5.2 Branch & Bound

The intuition behind *Branch & Bound* (*B&B*) is as follows.

- Organize a search space that does not preclude any solution. This could be the state space, in which a partial sequence of moves is extended in each move. This could also be a refinement space in which an abstract solution is refined.
- Continue looking for a solution by refining the cheapest candidate until
  - A complete solution is found
  - No candidate solution, partial or complete, with an estimated cost smaller than that of the complete solution exists

A high level algorithm is given in Figure 5.2. Here the task is to find the lowest cost path from a start node to a goal node, and the algorithm extends the cheapest cost partial solution at each stage.

```
B&B Procedure()
 1 open ← {(start, NIL, Cost(start))}
 2 closed ← {}
 3  while TRUE
 4   do if Empty(open)
 5        then return FAILURE
 6     Pick cheapest node n from open
 7     if GoalTest(Head(n))
 8        then return ReconstructPath(n)
 9        else  children ← MoveGen(Head(n))
10              for each m ∈ children
11                do Cost(m) ← Cost(n) + K(m, n)
12                   Add (m, Head(n), Cost(m)) to open
```

FIGURE 5.2 Branch and Bound extends the cheapest solution till the cheapest solution reaches the goal.

The basic idea behind *B&B* is to *ignore* those regions of a search space that are *known* not to contain a better solution. We look at an example in which the cost of a move corresponds to the length of an edge in a graph. Then the least cost solution corresponds to the shortest path in the graph. Let the graph in Figure 5.3 represent a tiny search space to illustrate the algorithm.

B&B begins with the start node *S*. The partial cost of *S* is zero. It expands *S*, generating partial paths *S-A* with cost 3, *S-B* with cost 4 and *S-C* with cost 8. These paths are stored in the list *OPEN*. *S* is transferred to list *CLOSED*, shown shaded in Figure 5.4.

B&B continues extending the cheapest partial path. It terminates when the goal node is picked for expansion.

The example in Figure 5.4 shows *Branch & Bound* extending partial solutions. This is an example of state space search. We have seen earlier (Chapter 3) that when the search space is made up of candidate solutions, we search in the solution space.

If the candidate solution is only partially specified then we can think of it as a set of solutions that share the specified part. A refinement operator partitions this set into two sets by specifying another component of the solution. Each (complete) candidate from the set is some complete refinement of the partially specified solution. Search, then involves, decisions amongst the different possible refinements of a given partial solution (Kambhampati, 1997). That is, search involves choosing a refinement of some candidate by specifying more information. It is interesting to note that the state space search algorithms

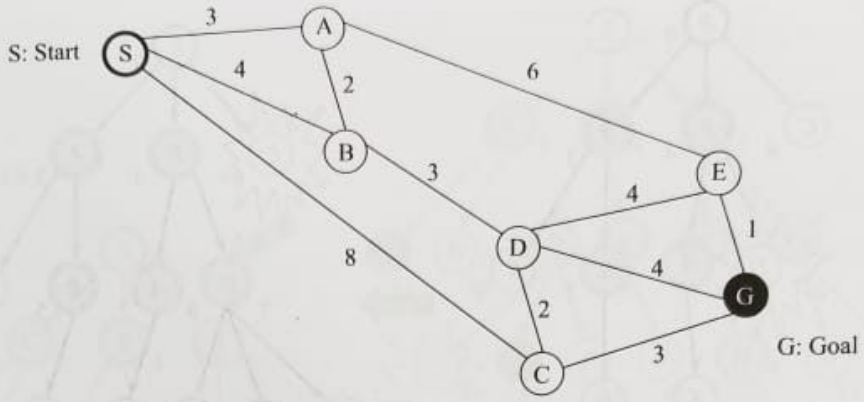

FIGURE 5.3 A tiny search graph.

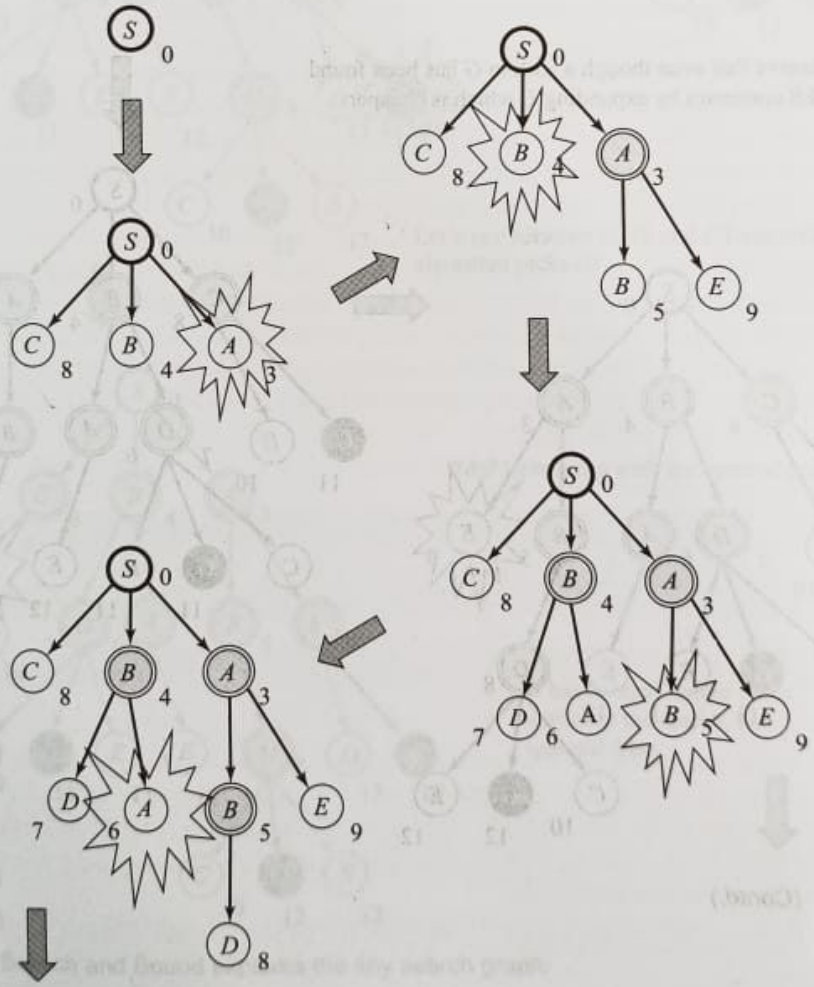

FIGURE 5.4 (*Contd.*)

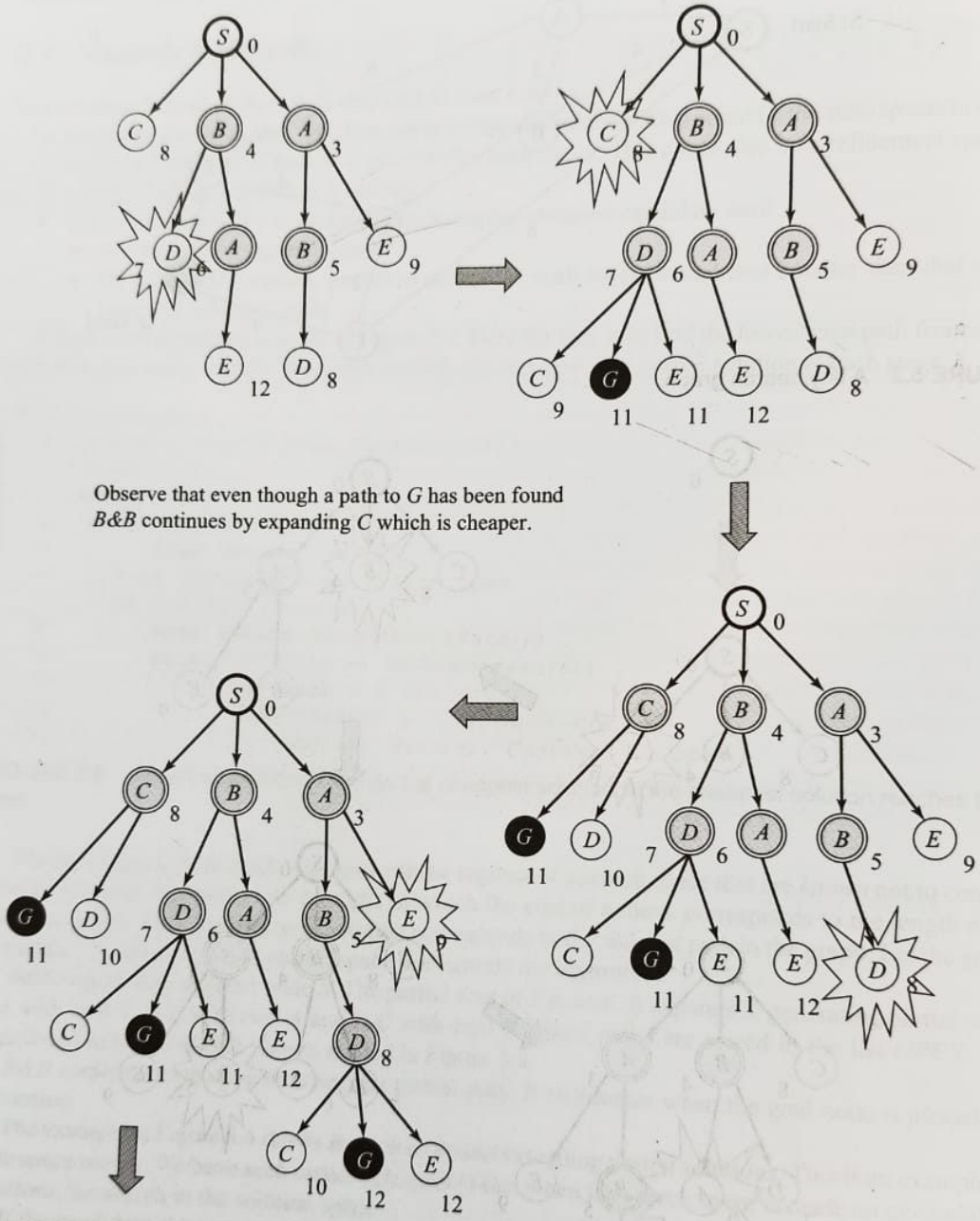

FIGURE 5.4 *(Contd.)*

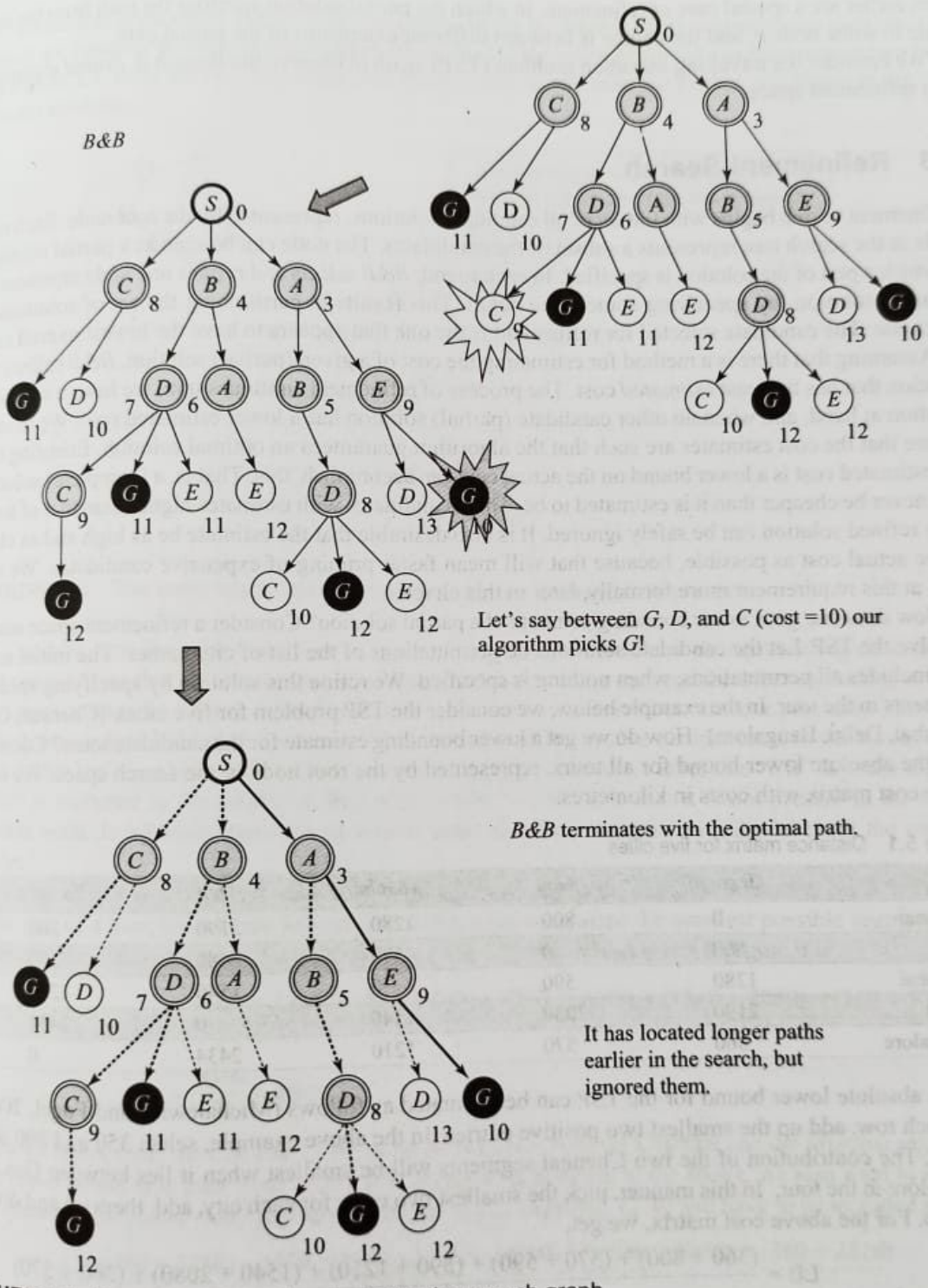

FIGURE 5.4 Branch and Bound explores the tiny search graph.

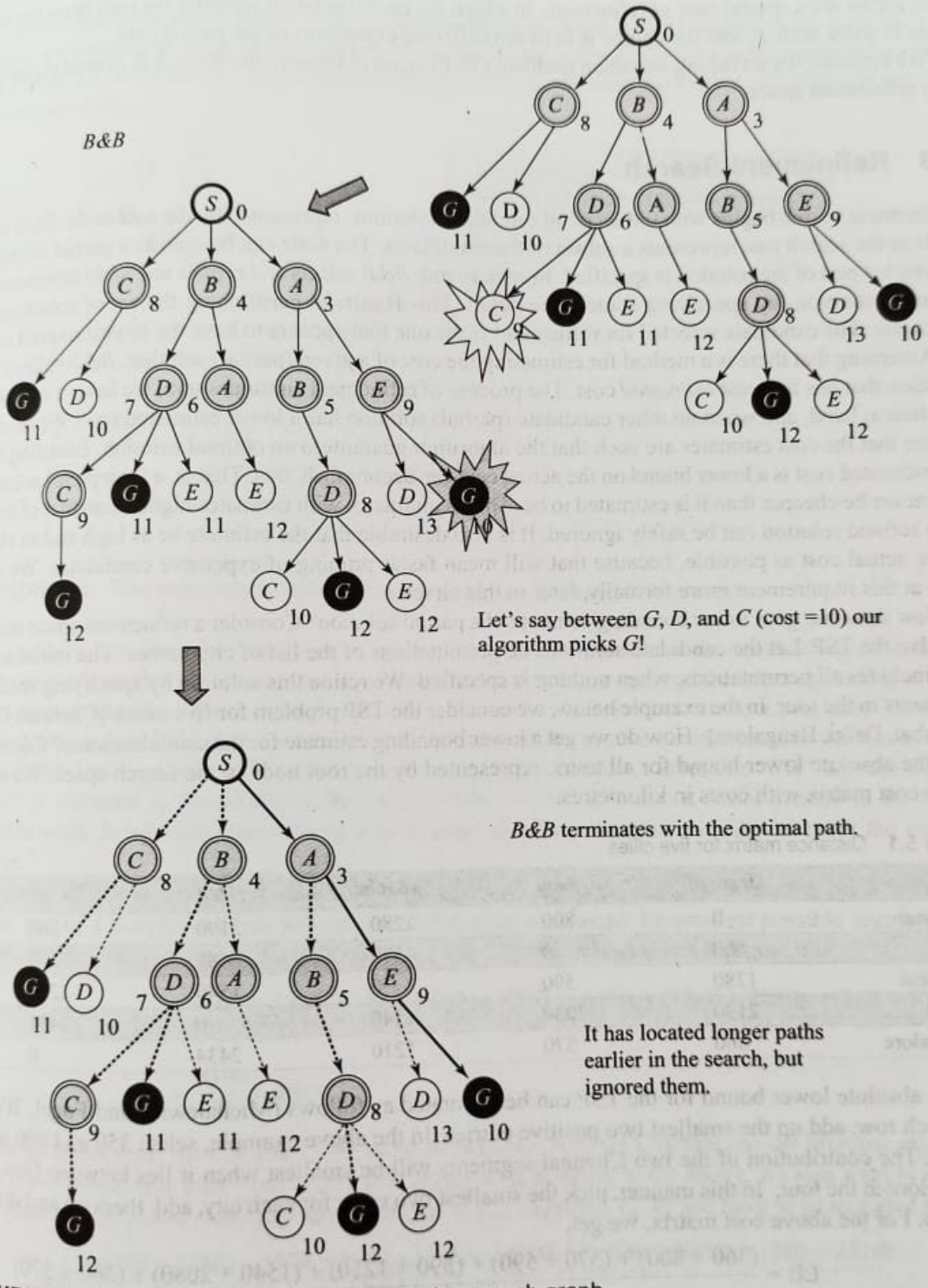

FIGURE 5.4 Branch and Bound explores the tiny search graph.

seen earlier are a special case of refinement, in which the partial solution specifies the path from the start node to some node n, and the choice is between different extensions of the partial path.

We consider the travelling salesman problem (TSP) again to observe the *Branch & Bound* algorithm in a refinement space.

5.3 Refinement Search

Refinement search begins with the *set* of all candidate solutions, represented by the root node. Each new node in the search tree represents a subset of the candidates. The node can be seen as a partial solution, in which a part of the solution is specified. In each round, *B&B* selects and refines one node representing a partial solution, by specifying some more detail. This results in partitioning the set of solutions in that node. The candidate selected for refinement is the one that appears to have the lowest overall cost.

Assuming that there is a method for estimating the cost of a given (partial) solution, *B&B* refines the solution that has the least *estimated* cost. The process of refinement continues until we have a complete solution at hand, and when no other candidate (partial) solution has a lower estimated cost. We need to ensure that the cost estimates are such that the algorithm guarantees an optimal solution. Ensuring that the estimated cost is a lower bound on the actual cost can accomplish this. That is, a (complete) solution will never be cheaper than it is estimated to be. Then, candidates with estimates higher than that of some fully refined solution can be safely ignored. It is also desirable that the estimate be as high and as close to the actual cost as possible, because that will mean faster pruning of expensive candidates. We will look at this requirement more formally, later in this chapter.

How does one get a lower bounding estimate of a partial solution? Consider a refinement space search to solve the TSP. Let the candidate solutions be permutations of the list of city names. The initial solution includes all permutations, when nothing is specified. We refine this solution by specifying specific segments in the tour. In the example below, we consider the TSP problem for five cities {Chennai, Goa, Mumbai, Delhi, Bangalore}. How do we get a lower bounding estimate for the candidate tours? Consider first the absolute lower bound for all tours, represented by the root node in the search space. We look at the cost matrix with costs in kilometres.

Table 5.1 Distance matrix for five cities

	Chennai	*Goa*	*Mumbai*	*Delhi*	*Bangalore*
Chennai	0	800	1280	2190	360
Goa	800	0	590	2080	570
Mumbai	1280	590	0	1540	1210
Delhi	2190	2080	1540	0	2434
Bangalore	360	570	1210	2434	0

An absolute lower bound for the TSP can be estimated as follows (Michalewicz and Fogel, 2004). For each row, add up the smallest two positive entries. In the above example, select 350 and 800 from row 1. The contribution of the two Chennai segments will be smallest when it lies between Goa and Bangalore in the tour. In this manner, pick the smallest two costs for each city, add them up and divide by two. For the above cost matrix, we get,

$$\text{LB} = \frac{(360 + 800) + (570 + 590) + (590 + 1210) + (1540 + 2080) + (360 + 570)}{2}$$

$$= \frac{8670}{2} = 4335$$

Observe that the above lower bound may not actually be feasible. This is because for each city (each row), we consider the nearest two neighbours as ones in the estimate. In the example map shown below in Figure 5.5, there are two long edges that must be part of any tour, but neither will figure in the lower bound estimate.

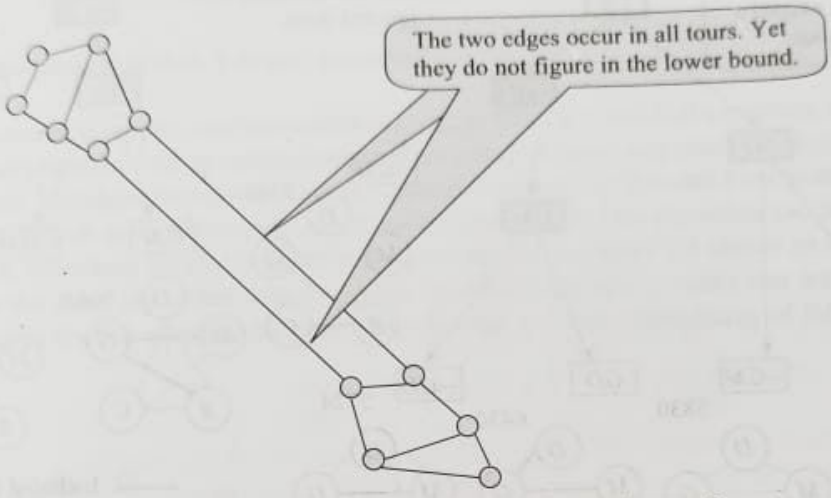

FIGURE 5.5 The lower bound estimated costs may not be feasible in practice.

Consider now the refinement search space. The root consists of a node representing all solutions. We can partition this set in two, one subset including a particular arc, and the other excluding it. These two sets can then be further refined recursively till each node describes one tour precisely. Part of the search tree is depicted below in Figure 5.6. It contains all the twelve distinct tours in the leaves. Some leaves that have not been fully refined contain more than one tour. A label like "CB" says the segment "CB" is included in the tour(s) in that node, while "¬CB" says that it is not present in the tour(s) in that node. In addition, the tours in a node must also be consistent with the labels at the ancestor nodes.

The next task is to estimate the cost of these partially refined solutions. Wherever an arc is known to be part of a tour, we add the known cost. Otherwise we choose the smallest possible segments. For example, the estimated cost of a tour including the *Chennai-Mumbai (CM)* segment is

$$\text{LB} = \frac{(360 + \mathit{1280}) + (570 + 590) + (590 + \mathit{1280}) + (1540 + 2080) + (360 + 570)}{2}$$

$$= \frac{9010}{2} = 4505$$

In the above case, the CM cost (1280) was included for both the Chennai and the Mumbai segment The lower bound cost then went up to 4505. Suppose in addition to the above, we also want to inclu the *Mumbai Bangalore (MB)* segment. The cost of this segment, 1210, will have to be included as we

$$\text{LB} = \frac{(\mathit{360} + 1280) + (570 + 590) + (\mathit{1210} + 1280) + (1540 + 2080) + (\mathit{360} + \mathit{1210})}{2}$$

$$= \frac{10480}{2} = 5240$$

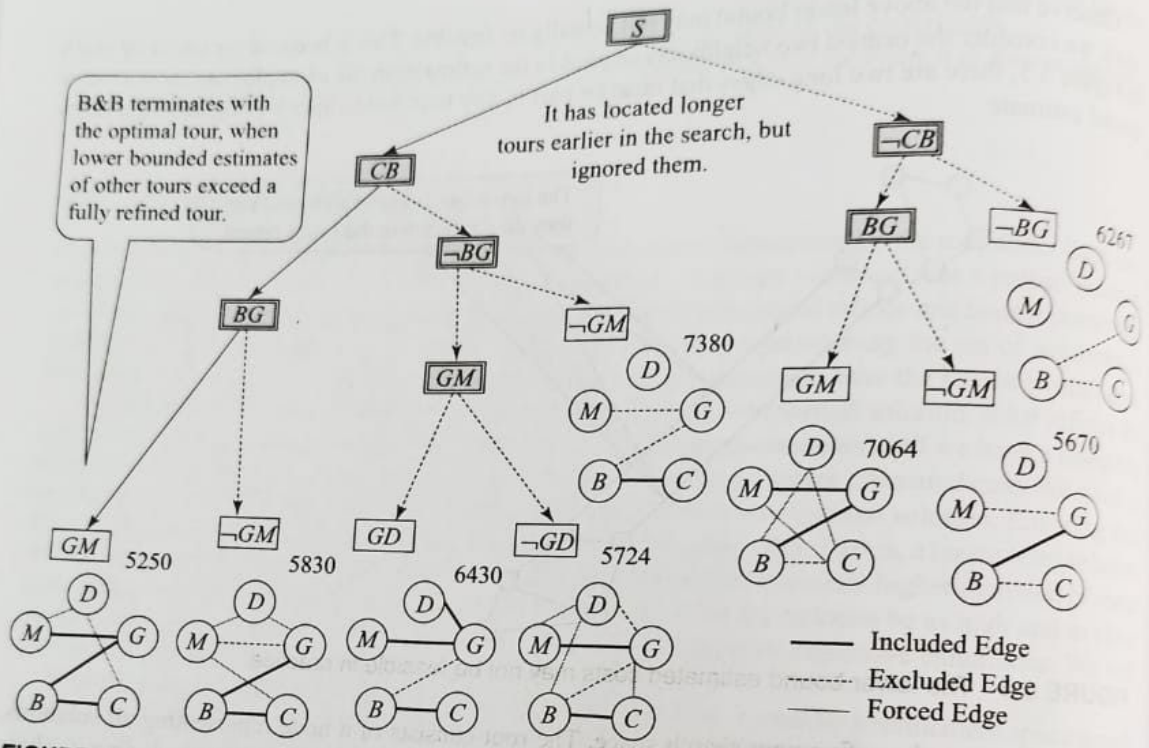

FIGURE 5.6 Branch and Bound on TSP in the refinement space.

While this does give us a tighter estimate, one can improve upon this with some reasoning. The fact that the CM segment and MB segments are included implies that the *Bangalore Chennai (BC)* segment with cost 360 should be excluded. Otherwise, we would have a C-M-B-C cycle, which is not allowed by the specification of TSP. The BC segment costing 360, contributes twice in the above estimate. One must replace the two occurrences with the next better costs, 800 and 570, as shown below.

$$LB = \frac{(800 + 1280) + (570 + 590) + (1210 + 1280) + (1540 + 2080) + (570 + 1210)}{2}$$

$$= \frac{11130}{2} = 5565$$

Both are incidentally to Goa, and therein lurks another case for reasoning, because each city can be connected only to two others. With further reasoning, we could get a still better estimate. This kind of trade off is not uncommon. Reasoning can prune the search space, but itself has an associated cost. In the above method for estimating costs for example, we may be counting more that two edges emanating out of a node. If we could avoid this, then our estimates would become more accurate.

As we refine the above partial solution, we get increasing and better estimates of cost. If at any point *all* the estimated costs were to become higher than a *known* cost of a complete solution in hand then we would stop refining the solution.

The *B&B* algorithm can thus be summarized as shown in Figure 5.7. The only condition is that the estimated cost must be a *lower bound* on the actual cost.

```
Generalized B&B Procedure()
1 Start with all possible solutions
2 repeat
3    Refine the least (estimated cost) solution further
4 until the cheapest solution s is fully refined
5 return s
```

FIGURE 5.7 The general *Branch & Bound* procedure.

Branch & Bound is a complete and admissible algorithm. That is, it will find a solution, if there exists one, and it is guaranteed to find an optimal solution. In terms of space and time complexity, however, it is not very good. The algorithm can be seen as a generalization of the Breadth First Search algorithm when each move has an associated cost. Like Breadth First Search, this algorithm too is uninformed and conservative, searching blindly without a sense of direction. Figure 5.8 shows an example map (to scale) where the *B&B* algorithm would spend a lot of time exploring nodes that are closer to the starting point before finding a path to the goal. Both the time and space complexity of *B&B* tends to be exponential.

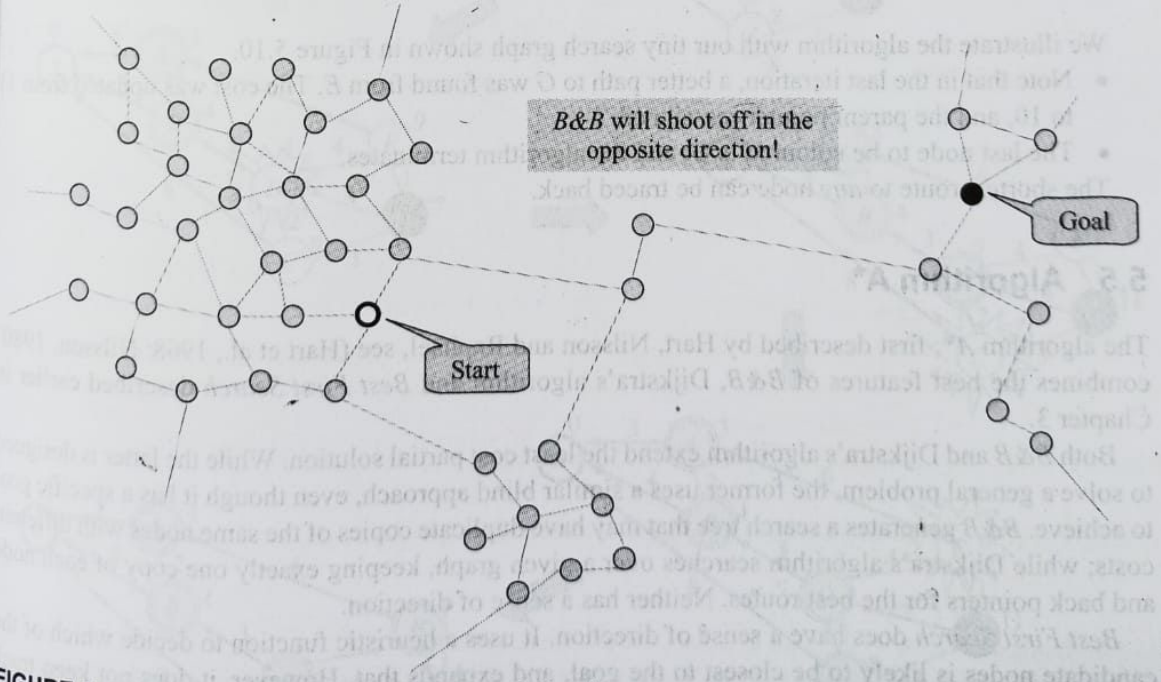

FIGURE 5.8 *Branch & Bound* has no sense of direction.