Chapter 5 Backtracking

- The Backtracking Technique
- The *n*-Queens Problem
- The Sum-of-Subsets Problem
- Graph Coloring
- The 0-1 Knapsack Problem

Backtracking

- maze puzzle
- following every path in maze until a dead end is reached.
- go back to a fork and pursue another path
- 2^n cases (exponential-time in the worst case)
- if we can find some ***signs*** while generating subsets, we can avoid unnecessary labor

5.1 The Backtracking Technique

5.1 The backtracking Technique

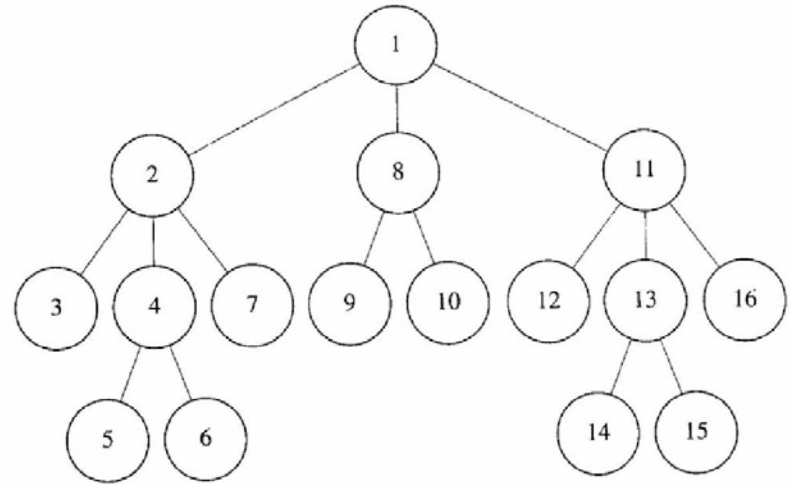
- a **sequence** of objects is chosen from a specified ***set***

  s.t. the sequence satisfies some *criterion*
- n-Queens Problem
- n Queens place in n×n chessboard s.t.

  no two Queens are in the same column, row, or diag
  - sequence (n positions)
  - set (n×n positions)
  - criterion (no two queens threaten each other)
- sequence generated by depth-first search
  - visiting root, left, right
  - see Fig. 5.1 pp. 199

5.1 The backtracking Technique

Figure 5.1 A tree with nodes numbered according to a depth-first search.



5.1 The backtracking Technique

- N-Queens Problem with n=4
- 4 queens on a 4×4 chessboard, no two queens threaten each other (same row, column, diag)
  - assigning each queen a different row I
  - checking which column combinations yield solutions
  - there are 4×4×4×4=256(44) candidate solutions
- Fig. 5.2, ***state space tree***
- a path from root to a leaf forms a candidate solution
- <i, j> node denotes to place i queen in row i column j
- depth first search to generate paths

5.1 The backtracking Technique

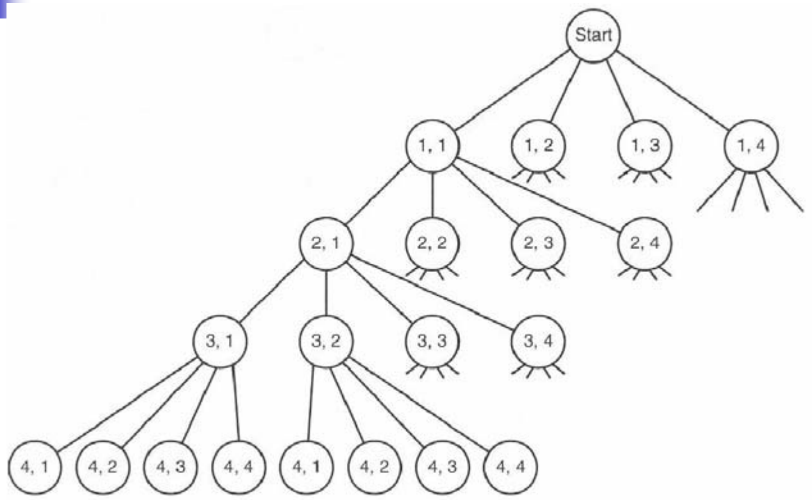
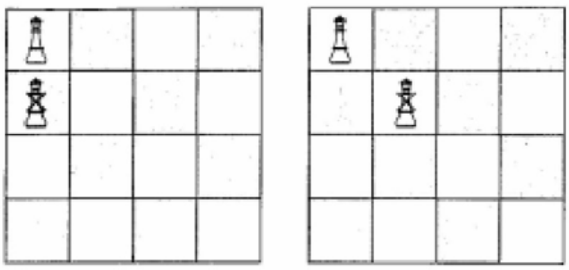


Figure 5.2 ● A portion of the state space tree for the instance of the n-Queens problem in which $n = 4$. The ordered pair $<i, j>$, at each node means that the queen in the i th row is placed in the j th column. Each path from the root to a leaf is a candidate solution.

5.1 The backtracking Technique

Figure 5.3 Diagram showing that if the first queen is placed in column 1, the second queen cannot be placed in column 1 (a) or column 2 (b).



(a) (b)

5.1 The backtracking Technique

- a general algorithm for backtracking

```
void checknode (node v) ← root
 {
     if (promising (v)) → possible lead to a solution
           if (there is a solution at v)
                 write the solution;
           else            → not form a solution yet
                 for (each child u of v)
                    checknode(v);
 }
```

- See Example 5.1
- See pp. 204 last paragraph

5.1 The backtracking Technique

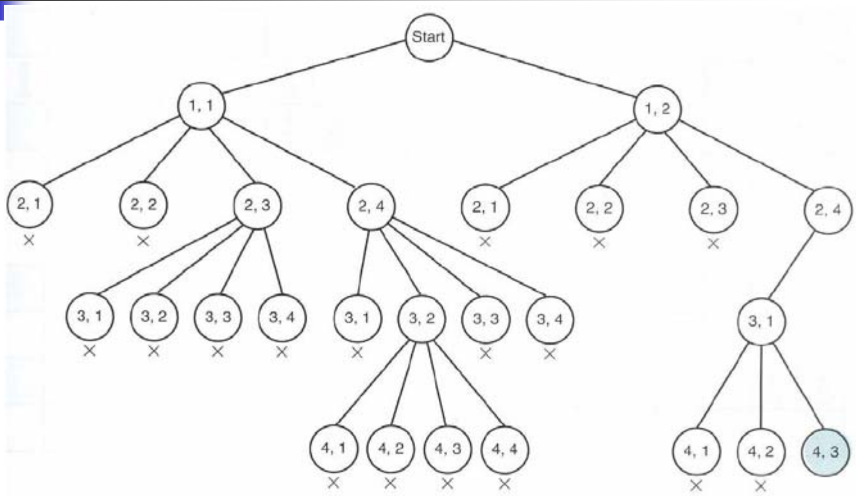
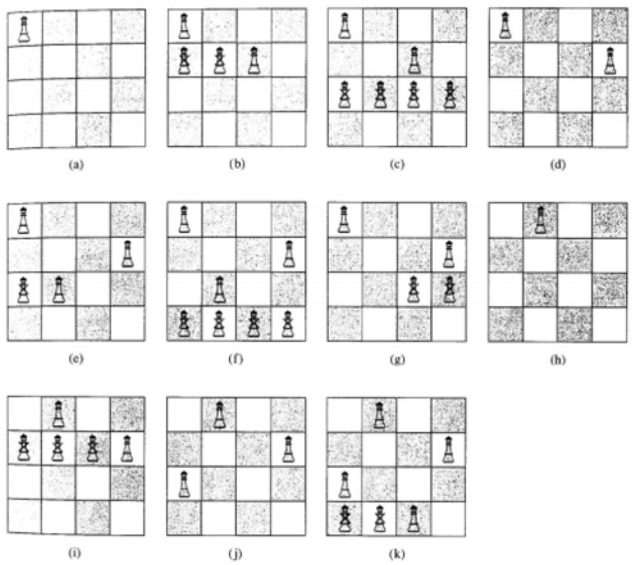


Figure 5.4 ● A portion of the pruned state space tree produced when backtracking is used to solve the instance of the n-Queens problems in which $n = 4$. Only the nodes checked to find the first solution are shown. That solution is found at the color-shaded node. Each nonpromising node is marked with a cross.

5.1 The backtracking Technique

Figure 5.5 The actual chessboard positions that are tried when backtracking is used to solve the instance of the n-Queens Problem in which $n = 4$. Each nonpromising position is marked with a cross.



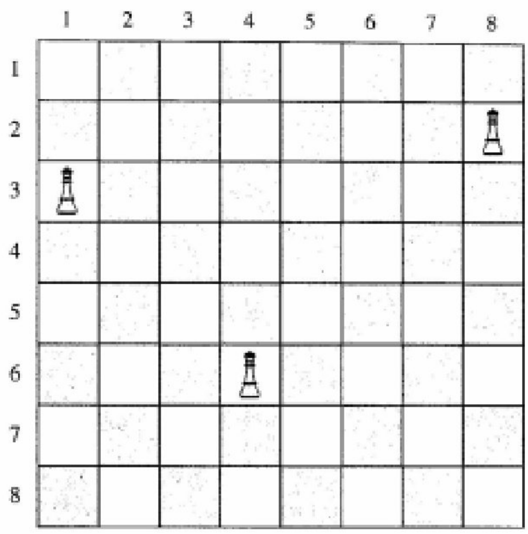
5.2 The n-Queens Problem

5.2 The n-Queens Problem

- promising(v): whether two queens are in the same ***column*** or ***diagonal***
- col(i) : the column where queen i in row i is located
- two queens i, k (note queens i, k are located in ***row*** i, k)
- in the same column➔ col(i)=col(k)
- in the same diagonal
- see Fig. 5.6 pp. 206
  - col(i)-col(k) = i−k or k−i
- See Algorithm 5.1
- See Table 5.1 for analysis, pp.209

5.2 The n-Queens Problem

Figure 5.6 The queen in row 6 is being threatened in its left diagonal by the queen in row 3 and in its right diagonal by the queen in row 2.



Algorithm 5.1 The Backtracking Algorithm for the *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 $n \times n$ chessboard so that no two queens threaten each other. Each output consists of an array of integers *col* indexed from 1 to *n*, where *col*[*i*] is the column where the queen in the *i*th row is placed.

```
void queens (index i)
{
  index j;
  if (promising(i))
    if (i == n)
      cout << col[1] through col[n];
    else
      for (j = 1; j <= n; j++) {        // See if queen in (i + 1)st row
        col[i + 1] = j;                 // can be positioned in each of
        queens(i + 1);                  // the n columns.
      }
}
  bool promising (index i)
  {
    index k;
    bool switch;

    k = 1;
    switch = true;                      // Check if any queen threatens
    while (k < i && switch) {           // queen in the ith row.
      if (col[i] == col[k] || abs(col[i] − col[k]) == i − k)
        switch = false;
      k++;
    }
    return switch;
  }
```


5.2 The n-Queens Problem

Table 5.1 An illustration of how much checking is saved by backtracking in the *n*-Queens Problem*

n	Number of Nodes Checked by Algorithm 1†	Number of Candidate Solutions Checked by Algorithm 2‡	Number of Nodes Checked by Backtracking	Number of Nodes Found Promising by Backtracking
4	341	24	61	17
8	19,173,961	40,320	15,721	2057
12	9.73×10^{12}	4.79×10^{8}	1.01×10^{7}	8.56×10^{5}
14	1.20×10^{16}	8.72×10^{10}	3.78×10^{8}	2.74×10^{7}

*Entries indicate numbers of checks required to find all solutions.
†Algorithm 1 does a depth-first search of the state space tree without backtracking.
‡Algorithm 2 generates the *n*! candidate solutions that place each queen in a different row and column.

5.4 The Sum-of-Subsets Problem

5.4 The Sum-of-Subsets Problem

- given n positive integers (weights) w1, w2, ...,wn
- given a positive integer W
- finding all subsets of n integers that sum to W
  - e.g., wi+wj+...+wk=W
  - See Example 5.2 pp. 214

5.4 The Sum-of-Subsets Problem

- create a state space tree
  - See Fig. 5.7 pp. 215
- each ***left*** edge denotes we ***include*** wi (weight wi)
- each ***right*** edge denotes we ***exclude*** wi (weight 0)
- any path from root to a leaf forms a subset
- See Fig. 5.8 pp. 216

5.4 The Sum-of-Subsets Problem

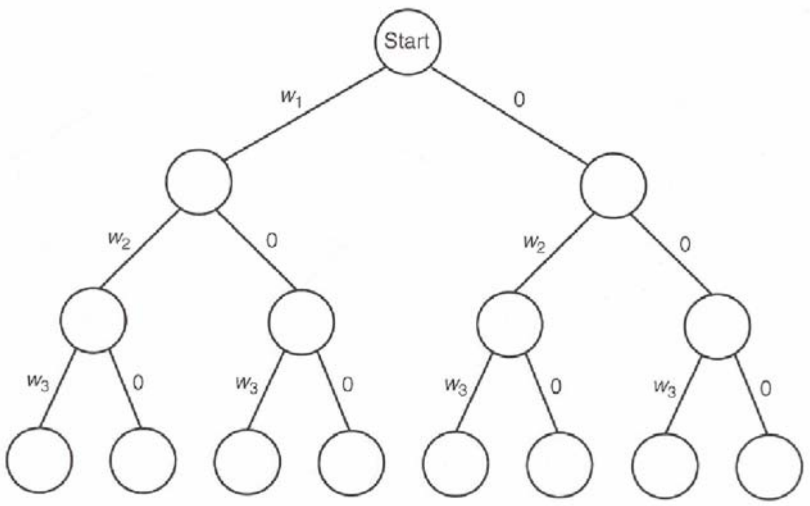


Figure 5.7 • A state space tree for instances of the Sum-of-Subsets problem in which $n = 3$.

5.4 The Sum-of-Subsets Problem

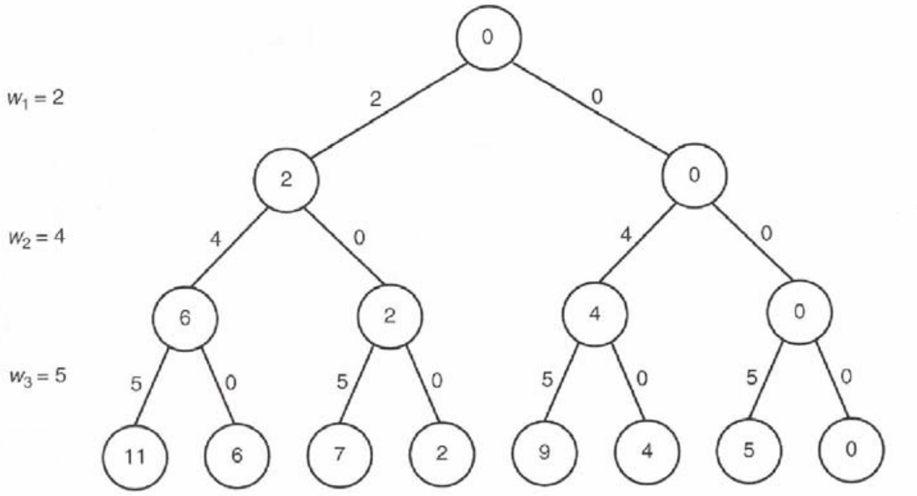


Figure 5.8 • A state space tree for the Sum-of-Subsets problem for the instance in Example 5.3. Stored at each node is the total weight included up to that node.

5.4 The Sum-of-Subsets Problem

- ***significant signs (backtracking)***
  - sorting the weights in nondecreasing order
  - weight be the subtotal from root to node i at level I
- weight $+w_{i+1} > W$
  - any descendant of node i will be nonpromising (because is w_{i+1} the lightest weight remaining)
- weight + all remaining items < W
  - any descendant of node i will be nonpromising
- Example 5.4 and Fig. 5.9 pp. 217
- See Algorithm 5.4

5.4 The Sum-of-Subsets Problem

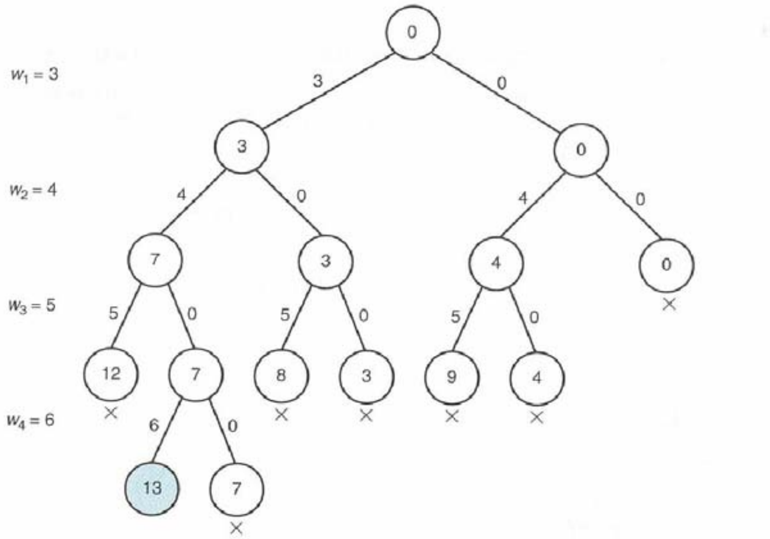


Figure 5.9 • The pruned state space tree produced using backtracking in Example 5.4. Stored at each node is the total weight included up to that node. The only solution is found at the shaded node. Each nonpromising node is marked with a cross.

Algorithm 5.4 The Backtracking Algorithm for the Sum-of-Subsets Problem

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)
      cout << include[1] through include[i];
    else {
      include[i + 1] = "yes";                   // Include w[i + 1].
      sum_of_subsets(i + 1, weight + w[i + 1], total − w[i + 1]);
      include[i + 1] = "no";                    // Do not include w[i + 1].
      sum_of_subsets(i + 1, weight, total − w[i + 1]);
    }
}

bool promising (index i);
{
return (weight + total >= W) && (weight == W || weight + w[i + 1] <= W);
}
```

5.5 Graph Coloring

5.5 Graph Coloring

- m-coloring problem
  - finding all ways to color vertices using at most m colors s.t. no two adjacent vertices are the same color
  - Example 5.5 pp. 220
- state space tree
  - Fig. 5.12 pp. 222
  - each possible color is tried for vertex vi at level i s.t. no two adjacent vertices are the same color

  ➔***sign (backtracking)***
- See Algorithm 5.5

5.5 Graph Coloring

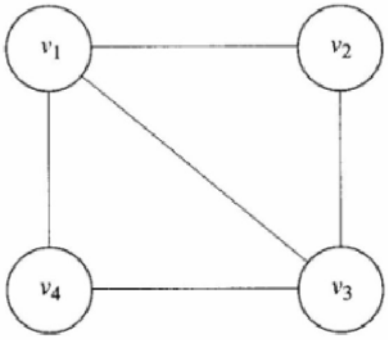
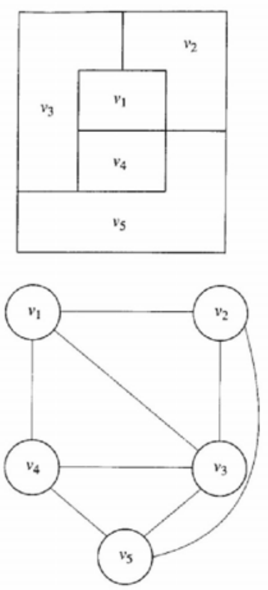


Figure 5.10 Graph for which there is no solution to the 2-Coloring Problem. A solution to the 3-Coloring Problem for this graph is shown in Example 5.5.

5.5 Graph Coloring

Figure 5.11 Map (top) and its planar graph representation (bottom).



5.5 Graph Coloring

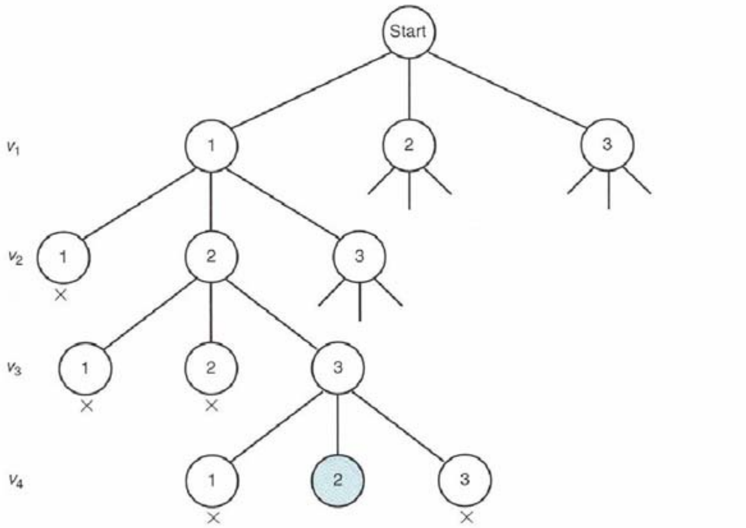


Figure 5.12 ● A portion of the pruned state space tree produced using backtracking to do a 3-coloring of the graph in Figure 5.10. The first solution is found at the shaded node. Each nonpromising node is marked with a cross.

Algorithm 5.5 The Backtracking Algorithm for the *m*-Coloring Problem

Problem: Determine all ways in which the vertices in an undirected graph can be colored, using only m colors, so that adjacent vertices are not the same color.

Inputs: positive integers n and m, 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 ith vertex and the jth vertex and false otherwise.

Outputs: all possible colorings of the graph, using at most m colors, so that no two adjacent vertices are the same color. The output for each coloring is an array *vcolor* indexed from 1 to n, where *vcolor*[i] is the color (an integer between 1 and m) assigned to the ith vertex.

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

  if (promising(i))
    if (i == n)
      cout << vcolor[1] through vcolor[n];
    else
      for (color = 1; color <= m; color++) {    // Try every color for
        vcolor[i + 1] = color;                  // next vertex.
        m_coloring(i + 1);
      }
}

bool promising (index i)
{
  index j;
  bool switch;

  switch = true;
  j = 1;
  while (j < i && switch) {                     // Check if an adjacent
    if (W[i][j] && vcolor[i] == vcolor[j])      // vertex is already this
      switch = false;                           // color.
    j++;
  }
  return switch;
}
```

5.7 The 0-1 Knapsack Problem

5.7 0-1 Knapsack Problem

- using a state space tree like the Sum-of-Subset Problem
- each level is used to decide whether to include an item *i* or not
  - (left edge: include it and right edge: exclude it)
- each path from root to a leaf is a candidate solution
  - 0-1 Knapsack Problem is an optimization problem;
- we can't know the optimal solution until the search is over

5.7 0-1 Knapsack Problem

```
void checknode (node v) ← root
        {
        if (value(v) is better than best)
            best=value(v)          ← total profit up to v
        if (promising (v))       ←stealing more items
                for (each child u of v)
                        checknode(v);
        }
```

★ best: the value of best solution found so far

5.7 0-1 Knapsack Problem

- promising (v): whether we can steal more items into knapsack
- 1. weight >= W ➔ nonpromising
- 2. greedy consideration
  - sort all items according to *pi /wi* in nondecreasing order
  - decide a node at level i be promising (expanding)
    - maxprofit : the best profit found so far
    - profit : the sum of profits up to the node
    - weight: the sum of weights up to the node

5.7 0-1 Knapsack Problem

- greedily grab itemi+1, itermi+2, ..., itemk (sorted)
  - s.t. total weight of item1,...,itemk above W
  - totweight = weight + $\sum_{j=i+1}^{k-1} w_j$
  - bound = (profit+ $\sum_{j=i+1}^{k-1} p_j$) + $(\frac{W - totweight}{w_k})$ xpk

    profit of k-1 items   profit of kth items

    bound <= maxprofit ⇨ node i is nonpromising
- See Example 5.6 pp. 229
- See Algorithm 5.7

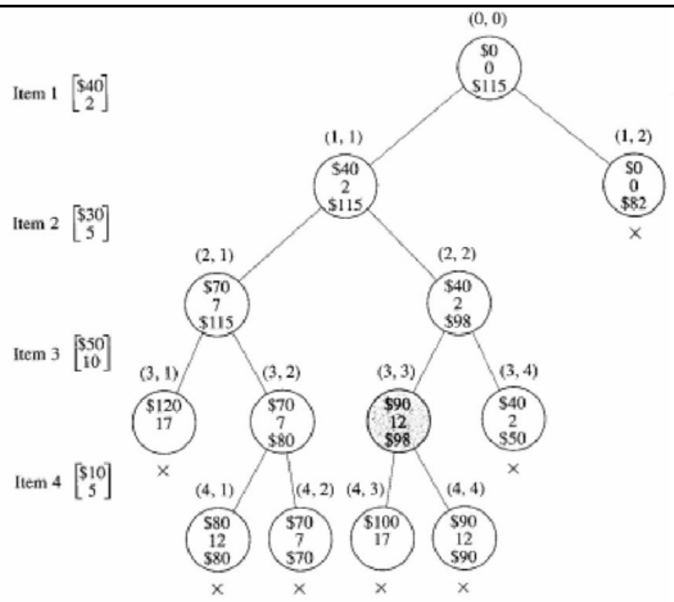


Figure 5.14 The pruned state space tree produced using backtracking in Example 5.6. Stored at each node from top to bottom are the total profit of the items stolen up to the node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The optimal solution is found at the shaded node. Each nonpromising node is marked with a cross.

Algorithm 5.7 The Backtracking Algorithm for the 0-1 Knapsack Problem

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 nonincreasing 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)
{
  if (weight <= W && profit > maxprofit) {      // This set is best so far.
    maxprofit = profit;
    numbest = i;                                // Set numbest to number
    bestset = include;                          // of items considered. Set
  }                                             // bestset to this solution.
  if (promising(i)) {
    include[i + 1] = "yes";                     // Include w[i + 1].
    knapsack(i + 1, profit + p[i + 1], weight + w[i + 1]);
    include[i + 1] = "no";                      // Do not include w[i + 1].
    knapsack(i + 1, profit, weight);
  }
}

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

  if (weight >= W)                              // Node is promising only
    return false;                               // if we should expand to
  else {                                        // its children. There must
    j = i + 1;                                  // be some capacity left for
    bound = profit;                             // the children.
    totweight = weight;
    while (j <= n && totweight + w[j] <= W) {   // Grab as many items as
      totweight = totweight + w[j];             // possible.
      bound = bound + p[j];
      j++;
    }
    k = j;                                      // Use k for consistency
    if (k <= n)                                 // with formula in text.
      bound = bound + (W - totweight) * p[k]/w[k]; // Grab fraction of kth
    return bound > maxprofit;                   // item.
  }
}
```


The End