Unit V

Branch and Bound Problems

Tackling Difficult Combinatorial Problems

- *exhaustive search* (brute force)
  - useful only for small instances

- *dynamic programming*
  - applicable to some problems (e.g., the knapsack problem)
- *backtracking*
  - eliminates some unnecessary cases from consideration
  - yields solutions in reasonable time for many instances but worst case is still exponential

- *branch-and-bound*
  - further refines the backtracking idea for optimization problems

Branch and Bound

- *An enhancement of backtracking*
- *Applicable to optimization problems*
- *For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)*
- *Uses the bound for:*
  - *ruling out certain nodes as "nonpromising" to prune the tree – if a node's bound is not better than the best solution seen so far*
  - *guiding the search through state-space*

Introduction …

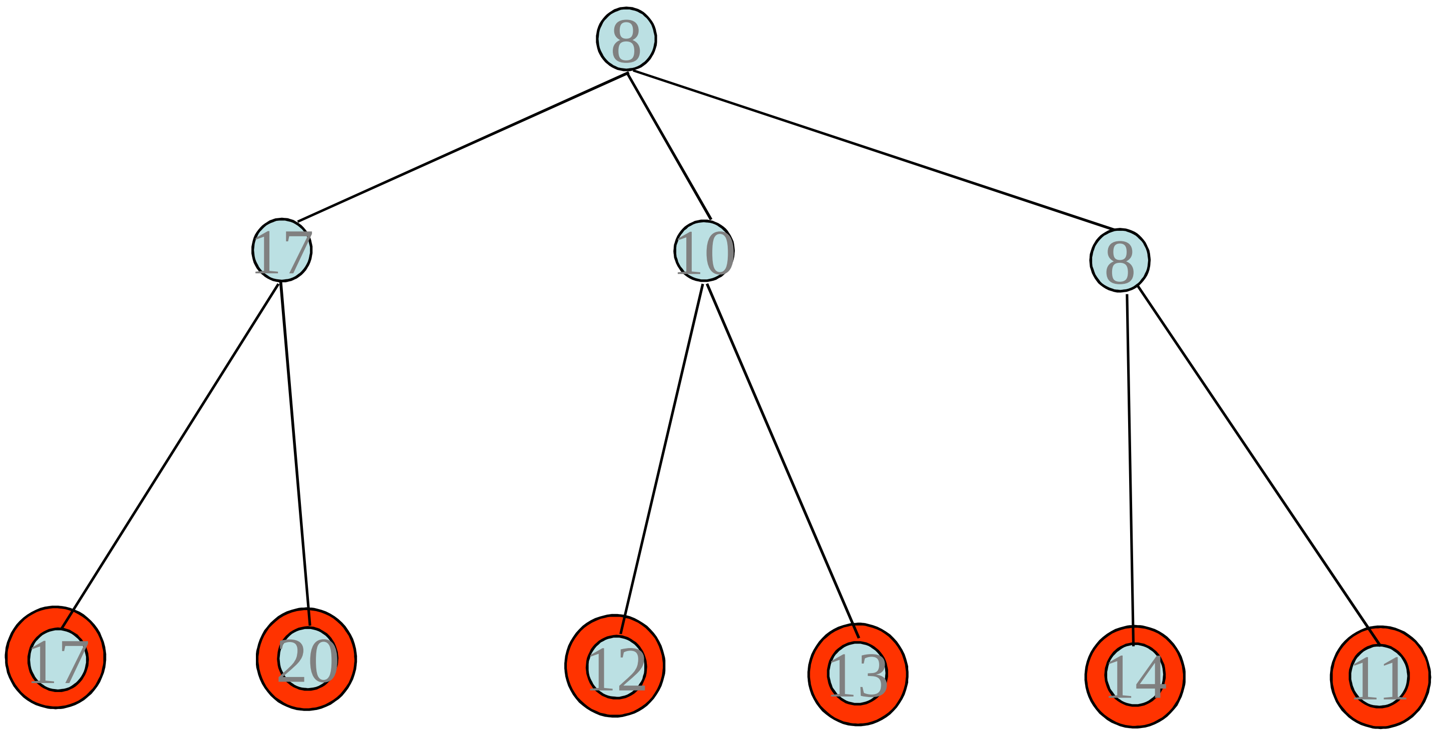
- *Besides using the bound to determine whether a node is promising, we can compare the bounds of promising nodes and visit the children of the one with the best bound.*

- *This approach is called best-first search with branch-and-bound pruning. The implementation of this approach is a modification of the breadth-first search with branch-and-bound pruning.*

0-1 Knapsack

- To learn about branch-and-bound, first we look at breadth-first search using the knapsack problem
- Then we will improve it by using best-first search.
- Remember the default strategy for the 0-1 knapsack problem was to use a depth-first strategy, not expanding nodes that were not an improvement on the previously found solution.

BackTracking (depth-first)



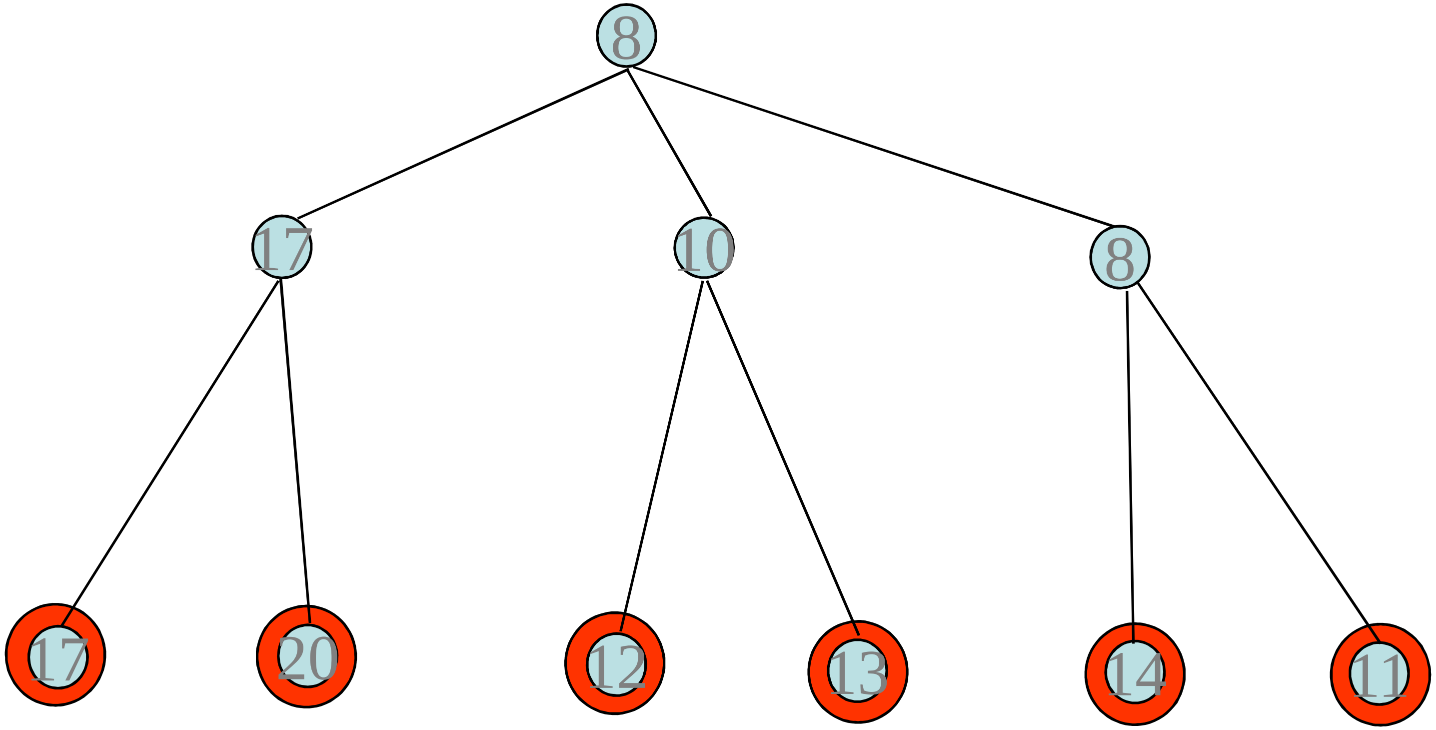
Breadth-first Search

- We can implement this search using a queue.
- All child nodes are placed in the queue for later processing if they are promising.
- Calculate an integer value for each node that represents the maximum possible profit if we pick that node.
- If the maximum possible profit is not greater than the best total so far, don't expand the branch.

Breadth-first Search

- The breadth-first search strategy has no advantage over a depth-first search (backtracking).
- However, we can improve our search by using our bound to do more than just determine whether a node is promising.

Branch and Bound (breadth-first)



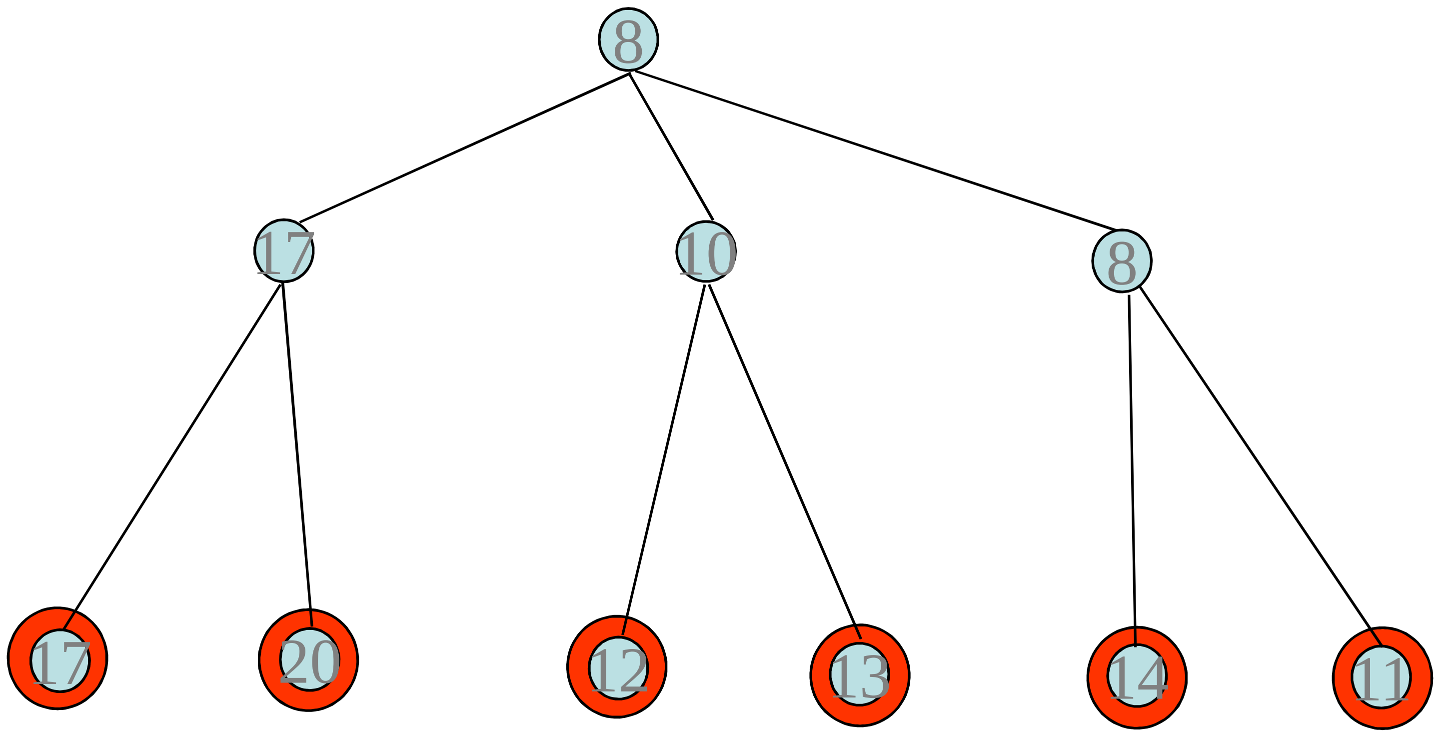
0-1 Knapsack

- 0-1 Knapsack using the branch and bound.
- Now look at all promising, unexpanded nodes and expand beyond the one with the **best** bound.
- We often arrive at an optimal solution more quickly than if we simply proceeded blindly in a predetermined order.

Best-first Search

- Best-first search expands the node with the best bounds next.
- How would you implement a best-first search?
  - Depth-first is a stack
  - Breadth-first is a queue
  - Best-first is a ???

Branch and Bound (best first)



0-1 Knapsack

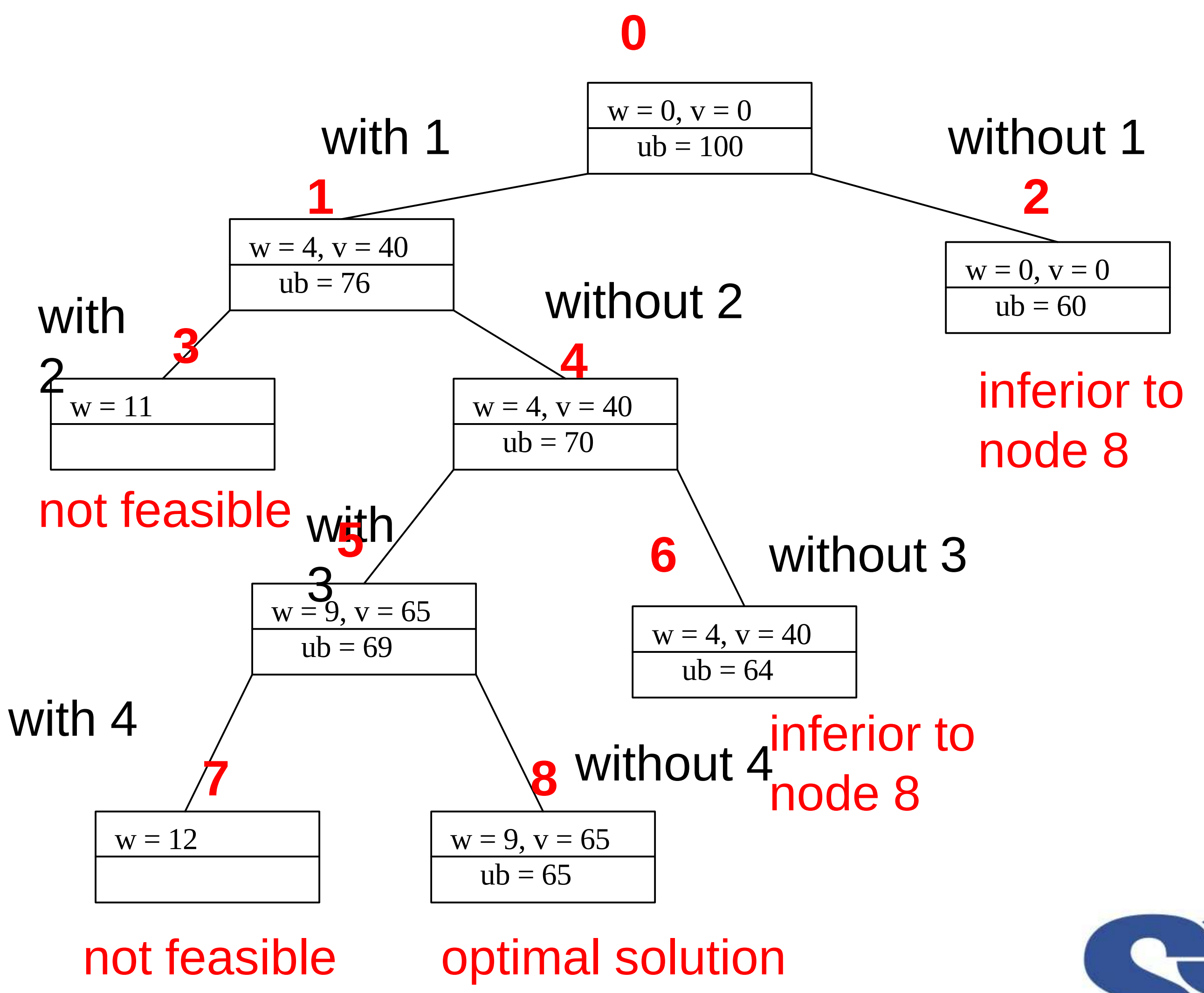
- Capacity W is 10
- Upper bound is $100 (use fractional value)

Item	Weight	Value	Value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

Computing Upper Bound

- To compute the upper bound, use
  - $ub = v + (W - w)(v_{i+1}/w_{i+1})$
- So the maximum upper bound is
  - pick no items, take maximum profit item
  - ub = (10 – 0)*($10) = $100
- After we pick item 1, we calculate the upper bound as
  - all of item 1 (4, $40) + partial of item 2 (7, $42)
  - $40 + (10-4)*6 = $76
- If we don’t pick item 1:
  - ub = (10 – 0) * ($6) = $60

State Space Tree



Bounding

- A bound on a node is a guarantee that any solution obtained from expanding the node will be:
  - Greater than some number (lower bound)
  - Or less than some number (upper bound)

- If we are looking for a maximal optimal (knapsack), then we need an upper bound
  - For example, if the best solution we have found so far has a profit of 12 and the upper bound on a node is 10 then there is no point in expanding the node
    - The node cannot lead to anything better than a 10

Bounding

- Recall that we could either perform a depth-first or a breadth-first search
  - Without bounding, it didn’t matter which one we used because we had to expand the entire tree to find the optimal solution
  - Does it matter with bounding?
    - Hint: think about when you can prune via bounding

Bounding

- We prune (via bounding) when:

  (currentBestSolutionCost >= nodeBound)
- This tells us that we get more pruning if:
  - The currentBestSolution is high
  - And the nodeBound is low
- So we want to find a high solution quickly and we want the highest possible upper bound
  - One has to factor in the extra computation cost of computing higher upper bounds vs. the expected pruning savings

Example: Assignment Problem

Select one element in each row of the cost matrix *C* so that:
- no two selected elements are in the same column
- the sum is minimized

Example

	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

<u>Lower bound</u>: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + 4)

Example: First two levels of the state-space tree

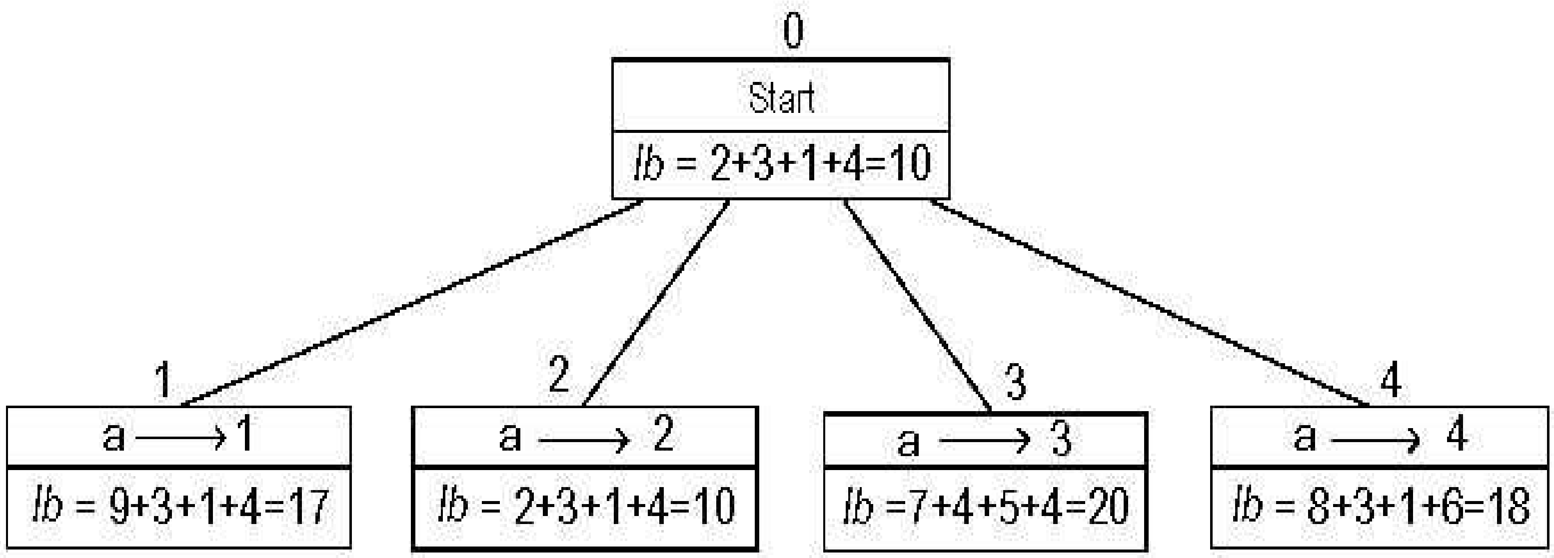


Figure 11.5 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person a and the lower bound value, lb, for this node.

Example (cont.)

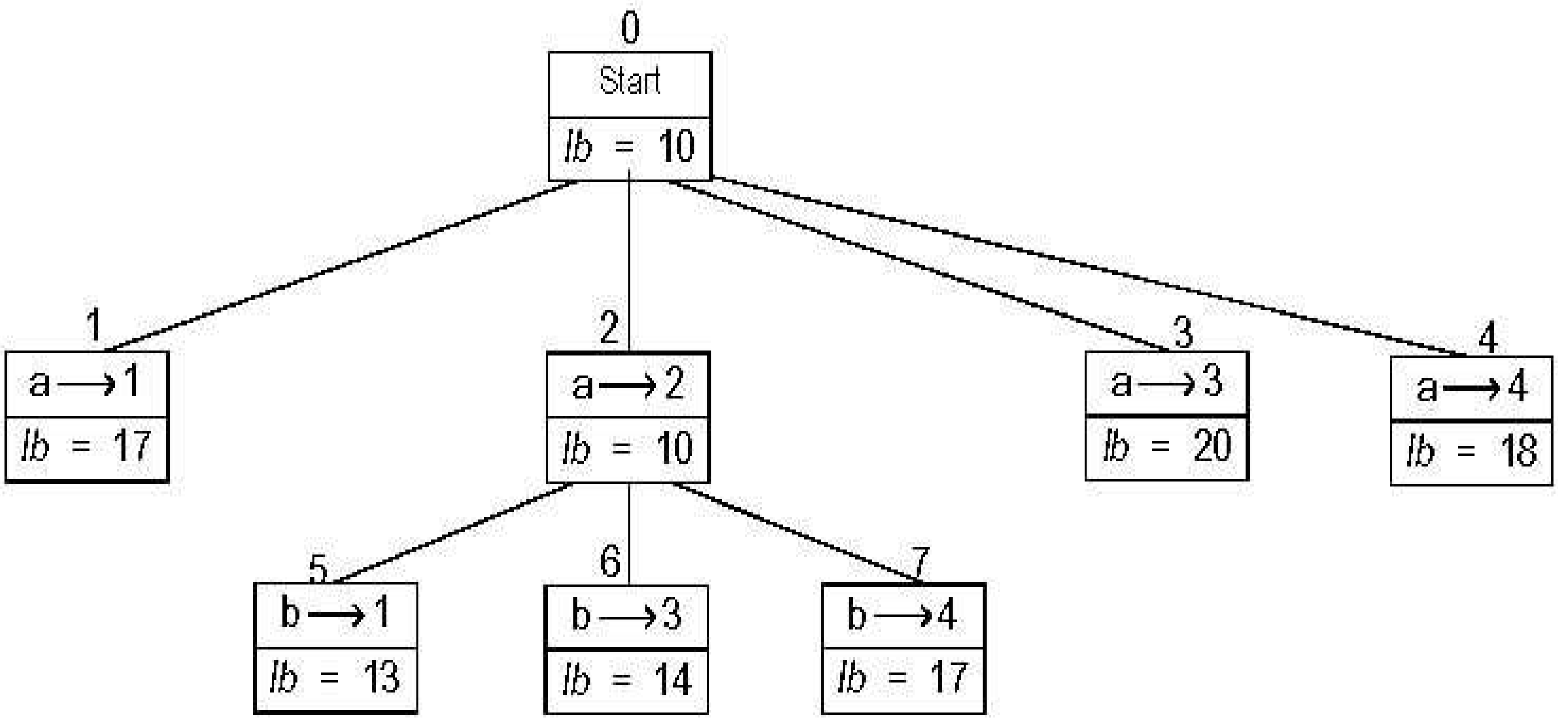


Figure 11.6 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

Example: Complete state-space tree

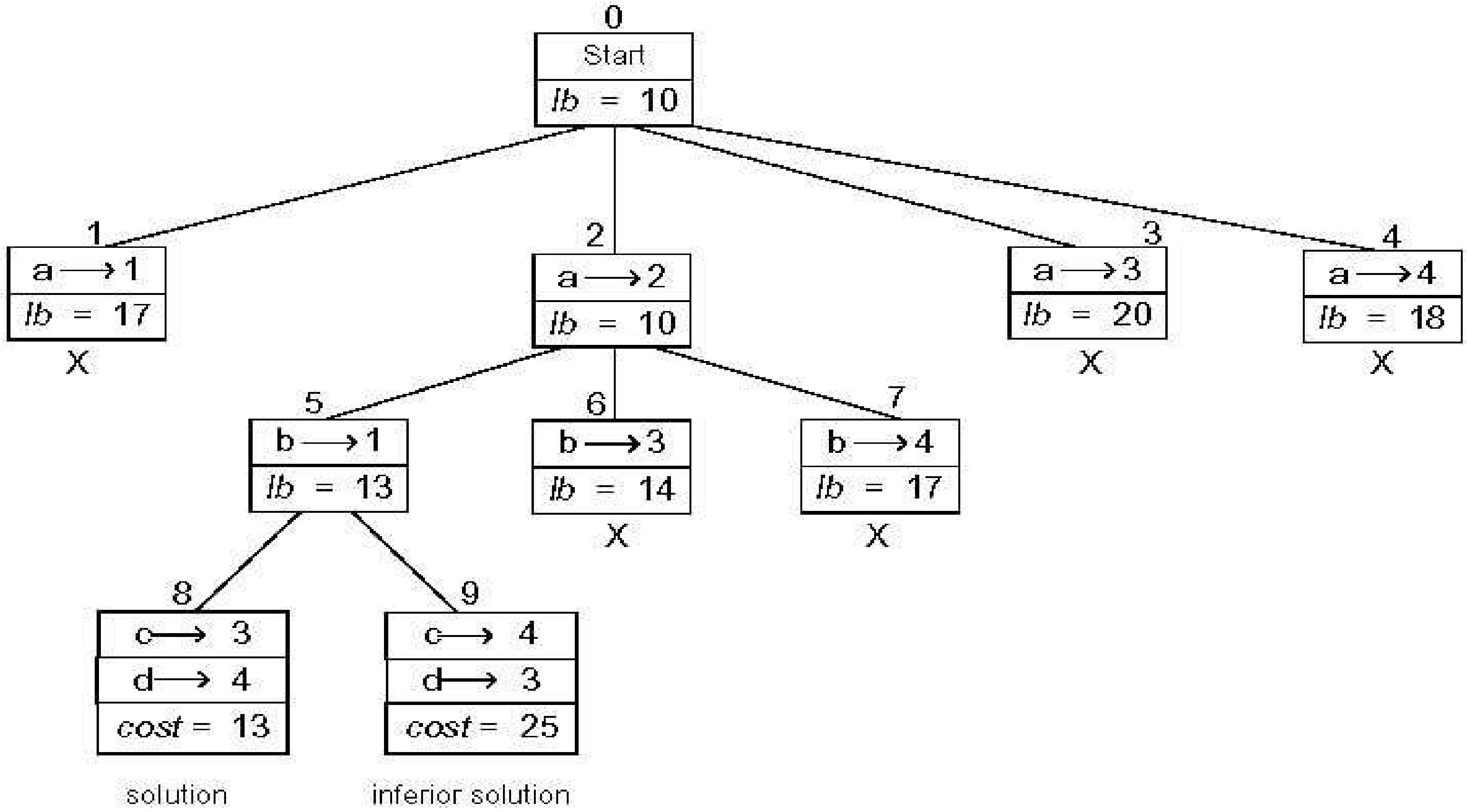


Figure 11.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

Example: Traveling Salesman Problem