Backtracking & Branch-and-Bound

Introduction

	Description
Backtracking is used for	Problems that are solved by searching a set of solutions or finding an optimal solution that satisfying some constraints
Representation of solution	n-tuple (x_1,x_2,…x_n)
Adv. of Backtracking over brute-force	Brute-force approach examine all possibilities of *n-tuples* , whereas backtrack algorithm has the ability to find the answer less than *m* trails. Where m is less than all possibilities
Idea	Build the solution vector one component at a time and to use modified criterion functions $P_i(x_1,x_2,…x_i)$***(known as bounding functions)*** to test whether the vector being formed has any chance of success.
Constraints	Explicit and Implicit constraints. **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. **Implicit constraints** are rules that determine which of the tuples in the solution space of I satisfy the criterion function.

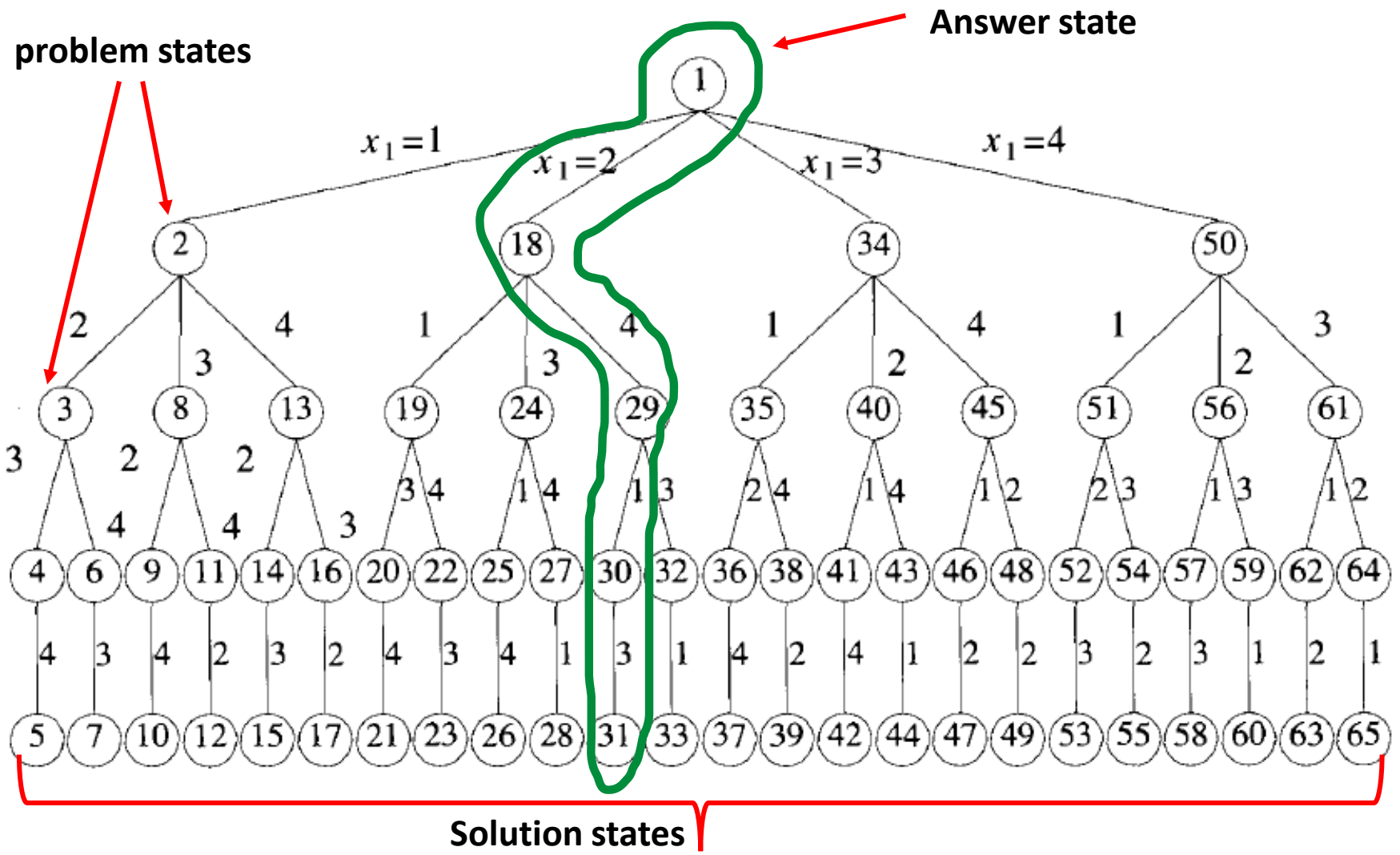
Example of Explicit Constraints	Common examples of explicit constraints are $x_i \geq 0$ or $S_i = \{\text{all nonnegative real numbers}\}$ $x_i = 0$ or 1 or $S_i = \{0, 1\}$ $l_i \leq x_i \leq u_i$ or $S_i = \{a : l_i \leq a \leq u_i\}$

- Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance.
- Tree organization is used for representation of searching the solution space.
- Each node in the tree is known as **problem state.**
- All paths from the root to other nodes define the state space of the problem.
- **Solution states** are problem states in S for which the path from the root to s define a tuple in the solution space.
- The solution states that forms a tuple in solution space that satisfies implicit constraints of the problem is known as **answer states**.

Terminology	Definition
Live node	A node which has been generated and all of whose children have not yet been generated.
E-node	It is a live node whose children are currently being generated.
Dead node	If the node is not expanded further or all its children have been generated.
Bounding functions	These are used **to kill the live nodes** without generating all its children
Backtracking	State generation using **depth first with bounding functions**
Branch-and-Bound	State generation methods in which the **E-node remains the E-node until it is dead.**

Example : State space tree for 4-queens

- Each node in the tree is known as **problem state.**
- All paths from the root to other nodes define the state space of the problem.
- **Solution states** are problem states in S for which the path from the root to s define a tuple in the solution space.
- The solution states that forms a tuple in solution space that satisfies implicit constraints of the problem is known as **answer states**.



General Algorithm of Backtracking

Recursive algorithm

```
1   Algorithm Backtrack(k)
2   // This schema describes the backtracking process using
3   // recursion. On entering, the first k − 1 values
4   // x[1], x[2], ..., x[k − 1] of the solution vector
5   // x[1 : n] have been assigned. x[ ] and n are global.
6   {
7       for (each x[k] ∈ T(x[1], ..., x[k − 1])) do
8       {
9           if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
10          {
11              if (x[1], x[2], ..., x[k] is a path to an answer node)
12                  then write (x[1 : k]);
13              if (k < n) then Backtrack(k + 1);
14          }
15      }
16  }
```

Iterative algorithm

```
1   Algorithm IBacktrack(n)
2   // This schema describes the backtracking process.
3   // All solutions are generated in x[1 : n] and printed
4   // as soon as they are determined.
5   {
6       k := 1;
7       while (k ≠ 0) do
8       {
9           if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
10              x[k − 1]) and B_k(x[1], ..., x[k]) is true) then
11          {
12              if (x[1], ..., x[k] is a path to an answer node)
13                  then write (x[1 : k]);
14              k := k + 1; // Consider the next set.
15          }
16          else k := k − 1; // Backtrack to the previous set.
17      }
18  }
```

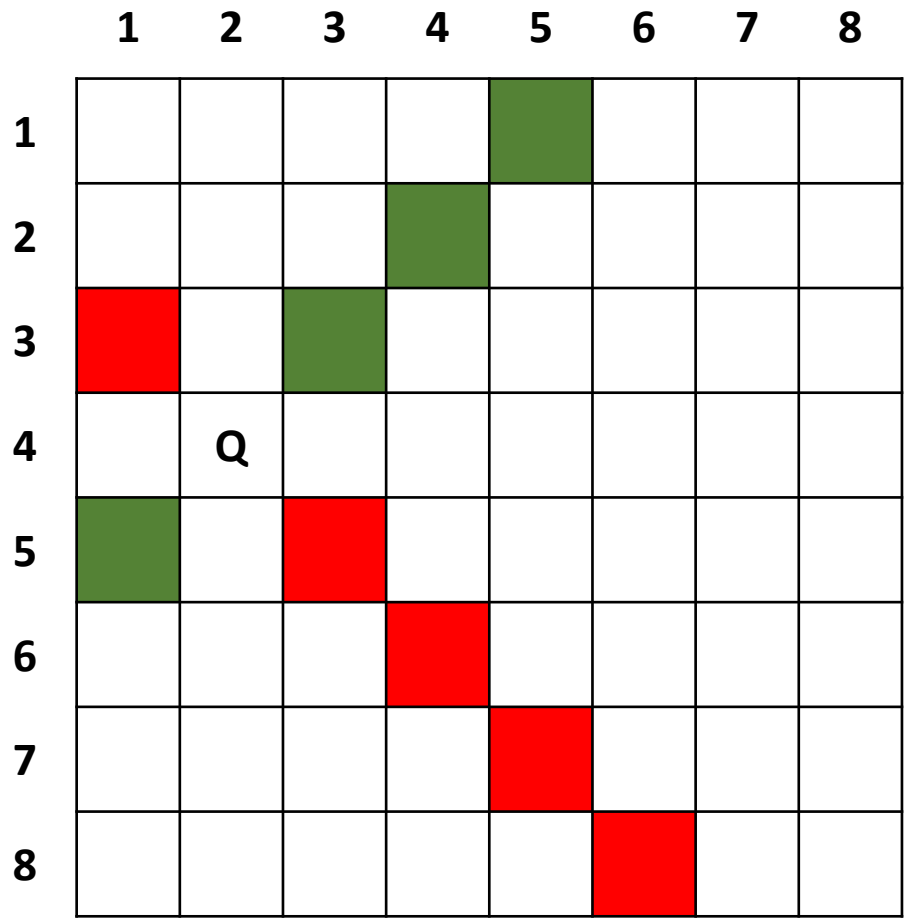
8-Queens Problem

- A classic Combinatorial Problem is to place eight queens on an 8x8 chessboard so that no two attack that is, no two of them are on the same row ,column, or diagonal.
- All solutions to the 8-queens problem can be represented as 8-tuples $(x_1,x_2,\ldots x_8)$ where x_i is the column on which queen *i* is placed.
- The explicit constraints for the problem is $S_i = \{1,2,3,4,5,6,7,8\}$ where $1 \le i \le 8$. Therefore, the solution space contains 8^8 8-tuples.
- The implicit constraints for the problem are that
  - no two x_i's can be same (i.e., all queens must be on different columns) and
  - no two queens can be on the same diagonal.
- The implicit constraint 1 reduces the solution space size from 8^8 to 8!.
- The one of the solution tuple is (4,6,8,2,7,1,3,5)

N - Queens problem is generalization of 8 – Queens problem for any n ≥ 4.

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			

N-Queens Solution



Let Queen location is (i, j) and the squares diagonal to it (say (k, l)) have the property

$i-j = k-l$ (upper left to lower right)

or

$i+j = k+l$ (upper right to lower left)

Example : Let Q location (i,j) = (4,2)

(i-j) = (k-l) = {(3,1),(5,3),(6,5),(7,5)(8,6)}

(i+j)= (k+l) = {(1,5),(2,4),(3,3),(5,1)}

Determining the two queens in same diagonal or not

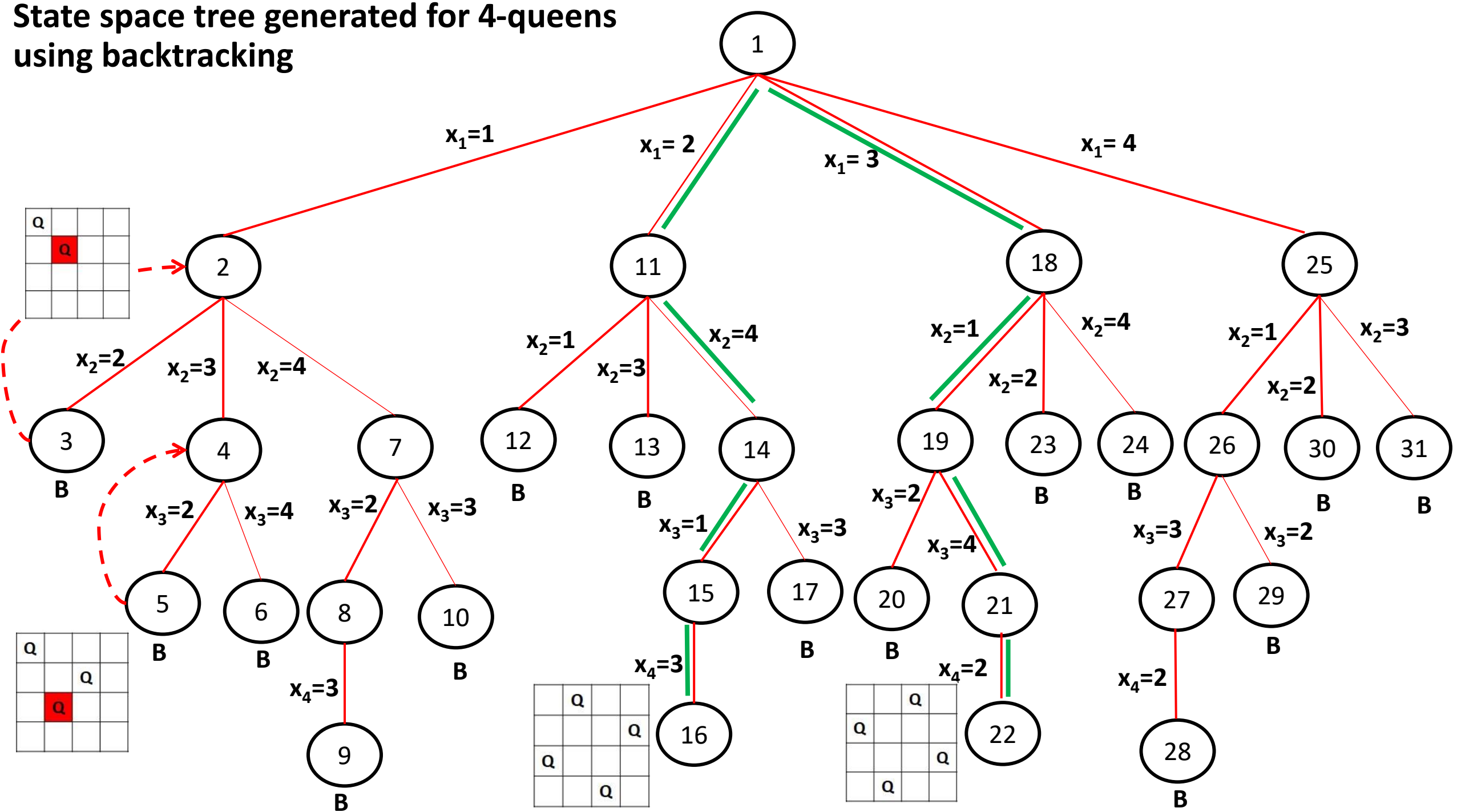
If Q_1 = (i, j) and Q_2= (k, j) then they belong to same diagonal or not if

$i-j = k-l$ or $i+j = k+l$

$l-j = k-i$ $\quad$ $j- l = k-i$

$$abs(l - j) = abs(k - i)$$

State space tree generated for 4-queens using backtracking



N – Queens Algorithm using Backtracking

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9                 or (Abs(x[j] − i) = Abs(j − k)))
10                    // or in the same diagonal
11                then return false;
12      return true;
13  }
```

SumOfSubsets Problem

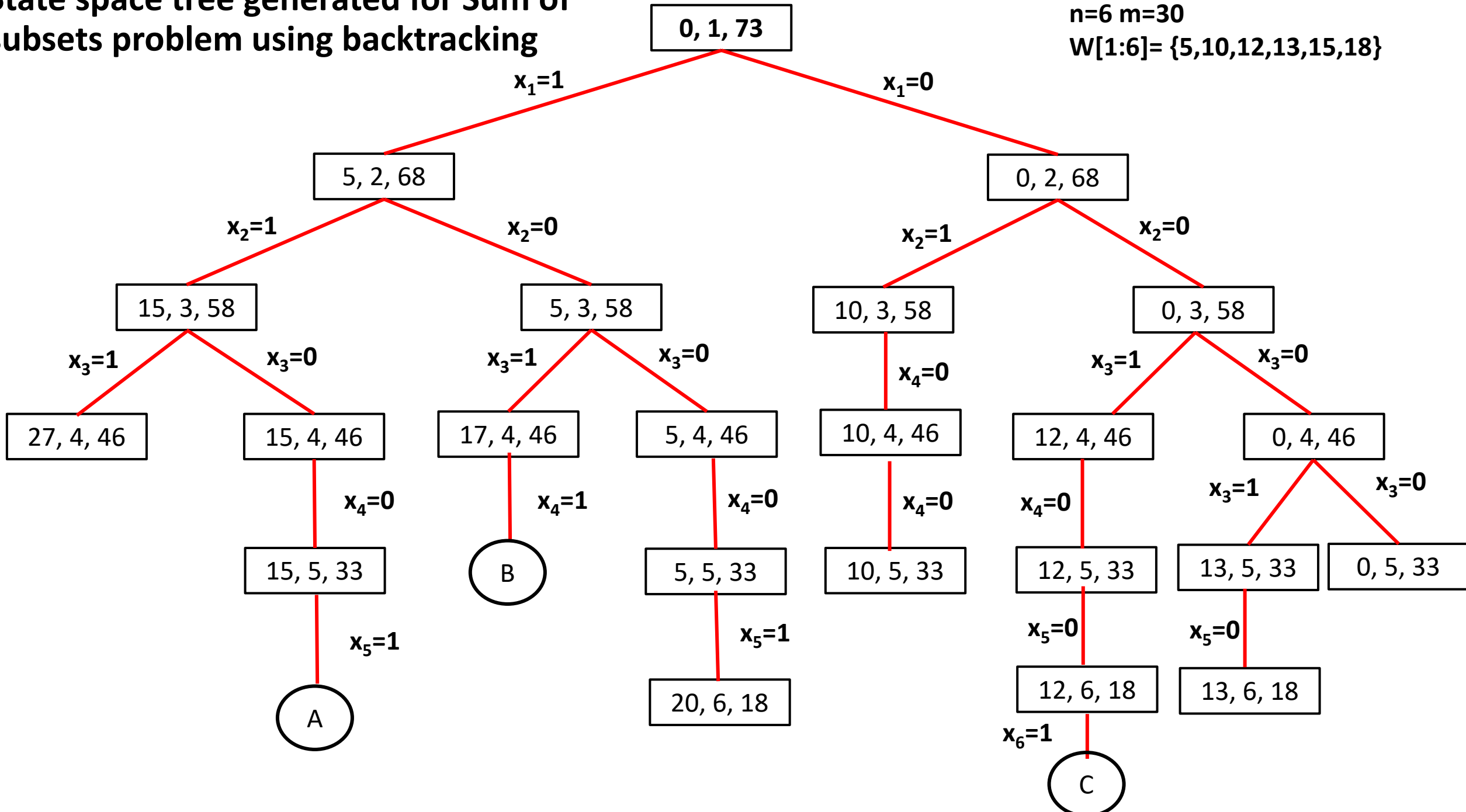
- Given n distinct positive numbers(usually called weights) , and find the all combinations of these numbers whose sum is equal to m.
- We can formulate the solution using fixed and variable sized tuple
- The explicit constraint is if x_i is included means weight w_i is included in the answer tuple ,so x_i=0 or 1
- The bounding function

$B_k(x_1,....x_k)$ = true iff $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$ and

$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$

State space tree generated for Sum of subsets problem using backtracking

n=6 m=30
W[1:6]= {5,10,12,13,15,18}



```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r - w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r - w[k]);
18      }
19  }
```

```
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  }
```

```
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  }
```

Branch-and-Bound

- FIFO BB : A BFS-like state space search is known as FIFO search (first-in-first-out) .
- LIFO BB: A D-search-like state space search will be called as LIFO search(last-in-last-out) search.
- In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rigid.
- In LC-Search ,the selection of next node E-node is based on cost function C(x).
- Both FIFO and LIFO are special cases of LC search method.
- An LC-search coupled with bounded functions is known as LC branch-and-bound search

```
1   Algorithm LCSearch(t)
2   // Search t for an answer node.
3   {
4       if *t is an answer node then output *t and return;
5       E := t; // E-node.
6       Initialize the list of live nodes to be empty;
7       repeat
8       {
9           for each child x of E do
10          {
11              if x is an answer node then output the path
12                  from x to t and return;
13              Add(x); // x is a new live node.
14              (x → parent) := E; // Pointer for path to root.
15          }
16          if there are no more live nodes then
17          {
18              write ("No answer node"); return;
19          }
20          E := Least();
21      } until (false);
22  }
```

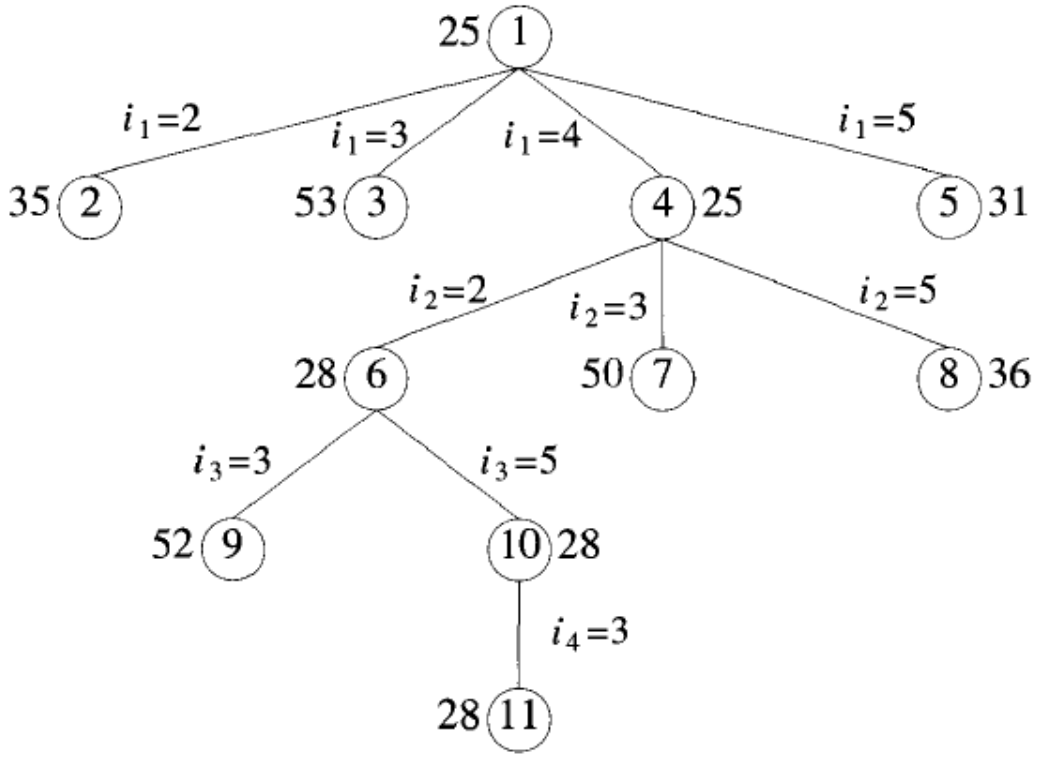
If the **list of live nodes** is implemented **as a queue with Least( ) and Add(x)** being algorithms to delete an element from and add an element to the queue , then LC Search will be transformed to a FIFO search

If the **list of live nodes** is implemented **as a stack with Least( ) and Add(x)** being algorithms to delete an element from and add an element to the queue , then LC Search will be transformed to a LIFO search

State space tree for TSP using LC-BB

Cost matrix

C	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	4	2
4	19	6	18	∞	3
5	16	4	7	16	∞



Optimal tour

1 -> 4 -> 2 -> 5 -> 3 -> 1

Process for generation of minimum cost

1. Row reduction – subtract minimum value of each row from that row
2. Column reduction – subtract minimum value of each column from that column

C	1	2	3	4	5	min
1	∞	20	30	10	11	**10**
2	15	∞	16	4	2	**2**
3	3	5	∞	4	2	**2**
4	19	6	18	∞	3	**3**
5	16	4	7	16	∞	**4**
						21

After row reduction

C	1	2	3	4	5	
1	∞	10	20	0	1	
2	13	∞	14	2	0	
3	1	3	∞	2	0	
4	16	3	15	∞	0	
5	12	0	3	12	∞	
min	**1**	**0**	**3**	**0**	**0**	**4**

After column reduction

C^1	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	2	0
4	15	3	12	∞	0
5	11	0	0	12	∞

Minimum cost = 25

i.e., minimum cost of tour is 25

C^1

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	2	0
4	15	3	12	∞	0
5	11	0	0	12	∞

$i_1=2$ $i_1=3$ $i_1=4$ $i_1=5$

C^2

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	2	0
3	0	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞

C^3

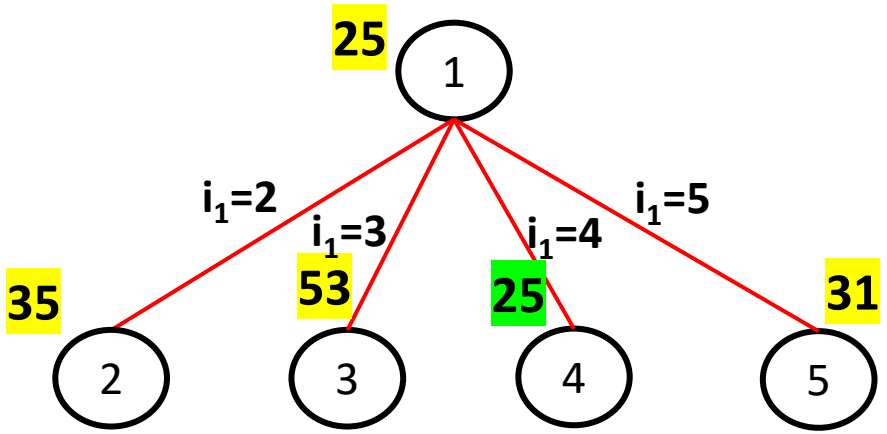
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	1	∞	∞	2	0
3	∞	3	∞	0	2
4	4	3	∞	∞	0
5	0	0	∞	12	∞

C^4

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

C^5

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞

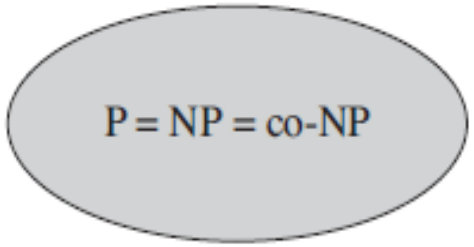


Complexity classes - P, NP, and NP-Complete

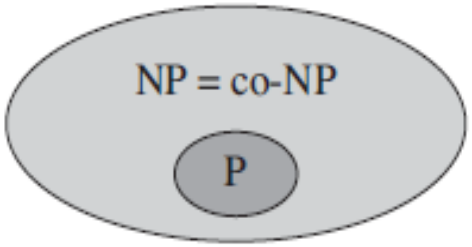
Class of Problem	Description
P class	A problem **that can be solved in polynomial-time algorithm** in worst case $O(n^k)$ for some constant k. Example : Searching, sorting, graph searching etc.
NP class	The problems ***that are verifiable in polynomial time***. Verifiable means if someone provided a solution , then we could verify whether the solution is correct in polynomial time.
NP-Complete	A Problem ***p* is said to be in NP-Complete** if • *p* belongs to NP (i.e., $p \in NP$) • *p* is as hard as any problem in NP (i.e., any problem in NP-Complete is polynomial time reducible to *p*) Example : 3-satisifiability, TSP, Hamiltonian cycle, vertex cover, clique etc.
NP-hard	Any Problem in NP-Complete can be polynomial time reducible to *p* and the p doesn't belongs to NP, then *p* is said to be NP-hard. (A Problem that satisfies ***only property 2*** ,**doesn't satisfy property 1** in the above definition.) Example : halting problem
Every NP-Complete problem is NP-hard, but there are problem which are NP-hard that is not NP-Complete	

P means – **P**olynomial-time NP means – **N**ondeterministic **P**olynomial time

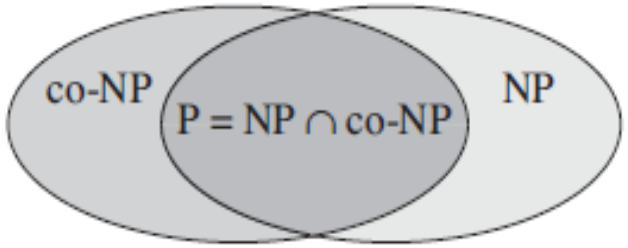
Relationship among P and NP classes from various researchers' point of view



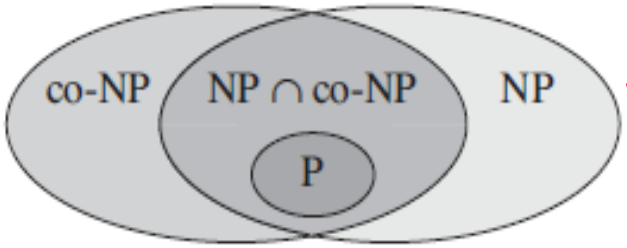
(a)



(b)



(c)



(d)

Most researchers regard this possibility as the most likely.

As researchers believe that P ≠ NP so , this introduces a new class known as **NP- Complete**

- As researchers believe that P ≠ NP so , this introduces a new class known as NP-Complete.
- Most theoretical computer scientists view the relationships among P, NP, and NPC. Both P and NPC are wholly contained with in NP, and P ∩ NPC = ∅