal to the length of the arc. For example, an arc with a length of 12 kilometers would be labelled [0, ∞, 12].
- Make the origin node a source of exactly one unit of flow, with no cost per unit of flow. The label on the phantom arc should be [1,1,0].
- Make the destination node a sink of exactly one unit of flow, with no cost per unit of flow. The label on the phantom arc should be [1,1,0].

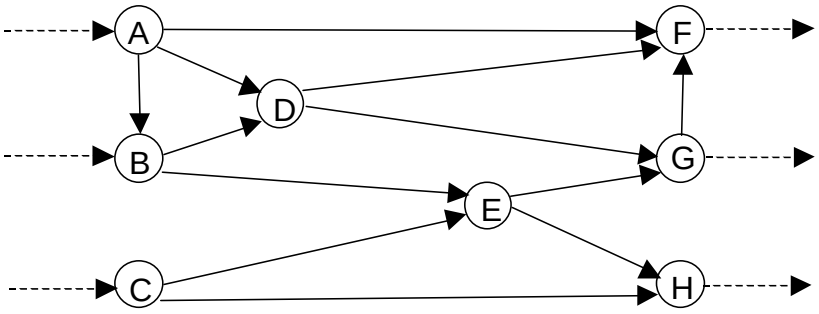


Figure 10.5: An unlabeled transshipment network.

The LP solution will assign flow to some arcs in the network, but not to other arcs. Arcs having a positive flow will be on the shortest route. In fact, if an arc has a positive flow, the flow value will be exactly one unit. Why? Because of the unimodularity property discussed earlier! It's a network flow value in which all of the node equations have integer right hand side constants, hence the network flows will all be integer. But the integers can't be any bigger than 1 because of the limitations placed on the single source node and the single sink node, so all arc flows will be either 0 or 1. And because of the minimum cost objective function, the LP will choose the "cheapest" (i.e. shortest) route to connect the input flow at the origin source node to the output flow at the destination sink node.

So there's no need to write or find an implementation of Dijkstra's algorithm to solve a shortest route problem when you have a linear programming solver at hand.

The Shortest Route Tree Problem

This is related to the shortest route problem, except that there is a single origin node, and many (if not all) of the other nodes in the network are destination nodes. The problem is to find the shortest route from the origin node to every one of the destination nodes. Of course, setting up and solving one shortest route problem for every destination node in the network can do this, but there is a simpler method that solves *all* of the shortest route problems simultaneously.

Again, it's just a matter of properly labeling the nodes and arcs and interpreting the LP solution. Let's assume that there are *n* destination nodes. Here are the main points of the problem setup:

- Create the network diagram,
- Label each arc with a lower flow bound of zero, an upper flow bound of infinity, and a cost per unit flow equal to the length of the arc. For example, an arc with a length of 12 kilometers would be labelled [0, ∞, 12].
- Make the origin node a source of exactly *n* units of flow, with no cost per unit of flow. The label on the phantom arc should be [*n*, *n*, 0].
- Make each destination node a sink of exactly one unit of flow, with no cost per unit of flow. The label on the phantom arc for each destination node should be [1,1,0].

Just as in the shortest route problem, the unimodularity property will permit only integer amounts of flow in an arc. Some arcs will have no flow, and some will have a flow equal to some positive integer less than or equal to *n*, the maximum amount of flow introduced at the origin node. If you mark the arcs that have a positive flow, then the marked arcs will form a tree on the diagram. The shortest route from the origin to each destination node is actually discovered backwards: trace the route from the destination node back to the origin via marked nodes. Because the marked arcs form a tree, there is only one shortest route for each destination node. The routes to several destination nodes may share an arc, which is why the flow in some arcs may be greater than one.

The Maximum Flow and Minimum Cut Problem

This problem too can be solved by proper formulation as a minimum cost network flow model. The main points of the problem setup:

- Create the network diagram,
- Label each arc with a lower flow bound of zero, the upper flow bound associated with the arc, and a cost per unit flow of zero. For example, an arc with an upper flow bound of 25 litres per minute would be labelled [0, 25, 0].
- Make the origin node a possible source of a very large volume of flow, with no cost per unit of flow. The label on the phantom arc should be [0, M, 0], where M represents a very large number.
- Make each destination node a possible sink of a very large volume of flow, with a cost per unit of flow of -1. The label on the phantom arc for the destination node should be [0, M, -1], where M again represents a very large number.

Now what is the effect of this strange labeling? Consider the destination node: it has a cost per unit of flow of –1. Since the objective function will automatically attempt to minimize costs, it will try to get as much flow as possible out via the destination node in order to drive the cost function as negative as possible. This is just like attaching a giant vacuum cleaner to the destination node, which attempts to suck as much flow as it can out of the network. At the other end, the origin node will certainly permit a very large volume of flow to enter the network. However, what limits the total flow is the upper flow bounds on the arc, just as it did in the Ford and Fulkerson algorithm we looked at earlier.

Thus, the minimum cost network flow solution will provide the following information:

- The flow out of the network at the destination node is the maximum total flow through the network.
- The flows in the arcs show a flow pattern that will provide this maximum flow. As usual, there may be other flow patterns that provide the same total flow.

The minimum cut is found in exactly the same way as it was after solving the maximum flow problem via the Ford and Fulkerson algorithm: mark the arcs that are full to capacity, then find a cut that uses only marked arcs.

A final question: what value should the M in the origin and destination nodes be set at? It should be a value that is at least as big as the maximum flow, but as small as possible to avoid any numerical problems in the computer solution. An easy way to find a suitable value is to take the cut-value for any random cut in the network diagram, because we know by the max-flow/min-cut theorem that the cut value will be greater than or equal to the maximum flow in the network.

Generalized Networks

In a *generalized network*, the arcs have *gain factors*. This is a number that multiplies the flow entering an arc to yield the flow leaving the arc. If all of the gain factors on all of the arcs are equal to 1, then this is a normal network. Using gain factors allows you to model a wider range of phenomena. For example, a gain factor of 0.9 might apply to a leaky pipe in a water network that loses 10% of its flow. Can you think of a reason for having a gain factor that is greater than 1? Obviously this represents an increase in the amount of flow leaving the arc as compared to the amount of flow entering the arc.

I asked this question to a graduate class some years back. They thought for a while and finally a student ventured an answer: "Suppose the flow represented chickens being transported to market, and meanwhile the chickens were laying eggs and they were hatching, thereby creating more chickens…". Creative answer. More realistically, a gain factor greater than 1 might be applied when the flow represents the value of an item, the arc represents the transportation of the item between two locations, and the item is worth more on arrival in the destination location.

Gain factors of zero are not allowed. Negative gain factors are allowed, but are not a good idea because they have the effect of reversing the flow at the head end of the arc, which destroys the intuitive meaning associated with the network diagram itself.

There are specialized fast networks for solving networks with side constraints.

Networks with Side Constraints

Sometimes the model is almost completely a network, but there are additional constraints that simply cannot be expressed as network relationships. For example, you may need to add a constraint like $13x_1 - 0.5x_2 + 12x_3 \geq 10$ to your network model. It's easy to see that this is not a network constraint in this case because the coefficients are not all +1 or –1. In a case like this you have a network with side constraints.

Networks with side constraints can of course be solved as ordinary general linear programs, but this sacrifices the solution speed that can be achieved for network models. Fortunately, there are specialized solution methods for networks with side constraints, which are quite fast. In fact, smart solvers will scan a model to see if it has network portions, and if it does, it may apply such specialized solvers. Some of the specialized solution algorithms work by solving the network portions by the fast network solution algorithms and then stitching the network portion together with the non-network portion. This is done numerous times in an iterative manner until the solutions for the two portions converge.

Processing Networks

Processing networks are extremely useful for modeling engineering systems, such as flows through factories. A processing network has regular nodes (constrained only by the usual conservation of flow), and at least one *processing node* in which the flows in the incident arcs are further constrained to have fixed proportions of flows relative to each other. For example, a processing node representing the energy flows in a large industrial boiler is shown in Figure 10.6. The numbers in the figure represent the fixed proportions of the energy flows. As you can see, the boiler is 80% efficient.

Processing nodes are shown as small squares on a network diagram to distinguish them from regular nodes, which are shown as small circles. If there are k incident arcs, then a processing node is represented by k-1 ratio equations. For example, the processing node in Figure 10.6 is completely represented by the following 3 ratio equations:

$$a/b = 1/0.01 \rightarrow 0.01a - b = 0$$

$$a/c = 1/0.21 \rightarrow 0.21a - c = 0$$

$$a/d = 1/0.8 \rightarrow 0.8a - d = 0$$

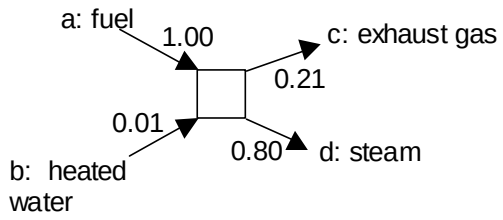


Figure 10.6: Processing node representation of an industrial boiler.

A complete model of a processing network is assembled by writing appropriate ratio equations for each processing node along with the flow conservation equations for the regular nodes. Then the usual flow bounds and minimum cost network flow objective function completes the model.

As you can see, a processing network is nothing more than a network with side constraints added because of the ratio equations generated by the processing nodes. The fast solution algorithms for networks with side constraints can then be used.

In a *flow-conserving processing network* the sum of the inflow proportions equals the sum of the outflow proportions. For example, in Figure 10.6, the inflow proportions sum to 1.01, as do the outflow proportions. This guarantees flow will be conserved in the processing node. However, there are many cases where flow conservation will not hold, especially when the units differ across the various flows incident on a processing node, as in an assembly model as shown in Figure 10.7. Here, the inflow proportions sum to 6, while the outflow proportions sum to 1.

However, this still simply defines a network model with side constraints, and so it can be solved by the usual methods.

As you have seen in this chapter, network models are incredibly versatile modeling tools with an appealing intuitive mapping to the system under study.

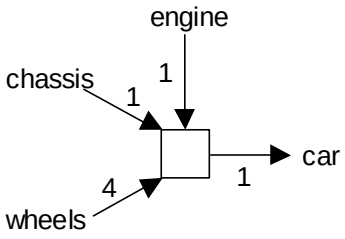


Figure 10.7: Processing node representation of an automobile assembly process