For example, node (1, 2) in Figure 6.3 is promising at the time we visit it. In our implementation, this is when we insert it in PQ. However, it becomes nonpromising when *maxprofit* takes the value \$90. In our implementation, this is before we remove it from PQ. We learn this by comparing its bound with *maxprofit* after removing it from PQ. In this way, we avoid visiting children of a node that becomes nonpromising after it is visited.

The specific algorithm for the 0-1 Knapsack problem follows. Because we need the bound for a node at insertion time, at removal time, and to order the nodes in the priority queue, we store the bound at the node. The type declaration is as follows:

```
struct node
{
  int level;          // the node's level in the tree
  int profit;
  int weight;
  float bound;
};
```

▶ Algorithm 6.2 **The Best-First Search with Branch-and-Bound Pruning 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 of positive integers w and p, each indexed from 1 to n, and each of which is sorted in nonincreasing order according to the values of $p[i]/w[i]$.

Outputs: an integer *maxprofit* that is the sum of the profits of an optimal set.

```
void knapsack3 (int n,
                const int p[], const int w[],
                int W,
                int& maxprofit)
{
  priority_queue_of_node PQ;
  node u, v;

  initialize(PQ);                                  // Initialize PQ to be
  v.level = 0; v.profit = 0; v.weight = 0;         // empty.
  maxprofit = 0;                                   // Initialize v to be the
  v.bound = bound(v);                              // root.
  insert(PQ, v);
```

```
while (!empty(PQ)){
    remove(PQ, v);                          // Remove node with
    if (v.bound > maxprofit){               // best bound.
        u.level = v.level + 1;              // Check if node is still
        u.weight = v.weight + w[u.level];   // promising.
        u.profit = v.profit + p[u.level];   // Set u to the child
        if (u.weight <= W && u.profit > maxprofit)   // that includes the
            maxprofit = u.profit;           // next item.
        u.bound = bound(u);
        if (u.bound > maxprofit)
            insert(PQ, u);
        u.weight = v.weight;                // Set u to the child
        u.profit = v.profit;                // that does not include
        u.bound = bound(u);                 // the next item.
        if (u.bound > maxprofit)
            insert(PQ, u);
    }
  }
}
```

best bound

Function *bound* is the one in Algorithm 6.1.

6.2 The Traveling Salesperson Problem

In Example 3.12, Nancy won the sales position over Ralph because she found an optimal tour for the 20-city sales territory in 45 seconds using a $\Theta(n^2 2^n)$ dynamic programming algorithm to solve the Traveling Salesperson problem. Ralph used the brute-force algorithm that generates all 19! tours. Because the brute-force algorithm takes over 3,800 years, it is still running. We last saw Nancy in Section 5.6 when her sales territory was expanded to 40 cities. Because her dynamic programming algorithm would take more than six years to find an optimal tour for this territory, she became content with just finding any tour. She used the backtracking algorithm for the Hamiltonian Circuits problem to do this. Even if this algorithm did find a tour efficiently, that tour could be far from optimal. For example, if there were a long, winding road of 100 miles between two cities that were 2 miles apart, the algorithm could produce a tour containing that road even if it were possible to connect the two cities by a city that is a mile from each of them. This means Nancy could be covering her territory very inefficiently using the tour produced by the backtracking algorithm. Given this, she might decide that she better go back to looking for an optimal tour. If the 40 cities were highly connected, having the backtracking algorithm produce all the tours would not work, because there would be a

Figure 6.4
Adjacency m
representatic
graph that h
edge from ev
vertex to eve
other vertex
and the nod
the graph ar
edges in an
tour (right).

```
float bound (node u)
{
   index j, k;
   int totweight;
   float result;
   if (u.weight >= W)
      return 0;
   else{
      result = u.profit;
      j = u.level + 1;
      totweight = u.weight;
      while (j <= n && totweight + w[j] <= W){
        totweight = totweight + w[j];          // Grab as many items
         result = result + p[j];               // as possible.
        j++;
      }
      k = j;
      if (k <=n)                               // Use k for consistenc
        result = result + (W- totweight) * p[k]/w[k];   // with formula in text
                                               // Grab fraction of kth
      return result;                           // item.
   }
}
```

We do not need to check whether *u.profit* exceeds *maxprofit* when the current item is not included because, in this case, *u.profit* is the profit associated with *u*'s parent, which means that it cannot exceed *maxprofit*. We do not need to store the bound at a node (as depicted in Figure 6.2) because we have no need to refer to the bound after we compare it with *maxprofit*.

Function *bound* is essentially the same as function *promising* in Algorithm 5.7. The difference is that we have written *bound* according to guidelines for creating branch-and-bound algorithms, and therefore *bound* returns an integer. Function *promising* returns a boolean value because it was written according to backtracking guidelines. In our branch-and-bound algorithm, the comparison with *maxprofit* is done in the calling procedure. There is no need to check for the condition $i = n$ in function *bound* because in this case the value returned by *bound* is less than or equal to *maxprofit*, which means that the node is not put in the queue.

Algorithm 6.1 does not produce an optimal set of items; it only determines the sum of the profits in an optimal set. The algorithm can be modified to produce an optimal set as follows. At each node we also store a variable *items*, which is the set of items that have been included up to the node, and