Backtracking

BackTracking Introduction

Backtracking is used to solve problems in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The classic example of the use of backtracking is in the n-Queens problem. The goal in this problem is to position n queens on an n × n chessboard so that no two queens threaten each other. That is, no two queens may be in the same row, column, or diagonal. The sequence in this problem is the n positions in which the queens are placed, the set for each choice is the n2 possible positions on the chessboard, and the criterion is that no two queens can threaten each other. The n-Queens problem is a generalization of its instance when n = 8, which is the instance using a standard chessboard.

Backtracking is a modified depth-first search of a tree.

Now let's illustrate the backtracking technique with the instance of the n-Queens problem when n = 4. Our task is to position four queens on a 4 × 4 chessboard so that no two queens threaten each other.
We can immediately simplify matters by realizing that no two queens can be in the same column. The instance can then be solved by assigning each queen a different column and checking which row combinations yield solutions. Because each queen can be placed in one of four rows, there are 4 × 4 × 4 × 4 = 256 candidate solutions.

We can create the candidate solutions by constructing a tree in which the row choices for the first queen (the queen in column 1) are stored in level-1 nodes in the tree (recall that the root is at level 0), the row choices for the second queen (the queen in column 2) are stored in level-2 nodes, and so on.

A path from the root to a leaf is a candidate solution. This tree is called a state space tree. The entire tree has 256 row leaves, one for each candidate solution.

Backtracking is the procedure whereby, after determining that a node can lead to nothing but dead ends, we go back ("backtrack") to the node's parent and proceed with the search on the next child. We call a node nonpromising if when visiting the node we determine that it cannot possibly lead to a solution. Otherwise, we call it promising. To summarize, backtracking consists of doing a depth-first search of a state space tree, checking whether each node is promising, and, if it is nonpromising, backtracking to the node's parent. This is called pruning the state space tree, and the subtree consisting of the visited nodes is called the pruned state space tree. A general algorithm for the backtracking approach is as follows:

```
void checknode (node v)
  {
 node u;
```

```
if (promising(v))
  if (there is a solution at v)
    write the solution;
  else
    for (each child u of v)
      checknode(u);
}
```

The root of the state space tree is passed to checknode at the top level. A visit to a node consists of first checking whether it is promising. If it is promising and there is a solution at the node, the solution is printed. If there is not a solution at a promising node, the children of the node are visited. The function promising is different in each application of backtracking. We call it the promising function for the algorithm. A backtracking algorithm is identical to a depth-first search, except that the children of a node are visited only when the node is promising and there is not a solution at the node. (Unlike the algorithm for the n-Queens problem, in some backtracking algorithms a solution can be found before reaching a leaf in the state space tree.) We have called the backtracking procedure checknode rather than backtrack because backtracking does not occur when the procedure is called. Rather, it occurs when we find that a node is nonpromising and proceed to the next child of the parent.

N-Queens problem

Problem: Position n queens on a chessboard so that no two are in the same row, column, or diagonal.

Inputs: positive integer n.

Outputs: all possible ways n queens can be placed on an n × n chessboard so that no two queens threaten each other. Each output consists of an array of integers col indexed from 1 to n, were row[i] is the column where the queen in the ith row is placed.

```
void queens(index i)
{
 index j;

 if (promising (i))
 {
    if (i == n) // solution is found
         print row[] array;
    else

// the for loop tries to place the queen in the (i+1)th column on each the n rows.
        for (j = 1; j <= n; j++)
```

```
        {
          row[i + 1] = j;

   queens (i + 1);

        }
   }
}
```

```
bool promising (index i)
{
   index k;
   bool okay;
   k = 1;
   okay = true;
   while (k < i && okay)
   {
      if (row[i] == row[k] ||
            abs (row[i] - row[k] == i-k)
         okay = false;
       k++;
  }
  return okay;
}
```

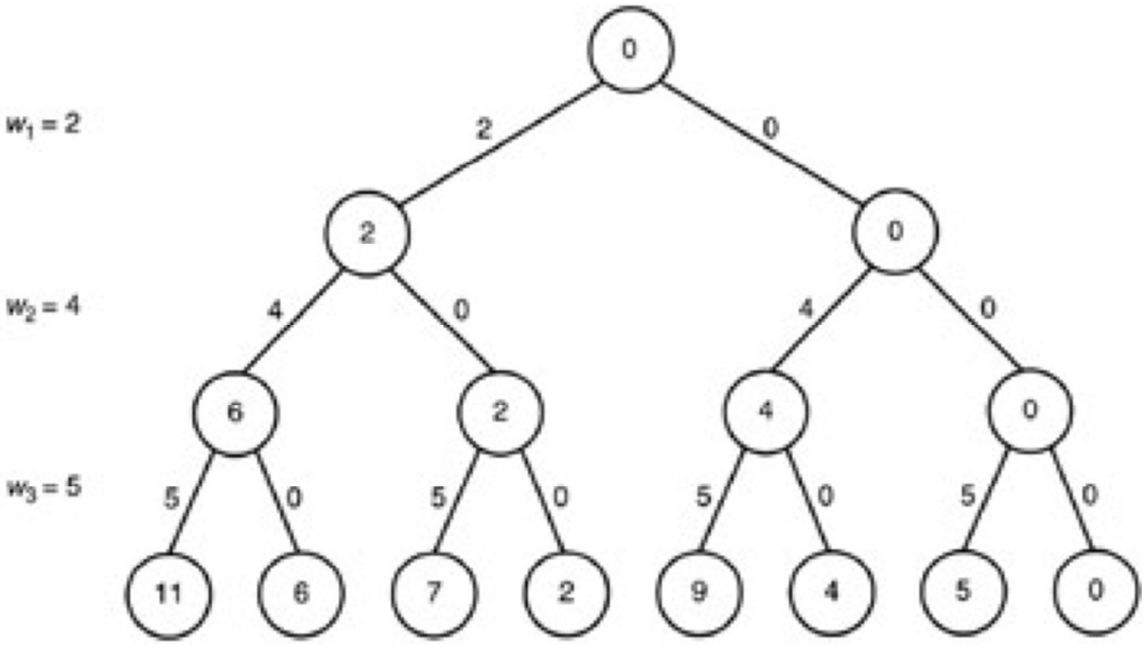
Sum-of-Subsets Problem

In the Sum-of-Subsets problem, there are n positive integers (weights) w_i and a positive integer W. The goal is to find all subsets of the integers that sum to W. As mentioned earlier, we usually state our problems so as to find all solutions

We create a state space tree. A possible way to structure the tree is:-

The state space tree for n = 3, **W** = 6, and

w1 = 2 w2 = 4 w3 = 5



We go to the left from the root to include w1, and we go to the right to exclude w1. Similarly, we go to the left from a node at level 1 to include w2, and we go to the right to exclude w2, etc. Each subset is represented by a path from the root to a leaf. When we include wi, we write wi on the edge where we include it. When we do not include wi, we write 0.∑

At each node, we have written the sum of the weights that have been included up to that point. Therefore, each leaf contains the sum of the weights in the subset leading to that leaf. The second leaf from the left is the only one containing a 6. Because the path to this leaf represents the subset {w1, w2}, this subset is the only solution.

If we sort the weights in nondecreasing order before doing the search, there is an obvious sign telling us that a node is nonpromising. If the weights are sorted in this manner, then wi+1 is the lightest weight remaining when we are at the ith level. Let weight be the sum of the weights that have been included up to a node at level i. If wi+1 would bring the value of weight above W, then so would any other weight following it. Therefore, unless weight equals W (which means that there is a solution at the node), a node at the ith level is nonpromising if

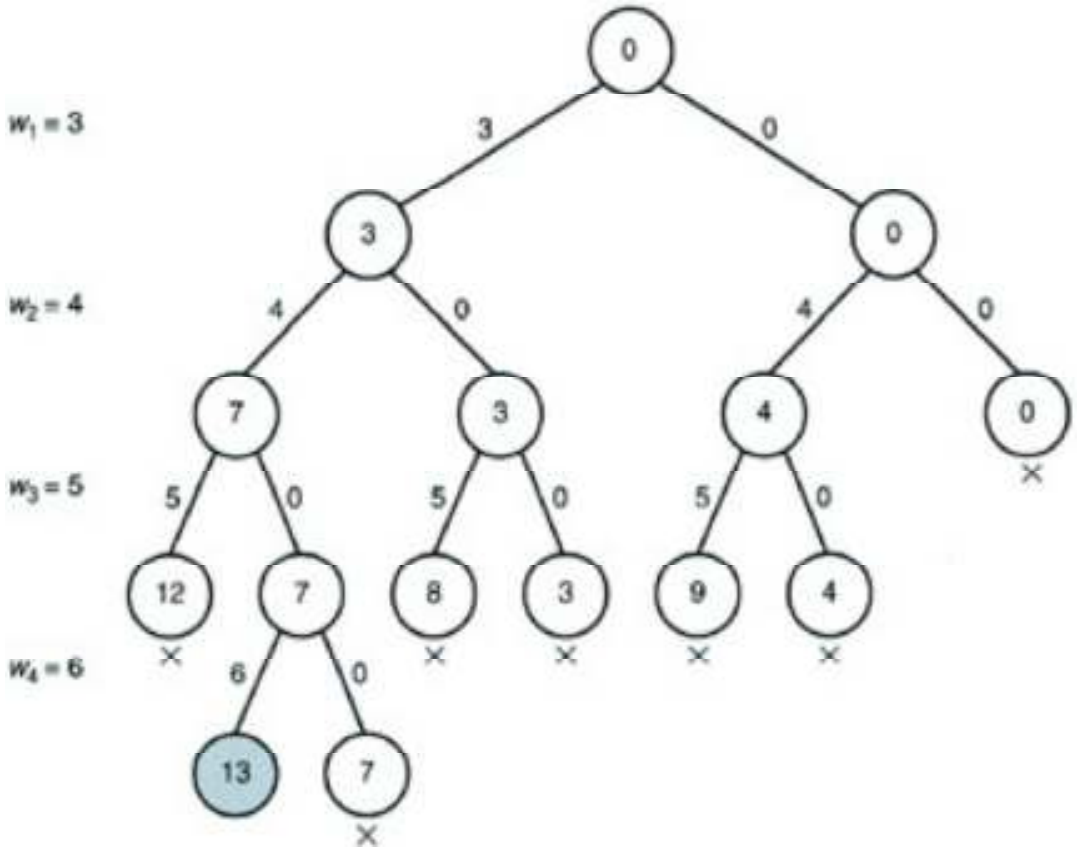
weight + w_{i+1} > W

There is another, less obvious sign telling us that a node is nonpromising. If, at a given node, adding all the weights of the remaining items to weight does not make weight at least equal to W, then weight could never become equal to W by expanding beyond the node. This means that if total is the total weight of the remaining weights, a node is nonpromising if

weight + total < W

The figure shows the pruned state space tree when backtracking is used with n = 4, W = 13, and

w1 = 3, w2 = 4, w3 = 5, w4 = 6



The pruned state space tree produced using backtracking. Stored at each node is the total weight included up to that node. Each nonpromising node is marked with a cross.

The only solution is found at the 5th level with total of 13. The solution is {w1, w2, w4}. The nonpromising nodes are marked with crosses. The nodes containing 12, 8, and 9 are nonpromising because adding the next weight (6) would make the value of weight exceed W. The nodes containing 7, 3, 4, and 0 are nonpromising because there is not enough total weight remaining to bring the value of weight up to W. Notice that a leaf in the state space tree that does not contain a solution is automatically nonpromising because there are no weights remaining that could bring weight up to W. The leaf containing 7 illustrates this. There are only 15 nodes in the pruned state space tree, whereas the entire state space tree contains 31 nodes.

When the sum of the weights included up to a node equals W, there is a solution at that node. Therefore, we cannot get another solution by including more items. This means that if

W = weight,

we should print the solution and backtrack. This backtracking is provided automatically by our general procedure checknode because it never expands beyond a promising node where a solution is found. This is an example of a backtracking algorithms where a solution is found before reaching a leaf in the state space tree.

Next we present the algorithm that employs these strategies. The algorithm uses an array include. It sets include[i] to"yes" if w[i] is to be included and to "no" if it is not.

Problem: Given n positive integers (weights) and a positive integer W, determine all combinations of the integers that sum to W.

Inputs: positive integer n, sorted (nondecreasing order) array of positive integers w indexed from 1 to n, and a positive integer W.

Outputs: all combinations of the integers that sum to W.

```
void sum_of_subsets (index i,int weight, int total)
{
 if (promising (i))
       if (weight == W)
            print include[] array;
       else{

          include [i + 1] = TRUE;

          sum_of_subsets ( i+1 , weight+w[i+1], total-w[i+1]);
          include [i + 1] = FALSE;
        sum_of_subsets (i+1, weight, total - w[i+1]);
       }
  }

bool promising (index i);
{
   bool okay = true;

   // adding the next weight will exceed W
   if (weight + w[i + 1] <= W)
      okay = false;

   //no matter what we add, it will never match up or add up to W.
   if (weight + total < W)
      okay = false;

   // redundant condition, only to understand that a solution is reached, and hence node is
promising
   if ((weight == W)
```

```
        okay = true;

    return okay;
}
```

From the main() method, we make a call to :-

sum_of_subsets(0,0,total);

The number of nodes in the state space tree searched by is equal to

$$1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1.$$

0/1 Knapsack

In this problem we have a set of items, each of which has a weight and a profit. The weights and profits are positive integers. A thief plans to carry off stolen items in a knapsack, and the knapsack will break if the total weight of the items placed in it exceeds some positive integer W. The thief's objective is to determine a set of items that maximizes the total profit under the constraint that the total weight cannot exceed W.

We can solve this problem using a state space tree exactly like the one in the Sum-of-Subsets problem. That is, we go to the left from the root to include the first item, and we go to the right to exclude it. Similarly, we go to the left from a node at level 1 to include the second item, and we go to the right to exclude it, and so on. Each path from the root to a leaf is a candidate solution.

This problem is different from the others discussed in this chapter in that it is an optimization problem. This means that we do not know if a node contains a solution until the search is over. Therefore, we backtrack a little differently. If the items included up to a node have a greater total profit than the best solution so far, we change the value of the best solution so far. However, we may still find a better solution at one of the node's descendants (by stealing more items). Therefore, for optimization problems we always visit a promising node's children. The following is a general algorithm for backtracking in the case of optimization problems.

```
void checknode (node v)
  {
 node u;

 if (value(v) is better than best)
       best = value(v);
 if (promising(v))
       for (each child u of v)
            checknode(u);
```

}

The variable best has the value of the best solution found so far, and value (v) is the value of the solution at the node. After best is initialized to a value that is worse than the value of any candidate solution, the root is passed at the top level. Notice that a node is promising only if we should expand to its children. Recall that our other algorithms also call a node promising if there is a solution at the node.

Next we apply this technique to the 0–1 Knapsack problem. First let's look for signs telling us that a node is non-promising. An obvious sign that a node is non-promising is that there is no capacity left in the knapsack for more items. Therefore, if weight is the sum of the weights of the items that have been included up to some node, the node is nonpromising if

weight > W

It is non-promising even if weight equals W because, in the case of optimization problems, "promising" means that we should expand to the children.

We can use considerations from the greedy approach to find a less obvious sign. Here we will only use greedy considerations to limit our search; we will not develop a greedy algorithm. To that end, we first order the items in non-increasing order according to the values of pi/wi, where wi and pi are the weight and profit, respectively, of the ith item. Suppose we are trying to determine whether a particular node is promising. No matter how we choose the remaining items, we cannot obtain a higher profit than we would obtain if we were allowed to use the restrictions in the Fractional Knapsack problem from this node on. Therefore, we can obtain an upper bound on the profit that could be obtained by expanding beyond that node as follows. Let profit be the sum of the profits of the items included up to the node. Recall that weight is the sum of the weights of those items. We initialize variables bound and totweight to profit and weight, respectively. Next we greedily grab items, adding their profits to bound and their weights to totweight, until we get to an item that if grabbed would bring totweight above W. We grab the fraction of that item allowed by the remaining weight, and we add the value of that fraction to bound. If we are able to get only a fraction of this last weight, this node cannot lead to a profit equal to bound, but bound is still an upper bound on the profit we could achieve by expanding beyond the node. Suppose the node is at level i, and the node at level k is the one that would bring the sum of the weights above W. Then

$$totweight = weight + \sum_{j=i+1}^{k-1} w_j, \quad \text{and}$$

$$bound = \underbrace{\left(profit + \sum_{j=i+1}^{k-1} p_j\right)}_{\text{Profit from first } k-1 \text{ items taken}} + \underbrace{(W - totweight)}_{\text{Capacity available for } k\text{th item}} \times \underbrace{\frac{p_k}{w_k}}_{\text{Profit per unit weight for } k\text{th item}}.$$

If maxprofit is the value of the profit in the best solution found so far, then a node at level i is nonpromising if

bound < = maxprofit

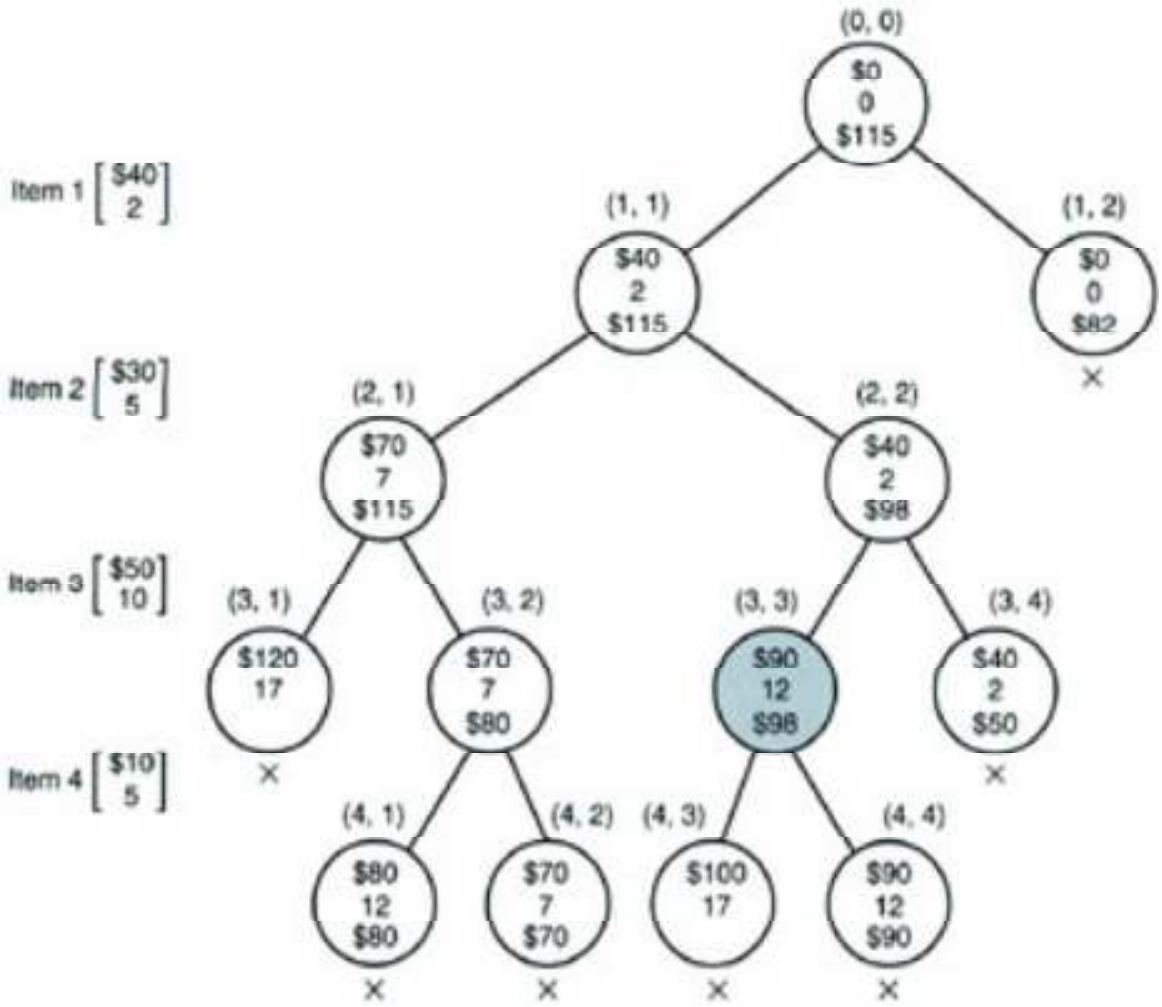
We are using greedy considerations only to obtain a bound that tells us whether we should expand beyond a node. We are not using it to greedily grab items with no opportunity to reconsider later (as is done in the greedy approach).

Before presenting the algorithm, we show an example.

Suppose that n = 4, W = 16, and we have the following:

i	p_i	w_i	$\frac{p_i}{w_i}$
1	$40	2	$20
2	$30	5	$6
3	$50	10	$5
4	$10	5	$2

We have already ordered the items according to pi/wi. For simplicity, we chose values of pi and wi that make pi/wi an integer. In general, this need not be the case. The following figure shows the pruned state space tree produced by using the backtracking considerations just discussed. The total profit, total weight, and bound are specified from top to bottom at each node. These are the values of the variables profit, weight, and bound mentioned in the previous discussion. The maximum profit is found at the node shaded in color. Each node is labeled with its level and its position from the left in the tree. For example, the shaded node is labeled (3, 3) because it is at level 3 and it is the third node from theleft at that level. Next we present the steps that produced the pruned tree. In these steps we refer to a node by its label.



The pruned state space tree produced using backtracking. Stored at each node from top to bottom are the total profit of the items stolen up to the node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The optimal solution is found at the node shaded in color. Each nonpromising node is marked with a cross.

1. Set maxprofit to $0.
2. Visit node (0, 0) (the root).

a. Compute its profit and weight:- profiit = 0 and weight = 0

b. Compute its bound. Because 2 + 5 + 10 = 17, and 17 > 16, the value of W, the third item would bring the sum of the weights above W. Therefore, k = 3, and we have

$$totweight = weight + \sum_{j=0+1}^{3-1} w_j = 0 + 2 + 5 = 7$$

$$bound = profit + \sum_{j=0+1}^{3-1} p_j + (W - totweight) \times \frac{p_3}{w_3}$$

$$= \$0 + \$40 + \$30 + (16 - 7) \times \frac{\$50}{10} = \$115.$$

c. Determine that the node is promising because its weight 0 is less than 16, the value of W, and its bound $115 is greater than $0, the value of maxprofit

3. Visit node (1, 1)

a. Compute its profit and weight.

profit = 0 + 40 = $40

weight = 0 + 2 = 2

b. Because its weight 2 is less than or equal to 16, the value of W, and its profit $40 is greater than $0, the value of maxprofit, set maxprofit to $40.

c. Compute its bound. Because 2 + 5 + 10 = 17, and 17 > 16, the value of W, the third item would bring the sum of the weights above W. Therefore, k = 3, and we have

$$totweight = weight + \sum_{j=1+1}^{3-1} w_j = 2 + 5 = 7$$

$$bound = profit + \sum_{j=1+1}^{3-1} p_j + (W - totweight) \times \frac{p_3}{w_3}$$

$$= \$40 + \$30 + (16 - 7) \times \frac{\$50}{10} = \$115.$$

d. Determine that the node is promising because its weight 2 is less than 16, the value of W, and its bound $115 is greater than $0, the value of maxprofit.

4. Visit node (2, 1).

a. Compute its profit and weight.

profit = 40 + 30 = 70

weight = 2 + 5 = 7

b. Because its weight 7 is less than or equal to 16, the value of W, and its profit $70 is greater than $40, the value of maxprofit, set maxprofit to $70.

c. Compute its bound.

d. Determine that the node is promising because its weight 7 is less than 16, the value of W, and its bound $115 is greater than $70, the value of maxprofit.

5. Visit node (3, 1).

a. Compute its profit and weight

profit = 70 + 50 = 120

weight = 7 + 10 = 17

b. Because its weight 17 is greater than 16, the value of W, maxprofit does not change.
c. Determine that it is nonpromising because its weight 17 is greater than or equal to 16, the value of W.
d. The bound for this node is not computed, because its weight has determined it to be nonpromising.

6. Backtrack to node (2, 1)
7. Visit node (3, 2).
a. Compute its profit and weight. Because we are not including item 3,
profit = 70, weight = 7

b. Because its profit $70 is less than or equal to $70, the value of maxprofit, maxprofit does not change.
c. Compute its bound. The fourth weight would not bring the sum of the items above W, and there are only four items. Therefore, k = 5, and

bound = 80

d. Determine that the node is promising because its weight 7 is less than 16, the value of W, and its bound $80 is greater than $70, the value of maxprofit. (From now on we leave the computations of profits, weights, and bounds as exercises. Furthermore, when maxprofit does not change, we will not mention it.)

8. Visit node (4, 1).
a. Compute its profit and weight to be $80 and 12.
b. Because its weight 12 is less than or equal to 16, the value of W, and its profit $80 is greater than $70, the value of maxprofit, set maxprofit to $80.
c. Compute its bound to be $80.

d. Determine that it is nonpromising because its bound $80 is less than or equal to $80, the value of maxprofit. Leaves in the state space tree are automatically nonpromising because their bounds are always less than or equal to maxprofit.

9. Backtrack to node (3, 2).
10. Visit node (4, 2).
a. Compute its profit and weight to be $70 and 7.
b. Compute its bound to be $70.
c. Determine that the node is nonpromising because its bound $70 is less than or equal to $80, the value of maxprofit.

11. Backtrack to node (1, 1)
12. Visit node (2, 2).
a. Compute its profit and weight to be $40 and 2.
b. Compute its bound to be $98.
c. Determine that it is promising because its weight 2 is less than 16, the value of W, and its bound $98 is greater than $80, the value of maxprofit.

13. Visit node (3, 3).
a. Compute its profit and weight to be $90 and 12.

b. Because its weight 12 is less than or equal to 16, the value of W, and its profit $90 is greater than $80, the value of maxprofit, set maxprofit to $90.

c. Compute its bound to be $98.
d. Determine that it is promising because its weight 12 is less than 16, the value of W, and its bound $98 is greater than $90, the value of maxprofit.

14. Visit node (4, 3).
a. Compute its profit and weight to be $100 and 17.
b. Determine that it is nonpromising because its weight 17 is greater than or equal to 16, the value of W.
c. The bound for this node is not computed because its weight has determined it to be nonpromising.
15. Backtrack to node (3, 3).
16. Visit node (4, 4).
a. Compute its profit and weight to be $90 and 12.
b. Compute its bound to be $90.
c. Determine that it is nonpromising because its bound $90 is less than or equal to $90, the value of maxprofit.

17. Backtrack to node (2, 2).
18. Visit node (3, 4).
a. Compute its profit and weight to be $40 and 2.
b. Compute its bound to be $50.
c. Determine that the node is nonpromising because its bound $50 is less than or equal to $90, the value of maxprofit.

19. Backtrack to root.
20. Visit node (1, 2).
a. Compute its profit and weight to be $0 and 0.
b. Compute its bound to be $82.
c. Determine that the node is nonpromising because its bound $82 is less than or equal to $90, the value of maxprofit.

21. Backtrack to root.
Root has no more children. We are done.

There are only 13 nodes in the pruned state space tree, whereas the entire state space tree has 31 nodes.
Next we present the algorithm. Because this is an optimization problem, we have the added task of keeping track of the current best set of items and the total value of their profits. We do this in an array bestset and a variable maxprofit. Unlike the other problems in this chapter, we state this problem so as to find just one optimal solution.

Problem: Let n items be given, where each item has a weight and a profit. The weights and profits are positive integers. Furthermore, let a positive integer W be given. Determine a set of items with maximum total profit, under the constraint that the sum of their weights cannot exceed W.

Inputs: Positive integers n and W; arrays w and p, each indexed from 1 to n, and each containing positive integers sorted in non-increasing order according to the values of p[i]/w[i].

Outputs: an array bestset indexed from 1 to n, where the values of bestset[i] is "yes" if the ith item is included in the optimal set and is "no" otherwise; an integer maxprofit that is the maximum profit.

```
void knapsack (index i,int profit, int weight)
{
   /* we try to find the best solution so far. The best configuration of the include array, ie the configuration of the include array that gives the maxprofit value is stored in bestset array */

   if (weight maxprofit)
   {
      maxprofit = profit;                numbest = i;                bestset = include;
   }

   if (promising(i))
   {   /* the 2 children of a node are to include the next item, or not include it */

      include [i + 1] = "yes";
     knapsack(i + 1, profit + p[i + 1],              weight + w[i + 1]);

      include [i + 1] = "no";
      knapsack (i + 1, profit, weight);
   }
}
```

```
bool promising (index i)
{
   index j, k;
   int totweight;
   float bound;

   //basic constraint: weight exceeding W
   if (weight >= W)        return false;

   else
   {
      /* the following section computes bound for the current node. Remember that bound is sum of profit and the sum of the profits of remaining items in fractional knapsack calculations */

      j = i + 1;
      bound = profit;
      totweight = weight;

/* the foll while loop goes on as long as there are items, ie 1st condition of j<=n AND as long as
```

the item can be selected ENTIRELY, which is the 2nd condition */

```
        while(j <= n && totweight+w[j] <= W)
        {
            totweight = totweight + w[j];
           bound = bound + p[j];
           j++;
        }
        k = j;

/* the foll if is for selecting a fraction of the last item */
        if (k <=n)
           bound = bound + (W - totweight) * p[k]/w[k];


     return bound > maxprofit;
    }
}
```

From the main method() we have the following initializations and steps to perform:-

```
numbest = 0;
maxprofit = 0;
knapsack(0, 0, 0);
print maxprofit
for (j = 1; j <= numbest; j++)
     print bestset[i];
```

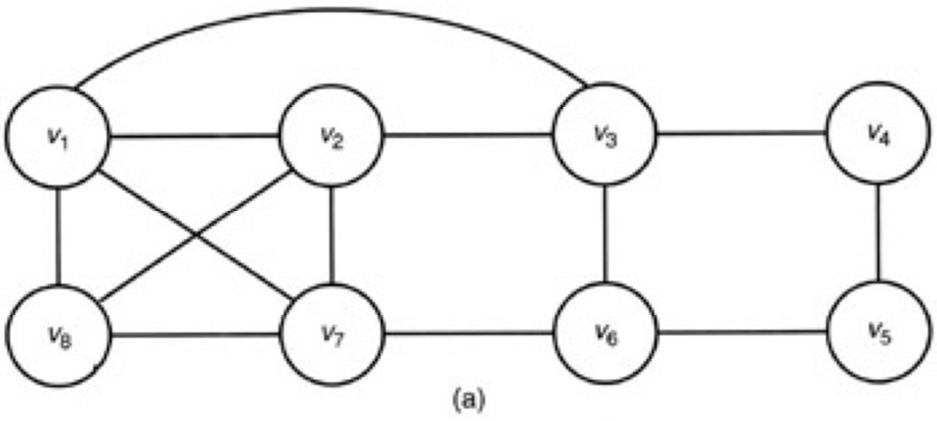
Recall that leaves in the state space tree are automatically nonpromising because their bounds cannot be greater than maxprofit. Therefore, we should not need a check for the terminal condition that i = n in function promising. Let's confirm that our algorithm does not need to do this check. If i = n, bound does not change from its initial value profit. Because profit is less than or equal to maxprofit, the expression bound>maxprofit is false, which means that function promising returns false.

The state space tree in the 0–1 Knapsack problem is the same as that in the Sum-of-Subsets problem. The number of nodes in that tree is

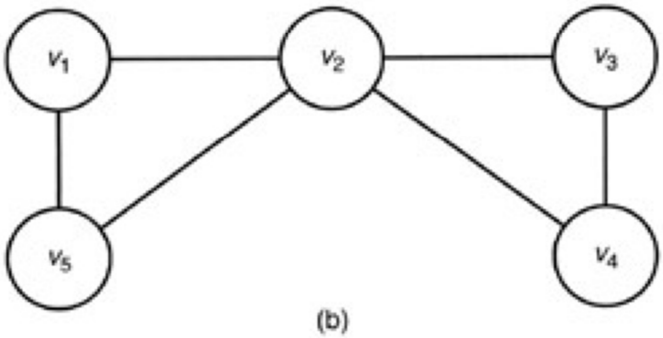
$2^{n+1} - 1$

Hamiltonian Circuit Problem

Given a connected, undirected graph, a Hamiltonian Circuit (also called a tour) is a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex. The graph (a) in the following figure contains the Hamiltonian Circuit [v1, v2, v8, v7, v6, v5, v4, v3, v1], but the next one (graph (b)) does not contain a Hamiltonian Circuit. The Hamiltonian Circuits problem determines the Hamiltonian Circuits in a connected, undirected graph.



(a)



(b)

A state space tree for this problem is as follows. Put the starting vertex at level 0 in the tree; call it the zeroth vertex on the path. At level 1, consider each vertex other than the starting vertex as the first vertex after the starting one. At level 2, consider each of these same vertices as the second vertex, and so on. Finally, at level n - 1, consider each of these same vertices as the (n - 1)st vertex.

The following considerations enable us to backtrack in this state space tree:

1. The ith vertex on the path must be adjacent to the (i - 1)st vertex on the path.
2. The (n - 1)st vertex must be adjacent to the 0th vertex (the starting one).
3. The ith vertex cannot be one of the first i - 1 vertices.

The algorithm that follows uses these considerations to backtrack. This algorithm is hard-coded to make v1 the starting vertex.

Problem: Determine all Hamiltonian Circuits in a connected, undirected graph.

Inputs: positive integer n and an undirected graph containing n vertices. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where W[i] [j] is true if there is an edge between the ith vertex and the jth vertex and false otherwise.

Outputs: For all paths that start at a given vertex, visit each vertex in the graph exactly once, and end up at the starting vertex. The output for each path is an array of indices vindex indexed from 0 to n - 1, where vindex[i] is the index of the ith vertex on the path. The index of the starting vertex is vindex[0].

```
void hamiltonian (index i)
{   if (promising (i)
    {
       if (i == n - 1)
          print the vindex[] array;
       else
       // the following for loop tries all colors for the next vertex
          for (j = 2; j <=n; j++)
          {
             vindex [i + 1] = j;
             hamiltonian (i + 1);
          }
}
```

```
bool promising (index i)
{
   index j;
   bool okay;

   if (i == n - 1 &&
      W[vindex[n - 1]][vindex[0]]= false)
       okay =false;
   else if (i > 0 &&
```

```
      W[vindex[i - 1]][vindex [i]]==false)
    okay = false;

   else
   {
      okay = true;
      j = 1;
      while (j < i && okay)
      {
         if (vindex[i] == vindex [j]

okay = false;

         j++;
      }// end while

   }// end if else
   return okay;
}// end promising
```

The top-level called to hamiltonian would be as follows:

```
vindex [0] = 1; //Make v1 the starting vertex.
hamiltonian (0);
```

The number of nodes in the state space tree for this algorithm is

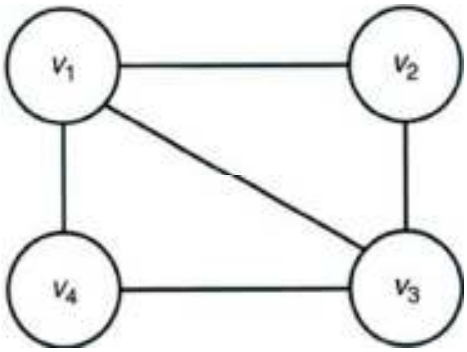
$$1 + (n-1) + (n-1)^2 + \cdots + (n-1)^{n-1} = \frac{(n-1)^n - 1}{n-2},$$

which is much worse than exponential. Although the following instance does not check the entire state space tree, it does check a worse-than-exponential number of nodes.

Graph Coloring (m-coloring)

The m-Coloring problem concerns finding all ways to color an undirected graph using at most m different colors, so that no two adjacent vertices are the same color. We usually call the m-Coloring problem a unique problem for each value of m.

Consider the following graph. There is no solution to the 2-Coloring problem for this graph because, if we can use at most two different colors, there is no way to color the vertices so that no adjacent vertices are the same color. One solution to the 3-Coloring problem for this graph is as follows:-

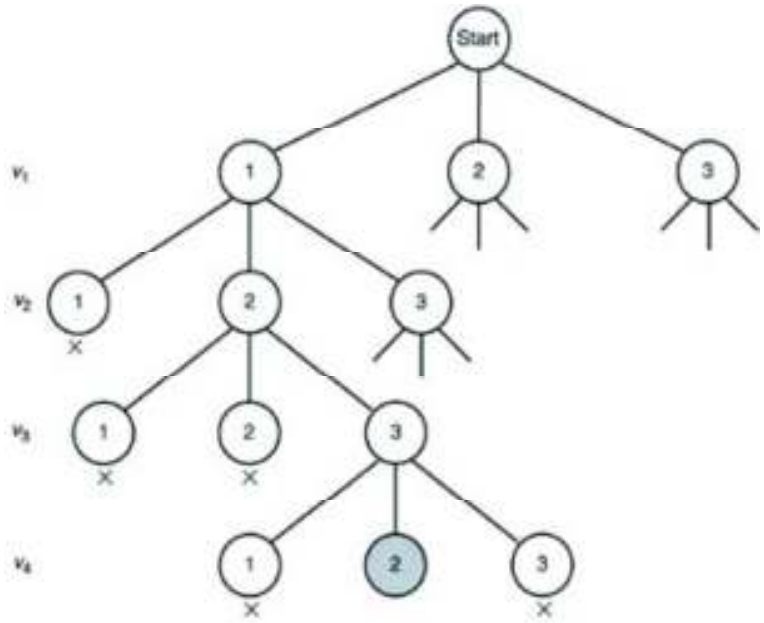


Vertex	Color
v1	color 1
v2	color 2
v3	color 3
v4	color 2

There are a total of six solutions to the 3-Coloring problem for this graph. However, the six solutions are only different in the way the colors are permuted. For example, another solution is to color v1 color 2, v2 and v4 color 1, and v3 color 3.

A straightforward state space tree for the m-Coloring problem is one in which each possible color is tried for vertex v1 at level 1, each possible color is tried for vertex v2 at level 2, and so on until each possible color has been tried for vertex vn at level n. Each path from the root to a leaf is a candidate solution. We check whether a candidate solution is a solution by determining whether any two adjacent vertices are the same color. To avoid confusion, remember in the following discussion that "node" refers to a node in the state space tree and "vertex" refers to a vertex in the graph being colored.

We can backtrack in this problem because a node is nonpromising if a vertex that is adjacent to the vertex being colored at the node has already been colored the color that is being used at the node. Following figure shows a portion of the pruned state space tree that results when this backtracking strategy is applied to a 3-coloring of the graph drawn above. The number in a node is the number of the color used on the vertex being colored at the node.



The first solution is found at the node with value 2 on the 4th level. Nonpromising nodes are labeled with crosses. After v1 is colored color 1, choosing color 1 for v2 is nonpromising because v1 is adjacent to v2. Similarly, after v1, v2, and v3 have been colored colors 1, 2, and 3, respectively, choosing color 1 for v4 is nonpromising because v1 is adjacent to v4.

```
void m_coloring (index i)
{
  int color;

    if (promising (i))
   {
       if (i == n)
          print the vcolor[] array;
      else
          // the following for loop tries every color for the next vertex
          for (color=1; color<=m; color++)
           {
            vcolor [i + 1] = color;
            m_coloring(i+1);
           }
}
```

```
bool promising (index i)
{
   bool okay = true;
    j = 1;

   while (j && okay)
   {
      // foll. if condition checks if an adjacent vertex already the same color
      if (W[i][j] && vcolor[i]==vcolor[j])
            okay = false;
       j++;
   }
   return okay;
}
```

The top level call to m_coloring would be

m_coloring(0)

The number of nodes in the state space tree for this algorithm is equal to

$$1 + m + m^2 + \cdots + m^n = \frac{m^{n+1} - 1}{m - 1}.$$