BRANCH AND BOUND

- An optimization problem seeks to minimize or maximize some objective function, usually subject to some constraints.
- A ***feasible solution*** is a point in the problem's search space that satisfies all the problem's constraints (for ex: a Hamiltonian circuit problem).
- An ***optimal solution*** is a feasible solution with the best value of the objective function(for ex: the shortest Hamiltonian circuit).
- **Branch and bound** is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.
- Compared to backtracking, branch-and-bound requires two additional items:
  - ✓ a way to provide, for every node of a state-space tree,
  - ✓ a bound on the best value of the objective function1 on any solution that can be obtained by adding further components to the partially constructed solution represented by the node the value of the best solution seen so far.
- The search path can be terminated at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:
  - ✓ The value of the node's bound is not better than the value of the best solution seen so far.
  - ✓ The node represents no feasible solutions because the constraints of the problem are already violated.
  - ✓ The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Knapsack Problem

- Given *n* items of known weights w_i and values v_i , $i = 1, 2, \ldots , n$, and a knapsack of capacity *W*, finding the most valuable subset of the items that fit in the sack is called Knapsack problem.
- It is convenient to order the items of a given instance in descending order by their value-to-weight ratios i.e.

$$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n.$$

- The state space tree of this problem is constructed as follows
  - ✓ Each node on the i^{th} level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items.
  - ✓ This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion.
  - ✓ We record the total weight w and the total value v of this selection in the node, along with some upper bound *ub* on the value of any subset that can be obtained by adding zero or more items to this selection.
  - ✓ The upper bound ub can be calculated by adding to v, the total value of the items already selected, the product of the remaining capacity of the knapsack $W - w$ and the best per unit payoff among the remaining items, which is v_{i+1}/w_{i+1}:

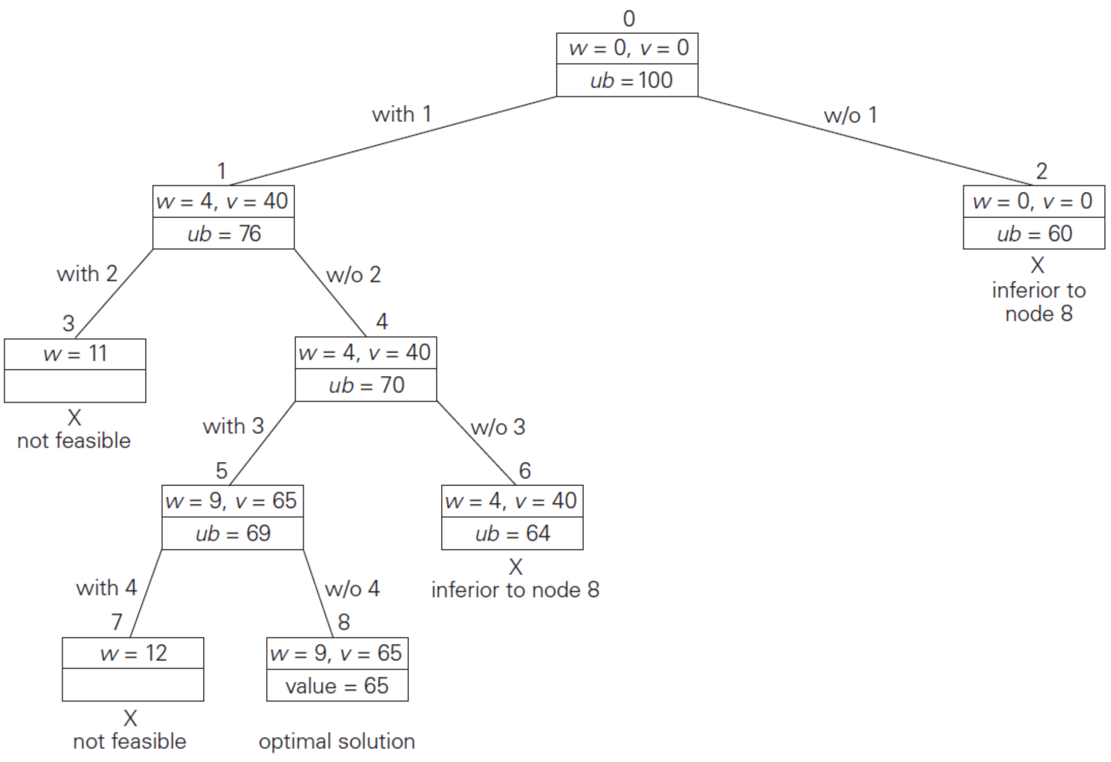
$$ub = v + (W - w)\,(v_{i+1}/w_{i+1})$$

<u>EXAMPLE</u>

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	\$40	10
2	7	\$42	6
3	5	\$25	5
4	3	\$12	4

The knapsack's capacity W is 10.

The state space tree of the above problem is shown below



- ✓ At the root of the state-space tree, no items have been selected: so w = 0 & v = 0,

  ub = 0 + (10 – 0) (10) = 100
- ✓ At node 1 (1st item included): w = 4, v = 40, ub = 40 + (10 – 4)(6) = 40 + 36 = 76
- ✓ At node 2 (1st item excluded): w = 0, v = 0, ub = 0 + (10 – 0)(6) = 60

  Node 1 is promising so continue from 1
- ✓ At node 3 (2nd item included): w = 4 + 7 = 11 > 10(Max. capacity of sack), not a feasible solution
- ✓ At node 4 (2nd item excluded): w = 4, v = 40, ub = 40 + (10 – 4)(5) = 40 + 30 = 70

  Node 4 is promising so continue from 4
- ✓ At node 5 (3rd item included): w = 9(4 + 5), v = 65(40 + 25), ub = 65 + (10 – 9)(4) = 65 + 4 = 69, Node 5 is promising so continue from 5
- ✓ At node 6 (3rd item excluded): w = 4, v = 40, ub = 40 + (10 – 4)(4) = 40 + 24 = 64, Node 5 is promising compared to 6 so continue from 5
- ✓ At node 7 (4th item included): w = 9 + 3 = 12 > 10(Max. capacity of sack), not a feasible solution
- ✓ At node 8 (4th item excluded): w = 9, v = 65, ub = 65 + (10 – 9)(0) = 65, Node 8 is the feasible solution.
- ✓ Compare to node 2 & node 6, node 8 is having highest value, so node 8 is the optimal solution.

- ✓ The solution set is {1, 3} [node 8 is reached by including 1, excluding 2, including 3 and excluding 4]

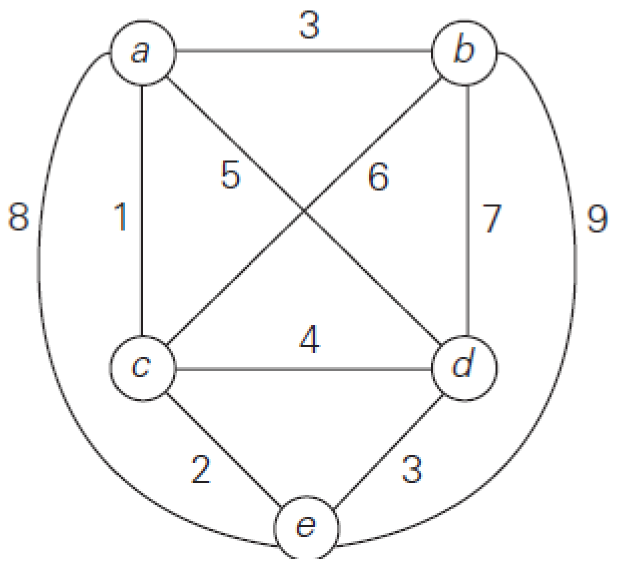
Traveling Salesman Problem

- Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.
- To solve this problem, lower bound can be computed on the length *l* of any tour as follows.
- For each city *i*, $1 \leq i \leq n$, find the sum s_i of the distances from city *i* to the two nearest cities; compute the sum *s* of these *n* numbers, divide the result by 2, and, if all the distances are integers, round up the result to the nearest integer:

$$lb = ceil(s/2)$$

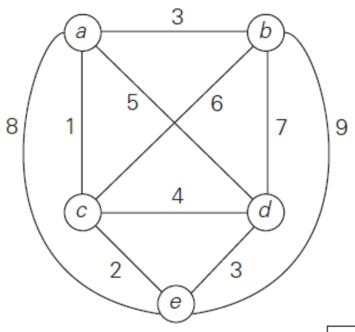
EXAMPLE

Find the optimal tour for the given graph

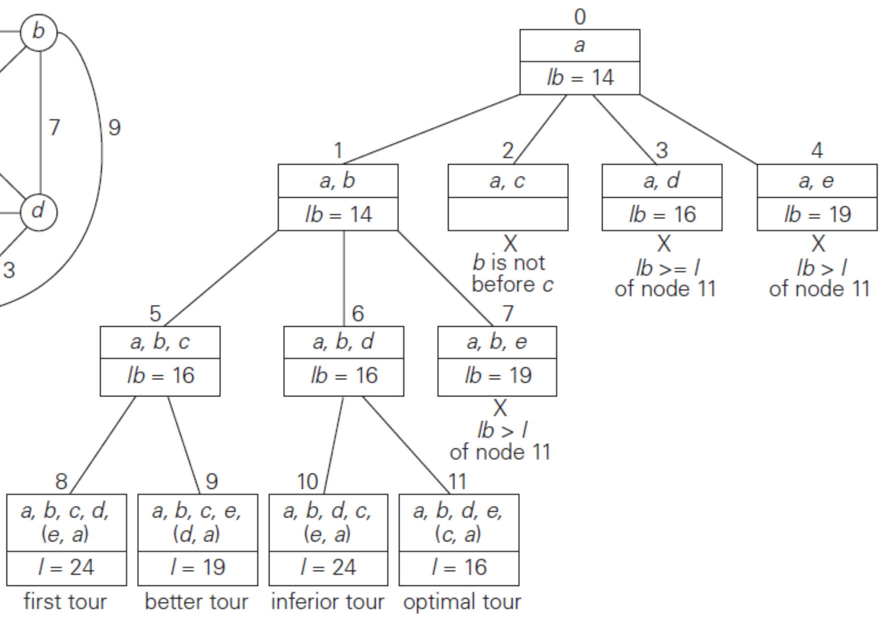


Solution

The state space tree is as follows



(a)



- For node 0: for each vertex, include 2 edges of least cost

  lb = ceil([(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)] / 2) = 14

- From a, we can traverse to b(a,b) / c(a,c) / d(a,d) / e(a,e),

  Include edges (a,b) and (b,a), when we calculate for (a,b)

  At node 1: lb = ceil([(3 + 1) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)]/2) = 14

  Include edges (a,c) and (c,a), when we calculate for (a,c)

  At node 2: lb = ceil([(3 + 1) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)]/2) = 14

  Include edges (a,d) and (d,a), when we calculate for (a,d)

  At node 3: lb = ceil([(1 + 5) + (3 + 6) + (1 + 2) + (5 + 3) + (2 + 3)]/2) = 16

  Include edges (a,e) and (e,a), when we calculate for (a,e)

  At node 4: lb = ceil([(8 + 1) + (3 + 6) + (1 + 2) + (3 + 4) + (8 + 2)]/2) = 19

  Choosing (a,b) of the 4 paths available, continue from (a,b)

- From (a,b), we can move to (a,b,c), (a,b,d) & (a,b,e)

  Include edges (a,b), (b,a), (b,c) and (c,b), when we calculate for (a,b,c)

  At node 5: lb = ceil([(3 + 1) + (3 + 6) + (6 + 1) + (3 + 4) + (2 + 3)]/2) = 16

  Include edges (a,b), (b,a), (b,d) and (d,b), when we calculate for (a,b,d)

  At node 6: lb = ceil([(3 + 1) + (3 + 7) + (1 + 2) + (7 + 3) + (2 + 3)]/2) = 16

  Include edges (a,b), (b,a), (b,e) and (e,b), when we calculate for (a,b,e)

  At node 7: lb = ceil([(3 + 1) + (3 + 9) + (1 + 2) + (3 + 4) + (2 + 9)]/2) = 19

Choosing 5 & 6 and continue them

- From (a,b,c), we can move to (a,b,c,d,(e,a)) or (a,b,c,e,(d,a))

  At node 8: length of the tour (l) = 3 + 6 + 4 + 3 + 8 = 24 (first tour)

  At node 9: length of the tour (l) = 3 + 6 + 2 + 3 + 5 = 19 (better tour)
- Similarly, from (a,b,d), we can move to (a,b,d,c,(e,a)) or (a,b,d,e,(c,a))

  At node 10: length of the tour (l) = 3 + 7 + 4 + 2 + 8 = 24 (inferior tour)

  At node 11: length of the tour (l) = 3 + 7 + 3 + 2 + 1 = 16 (optimal tour)
- The optimal tour is a → b → d → e → c → a and the cost is **16**

DIFFERENCE BETWEEN BACKTRACKING AND BRANCH AND BOUND METHOD

Backtracking

- ✓ It is used to find all possible solutions available to the problem.
- ✓ It traverse tree by DFS(Depth First Search).
- ✓ It realizes that it has made a bad choice & undoes the last choice by backing up.
- ✓ It search the state space tree until it found a solution.
- ✓ It involves feasibility function.

Branch-and-Bound (BB)

- ✓ It is used to solve optimization problem.
- ✓ It may traverse the tree in any manner, DFS or BFS.
- ✓ It realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution.
- ✓ It completely searches the state space tree to get optimal solution.
- ✓ It involves bounding function