Chapter 12: Integer/Discrete Programming via Branch and Bound

Thus far we have been dealing with models in which the variables can take on real values, for example a solution value of 7.3 is perfectly fine. But the variables in some models are restricted to taking only integer or discrete values. You can assign 6 or 7 people to a team, for example, but not 6.3 people; or you can choose to make a transistor from silicon dioxide or gallium arsenide, but not some mixture. Binary variables are a subset of integer/discrete variables that are restricted to 0/1 values. Binary variables are usually associated with yes/no decisions, e.g. to undertake a project or not.

You will often encounter integer models that look like they can be solved by methods suitable for real-valued variables. For example, it is common to find models in which the objective function and constraints are linear relations defined on integer variables. This looks exactly like a standard linear program, except that the variables cannot take on real values. There is an overwhelming temptation to just solve the problem by standard linear programming and then to round any non-integer variable values to the closest integer value. Don't do this! The following simple example (due to Hillier and Lieberman) shows how misleading this can be.

Maximize $Z = x_1 + 5x_2$
Subject to: $x_1 + 10x_2 \leq 20$
$x_1 \leq 2$
$x_1, x_2 \geq 0$ and integer.

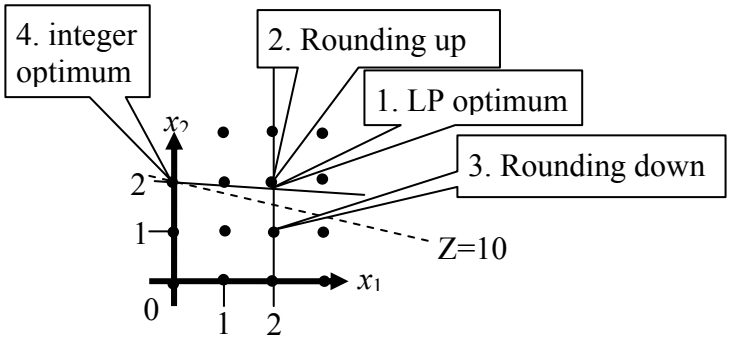

Figure 12.1: Why rounding doesn't work.

The linear programming solution to this problem yields Z=11 at (2, 1.8). The first instinct is to round x_2 to the nearest integer value, i.e. x_2=2. Unfortunately (2, 2) is infeasible by the first constraint. The natural inclination is then to try rounding x_2 in the opposite direction, i.e. x_2=1. This yields a feasible point (2, 1) with Z=7.

However, this is not the optimum point! The integer-feasible optimum point is at (0, 2) where Z=10. Figure 12.1 shows a sketch of this problem. The dots in Figure 12.1 show points at which both x_1 and x_2 have integer values simultaneously.

Notice that the integer optimum point is far away from the LP optimum. Notice also how you can't get to the integer optimum (0,2) by rounding the LP optimum (2, 1.8). The moral of the story is that you simply can't use real-valued solution methods to optimize integer-valued models. Completely different methods are needed.

The first thing you might think of for solving integer-valued problems is to simply enumerate all of the possible solutions and then to choose the best one. This will actually work for very small problems, but it very rapidly becomes unworkable for even small- to medium-size problems, let alone industrial-scale problems. For example, consider a binary 0/1 problem that has 20 variables. This will have 2^{20}=1,048,576 solutions to enumerate, which might be possible to do via computer. Now consider a slightly larger binary problem having 100 variables. Now there are $2^{100}=1.268\times10^{30}$ solutions to enumerate, which is likely impossible, even for a very fast computer. The combinatorial explosion is even worse for general integer variables that can take on even more values than just the two possibilities for binary variables. It is easy to construct problems in which the number of solutions is greater than the total number of atoms in the universe! Enumeration just won't work for most real-world problems – we need a better way of tackling the combinatorial explosion.

The *branch and bound* method is the basic workhorse technique for solving integer and discrete programming problems. The method is based on the observation that the enumeration of integer solutions has a tree structure. For example, consider the complete enumeration of a model having one general integer variable x_1, and two binary variables x_2 and x_3, whose ranges are $1\leq x_1\leq3$, $0\leq x_2\leq1$, and $0\leq x_3\leq1$. Figure 12.2 shows the complete enumeration of all of the solutions for these variables, even those which might be infeasible due to other constraints on the model.

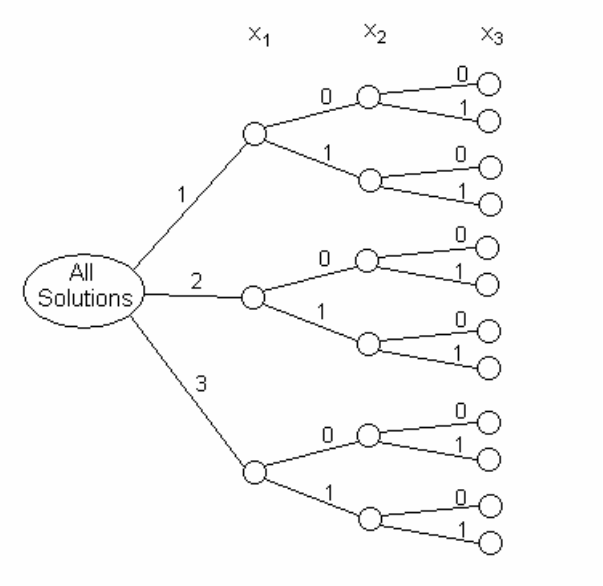

Figure 12.2: A full enumeration tree.

The structure in Figure 12.2 looks like a tree lying on its side with the *root* (or root node) on the left, labeled "all solutions", and the *leaves* (or leaf nodes) on the right. The leaf nodes represent the actual enumerated solutions, so there are 12 of them: (3 possible values of x_1) × (2 possible values of x_2) × (2 possible values of x_3). For example, the node at the upper right represents the solution in which x_1=1, x_2=0, and x_3=0. The other nodes can be thought of as representing sets of possible solutions. For example, the root node represents all solutions that can be generated by growing the tree. Another intermediate node, e.g. the first node directly to the right of the root node, represents another subset of all of the possible solutions, in this case, all of the solutions in which x_1=2 and the other two variables can take on any of their possible values. For any two directly connected nodes in the tree, the *parent* node is the one closer to the root, and the *child* node is the one closer to the leaves.

Now the main idea in branch and bound is to avoid growing the whole tree as much as possible, because the entire tree is just too big in any real problem. Instead branch and bound grows the tree in stages, and grows only the most promising nodes at any stage. It determines which node is the most promising by estimating a bound on the best value of the objective function that can be obtained by growing that node to later stages. The name of the method comes from the

branching that happens when a node is selected for further growth and the next generation of children of that node is created. The *bounding* comes in when the bound on the best value attained by growing a node is estimated. We hope that in the end we will have grown only a very small fraction of the full enumeration tree.

Another important aspect of the method is *pruning*, in which you cut off and permanently discard nodes when you can show that it, or any its descendents, will never be either feasible or optimal. The name derives from gardening, in which pruning means to clip off branches on a tree, exactly what we will do in this case. Pruning is one of the most important aspects of branch and bound since it is precisely what prevents the search tree from growing too much.

To describe branch and bound in detail, we first need to introduce some terminology:

- *Node*: any partial or complete solution. For example, a node that is two levels down in a 5-variable problem might represent the partial solution 3-17-?-?-?, in which the first variable has a value of 3 and the second variable has a value of 17. The values of the last three variables are not yet set.
- *Leaf (leaf node)*: a complete solution in which all of the variable values are known.
- *Bud (bud node)*: a partial solution, either feasible or infeasible. Think of it as a node that might yet grow further, just as on a real tree.
- *Bounding function*: the method of estimating the best value of the objective function obtainable by growing a bud node further. Only bud nodes have associated bounding function values. Leaf nodes have objective function values, which are actual values and not estimates. It is important that the bounding function be an optimistic estimator. In other words, if you are minimizing, it must underestimate the actual best achievable objective function value; if maximizing it must overestimate the best achievable objective function value. You want it to be as accurate an estimator as possible so that the resulting branch and bound tree is as small as possible, but it must err in the optimistic direction. The bounding function is the real magic in branch and bound. It takes ingenuity sometimes to find a good bounding function, but the payoff in increased efficiency is tremendous. Every problem is different.
- *Branching, growing, or expanding a node*: the process of creating the child nodes for a bud node. One child node is created for each possible value of the next variable. For example, if the next variable is binary, there will be one child node associated with the value zero and one child node associated with the value one.
- *Incumbent*: the best complete feasible solution found so far. There may not be an incumbent when the solution process begins. In that case, the first complete feasible solution found during the solution process becomes the first incumbent.

Branch and bound is a very general framework. To completely specify how the process is to proceed, you also need to define policies concerning selection of the next node, selection of the next variable, how to prune, and when to stop. We'll discuss these next.

At any intermediate point in the algorithm, we have the current version of the branch and bound tree, which consists of bud nodes labeled with their bounding function values and other nodes

labeled in various ways that we will see later. The *node selection policy* governs how to choose the next bud node for expansion. There are three popular policies for node selection:

- *Best-first* or *global-best node selection:* choose the bud node that has the best value of the bounding function anywhere on the branch and bound tree. If we are minimizing, this means choosing the bud node with the smallest value of the bounding function; if maximizing choose the bud node with the largest value of the bounding function.
- *Depth-first:* choose only from among the set of bud nodes just created. Choose the bud node with the best value of the bounding function. Depth-first node selection takes you one step deeper into the branch and bound tree at each iteration, so it reaches the leaf nodes quickly. This is one way of achieving an early incumbent solution. If you cannot proceed any deeper into the tree, back up one level and choose another child node from that level.
- *Breadth-first:* expand bud nodes in the same order in which they were created.

Similarly, once a bud node has been chosen for expansion, how do we choose the next variable to use in creating the child nodes of the bud node? The *variable selection policy* governs this choice. There are few standard policies for variable selection. The variables are often selected just in their natural order, though a good variable selection policy can improve efficiency greatly.

We also need to establish policies and rules for pruning bud nodes. As mentioned above, there are two main reasons to prune a bud node: you can show that no descendent will be feasible, or you can show that no descendent will be optimal.

The method for showing that no descendent will be optimal is standard: if the bud node bounding function value is worse than the objective function value for the incumbent, then the bud node can be pruned. This is because the bounding function is an optimistic estimator. Suppose you are maximizing, and the incumbent solution has an objective function value of 87. If the optimistic bounding function overestimate for the bud node has a value of only 79, then you know that no descendent of the bud node will ever exceed 79, let alone 87, and so none of the descendents can ever be optimal. So the bud node is pruned. The same reasoning applies in reverse if minimizing.

Methods for showing that no descendent can ever be feasible vary with the specific problem. In problems that include standard arithmetic constraints, it is sometimes easy to detect this condition. For example, consider a partial solution $(x_1,x_2,x_3,x_4)=(1,1,?,?)$ in an all-binary problem which has the constraint $-10x_1-5x_2+6x_3+4x_4\geq 0$. Now that x_1 and x_2 are both set to 1, there are no possible settings of x_3 and x_4 which will satisfy the constraint. Hence we can deduce that all of the descendents of this bud node will be infeasible, so the node is pruned.

There is one other case in which the expansion of a bud node can be halted: when the best possible value of the objective function obtainable by expansion can be seen directly. This is known as *fathoming* a node. This is sometimes a by-product of the bounding function calculations. Bounding functions often work by solving a simpler problem that is created by ignoring some of the constraints on the real problem. Sometimes the solution to this simpler problem actually does satisfy all of the constraints on the original problem, hence it is the best possible solution for the original problem, and is obtained without expanding the bud node any

further. All that is necessary at this point is to compare this solution to the incumbent: if it is better than the incumbent, then it replaces it, otherwise the node can be pruned.

Finally, we need a *terminating rule* to tell us when to stop expanding the branch and bound tree. To guarantee that we have reached optimality, we stop when the incumbent solution's objective function value is better than or equal to the bounding function value associated with all of the bud nodes. This means that none of the bud nodes could possibly develop into a better solution than the complete feasible solution we already have in hand, so there is no point in expanding the tree any further. Of course, according to our policies for pruning, all bud nodes in this condition will already have been pruned, so this terminating rule amounts to saying that we stop when there are no more bud nodes left to consider for further growth! This also proves that the incumbent solution is optimum.

That's a lot of theory, so it's time for an example. Let's revisit the person-task assignment problem. Recall, though, that this problem can be cast in a network-flow format and solved quickly by linear programming, so you would never actually solve it via branch and bound in real life. However it's an easy problem to understand, so we will re-use it to demonstrate a branch and bound solution.

We have four persons, A though D, to assign to four tasks, 1 through 4. The table below shows the number of minutes it takes for each person to do each task. Each person can do exactly one task, and all tasks must have an assigned person. The objective is to minimize the total minutes taken. How many possible assignments of persons to tasks are there? There are 4!=24. In general, when there are *n* persons and *n* tasks there are *n*! possible assignments.

		task			
		1	**2**	**3**	**4**
person	**A**	9	5	4	5
	B	4	3	5	6
	C	3	1	3	2
	D	2	4	2	6

The formal branch and bound formulation follows. Any complete formulation must address all of these items.

- *Meaning of a node in the branch and bound tree*: a partial or complete assignment of persons to tasks. For example, a complete assignment ACBD represents the assignment of person A to task 1, person C to task 2, etc.
- *Node selection policy:* global best value of the bounding function.
- *Variable selection policy:* choose the next task in the natural order 1 to 4.
- *Bounding function:* for unassigned tasks, choose the best unassigned person, even if that person is chosen more than once. This is a relaxation of the original problem in which each person can do exactly one task.
- *Terminating rule:* when the incumbent solution objective function value is better than or equal to the bounding function values associated with all of the bud nodes.

- *Fathoming:* the "solution" generated by the bounding function is feasible if every task is assigned to a different person.

As an example of the calculation of the bounding function, let's look at the first-level node associated with assigning person A to task 1. The set of solutions represented by this node is A???. The bounding function value is calculated as follows:

- Actual value associated with assigning A to task 1: 9.
- Best unassigned person for task 2 is C, value: 1.
- Best unassigned person for task 3 is D, value: 2.
- Best unassigned person for task 4 is C, value: 2.
- The bounding function "solution" is ACDC, with total cost = 9+1+2+2=14. At this point we know that the very best objective function value that we might find at a leaf node descended from A??? is 14. Since person C is repeated, this is not a feasible solution, so the node is not fathomed. Note that person A is actually assigned, so we will not see A repeated in the bounding function calculations at this node or at any of the descendent nodes.

In practice, this bounding function amounts to crossing out the tasks (columns) that have assigned people, and the people (rows) that have assigned tasks, and then choosing the best number in each of the remaining columns, even if a particular person (row) is chosen more than once.

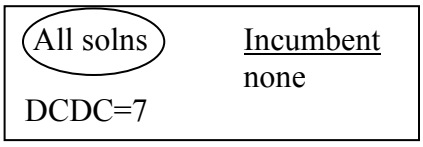

Figure 12.3: Stage 1: the root node.

Now we are ready to develop the branch and bound tree. First we create the root node. It's always worth finding the "solution" generated by the bounding function at the root node, since there is a chance that if you are fantastically lucky this "solution" will fathom the entire tree! Figure 12.3 shows the root node. In all of the figures, each node is labeled with the "solution" generated by the bounding function, and the bounding function value. Pruned nodes are indicated by dashed borders, and feasible nodes are indicated by bold borders. Pruned feasible nodes have both features.

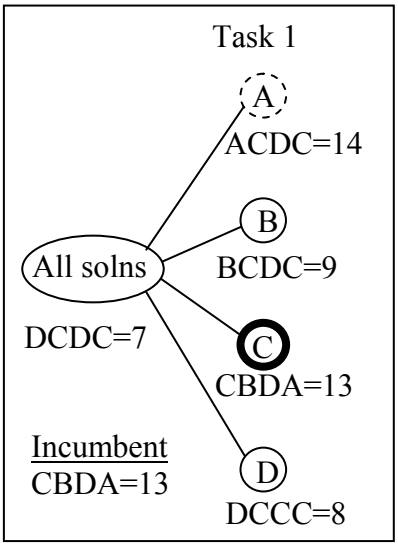

Figure 12.4: Stage 2.

Figure 12.4 shows the next level of the tree, generated from the root node by enumerating the possible persons who can do task 1. Node C??? is fathomed, providing our first feasible solution, and hence the first incumbent solution, CBDA=13. This allows us to prune node A??? whose bounding function value is 14.

There are two bud nodes in Figure 12.4 that show promise of improving on the incumbent solution: B??? with a bounding function value of 9 and D??? with a bounding function value of 8. Since we are using the global-best node selection policy, we

choose node D??? for further expansion, which results in Figure 12.5. Note that the child nodes of D??? are labeled A, B, and C, but not D. Why? Because person D is already assigned, and so can't be assigned again in descendent nodes. There are no new feasible solutions, so the incumbent solution is unchanged, and none of the new nodes can be pruned by comparison with the incumbent, or fathomed. So we choose the global best value of the bounding function between nodes B??? with value 9, DA?? with value 12, DB?? with value 10 and DC?? with value 12. The best of these is node B???, which is expanded in Figure 12.6.

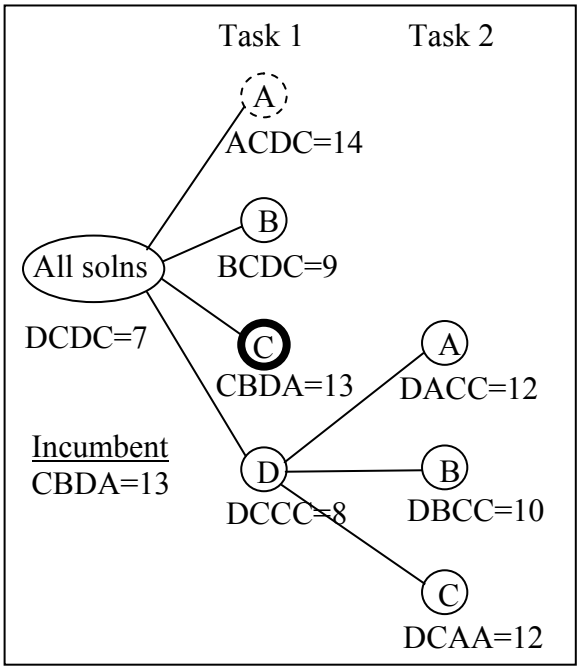

Figure 12.5: Stage 3.

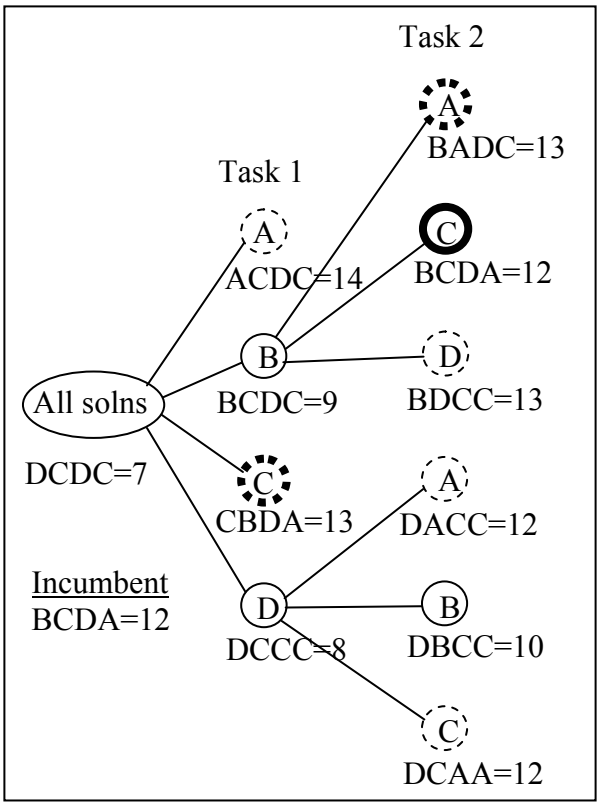

Figure 12.6: Stage 4.

Figure 12.6 shows that there are two new fathomed nodes, BA?? and BC??. The feasible bounding function "solution" associated with BC?? has a lower value than the current incumbent, and so replaces it and prunes the old incumbent. Node BA??, even though feasible, is pruned by comparison with the new incumbent. Nodes DA?? and DC?? are also pruned by comparison with the incumbent (these could be kept for potential later exploration if we wanted to find all possible optimum solutions instead of just one). There remains only a single bud node which has the possibility of producing a better solution than the incumbent: node DB??. This node is expanded to produce Figure 12.7.

Figure 12.7 shows the expansion of DB?? to produce two child nodes. Both of these are feasible since once the first 3 persons are chosen, the only unassigned person is assigned to the final task, producing a feasible solution. The newly produced solution DBAC has a better objective function value than the incumbent, and so replaces it and prunes the old incumbent. The other newly created solution, DBCA, is pruned by comparison with the new incumbent. At this point there are no bud nodes remaining for possible expansion, so we stop. The final solution is given by the current incumbent: DBAC with a minimum total time of 11 minutes for the assignment.

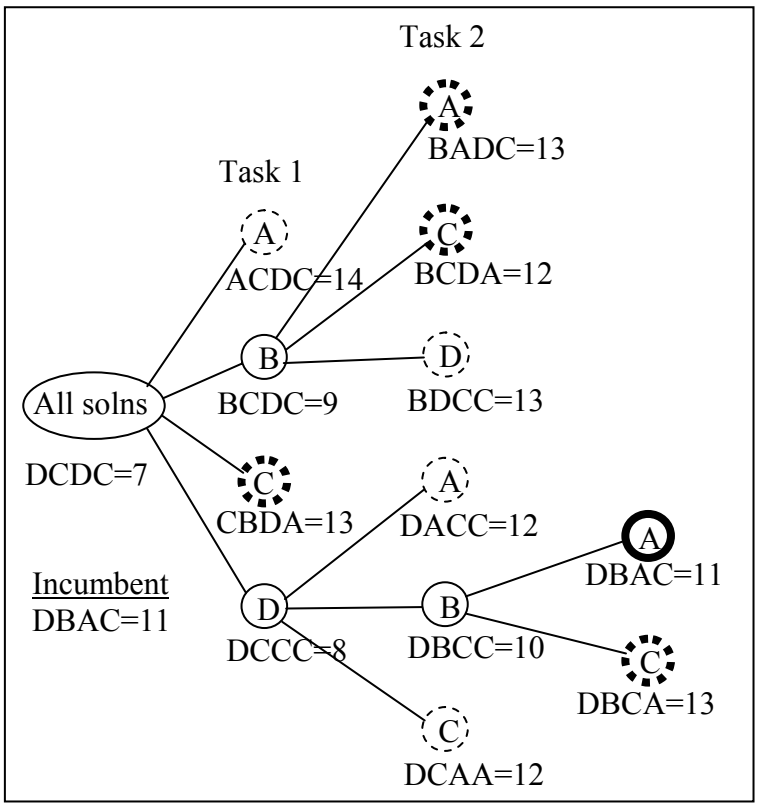

Figure 12.7: Stage 5. The tree is complete.

Now how much work did we do? We evaluated 13 nodes, including the root node. This is about half of the work of a full enumeration of the 24 possible solutions, quite a good speed-up. However, branch and bound solutions for large problems should do a much smaller fraction of the work, more like a tenth of a percent or less.

Some final notes on branch and bound methodology. First, how do we deal with ties for the next node to choose for expansion? You can simply choose arbitrarily: if you choose wrongly, the branch and bound method will eventually bring you back to the unchosen node. A general rule of thumb is to first choose the node that is farthest away from the root, since it is more likely to be closer to a feasible solution.

Second, suppose the incumbent solution objective function value is tied with the bounding function value at some bud nodes. If we are only interested in a single solution to the problem, then the bud nodes can be pruned because the best that they can do is to eventually grow into solutions equal to the one we already have on hand, and even that is not very likely. On the other hand, if we are interested in finding <u>all</u> of the optimal solutions, then those bud nodes are grown until they either yield an equivalent solution or are pruned. This allows us to generate all of the equivalent optimum solutions. In Figure 12.6 we pruned two bud nodes that had bounding function values equal to the incumbent objective function value because we were interested in finding just one single optimum solution.

More on Bounding Functions

A good bounding function is really what makes branch and bound work. Sometimes it takes a bit of ingenuity to find a good one. Let's consider a bounding function that we can use in a completely different problem. In the well-known traveling salesman problem we are given a graph whose arcs are labeled with distances. The goal is to find a *tour* that visits each node in the graph exactly once and returns to the original node, and has the shortest total length. This is the optimum route for the traveling salesman's as he visits each node (city) to sell his wares.

In this case a node in the branch and bound tree represents a partial tour, and it is the bounding function's job to estimate the length of the shortest tour that might result from continuing this

tour. Let's suppose the partial tour is as shown in boldface in Figure 12.8. How can we estimate the length of the shortest complete tour that incorporates the partial tour shown in Figure 12.8?

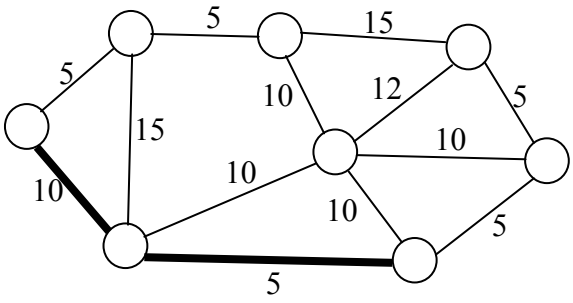

Figure 12.8: A partial tour in a traveling salesman problem.

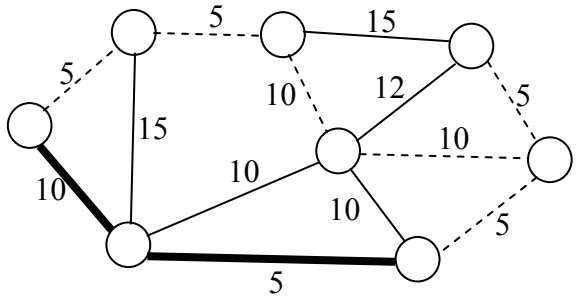

Figure 12.9: Bounding function: partial tour plus minimum spanning tree.

As usual the bounding function will solve a simpler problem that violates some of the constraints on the original problem. One clever bounding function solves a minimum spanning tree problem over the unvisited nodes and the two end nodes on the partial tour, as shown in Figure 12.9 (the minimum spanning tree arcs are dashed). The bounding function value associated with Figure 12.9 is given by (length of partial tour) + (length of minimum spanning tree) = (10+5) + (5+5+10+10+5+5) = 55. The "solution" generated by the bounding function is not feasible in this case (some cities must be visited more than once if this route is used), but it is a good underestimate of the shortest route. It also has the advantage that it is easy to recognize a feasible solution if one is generated by the bounding function, so fathoming is easy.

The traveling salesman problem is one of those interesting problems that are easy to state but hard to solve. It has been the subject of a great deal of work over many years since it has many practical applications (e.g. the routing of the welding head wielded by a robot on an automobile assembly line). There are now some reasonably good heuristic methods for fairly large problems.

Keys to Success

There are several keys to using branch and bound successfully.

Have a good bounding function. It must be optimistic, but as close to feasibility as possible. The more exact it is, the smaller the resulting search tree will be. There are often numerous different ways to construct a bounding function for a particular problem. Be creative!

Get a good incumbent early. This is tremendously useful in pruning because many buds will never be expanded if their bounding function value is worse than the objective function value of the incumbent. Consider using a heuristic to generate an incumbent solution even before beginning the branch and bound process. As an example, consider the following heuristic for generating an initial incumbent in the person-task assignment problem. First choose the smallest number anywhere in the table, then remove the associated task and person from the table. Repeat this process until all tasks and people have been assigned. You could also use depth-first node selection until the first incumbent is found, then switch to another node selection policy.

Find ways to identify nodes that have no feasible descendents. This prevents nodes from being expanded.

Order constraints and variables so that the most restrictive are tested first. The general idea is to encourage early failure of nodes on the tree. The closer to the root that a node is pruned, the more tree is cut off. If one variable has a very restricted range or is involved in a very restrictive constraint, generate child nodes based on that variable first.

Sub-optimizing in Very Large Problems

Branch and bound slows down the combinatorial explosion in integer programs, but it doesn't stop it altogether. There are many problems in which even the standard branch and bound tree is too large. In cases like this we can sacrifice the guarantee of optimality that is provided by branch and bound in favour of getting a reasonable answer quickly, or within the memory limitations of our computer. There are three main heuristics, all based on branch and bound.

Stopping with a guarantee of closeness to optimality. Choose an acceptable distance to optimality, e.g. 5%. Now you simply halt the branch and bound process when the incumbent solution is within 5% of the best bud node bounding function value. The incumbent solution is definitely within 5% of the optimum, and it could be much closer, even optimum. It just takes a lot more branch and bound nodes to prove it.

For example, global optimization of nonlinear functions can be done by a branch and bound procedure which subdivides the search space as it goes. A recent Ph.D. student of mine used this approach. We discovered that his procedure very often reached the optimum solution in the first 5 seconds of computing time, but it then took another 24 hours of computing time to prove that it was really the optimum! Stopping within some stated fraction of the optimum can reduce calculation time a great deal, sometimes without much effect on the final solution.

Beam search. This technique is especially useful if computer memory is limited. Set an upper limit on the number of bud nodes that will be maintained in memory, e.g. 1000. If this limit is reached, rank order all of the bud nodes based on their bounding function values and keep only the best 1000. Of course the guarantee of optimality is lost (one of the discarded nodes may have been the one which ultimately lead to the optimum solution), but the search can run in the limited memory space available.

Depth-first search to first incumbent. If time is limited, depth-first search is preferred since it is the most likely to reach a feasible incumbent solution first. If time runs out, at least one feasible solution is available even though the branch and bound solution is halted prematurely. As we will see later, depth-first search is especially preferred if linear programming is being used in the bounding function since each LP solution is quite similar to the last one, so advanced starts can be used.

Branch and bound methods can be customized to handle special situations. We will address some of these in the next chapter.