MODULE – 5

5.1. Backtracking

Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the *backtracking* formulation.

- Name was first coined by D H Lehmer in 1950s.
- The desired solution is expressible as an *n-tuple* $(x_1,\ldots,x_n)$ where, x_i are chosen from some finite set S_i.
- The problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a *criterion function* $P(x_1,\ldots,x_n)$.Sometimes it seeks all vectors that satisfy *P*.

  For example, sorting the array of integers in *a[1:n]* is a problem whole solution is expressible by an *n-tuple*, where x_i is the index in *a* of the i^{th} smallest solution.

  - The criterion function *P* is the inequality $a[x_i]<=a[x_i+1]$ for $1<=i<n$.
  - The set S_i is finite and includes integers *1* through *n*.
- Consider that $m_i$is the size of the set S_i. There are $m=m_1m_2\ldots m_n$ *n-tuples* that are possible candidates for satisfying the function *P*.The backtrack algorithm yields the same answer with far fewer than *m* trials.

Basic Idea

- Build up solution vector one component at a time.
- Use modified criterion functions $P_i(x_1,\ldots,x_i)$ called bounding functions to test whether the vector being formed has any chance of success.

Advantage

If it is realized that the partial vector $(x_1,x_2,\ldots x_i)$ can in no way lead to an optimal

solution, then $m_{i+1}.....m_n$possible test vectors can be ignored entirely.

5.1.1. General Approach

Definition 5.1

Explicit constraints are rules that restrict each x_i to take on values only from a given set.

Common examples of explicit constraints are

$$\begin{array}{rcl} x_i \geq 0 \quad \text{or} & S_i & = \{\text{all nonnegative real numbers}\} \\ x_i = 0 \text{ or } 1 \quad \text{or} & S_i & = \{0,1\} \\ l_i \leq x_i \leq u_i \quad \text{or} & S_i & = \{a : l_i \leq a \leq u_i\} \end{array}$$

The explicit constraints depend on the particular instance *I* of the problem being solved. All tuples that satisfy the explicit constraints define a possible *solution space* for *I*.

Definition 5.2

The implicit constraints are rules that determine which of the tuples in the solution space of *I* satisfy the criterion function. Thus, implicit constraints describe the way in which the x_i must relate to each other.

Example 5.1: 8-queens

Place 8 queens on an 8×8 chessboard so that no two "attack" → no two are on the same row, column, or diagonal.

- Numberthe rows and columns of the chessboard 1 through 8.
- Number the queens also 1 through 8.
- Assume that given queen *i*is to be placed on row *i*, since each queen must beon different row.

- All solutions for 8-queens problem can be represented as 8-tuples $(x_1,....,x_8)$. Here, x_i represents column on which queen *i* is placed.

- The explicit constraints are $S_i=\{1,2,3,4,5,6,7,8\}$, $1 \le i \le 8$.
    - The solution space consists of 8^8 8-tuples.
- The implicit constraints are that
    - No two x_is can be the same.
    - All queens must be on different columns,
    - No two queens can be on the same diagonal.
- The first of these two constraint implies that all solutions are permutations of the 8-tuple (1,2,3,4,5,6,7,8). →size of the solution space is reduced from 8^8 tuples to 8! tuples.
- The solution in Figure 5.1 is (4,6,8,2,7,1,3,5) expresses as 8-tuple

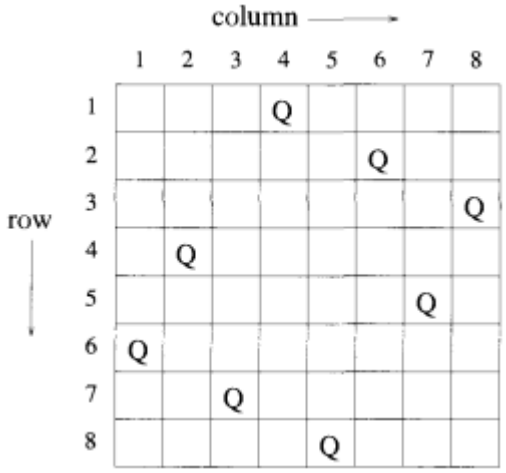

Figure 5.1: One solution to the n-queens problem

Example 5.2: Sum of Subsets

Given the positive numbers w_i, $1 \le i \le n$, and m, find all the subsets of the w_i whose sums are m.

For examples, consider $n=4$, $(w_1,w_2,w_3,w_4)=(11,13,24,7)$ and

$m=31$,

The desired subsets are *(11,13,7)* and *(24,7)*.

Rather than representing the solution vector by w_i, represent the solution vector by giving indices these w_i which sum to *m*.

Hence the solution vectors are *(1,2,4)* and *(3,4)*.

Therefore, the solutions are *k*-tuples $(x_1,x_2,\ldots,x_k)$, $1\le k\le n$ and different solutions have different sized tuples.

Formulation 1

- The explicit constraints require $x_i \in \{j | j$ is an integer and $1\le j\le n\}$
- The implicit constraints require that
    - No two be the same and
    - The sum of the corresponding w_i's be *m*.
- Avoid generating multiple instances of the same subset (example, (1,2,4) & (1,4,2) represent same solution). Hence, another implicit constraint is $x_i<x_{i+1}, 1\le i\le k$.

Formulation 2

- Each solution subset is represented by an n-tuple $(x_1,x_2,\ldots,x_n)$ such that $x_i \in \{0,1\}, 1\le i\le n$.
    - $x_i=0$ if w_i is not chosen
    - $x_i=1$ if w_i is chosen.
- Therefore, the solutions for the above instances are (1,1,0,1) and (0,0,1,1). →solution using a fixed tuple.
- The solution space consists of 2^n distinct tuples.

5.1.2. n-Queens Problem

Problem: place n queens on an n×n chessboard so that no two queens attack each other by being in the same row or in the same column or on the

same diagonal.

- Forn=1,the problem has a trivial solution
- There is no solution forn=2 andn=3.
- Consider 4-queens problem and solve by backtracking.

Since each of the four queenshas to be placed in its own row, all we need to do is to assign a column for eachqueen on the board presented in Figure 5.2.

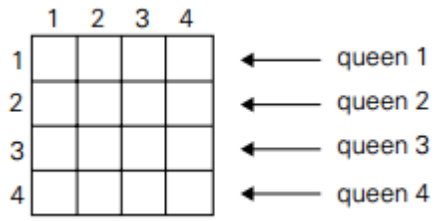

Figure 5.2: Board for the 4-queens problem

i. Start with the empty board

ii. Place queen 1 in the first possibleposition of its row, which is in column 1 of row 1

iii. Place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3.

This proves to be a dead end because there is no acceptable position for queen 3.

iv. The algorithm backtracks and puts queen 2 in the next possible position at (2, 4)

v. Then queen 3 is placedat (3, 2), which **proves to be another dead end**.

vi. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2).

vii. Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3),

which is a solution to the problem.

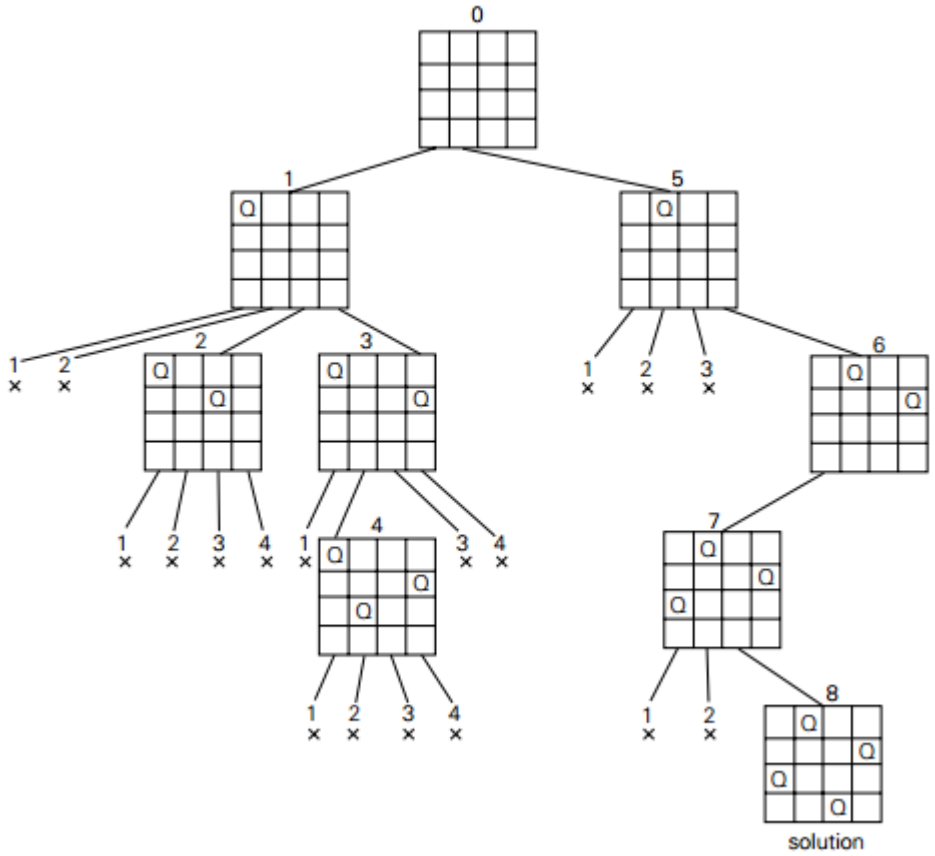

Figure 5.3: State-space tree of solving the four-queens problem by backtracking.×denotes an unsuccessful attempt to place a queen in the indicatedcolumn. The numbers above the nodes indicate the order in which thenodes are generated

To find other solution, the algorithm can simply resume its operations at the leaf atwhich it stopped.

NOTE:

A single solution to then-queens problemfor anyn≥4 can be found in linear time.

5.1.3. Sum of Subsets Problem

Problem: find a subset of a givensetA={a_1,...,a_n}of*n*positive integers whose sum is equal to a given positiveinteger*d*.

For example, *A={1,2,5,6,8}*and*d=9*,there are two solutions:

- {1, 2, 6} and
- {1, 8}

Procedure:

Sort the set's elements in increasing order.

Assume, $a_1<a_2<....<a_n$

The state-space tree can be constructed as a binary tree like that in Figure 5.4forthe instanceA={3, 5,6,7}andd=15.

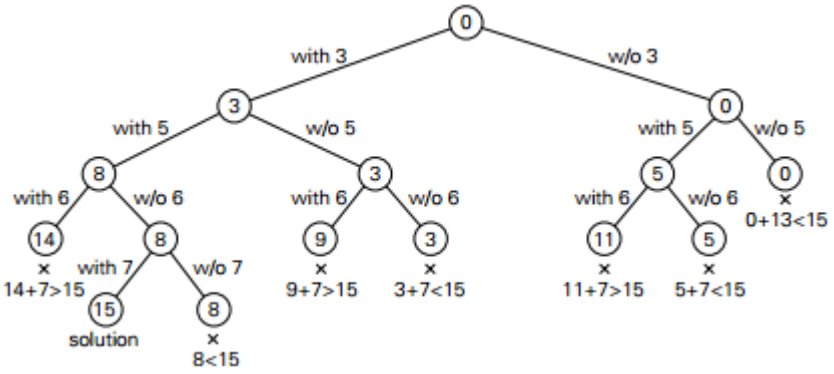

Figure 5.4: Complete state-space tree of the backtracking algorithm applied to theinstance A={3, 5, 6, 7} andd=15of the subset-sum problem. Thenumber inside a node is the sum of the elements already included in thesubsets represented by the node. The inequality below a leaf indicatesthe reason for its termination

- The root of the tree represents the starting point, with no decisions about the given elements made
- Its left and right children represent, respectively, inclusion and exclusionofa_1 in a set being sought.
- Going to the left from a node of the first level corresponds to inclusion of$a_2$while going to the right corresponds to its exclusion, and so on.

A pathfrom the root to a node on the i^{th} level of the tree indicates which of the first i numbers have been included in the subsets represented by that node.

Record the value of s, the sum of these numbers, in the node.

- If s is equalto d, we have a solution to the problem.
- If all the solutions need to be found, continue by backtracking to the node'sparent.
- If s is not equal to d terminate the node as non-promising if either of the following two inequalities hold

$$s + a_{i+1} > d \quad \text{(the sum } s \text{ is too large)},$$
$$s + \sum_{j=i+1}^{n} a_j < d \quad \text{(the sum } s \text{ is too small)}.$$

Backtracking Algorithm

An output of a backtracking algorithm can be thought of as ann-tuple $(x_1,x_2,...,x_n)$ where each coordinate x_i is an element of some finite linearly ordered set S_i. Each S_i is the setof integers (column numbers) 1 through n.

- The tuple may need to satisfy someadditional constraints. Depending on the problem, all solution tuples can be of the same lengthand of different lengths.

A backtracking algorithm generates, explicitly or implicitly, a state-space tree; its nodes represent partially constructed tuples with thefirst i coordinates defined by the earlier actions of the algorithm.

- If such a tuple $(x_1,x_2,...,x_i)$ is not a solution, the algorithm finds the next element in S_{i+1} that is consistent with the values of $(x_1,x_2,...,x_i)$ and the problem's constraints, and adds it to the tuple as its $(i+1)^{st}$ coordinate.
- If such an element does not exist, the algorithm backtracks to consider the next value of x_i ,and so on.

To start a backtracking algorithm, the following pseudocode can be called fori=0;X[1..0] represents the empty tuple.

```
ALGORITHM Backtrack(X[1..i])
    //Gives a template of a generic backtracking algorithm
    //Input: X[1..i] specifies first i promising components of a solution
    //Output: All the tuples representing the problem's solutions
    if X[1..i] is a solution write X[1..i]
    else      //see Problem 9 in this section's exercises
        for each element x ∈ S_{i+1} consistent with X[1..i] and the constraints do
            X[i + 1] ← x
            Backtrack(X[1..i + 1])
```

Tricks to reduce the size of a state-space tree

i. Exploit the symmetry often present in combinatorial problems, or

ii. Pre-assign values to one or more components of a solution

Estimate the size of the state-space tree of a backtracking algorithm

Generating a random path from the root toa leaf and using the information about the number of choices available duringthe path generation to estimate the size of the tree.

NOTE on BACKTRACKING

i. It is typically applied to difficult combinatorial problems for which no efficient algorithms for finding exact solutions possibly exist

ii. Backtracking at least holds a hope for solving some instances of nontrivial sizes in an acceptable amount of time.

iii. Even if backtracking does not eliminate any elements of a problem's state space and ends up generating all its elements, it provides a specific technique for doing so, which can be of value in its own right.

5.1.4. Graph Coloring

Let *G* be a graph and *m* be a given positive integer.

Problem: Color the node of G in such a way that no two adjacent nodes

have the same color yet only m colors are used.→m-colorability decision problem

If *d* is the degree of the given graph, then it can be colored with *d+1* colors.

m-colorability optimization problem→considers smallest integer *m* for which the graph *G* can be colored.Where *m* is *chromatic number* of the graph.

Consider the graph in Figure 5.5.

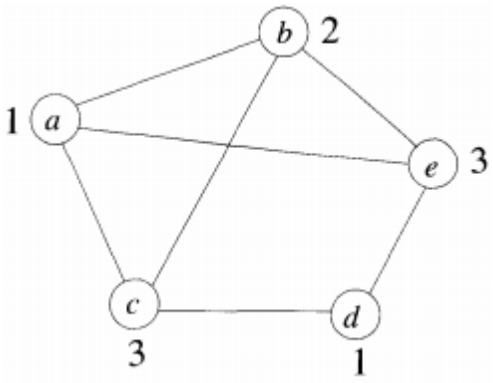

Figure 5.5: An example graph and its coloring.

- The above graph can be colored with three colors 1,2, and 3 represented next to each node.
- Three colors are needed to color this graph and hence its chromatic number is 3.

Planar graph

A planar graph can be drawn in a plane such that no two edges cross each other.

Consider an example for m-colorability decision problem i.e. 4-color problem for planar graph.

Problem: Given any map, can the regions be colored in such a way that no two adjacent regions have the same color, yet only 4 colors are needed.

Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge.

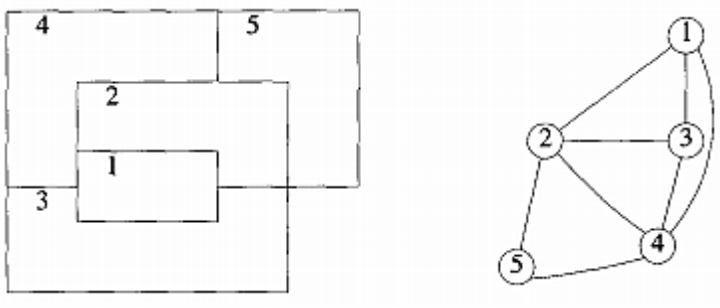

Figure 5.6: A map and its planar graph representation.

Figure 5.6 shows a map with 5 regions and its corresponding graph. This map requires 4 colors.

NOTE: No map that required more than 4 colors had ever been found.

Consider, representing a graph by its adjacency matrix *G[1:n, 1:n]*, where *G[i,j]=1* if *(i,j)* is an edge of *G*, and *G[i,j]=0* otherwise.

The colors are represented by the integers *1,2,…,m* and the solutions are given by the n-tuple $(x_1,x_2,\ldots,x_n)$, where x_i is the color of the node *i*.

Using the recursive backtracking algorithm, the resulting algorithm is *mColoring*as given below.

```
1   Algorithm mColoring(k)
2   // This algorithm was formed using the recursive backtracking
3   // schema. The graph is represented by its boolean adjacency
4   // matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
5   // vertices of the graph such that adjacent vertices are
6   // assigned distinct integers are printed. k is the index
7   // of the next vertex to color.
8   {
9       repeat
10      {// Generate all legal assignments for x[k].
11          NextValue(k); // Assign to x[k] a legal color.
12          if (x[k] = 0) then return; // No new color possible
13          if (k = n) then     // At most m colors have been
14                              // used to color the n vertices.
15              write (x[1 : n]);
16          else mColoring(k + 1);
17      } until (false);
18  }
```

State space tree used → degree *m*

→height*n+1*

- Each node at level *i* has *m* children corresponding to *m* possible assignments to x_i, $1 \le i \le n$.
- Nodes at level *n+1* are leaf nodes

Figure 5.7.shows the state space tree when *n=3* and *m=3.*

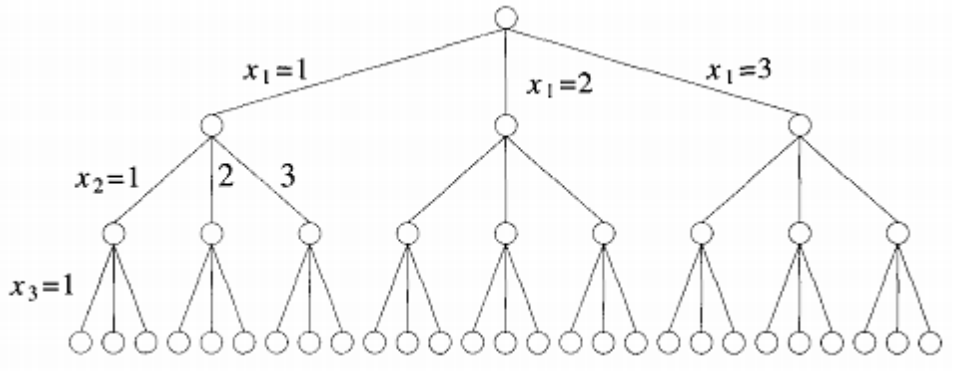

Figure 5.7: State space tree for mColoring when n=3 and m=3.

mColoring

- First assign the graph to its adjacency matrix, setting the array *x[ ]* to *0*.
- Invoke the statement *mColoring(1)*.
- The main loop of *mColoring*
  - Repeatedly picks an element from the set of possibilities,
  - Assigns it to x_k, and
  - Calls *mColoring* recursively.

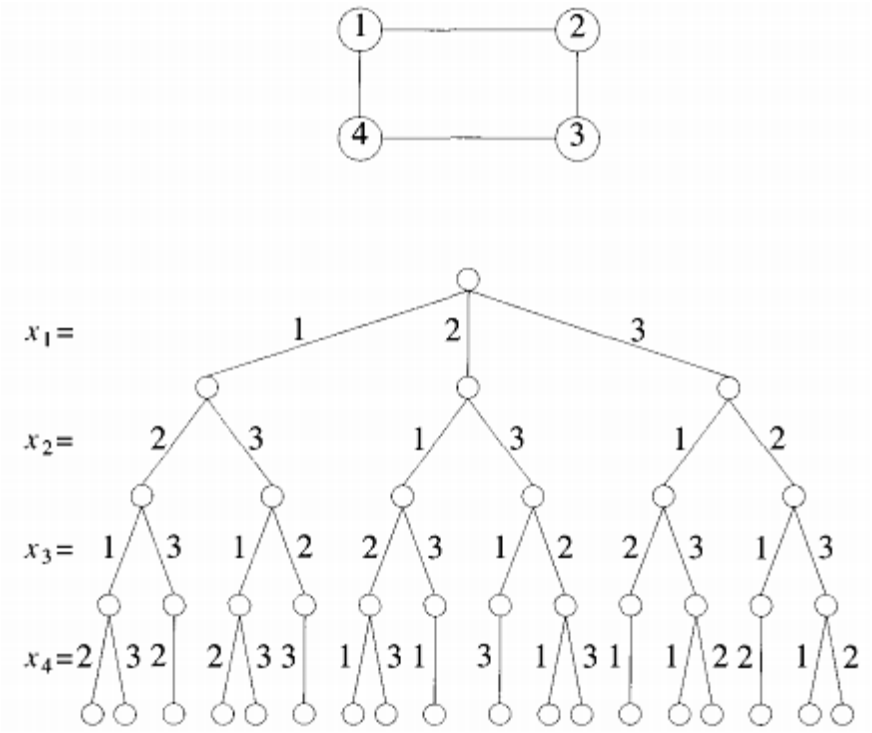

Figure 5.7: A 4-node graph and all possible 3-colorings

In Figure 5.7, the tree is generated by *mColoring*.

- Each path to leaf represents a coloring using at most 3 colors.
- After choosing $x_1=2$ and $x_2=1$, the possible choices for x_3 are *2* and *3*.
- After choosing $x_1=2$ and $x_2=1$, and $x_3=2$, possible values for x_4 are *1* and *3*. And so on.

Computing time for mColoring:

The number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$.

At each internal node, *O(mn)* time is sent by *NextValue* to determine the children corresponding to legal coloring.

The total time is bounded by

$$\sum_{i=0}^{n-1} m^{i+1}n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2)/(m-1) = O(nm^n).$$

```
1   Algorithm NextValue(k)
2   // x[1], ..., x[k − 1] have been assigned integer values in
3   // the range [1, m] such that adjacent vertices have distinct
4   // integers. A value for x[k] is determined in the range
5   // [0, m]. x[k] is assigned the next highest numbered color
6   // while maintaining distinctness from the adjacent vertices
7   // of vertex k. If no such color exists, then x[k] is 0.
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
12          if (x[k] = 0) then return; // All colors have been used.
13          for j := 1 to n do
14          {   // Check if this color is
15              // distinct from adjacent colors.
16              if ((G[k, j] ≠ 0) and (x[k] = x[j]))
17              // If (k, j) is and edge and if adj.
18              // vertices have the same color.
19                  then break;
20          }
21          if (j = n + 1) then return; // New color found
22      } until (false); // Otherwise try to find another color.
23  }
```

5.1.5. Hamilton Cycles

Let *G=(V,E)* be a connected graph with *n* vertices.

*A Hamilton Cycle is a round-trip path along 'n' edges of 'G' that visits every vertex once and returns to its starting positi*on.→ Sir William Hamilton

If a Hamilton Cycle begins at some vertex $v_1 \epsilon G$ and the vertices of *G* are visited in order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in *E*, 1in and the vi are distinct except for v_1 and v_{n+1}which are equal.

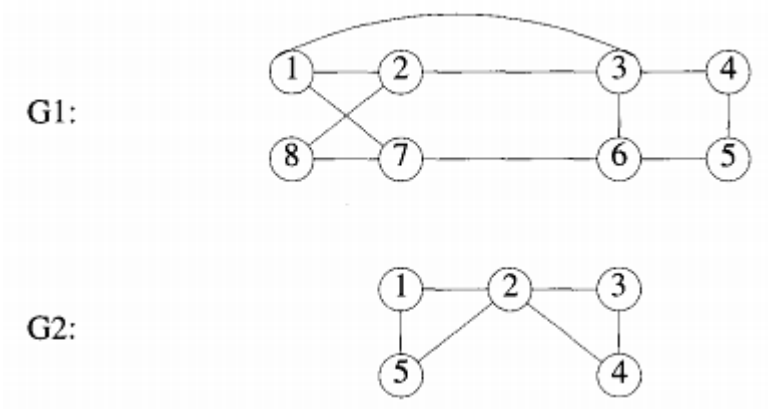

Figure 5.8: Two graphs, one containing a Hamilton Cycle.

In Figure 5.8,

- The graph *G1* contains a Hamilton Cycle 1,2,8,7,6,5,4,3,1
- The graph *G2* contains no Hamilton Cycle.

To find whether a graph contains Hamilton Cycle

Solution: Backtracking algorithm

The graph is either directed or undirected.

Output: Distinct cycles.

The backtracking solution vector $(x_1,\ldots,x_n)$ is defined so that x_i represents the i^{th} visited vertex of the proposed cycle.

To compute the set of possible vertices for x_k if $(x_1,..,x_{k-1})$ is already chosen,

- If $k=1$, then x_1 can be any one of the n vertices.
- Set $x_1=1$ to avoid printing same cycle n times.
- If 1<k<n, the x_k can be any vertex v that is distinct from $x_1,x_2,\ldots x_{k-1}$, and v is connected by an edge to x_{k-1}.
- The vertex x_n can only be the one remaining vertex, and it must be connected to x_{n-1} and x_1.

The *NextValue(k)* given below determines next vertex for the proposed cycle.

```
1   Algorithm NextValue(k)
2   // x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
3   // no vertex has as yet been assigned to x[k]. After execution,
4   // x[k] is assigned to the next highest numbered vertex which
5   // does not already appear in x[1 : k − 1] and is connected by
6   // an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
7   // in addition x[k] is connected to x[1].
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
12          if (x[k] = 0) then return;
13          if (G[x[k − 1], x[k]] ≠ 0) then
14          { // Is there an edge?
15              for j := 1 to k − 1 do if (x[j] = x[k]) then break;
16                                  // Check for distinctness.
17              if (j = k) then // If true, then the vertex is distinct.
18                  if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
19                      then return;
20          }
21      } until (false);
22  }
```

Using *NextValue*a recursive backtracking schema can be particularized to find all Hamilton Cycles as shown in algorithm given below:

```
1   Algorithm Hamiltonian(k)
2   // This algorithm uses the recursive formulation of
3   // backtracking to find all the Hamiltonian cycles
4   // of a graph. The graph is stored as an adjacency
5   // matrix G[1 : n, 1 : n]. All cycles begin at node 1.
6   {
7       repeat
8       { // Generate values for x[k].
9           NextValue(k); // Assign a legal next value to x[k].
10          if (x[k] = 0) then return;
11          if (k = n) then write (x[1 : n]);
12          else Hamiltonian(k + 1);
13      } until (false);
14  }
```

- The algorithm starts by first initializing the adjacency matrix *G[1:n,1:n]*
- Then sets *x[2:n]* to zero and *x[1]* to 1
- Then execute Hamiltonian(2).

5.2. Branch and Bound

A *feasible solution* is a point in the problem's search space that satisfies all the problem's constraints.

An *optimal solution* is a feasible solution with the best value of the objective function

Compared to backtracking, branch-and-bound requires two additional items:

- A way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node.
- Thevalue of the best solution seen so far.

With this information, compare a node's bound value withthe value of the best solution seen so far.

The principal idea of the branch-and-bound technique.

- If the bound value is not better than the value of the best solution seen so far, the node is non-promising and can be terminated.
- No solutionobtained from it can yield a better solution than the one already available

A search path is terminated at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

i. The value of the node's bound is not better than the value of the best solution seen so far.

ii. The node represents no feasible solutions because the constraints of the problem are already violated.

iii. The subset of feasible solutions represented by the node consists of a single pointand hence no further choices can be made

5.2.1. Assignment Problem

Problem: assigning n people to n jobs so that the total cost of the assignment is as small as possible.

An instance of the assignment problem is specifiedby ann×ncost matrixCso that we can state the problem as follows:

"Select oneelement in each row of the matrix so that no two selected

elements are in thesame column and their sum is the smallest possible."

Consider C as follows:

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

To find a lower bound on the cost of an optimal selection withoutactually solving the problem:

- The cost of any solution, including an optimal one, cannot be smaller than the sumof the smallest elements in eachof thematrix's rows.
- For the instance sum is 2+3+1+4=10.
- For any legitimate selection that selects 9 from the first row, the lower bound will be 9+3+1+4=17.

The problem's state space tree deals with the order in which the tree nodes will be generated. Rather than generating a single child of the last promising node as in backtracking, generate all the children of the most promisingnode among onterminated leaves in the current tree.

Which of the nodes is most promising?

- Compare the lower bounds of the live nodes.
- Consider a node with the best bound as most promising,

This variation of thestrategy is called the*best-first branch-and-bound*.

Startwith the root that corresponds to no elements selected from the cost matrix C.

- The lower-bound value for the root, denoted*lb*, is 10.
- Thenodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person *a*(Figure 5.9).

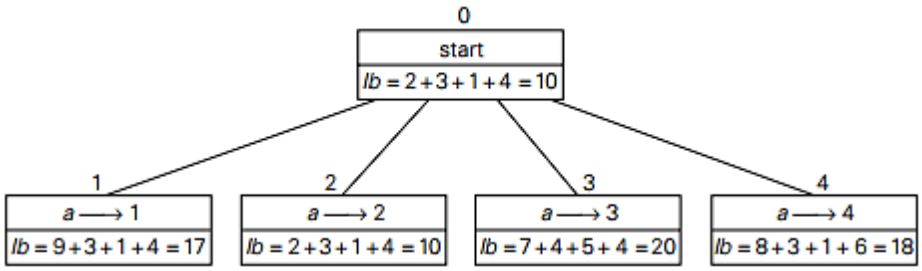

Figure 5.9: Levels 0 and 1 of the state-space tree for the instance of the assignmentproblem being solved with the best-first branch-and-bound algorithm. Thenumber above a node shows the order in which the node was generated.A node's fields indicate the job number assigned to person *a*and thelower bound value, *lb*,for this node.

- There are four live leaves—nodes 1 through 4—that may contain an optimal solution
- The most promising of them is node 2 because it has the smallest lower-bound value.
- Using best-first search strategy, branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column—the three different jobs that can be assigned to person *b* (Figure 5.10)

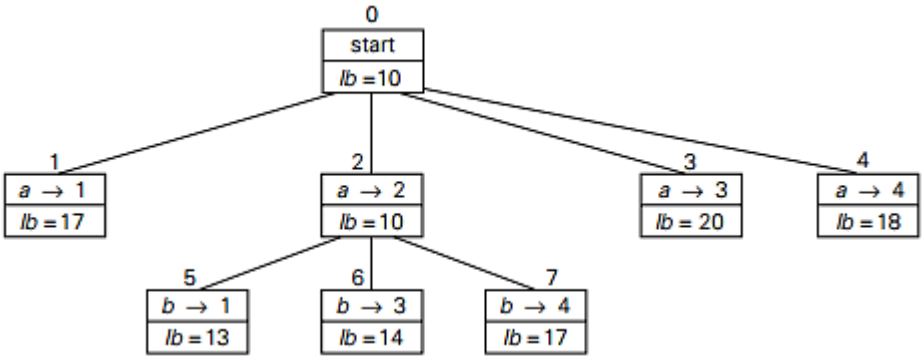

Figure 5.10: Levels 0, 1, and 2 of the state-space tree for the instance of the assignmentproblem being solved with the best-first branch-and-bound algorithm.

- Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, choose the one with the smallest lower bound, node 5.
    - First, we consider selecting the third column's element from *c*'s row,this leaves us with no choice but to select the element from the fourth column of *d*'s row.
    - This yields leaf 8 (Figure 5.11)which corresponds to the feasible

solution{a→2,b→1,c→3,d→4}with the total cost of 13.

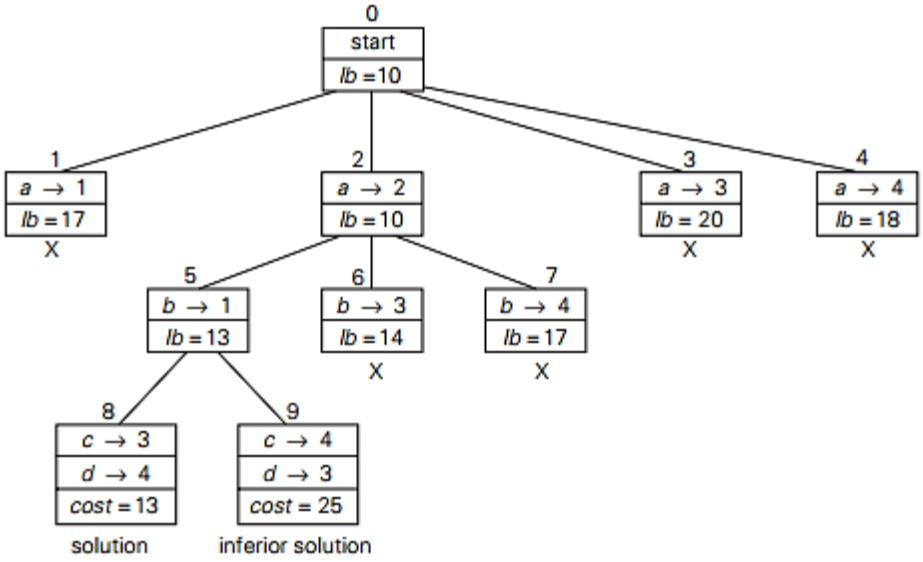

Figure 5.11: Complete state-space tree for the instance of the assignment problemsolved with the best-first branch-and-bound algorithm

- Its sibling, node 9, corresponds to the feasible solution {a→2, b→1,c→4,d→3}with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated.
- The lower-bound values of nodes1, 3, 4, 6, and 7 are not smaller than 13, the value of the best selection seen so far (leaf 8).

  Hence, terminate all of them and recognize the solution represented by leaf 8 as theoptimal solution to the problem

5.2.2. Travelling Sales Person Problem

It is possible to apply the branch-and-bound technique to instances of thetraveling salesman problem with a reasonable lower bound on tourlengths.

To obtain lower bound find the smallest element in the intercity distance matrix *D* and multiply it by the number of cities *n*.

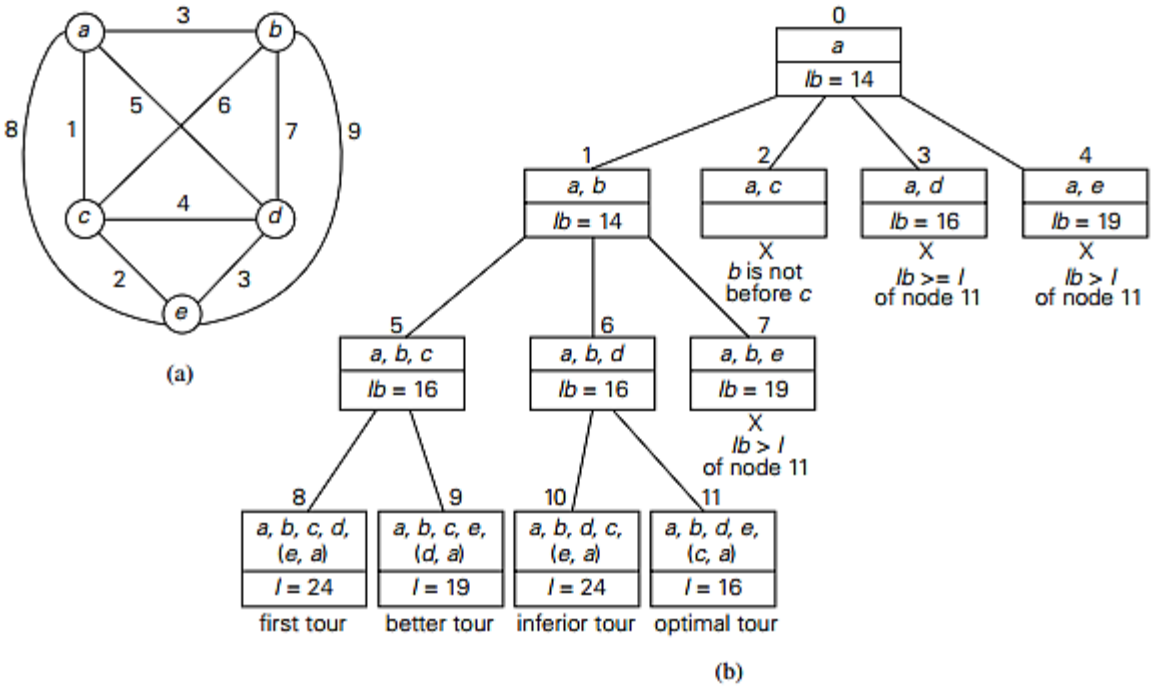

Figure 5.12: (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithmto find a shortest Hamiltonian circuit in this graph. The list of vertices ina node specifies a beginning part of the Hamiltonian circuits representedby the node.

It is possible compute alower bound on the length*l* of any tour as follows.

i. For each cityi,1≤i≤n, find the sums *i* of the distances from city *i* to the two nearest cities

ii. Compute the sum *s*of these*n*numbers

iii. Divide the result by 2, and

iv. if all the distances are integers, round up the result to the nearest integer:

$$lb = \lceil s/2 \rceil. \qquad (5.1)$$

For the instance in Figure 5.12a, formula 5.1) yields

$$lb = \lceil [(1+3)+(3+6)+(1+2)+(3+4)+(2+3)]/2 \rceil = 14.$$

v. For any subset of tours that must include particular edges of a given graph, modify lower bound (lb) accordingly.

For example, for all theHamiltonian circuits of the graph in Figure 5.12a that must include edge(a, d) by summing up the lengths of the two shortestedges incident with each of the

vertices, with the required inclusion of edges(a, d)and(d, a), the lower bound is:

$$\lceil[(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2\rceil = 16.$$

Apply the branch-and-bound algorithm, with the bounding functiongiven by formula 5.1.to find the shortest Hamiltonian circuit for the graph in Figure 5.12a.

To reduce the amount of potential work

i. Without loss of generality, consider only tours that start at*a*.

ii. Since the graph is undirected, generate only tours in which*b*is visited before*c*.

iii. After visitingn−1=4 cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one. The State space tree is given in Figure 5.12b.

5.2.3.0/1 Knapsack Problem

Problem: givennitemsof known weights$w_i$and valuesv_i,i=1,2,. . .,n,and a knapsack of capacityW,find themost valuable subset of the items that fit in the knapsack.

- Order the items of a given instance in descending order by their value-to-weight ratios.
- The first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n.$$

Structure the state-space tree for this problem as a binary treeconstructed as follows:

- Each node on the i^{th} level of this tree, 0≤i≤n,represents all the subsets of*n*items that include a particular selection made from the first*i* ordered items.
- This is the path from root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion.

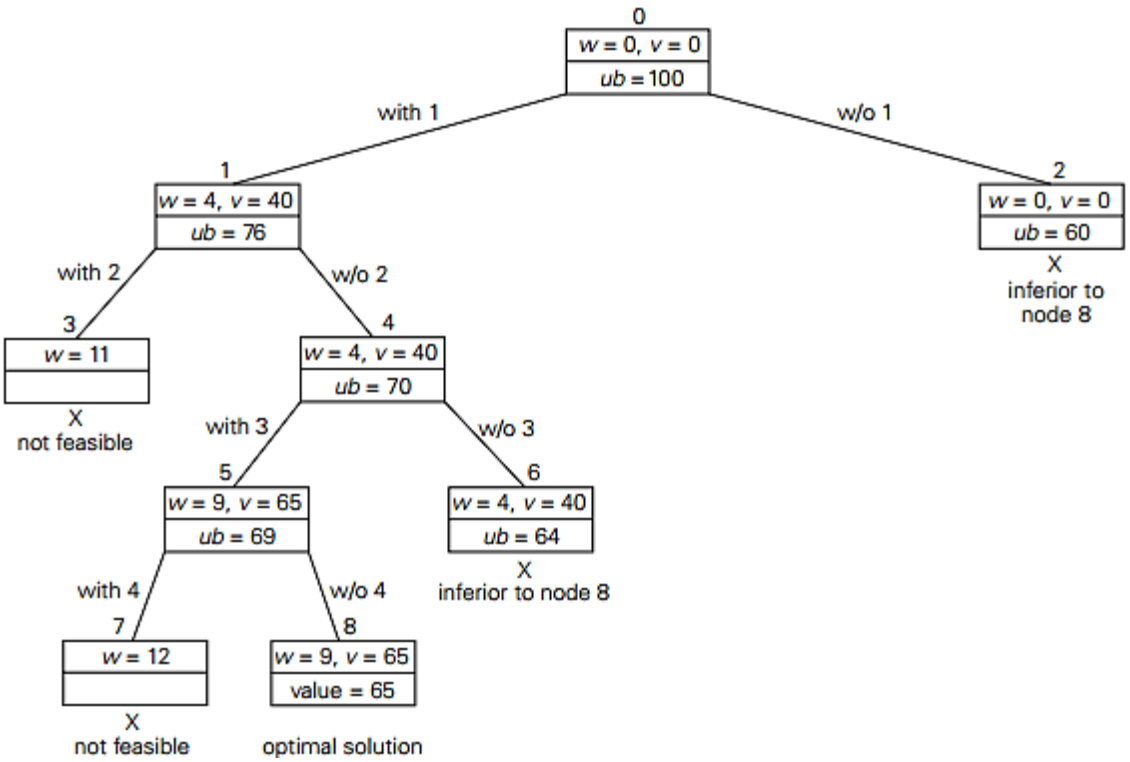

Figure 5.13: State-space tree of the best-first branch-and-bound algorithm for theinstance of the knapsack problem.

- Record the
  - Total weight *w* and
  - The total value *v* of this selection in the node,
  - Upper bound *ub* on the value of any subset that can be obtained by adding zero or more items to this selection.
- To compute the upper bound*ub*: add to*v*,
  - The total value of the items already selected,
  - the product of the remaining capacity of the knapsack*W*−*w*and
  - thebest per unit payoff among the remaining items, which isv_{i+1}/w_{i+1}

$$ub = v + (W - w)(v_{i+1}/w_{i+1}). \quad (5.2)$$

For example, apply the branch-and-bound algorithm to the following instance of knapsack problem:

item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10.

In Figure 5.13,

- At the root no items have been selected as yet. Hence, both the total weight of the items already selected *w* and their total value *v* are equal to 0.
- The value of the upper bound computed by formula (5.2) is $100.
- Node 1, the left child of the root, represents the subsets that include item 1.
    - The total weight and value of the items already included are 4 and $40, respectively; the value of the upper bound is 40 +(10−4)∗6 = $76.
- Node 2 represents the subsets that do not include item 1.
    - w=0, v=$0,andub=0+(10−0)∗6 = $60.

Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem and hence, branch from node 1 first.

- Its children—nodes 3 and 4—represent subsets with item 1 and with and without item 2, respectively
    - The total weight *w* of every subset represented by node 3 exceeds the knapsack's capacity; node 3 can be terminated immediately.
    - Node 4 has the same values of *w* and *v* as its parent; the upper boundubis equal to 40 +(10−4)∗5 = $70.
    - Selecting node 4 over node 2 for the next branching, nodes 5 and 6 by respectively are obtained including and excluding item3.

The total weights and values as well as the upper bounds forthese nodes are computed in the same way as for the preceding nodes.

- Branching from node 5 yields node 7, which represents no feasible solutions, and node 8, which represents just a single subset {1, 3} of value $65.
- The remaining live nodes 2 and 6 have smaller upper bound values than the value of the solution representedby node 8.

Hence, both can be terminated making the subset {1, 3} of node 8 theoptimal solution to the problem.

5.2.3.1. LC Branch and Bound solution

Example [LCBB]:

Consider the knapsack instance $n = 4$, $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$, $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, and $m = 15$. Let us trace the working of an LC branch-and-bound search using $\hat{c}(\cdot)$ and $u(\cdot)$ as defined previously. We continue to use the fixed tuple size formulation. The search begins with the root as the E-node. For this node, node 1 of Figure 8.8, we have $\hat{c}(1) = -38$ and $u(1) = -32$.

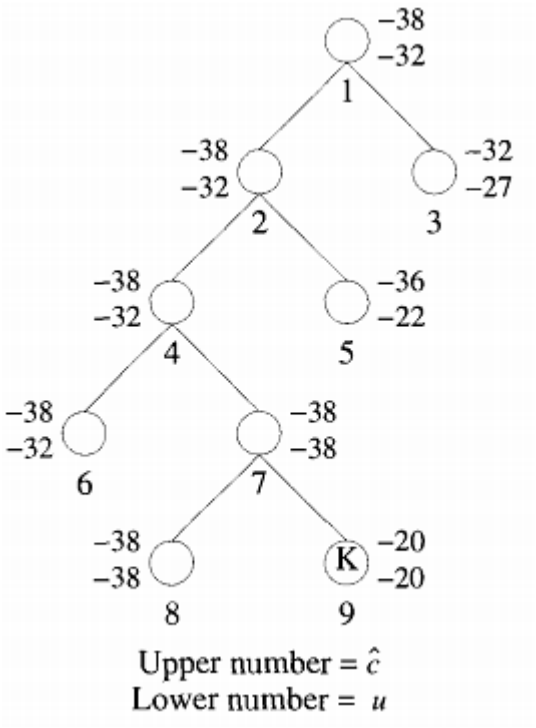

Figure 5.14: LC branch and bound tree

The Computation of *u(1)* and *c(1)* is done as follows:

- The bound *u(1)* has a value *UBound(0,0,0,15)*.
  - UBound scans through the objects from left to right starting

from *j*,

- it adds these objects into the knapsack until the first object that doesn't fit is encountered.
- This returns the negation of the total profit of all the objects in the knapsack plus *cw*.

- In function *UBound*,
  - *c*and *b* start with the value of zero.
  - For *i=1,2, and 3*
    - *c* gets incremented by *2,4, and 6* respectively.
    - *b* gets decremented by 10,0, and 12 respectively.
  - When *i=4*, the test (c+w[i]≤m) fails and return value -32.
- Function *Bound* is similar to *UBound*, except that it also considers a fraction of the first object that doesn't fit the knapsack. Example, the object 4 doesn't fit and value is 9. Bound considers a fraction 3/9 of the object 4 and returns -32-3/9*18=-38
- LCBB sets ans=0 and upper=-32.
  - The E-node is expanded and two children 2 and 3 are generated. →live nodes.
  - The cost c(2)=-32, c(3)=-32, u(2)=-32, and u(3)=-27.
  - Node-2 is next E-node. Expand and generate 4 and 5. →live nodes.
  - 4 has least c value and is next E-node. Generate 6 and 7. →live nodes.
  - Upper is -38.
  - Assume 7is next E-node. Generate 8 and 9.
  - Node 8 is a solution node. Update upper to -38. Set 8 as live node.
  - Node 9 with c(9) > upper is terminated.
  - Node 6 and 8 are two live nodes with least c.
  - Search terminates at node 8 as answer node.

- Value is -38 with the path 8,7,4,2,1

- Assign values to x_i such that $\sum p_i x_i = upper$.
- To extract values of x_i:
  - Associate with each node, a one bit field, *tag*, such that the sequence of the tag bits from answer node to the root gives x_i values.
  - Example, tag(2)=tag(4)=tag(6)=tag(8)=1
  - Tag sequence for the path 8,7,4,2,1 is 1011. x_4=1,x_3=0,x_2=1, and x_1=1.

To solve knapsack problem using LCBB, specify

i. The structure of nodes in the state space tree being searched.

ii. How to generate the children of a given node.

iii. How to recognize a solution node

iv. A representation of a list of live nodes and a mechanism for adding anode into the list as well as identifying the least cost node.

To generate x's children:

- ✓ Identify the level of x in state space tree. →field used *level.*
- ✓ Set $x_{level(x)}$=1 for left child and $x_{level(x)}$=0 for right child.
- ✓ Retain the value *cu (capacity unused)*
- ✓ The evaluation of c(x) and u(x) requires value of profit. $\sum_{1 \le i < level(x)} p_i x_i$
- ✓ This value can be retained in the field pe.
- ✓ In order to determine the live node, with least c value , compute c(x).
- ✓ The c(x) is explicitly stored in ub.

We need nodes with six fields: parent, level, tag, cu, pe, and ub using which the children of any node can be easily determined.

- ✓ The left child is feasible iff cu(x)>= $w_{level(x)}$.
  - Parent(y)=x
  - Level(y)=level(x)+1

- cu(y)=cu(x)-$w_{level(x)}$.
- pe(y)=pe(x)+$P_{level(x)}$.
- tag(y)=1, and
- ub(y)=ub(x)

The functions to be performed on list of live nodes:

i. test if the list is empty

ii. add nodes, and

iii. delete a node with a least ub.

5.2.3.2. FIFO Branch and Bound solution

Example:

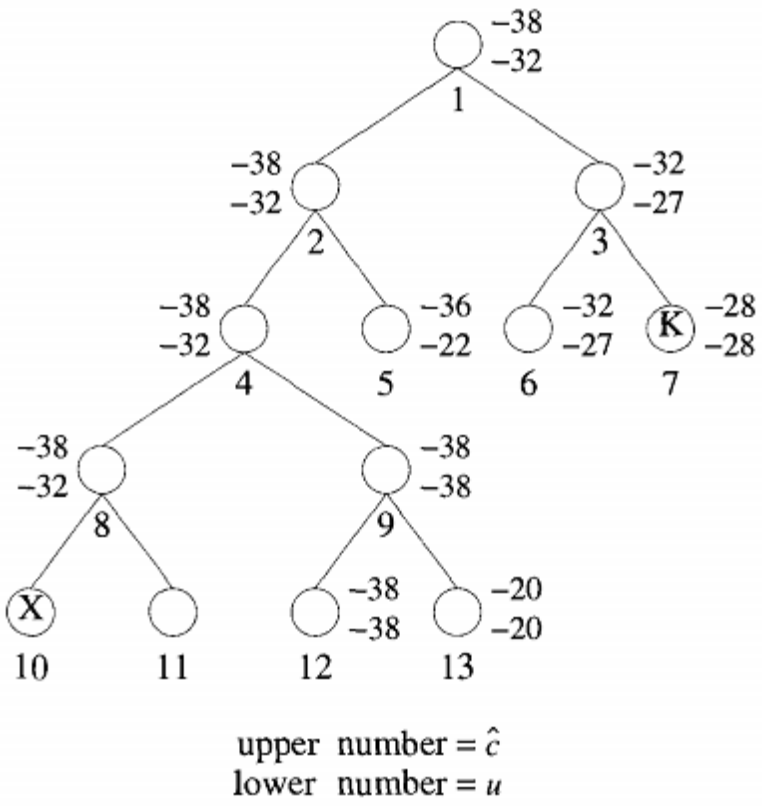

Figure 5.16: FIFO branch and bound example

- ✓ Initially, node 1 is the E-node and queue of live nodes is empty. upper is initialized to u(1)= -32.
- ✓ Generate nodes from left to right.

- ✓ Generate 2 and 3 and add to the queue. Upper value unchanged.
- ✓ Node 2 is next E-node. Generate 4 and 5 and add into queue.
- ✓ Expand node 3, the next E-node. Generate 6 and 7. c(7) > upper so node 7 immediately killed.
- ✓ Expand node 4.generate 8 and 9 and add to querue.
- ✓ Update *upper to -38.*
- ✓ 5 and 6 are next. Both are not expanded since there c value is > upper.
- ✓ Node 8 is next E-node. Generate nodes 10 and 11.
    - o Node 10 is killed due to infeasibility.
    - o Node 11's c value is > upper so killed.
- ✓ Expand node 9. Generate 12and 13.
    - o 13 is killed.its c value>upper.
    - o Add 12 in queue. It has no children and the search is terminated.
- ✓ Output→ the value of upper and path from node 12 to root.

5.3. NP-Complete and NP-Hard problems

5.3.1. Basic Concepts

The best algorithms for their solutions have computing times that cluster into two groups:

i. Problems whose solutions times are bounded by polynomials of small degree (*O(log n), O(n)*).

ii. Problems whose best-known algorithms are non-polynomial ($O(n^2 2^n)$, $O(2^{n/2})$).

NOTE:

- Algorithms whose computing times are greater than polynomial very quickly require vast amounts of time to execute.
- Many of the problems for which there are no known polynomial time algorithms are computationally related. →***NP-Hard and NP-Complete Problems.***

NP-Complete and NP-Hard Problems

- ✓ A problem that is NP-Complete has the property that it can be solved in polynomial time if and only if all other NP-Complete problems can also be solved in polynomial time.
- ✓ If an NP-Hard problem can be solved in polynomial time, then all NP-Complete problems can be solved in polynomial time.
- ✓ All NP-Complete problems are NP-Hard but some NP-Hard problems are not known to be NP-Complete.

5.3.2. Nondeterministic Algorithms

The algorithms for which the result of every operation is uniquely defined are called *deterministic algorithms*.

Algorithms may be made to contain operations whose outcomes are not uniquely defined but are limited to specified sets of possibilities. The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition. Such algorithms are named as *nondeterministic algorithms*.

To specify nondeterministic algorithms, three new functions are introduced:

i. Choice(S) → arbitrarily chooses one of the elements of set S.

ii. Failure( ) → signals an unsuccessful completion

iii. Success ( ) → signals a successful completion

The assignment statement *x:=Choice(1,n)* can result in *x* being assigned any of the integers in range [1,n].

The *Failure()* and *Success()* signals are used to define computation of the algorithm. →cannot be used to effect a **return.**

NOTE:

- ✓ A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal.
- ✓ The computation times for *Choice, Success, and Failure* is taken as *O(1)*.
- ✓ Machine capable of executing a nondeterministic is called a *nondeterministic* machine.

Example:

Consider the problem of searching for an element x in a given set of elements $A[1:n]$, $n \geq 1$. We are required to determine an index j such that $A[j] = x$ or $j = 0$ if x is not in A. A nondeterministic algorithm for this is Algorithm 11.1.

```
1   j := Choice(1, n);
2   if A[j] = x then {write (j); Success();}
3   write (0); Failure();
```

Algorithm : Nondeterministic Search

- ✓ No=umber 0 can be output if and only if there is no *j* such that *A[j]= x.*
- ✓ The complexity of above algorithm is *O(1).*

Example [Sorting]:

Let $A[i]$, $1 \leq i \leq n$, be an unsorted array of positive integers. The nondeterministic algorithm NSort(A, n) sorts the numbers into nondecreasing order and then outputs them in this order. An auxiliary array $B[1:n]$ is used for convenience. Line 4 initializes B to zero though any value different from all the $A[i]$ will do. In the **for** loop of lines 5 to 10, each $A[i]$ is assigned to a position in B. Line 7 nondeterministically determines this position. Line 8 ascertains that $B[j]$ has not already been used. Thus, the order of the numbers in B is some permutation of the initial order in A. The **for** loop of lines 11 and 12 verifies that B is sorted in nondecreasing order. A successful completion is achieved if and only if the numbers are output in nondecreasing order. Since there is always a set of choices at line 7 for such an output order, algorithm NSort is a sorting algorithm. Its complexity is $O(n)$.]

```
1   Algorithm NSort(A, n)
2   // Sort n positive integers.
3   {
4       for i := 1 to n do B[i] := 0; // Initialize B[ ].
5       for i := 1 to n do
6       {
7           j := Choice(1, n);
8           if B[j] ≠ 0 then Failure();
```

```
9            B[j] := A[i];
10       }
11       for i := 1 to n − 1 do  // Verify order.
12            if B[i] > B[i + 1] then Failure();
13       write (B[1 : n]);
14       Success();
15   }
```

Algorithm: Nondeterministic Sorting

Definition:

Any problem for which the answer is zero or one is called a ***decision algorithm.*** *Any algorithm that involves the identification of an optimal value of a given cost function is known as an* ***optimization problem.*** *An* ***optimization algorithm*** *is used to solve optimization problem*

Many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time if and only if the corresponding optimization problem can.

Example: Maximum Clique

A maximal complete subgraph of a graph $G = (V, E)$ is a *clique*. The size of the clique is the number of vertices in it. The *max clique problem* is an optimization problem that has to determine the size of a largest clique in G. The corresponding decision problem is to determine whether G has a clique of size at least k for some given k. Let DClique(G, k) be a deterministic decision algorithm for the clique decision problem. If the number of vertices in G is n, the size of a max clique in G can be found by making several applications of DClique. DClique is used once for each k, $k = n, n-1, n-2, \ldots$, until the output from DClique is 1. If the time complexity of DClique is $f(n)$, then the size of a max clique can be found in time $\leq n\ f(n)$. Also, if the size of a max clique can be determined in time $g(n)$, then the decision problem can be solved in time $g(n)$. Hence, the max clique problem can be solved in polynomial time if and only if the clique decision problem can be solved in polynomial time. □

Example: 0/1 Knapsack Problem

The knapsack decision problem is to determine whether there is a 0/1 assignment of values to x_i, $1 \leq i \leq n$, such that $\sum p_i x_i \geq r$ and $\sum w_i x_i \leq m$. The r is a given number. The p_i's and w_i's are nonnegative numbers. If the knapsack decision problem cannot be solved in deterministic polynomial time, then the optimization problem cannot either.

Let *n* be the length of the input to the algorithm. Consider that all inputs are integers.

- ✓ The length of the input is measured assuming a binary representation. Example, 10 is represented as *1010* whose length is *4*.
- ✓ A positive integer *k* has a length of $\lfloor \log_2 k \rfloor + 1$ bits when represented in binary. The length of binary representation of *0* is *1*.
- ✓ The size *n* of the input to an algorithm is *the sum of the lengths of the individual numbers being input.*

Consider a radix *r* for an input with different representation. Then the length of the positive number *k*is $\lfloor \log_r k \rfloor + 1$.

Since, $\log_r k = \log_2 k / \log_2 r$, the length of any input using radix *r (r>1)* representation is *c(r)n*.

Note: Input is *unary form* if *r=1*. →length of positive integer *k* is *k*.

Example: Max Clique

The input to the max clique decision problem can be provided as a sequence of edges and an integer k. Each edge in $E(G)$ is a pair of numbers (i, j). The size of the input for each edge (i, j) is $\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2$ if a binary representation is assumed. The input size of any instance is

$$n = \sum_{\substack{(i,j) \in E(G) \\ i<j}} (\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2) + \lfloor \log_2 k \rfloor + 1$$

Note that if G has only one connected component, then $n \geq |V|$. Thus, if this decision problem cannot be solved by an algorithm of complexity $p(n)$ for some polynomial $p()$, then it cannot be solved by an algorithm of complexity $p(|V|)$. □

Example: 0/1 Knapsack

Assuming p_i, w_i, m, and r are all integers, the input size for the knapsack decision problem is

$$q = \sum_{1<i\leq n} (\lfloor \log_2 p_i \rfloor + \lfloor \log_2 w_i \rfloor) + 2n + \lfloor \log_2 m \rfloor + \lfloor \log_2 r \rfloor + 2$$

Note that $q > n$. If the input is given in unary notation, then the input size s is $\sum p_i + \sum w_i + m + r$. Note that the knapsack decision and optimization problems can be solved in time $p(s)$ for some polynomial $p()$ (see the dynamic programming algorithm). However, there is no known algorithm with complexity $O(p(n))$ for some polynomial $p()$. □

Definition:

The time required by a nondeterministic algorithm performing on any given input is the minimum number of steps needed to reach a successful completion if there exists a sequence of choices leading to such a completion. If not, the time required is *O(1)*.

The complexity of nondeterministic algorithm is *O(f(n))* for the inputs of size *n*.

Example: Knapsack Decision Problem

DKP is a nondeterministic polynomial time algorithm for the knapsack decision problem. The **for** loop of lines 4 to 8 assigns 0/1 values to $x[i]$, $1 \leq i \leq n$. It also computes the total weight and profit corresponding to this choice of $x[\,]$. Line 9 checks to see whether this assignment is feasible and whether the resulting profit is at least r. A successful termination is possible iff the answer to the decision problem is yes. The time complexity is $O(n)$. If q is the input length using a binary representation, the time is $O(q)$. □

```
1   Algorithm DKP(p, w, n, m, r, x)
2   {
3       W := 0; P := 0;
4       for i := 1 to n do
5       {
6           x[i] := Choice(0, 1);
7           W := W + x[i] * w[i]; P := P + x[i] * p[i];
8       }
9       if ((W > m) or (P < r)) then Failure();
10      else Success();
11  }
```

Algorithm: Nondeterministic knapsack algorithm

Example: Max Clique

DCK is a nondeterministic algorithm for the clique decision problem. The algorithm begins by trying to form a set of k distinct vertices. Then it tests to see whether these vertices form a complete subgraph. If G is given by its adjacency matrix and $|V| = n$, the input length m is $n^2 + \lfloor \log_2 k \rfloor + \lfloor \log_2 n \rfloor + 2$. The **for** loop of lines 4 to 9 can easily be implemented to run in nondeterministic time $O(n)$. The time for the **for** loop of lines 11 and 12 is $O(k^2)$. Hence the overall nondeterministic time is $O(n + k^2) = O(n^2) = O(m)$. There is no known polynomial time deterministic algorithm for this problem. □

```
1   Algorithm DCK(G, n, k)
2   {
3       S := ∅; // S is an initially empty set.
4       for i := 1 to k do
5       {
6           t := Choice(1, n);
7           if t ∈ S then Failure();
8           S := S ∪ {t} // Add t to set S.
9       }
10      // At this point S contains k distinct vertex indices.
11      for all pairs (i, j) such that i ∈ S, j ∈ S, and i ≠ j do
12          if (i, j) is not an edge of G then Failure();
13      Success();
14  }
```

Algorithm: Nondeterministic clique pseudocode

Example: Satisfiability

Let $x_1, x_2, \ldots$ denote boolean variables (their value is either true or false). Let $\bar{x}_i$ denote the negation of x_i. A *literal* is either a variable or its negation. A formula in the propositional calculus is an expression that can be constructed using literals and the operations **and** and **or**. Examples of such formulas are $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ and $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$. The symbol $\vee$ denotes **or** and $\wedge$ denotes **and**. A formula is in *conjunctive normal form* (CNF) if and only if it is represented as $\wedge_{i=1}^{k} c_i$, where the c_i are clauses each represented as $\vee l_{ij}$. The l_{ij} are literals. It is in *disjunctive normal form* (DNF) if and only if it is represented as $\vee_{i=1}^{k} c_i$ and each clause c_i is represented as $\wedge l_{ij}$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ is in DNF whereas $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ is in CNF. The **satisfiability** problem is to determine whether a formula is true for some assignment of truth values to the variables. *CNF-satisfiability* is the satisfiability problem for CNF formulas.

```
1   Algorithm Eval(E, n)
2   // Determine whether the propositional formula E is
3   // satisfiable. The variables are x_1, x_2, ..., x_n.
4   {
5       for i := 1 to n do  // Choose a truth value assignment.
6           x_i := Choice(false, true);
7       if E(x_1, ..., x_n) then Success();
8       else Failure();
9   }
```

Algorithm: Nondeterministic satisfiability

5.3.3. The Classes NP-hard and NP-Complete

An algorithm *A* is *polynomial complexity* if thereexists a polynomial *p()* such that the computing time of *A* is *O(p(n))* for every input of size *n*.

Definition

*P*is the set of all decision problems solvable by deterministic algorithms in polynomial time. *NP* is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Conclusion: $\mathcal{P} \subseteq \mathcal{NP}$.

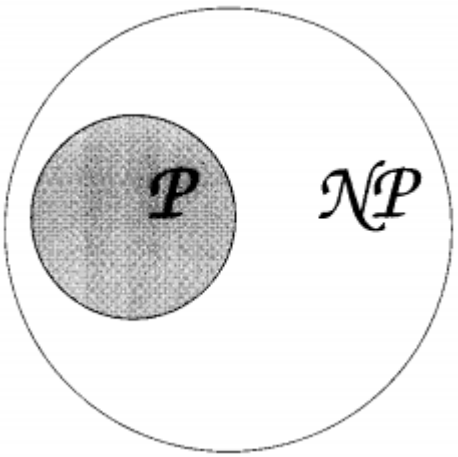

Figure 5.14: Commonly believed relationship between P and NP

Figure 5.14 assumes that P≠NP

Theorem [Cook]: Satisfiability is in *P* if and only if *P=NP.*

Definition:

Let L_1 and L_2 be problems. Problem L_1 *reduces* to L_2 (also written $L_1 \propto L_2$) if and only if there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time. □

Here, α is a transitive relation.

Definition

A problem L is $\mathcal{NP}$-hard if and only if satisfiability reduces to L (satisfiability $\propto L$). A problem L is $\mathcal{NP}$-complete if and only if L is $\mathcal{NP}$-hard and $L \in \mathcal{NP}$. □

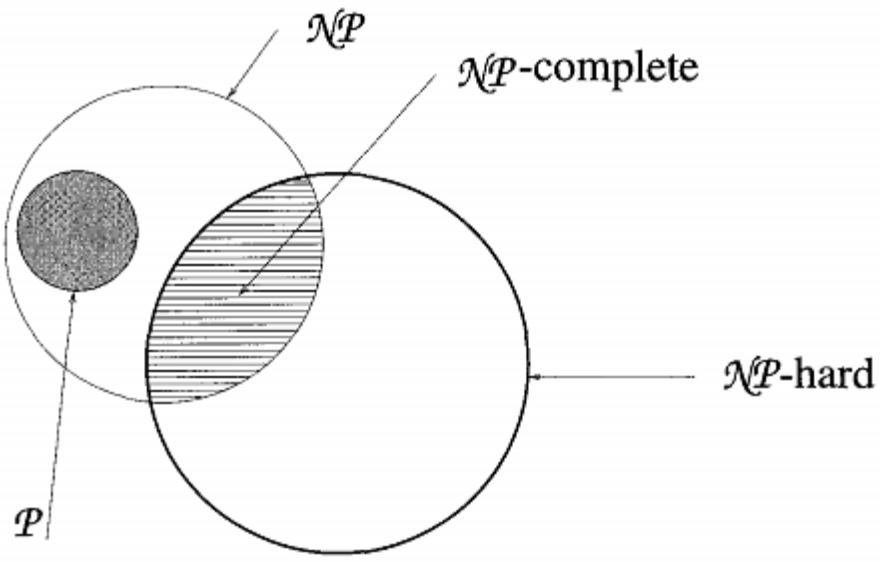

Figure 5.15: Commonly believed relationship among P, NP, NP-Complete and NP-Hard problems

- ✓ There are NP-hard problems that are not NP-complete.
- ✓ Only a decision problem can be NP-complete.
- ✓ An optimization problem can be NP-hard.
- ✓ Optimization problem cannot be NP-complete whereas decision problem can.
- ✓ There also exists NP-hard decision problems that are not NP-complete.

Example:

As an extreme example of an $\mathcal{NP}$-hard decision problem that is not $\mathcal{NP}$-complete consider the halting problem for deterministic algorithms. The *halting problem* is to determine for an arbitrary deterministic algorithm A and an input I whether algorithm A with input I ever terminates (or enters an infinite loop). It is well known that this problem is undecidable. Hence, there exists no algorithm (of any complexity) to solve this problem. So, it clearly cannot be in $\mathcal{NP}$. To show satisfiability $\propto$ the halting problem, simply construct an algorithm A whose input is a propositional formula X. If X has n variables, then A tries out all 2^n possible truth assignments and verifies whether X is satisfiable. If it is, then A stops. If it is not, then A enters an infinite loop. Hence, A halts on input X if and only if X is satisfiable. If we had a polynomial time algorithm for the halting problem, then we could solve the satisfiability problem in polynomial time using A and X as input to the algorithm for the halting problem. Hence, the halting problem is an $\mathcal{NP}$-hard problem that is not in $\mathcal{NP}$. □

Definition:

Two problems L_1 and L_2 are said to be polynomially equivalent if and only if $L_1 \propto L_2$ and $L_2 \propto L_1$.

To show that a problem L_2 is $\mathcal{NP}$-hard, it is adequate to show $L_1 \propto L_2$, where L_1 is some problem already known to be $\mathcal{NP}$-hard. Since $\propto$ is a transitive relation, it follows that if satisfiability $\propto L_1$ and $L_1 \propto L_2$, then satisfiability $\propto L_2$. *To show that an $\mathcal{NP}$-hard decision problem is $\mathcal{NP}$-complete, we have just to exhibit a polynomial time nondeterministic algorithm for it.*