

# Exploring Graphs

# Depth-first Search - Directed Graphs

- 
1.  $dfs(1)$  initial call
  2.  $dfs(2)$  recursive call
  3.  $dfs(3)$  recursive call; progress is blocked
  4.  $dfs(4)$  a neighbour of node 1 has not been visited
  5.  $dfs(8)$  recursive call
  6.  $dfs(7)$  recursive call; progress is blocked
  7.  $dfs(5)$  new starting point
  8.  $dfs(6)$  recursive call; progress is blocked
  9. there are no more nodes to visit

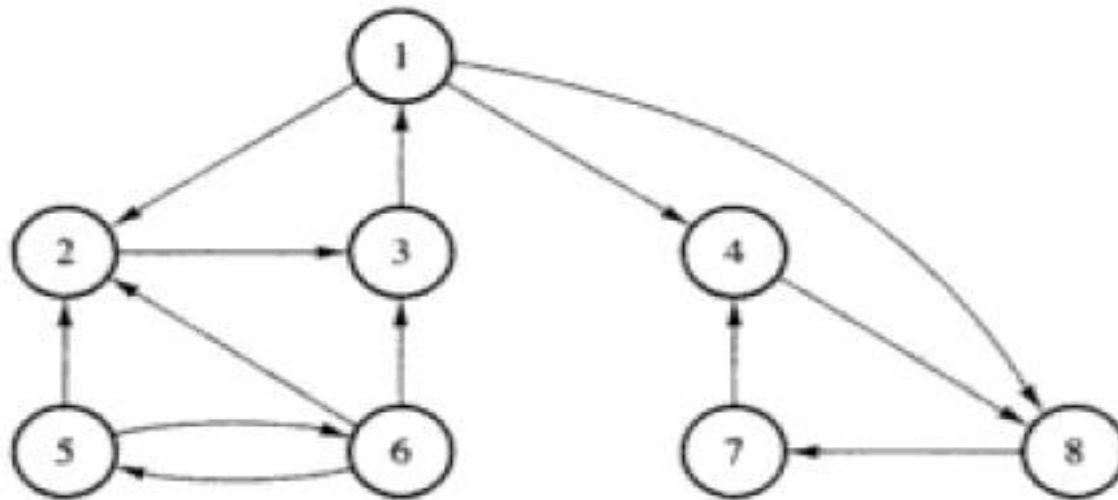


Figure 9.5. A directed graph

# Depth-first Search - Directed/ Undirected Graphs



```
procedure dfsearch(G)  
  for each  $v \in N$  do mark[v] ← not-visited  
  for each  $v \in N$  do  
    if mark[v] ≠ visited then dfs(v)  
  
procedure dfs(v)  
  {Node v has not previously been visited}  
  mark[v] ← visited  
  for each node w adjacent to v do  
    if mark[w] ≠ visited then dfs(w)
```

# Backtracking

- Many problems translate to searching for a specific node, path or pattern in the associated graph.

# Backtracking

- Many problems translate to searching for a specific node, path or pattern in the associated graph.
- If the graph contains a large number of nodes, and particularly if it is infinite, it may be wasteful or infeasible to build it explicitly in computer storage before applying one of the search techniques we have discussed so far.

# Backtracking

- Many problems translate to searching for a specific node, path or pattern in the associated graph.
- If the graph contains a large number of nodes, and particularly if it is infinite, it may be wasteful or infeasible to build it explicitly in computer storage before applying one of the search techniques we have discussed so far.
- In such a situation we use an implicit graph for which description of its nodes and edges is available and relevant parts of it can be built as the search progresses.

# Backtracking

- Many problems translate to searching for a specific node, path or pattern in the associated graph.
- If the graph contains a large number of nodes, and particularly if it is infinite, it may be wasteful or infeasible to build it explicitly in computer storage before applying one of the search techniques we have discussed so far.
- In such a situation we use an implicit graph for which description of its nodes and edges is available and relevant parts of it can be built as the search progresses.
- In this way, computing time is saved whenever the search succeeds before the entire graph has been constructed.

# Backtracking

- Many problems translate to searching for a specific node, path or pattern in the associated graph.
- If the graph contains a large number of nodes, and particularly if it is infinite, it may be wasteful or infeasible to build it explicitly in computer storage before applying one of the search techniques we have discussed so far.
- In such a situation we use an implicit graph for which description of its nodes and edges is available and relevant parts of it can be built as the search progresses.
- In this way, computing time is saved whenever the search succeeds before the entire graph has been constructed.
- The economy in storage space can also be dramatic, especially when nodes that have already been searched can be discarded, making room for subsequent nodes to be explored.



# Backtracking

- If the graph involved is infinite, such a technique offers us the only way of exploring it.

# Backtracking

- If the graph involved is infinite, such a technique offers us the only way of exploring it.
- In its basic form, backtracking resembles a depth-first search in a directed graph.

# Backtracking

- If the graph involved is infinite, such a technique offers us the only way of exploring it.
- In its basic form, backtracking resembles a depth-first search in a directed graph.
- The graph concerned is usually a tree, or at least it contains no cycles.

# Backtracking

- If the graph involved is infinite, such a technique offers us the only way of exploring it.
- In its basic form, backtracking resembles a depth-first search in a directed graph.
- The graph concerned is usually a tree, or at least it contains no cycles.
- Whatever its structure, the graph exists only implicitly.

# Backtracking

- 0/1 Knapsack Problem:
  - We have certain number of objects and a knapsack.
  - Specifically we have  $n$  types of object (**not  $n$  objects**), and that an adequate number of objects of each type are available.
- Instance:
  - Four types of objects, whose weights are respectively 2, 3, 4 and 5 units, and whose values are 3, 5, 6, and 10. The knapsack can carry a maximum of 8 units of weight.

# Backtracking

## ➤ 0/1 Knapsack Problem:

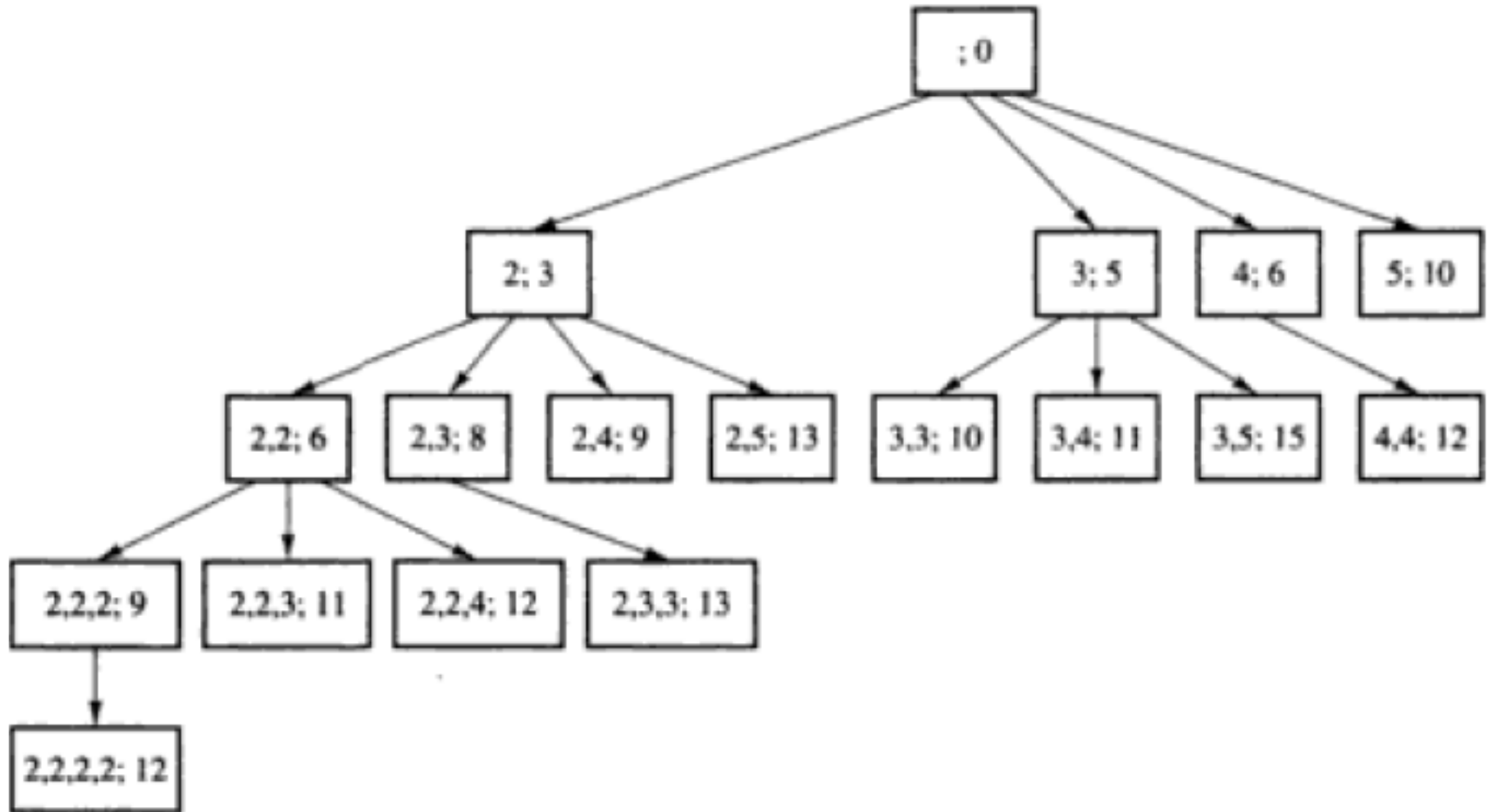


Figure 9.12. The implicit tree for a knapsack problem

# Backtracking

- 0/1 Knapsack Problem:
  - Here, a node such as (2,3; 8) corresponds to a partial solution of the problem.
  - The figures to the left of the semicolon are the weights of the objects we have decided to include, and the figure to the right is the current value of the knapsack.
  - Moving down from a node to one of its children corresponds to deciding which kind of object to put into the knapsack next.
  - We may agree to load objects into the knapsack in order of increasing weight.
  - This is not essential, and indeed any other order - by decreasing weight, for example, or by value-would work just as well.

# Backtracking

- 0/1 Knapsack Problem:
  - Initially the partial solution is empty.
  - The backtracking algorithm explores the tree in a dfs manner, constructing nodes and partial solution as it goes.
  - In the example, the first node visited is (2; 3), the next is (2, 2; 6), the third is (2, 2, 2; 9) and the fourth (2, 2, 2, 2; 12).
  - As each new node is visited, the partial solution is extended.
  - After visiting these four nodes, the dfs is blocked: node (2, 2, 2, 2; 12) has no unvisited successors (indeed no successors at all), since adding more items to this partial solution would violate the capacity constraint.
  - Since this partial solution may turn out to be the optimal solution to our instance, we memorize it.



# Backtracking

- 0/1 Knapsack Problem:
  - The dfs now backs up to look for other solution. At each step back up the tree, the corresponding item is removed from the partial solution.
  - In the example, the search first backs up to  $(2, 2, 2; 9)$ , which also has no unvisited successor; one step further up the tree, however, at node  $(2, 2; 6)$ , two successors remain to be visited.
  - After exploring nodes  $(2, 2, 3; 11)$  and  $(2, 2, 4; 12)$ , neither of which improves on the solution previously memorized, the search backs up one stage further, and so on.

# Backtracking

- 0/1 Knapsack Problem:
  - Exploring the tree in this way, (2, 3, 3; 13) is found to be a better solution than the one we have, and later (3, 5; 15) is found to be better still. Since no other improvement is made before the search ends, this is the optimal solution to the instance.

# Backtracking

## ➤ 0/1 Knapsack Problem:

Programming the algorithm is straightforward, and illustrates the close relation between recursion and depth-first search. Suppose the values of  $n$  and  $W$ , and of the arrays  $w[1..n]$  and  $v[1..n]$  for the instance to be solved are available as global variables. The ordering of the types of item is unimportant. Define a function *backpack* as follows.

```
function backpack( $i, r$ )  
    {Calculates the value of the best load that can  
     be constructed using items of types  $i$  to  $n$   
     and whose total weight does not exceed  $r$ }  
     $b \leftarrow 0$   
    {Try each allowed kind of item in turn}  
    for  $k \leftarrow i$  to  $n$  do  
        if  $w[k] \leq r$  then  
             $b \leftarrow \max(b, v[k] + \textit{backpack}(k, r - w[k]))$   
    return  $b$ 
```

Now to find the value of the best load, call *backpack*(1,  $W$ ).

# Branch-and-Bound



Like backtracking, branch-and-bound is a technique for exploring an implicit directed graph. Again, this graph is usually acyclic or even a tree. This time, we are looking for the optimal solution to some problem. At each node we calculate a bound on the possible value of any solutions that might lie farther on in the graph. If the bound shows that any such solution must necessarily be worse than the best solution found so far, then we need not go on exploring this part of the graph.

In the simplest version, calculation of the bounds is combined with a breadth-first or a depth-first search, and serves only, as we have just explained, to prune certain branches of a tree or to close paths in a graph. More often, however, the calculated bound is also used to choose which open path looks the most promising, so it can be explored first.

# Branch-and-Bound



## 9.7.1 The assignment problem

In the *assignment problem*,  $n$  agents are to be assigned  $n$  tasks, each agent having exactly one task to perform. If agent  $i$ ,  $1 \leq i \leq n$ , is assigned task  $j$ ,  $1 \leq j \leq n$ , then the cost of performing this particular task will be  $c_{ij}$ . Given the complete matrix of costs, the problem is to assign agents to tasks so as to minimize the total cost of executing the  $n$  tasks.

For example, suppose three agents  $a$ ,  $b$  and  $c$  are to be assigned tasks 1, 2 and 3, and the cost matrix is as follows:

	1	2	3
$a$	4	7	3
$b$	2	6	1
$c$	3	9	4

If we allot task 1 to agent  $a$ , task 2 to agent  $b$ , and task 3 to agent  $c$ , then our total cost will be  $4 + 6 + 4 = 14$ , while if we allot task 3 to agent  $a$ , task 2 to agent  $b$ , and task 1 to agent  $c$ , the cost is only  $3 + 6 + 3 = 12$ . In this particular example, the reader may verify that the optimal assignment is  $a \rightarrow 2$ ,  $b \rightarrow 3$ , and  $c \rightarrow 1$ , whose cost is  $7 + 1 + 3 = 11$ .

# Branch-and-Bound

Suppose we have to solve the instance whose cost matrix is shown in Figure 9.13. To obtain an upper bound on the answer, note that  $a \rightarrow 1, b \rightarrow 2, c \rightarrow 3, d \rightarrow 4$  is one possible solution whose cost is  $11 + 15 + 19 + 28 = 73$ . The optimal solution to the problem cannot cost more than this. Another possible solution is  $a \rightarrow 4, b \rightarrow 3, c \rightarrow 2, d \rightarrow 1$  whose cost is obtained by adding the elements in the other diagonal of the cost matrix, giving  $40 + 13 + 17 + 17 = 87$ . In this case the second solution is no improvement over the first. To obtain a lower bound on the solution, we can argue that whoever executes task 1, the cost will be at least 11; whoever executes task 2, the cost will be at least 12, and so on. Thus adding the smallest elements in each column gives us a lower bound on the answer. In the example, this is  $11 + 12 + 13 + 22 = 58$ . A second lower bound is obtained by adding the smallest elements in each row, on the grounds that each agent must do something. In this case we find  $11 + 13 + 11 + 14 = 49$ , not as useful as the previous lower bound. Pulling these facts together, we know that the answer to our instance lies somewhere in  $[58..73]$ .

	1	2	3	4
<i>a</i>	11	12	18	40
<i>b</i>	14	15	13	22
<i>c</i>	11	17	19	23
<i>d</i>	17	14	20	28

Figure 9.13. The cost matrix for an assignment problem

# Branch-and-Bound



To solve the problem by branch-and-bound, we explore a tree whose nodes correspond to partial assignments. At the root of the tree, no assignments have been made. Subsequently, at each level we fix the assignment of one more agent. At each node we calculate a bound on the solutions that can be obtained by completing the corresponding partial assignment, and we use this bound both to close off paths and to guide the search. Suppose for example that, starting from the root, we decide first to fix the assignment of agent  $a$ . Since there are four ways of doing this, there are four branches from the root. Figure 9.14 illustrates the situation.

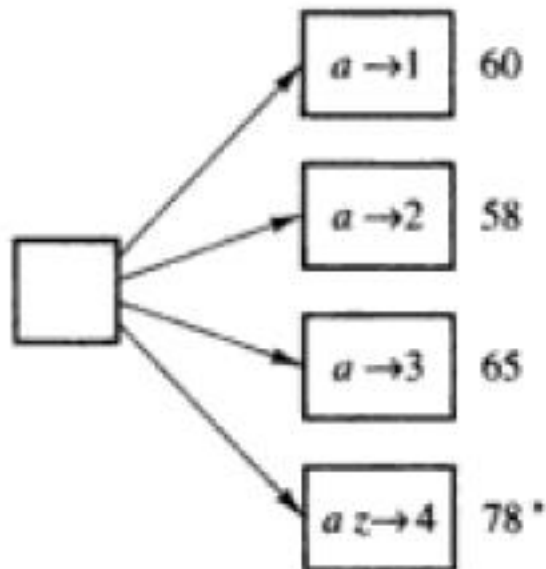


Figure 9.14. After assigning agent  $a$

# Branch-and-Bound

Here the figure next to each node is a lower bound on the solutions that can be obtained by completing the corresponding partial assignment. We have already seen how the bound of 58 at the root can be obtained. To calculate the bound for the node  $a \rightarrow 1$ , for example, note first that with this partial assignment task 1 will cost 11. Task 2 will be executed by  $b$ ,  $c$  or  $d$ , so the lowest possible cost is 14. Similarly tasks 3 and 4 will also be executed by  $b$ ,  $c$  or  $d$ , and their lowest possible costs will be 13 and 22 respectively. Thus a lower bound on any solution obtained by completing the partial assignment  $a \rightarrow 1$  is  $11 + 14 + 13 + 22 = 60$ . Similarly for the node  $a \rightarrow 2$ , task 2 will be executed by agent  $a$  at a cost of 12, while tasks 1, 3 and 4 will be executed by agents  $b$ ,  $c$  and  $d$  at a minimum cost of 11, 13 and 22 respectively. Thus any solution that includes the assignment  $a \rightarrow 2$  will cost at least  $12 + 11 + 13 + 22 = 58$ . The other two lower bounds are obtained similarly. Since we know the optimal solution cannot exceed 73, it is already clear that there is no point in exploring the node  $a \rightarrow 4$  any further: any solution obtained by completing this partial assignment will cost at least 78, so it cannot be optimal. The asterisk on this node indicates that it is “dead”. However the other three nodes are still alive. Node  $a \rightarrow 2$  has the smallest lower bound. Arguing that it therefore looks more promising than the others, this is the one to explore next. We do this by fixing one more element in the partial assignment, say  $b$ . In this way we arrive at the situation shown in Figure 9.15.



# Branch-and-Bound

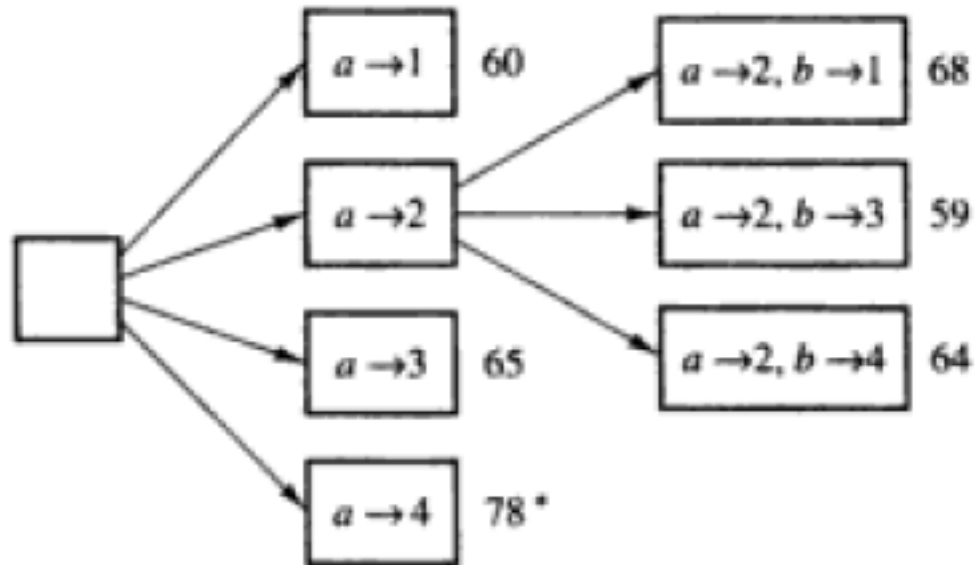


Figure 9.15. After assigning agent  $b$

# Branch-and-Bound

Again, the figure next to each node gives a lower bound on the cost of solutions that can be obtained by completing the corresponding partial assignment. For example, at node  $a \rightarrow 2, b \rightarrow 1$ , task 1 will cost 14 and task 2 will cost 12. The remaining tasks 3 and 4 must be executed by  $c$  or  $d$ . The smallest possible cost for task 3 is thus 19, while that for task 4 is 23. Hence a lower bound on the possible solutions is  $14 + 12 + 19 + 23 = 68$ . The other two new bounds are calculated similarly.

The most promising node in the tree is now  $a \rightarrow 2, b \rightarrow 3$  with a lower bound of 59. To continue exploring the tree starting at this node, we fix one more element in the partial assignment, say  $c$ . When the assignments of  $a$ ,  $b$  and  $c$  are fixed, however, we no longer have any choice about how we assign  $d$ , so the solution is complete. The right-hand nodes in Figure 9.16, which shows the next stage of our exploration, therefore correspond to complete solutions.

# Branch-and-Bound

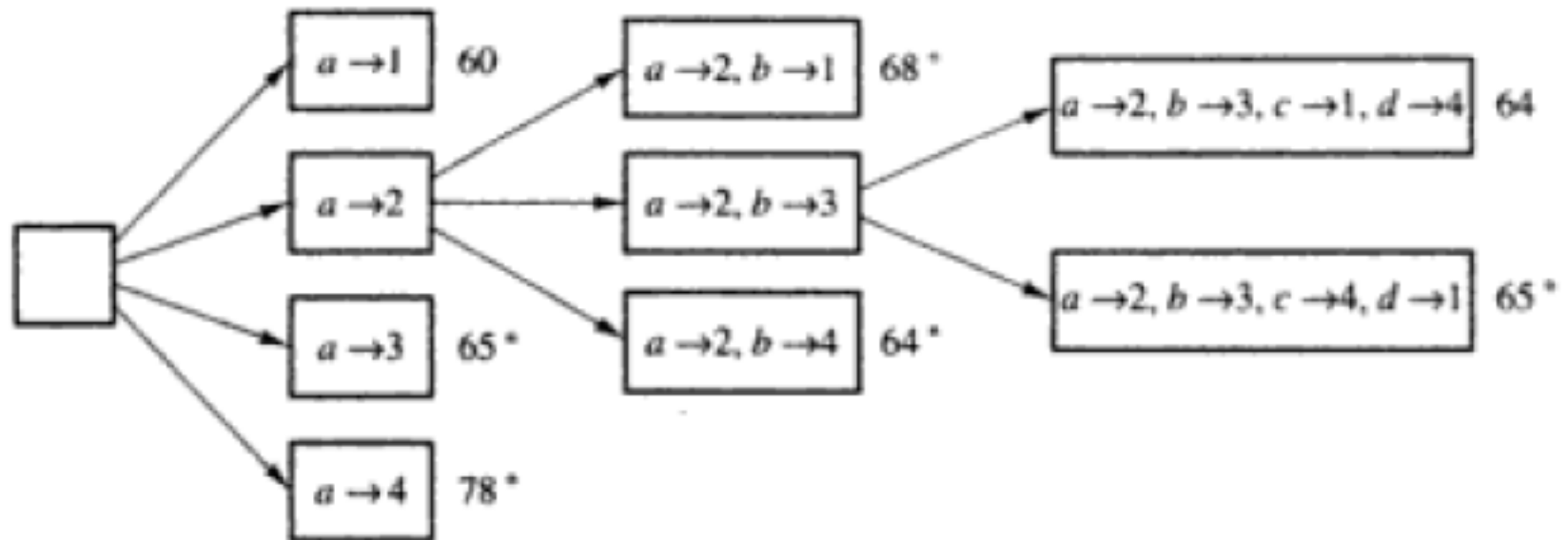


Figure 9.16. After assigning agent  $c$

The solution  $a \rightarrow 2, b \rightarrow 3, c \rightarrow 1, d \rightarrow 4$ , with a cost of 64, is better than either of the solutions we found at the outset, and provides us with a new upper bound on the optimum. Thanks to this new upper bound, we can remove nodes  $a \rightarrow 3$  and  $a \rightarrow 2, b \rightarrow 1$  from further consideration, as indicated by the asterisks. No solution that completes these partial assignments can be as good as the one we have just found. If we only want one solution to the instance, we can eliminate node  $a \rightarrow 2, b \rightarrow 4$  as well.

# Branch-and-Bound



The only node still worth exploring is  $a \rightarrow 1$ . Proceeding as before, after two steps we obtain the final situation shown in Figure 9.17. The best solution found is  $a \rightarrow 1, b \rightarrow 3, c \rightarrow 4, d \rightarrow 2$  with a cost of 61. At the remaining unexplored nodes the lower bound is greater than 61, so there is no point in exploring them further. The solution above is therefore the optimal solution to our instance.

The example illustrates that, although at an early stage node  $a \rightarrow 2$  was the most promising, the optimal solution did not in fact come from there. To obtain our answer, we constructed just 15 of the 41 nodes (1 root, 4 at depth 1, 12 at depth 2, and 24 at depth 3) that are present in a complete tree of the type illustrated. Of the 24 possible solutions, only 6 (including the two used to determine the initial upper bound) were examined.

# Branch-and-Bound

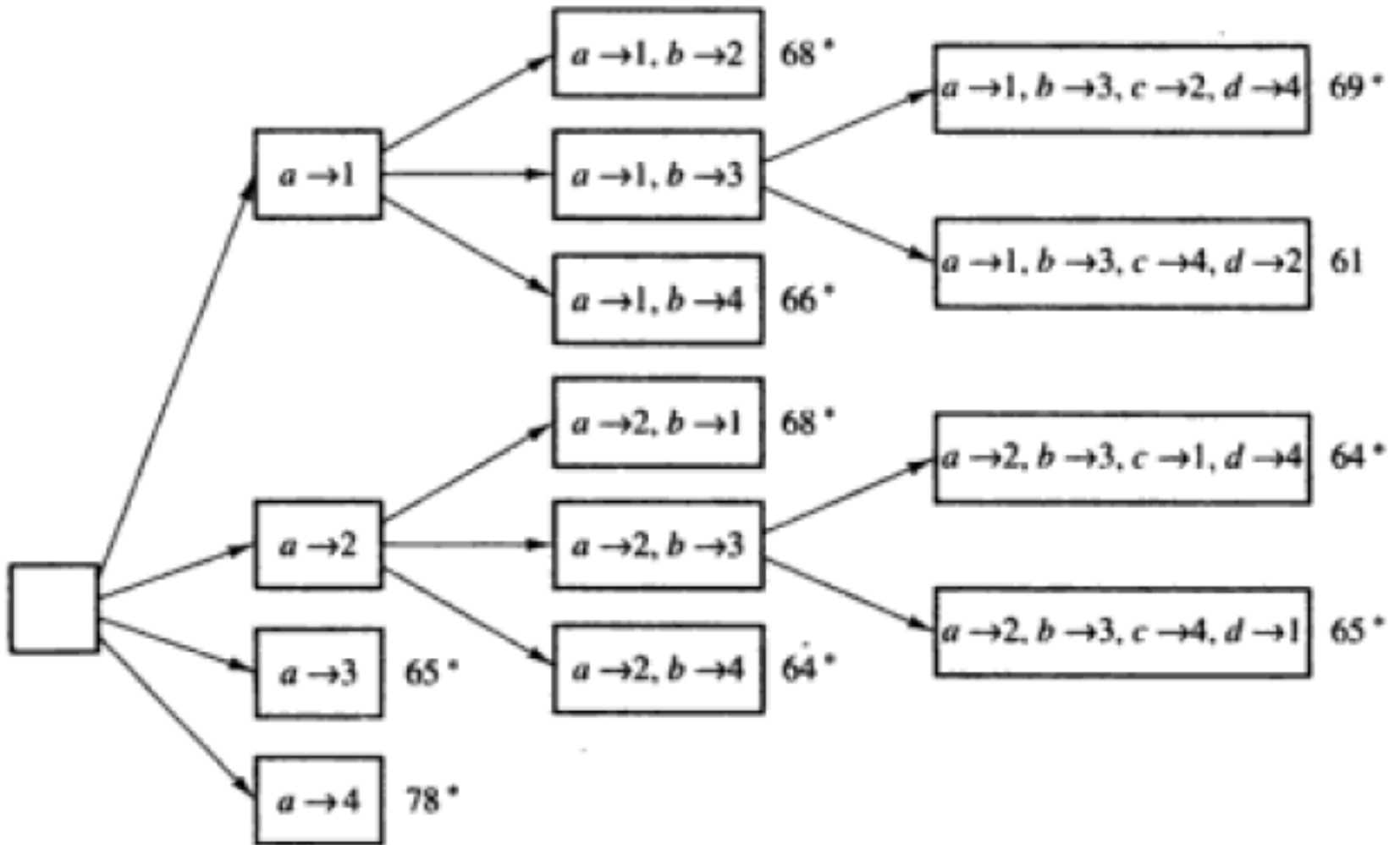


Figure 9.17. The tree completely explored

# Hungarian Algorithm for Assignment Problem

➤ You work as a manager for a chip manufacturer, and you currently have 3 people on the road meeting clients. Your sales-people are in Jaipur, Pune and Bangalore, and you want them to fly to three other cities: Delhi, Mumbai and Kerala. The table below shows the cost of airline tickets in INR between the cities:

	Delhi	Kerala	Mumbai
Jaipur	2500	4000	3500
Pune	4000	6000	3500
Bangalore	2000	4000	2500

➤ The question: where would you send each of your salespeople in order to minimize fair?

# Hungarian Algorithm for Assignment Problem

- For each row of the matrix, find the smallest element and subtract it from every element in its row.
- Do the same (as step 1) for all columns.
- Cover all zeros in the matrix using minimum number of horizontal and vertical lines.
- Test for Optimality: If the minimum number of covering lines is  $n$ , an optimal assignment is possible and we are finished. Else if lines are lesser than  $n$ , we haven't found the optimal assignment, and must proceed to step 5.
- Determine the smallest entry not covered by any line. Subtract this entry from each uncovered row, and then add it to each covered column. Return to step 3.

# Hungarian Algorithm for Assignment Problem

- Below is the cost matrix of example given in above diagrams.

2500 4000 3500

4000 6000 3500

2000 4000 2500

- Step 1: Subtract minimum of every row.

0 1500 1000

500 2500 0

0 2000 500

- Step 2: Subtract minimum of every column.

0 0 1000

500 1000 0

0 500 500



# Hungarian Algorithm for Assignment Problem

- Step 3: Cover all zeroes with minimum number of horizontal and vertical lines.

0	0	1000
500	1000	0
0	500	500

- Step 4: Since we need 3 lines to cover all zeroes, we have found the optimal assignment.

2500 **4000** 3500

4000 6000 **3500**

**2000** 4000 2500

- So the optimal cost is  $4000 + 3500 + 2000 = 9500$

# Hungarian Algorithm for Assignment Problem - Another Example

➤ cost matrix:

1500 4000 4500

2000 6000 3500

2000 4000 2500

➤ Step 1: Subtract minimum of every row.

0 2500 3000

0 4000 1500

0 2000 500

➤ Step 2: Subtract minimum of every column.

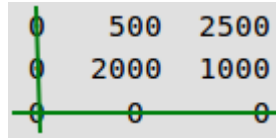
0 500 2500

0 2000 1000

0 0 0

# Hungarian Algorithm for Assignment Problem - Another Example

- Cover all zeroes with minimum number of horizontal and vertical lines.



0	500	2500
0	2000	1000
0	0	0

- Since we only need 2 lines to cover all zeroes, we have NOT found the optimal assignment.

- Step 5: We subtract the smallest uncovered entry from all uncovered rows.

-500	0	2000
-500	1500	500
0	0	0

- Then we add the smallest entry to all covered columns, we get

0	0	2000
0	1500	500
500	0	0

# Hungarian Algorithm for Assignment Problem - Another Example

➤ Now we return to Step 3:. Here we cover again using lines. and go to Step 4:. Since we need 3 lines to cover, we found the optimal solution.

1500 **4000** 4500

**2000** 6000 3500

2000 4000 **2500**

➤ So the optimal cost is  $4000 + 2000 + 2500 = 8500$

# Disclaimer

➤ Contents of this presentation are not original and they have been prepared from various sources just for the teaching purpose.