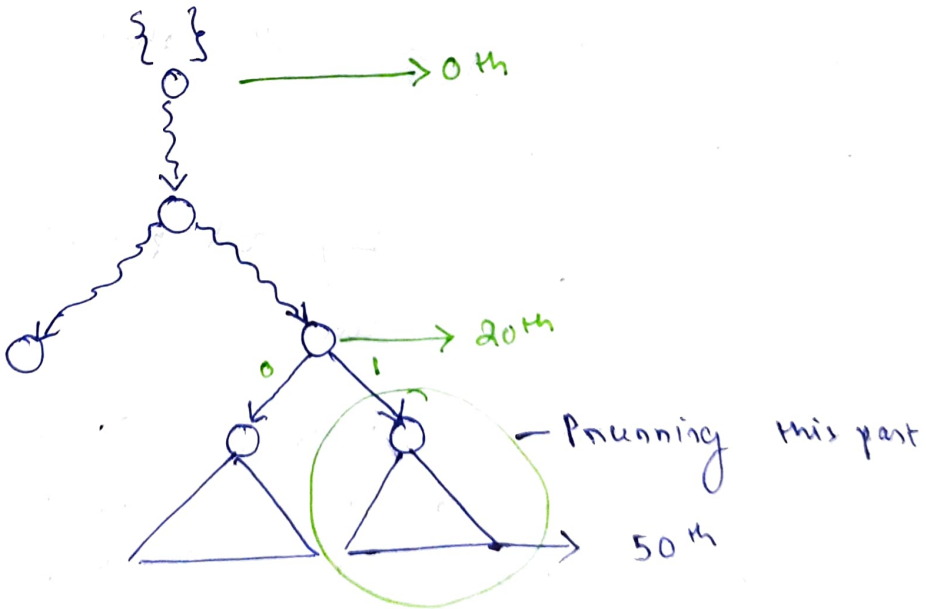
Backtracking

(i) Introduction
(ii) Control Abstraction.
(iii) General Algo for Backtracking Method
(iv) Constructing All subsets of a given set.
(v) Constructing All permutation of a given set
(vi) Constructing all paths in a graph G.
(vii) x Subset sum
(viii) x N-Queen } by using backtracking.
(ix) x Sudoku

(i) Introduction



→ Backtracking is an algo. design technique for listing all possible soln for combinatorial searching problem.

→ It uses recursive calling to find the solution by building a soln step by step increasing values with time.

→ It removes the solutions that doesnot give rise to the goal ~~of the soln~~ of the problem based on constrains given to solve problem.

→ Backtracking is similar to ~~prob~~ permutation n combinatn where we try different ~~prob~~ path/soln of the same problem.

→ We start with a possible partial soln of the prob. and move ahead, with this approach towards one of the soln that will satisfy all constrains.

→ We could find 1/all possible soln's for the problems we are solving.

→ At each step, we look for candidate if the path taken is not leading us to goal of the prob. we backtrack 1 level back and start with new candidate if that level again doesn't lead to correct soln we backtrack to 1 level further up till we reach the root, we say that soln is not available for combination with said constrains.

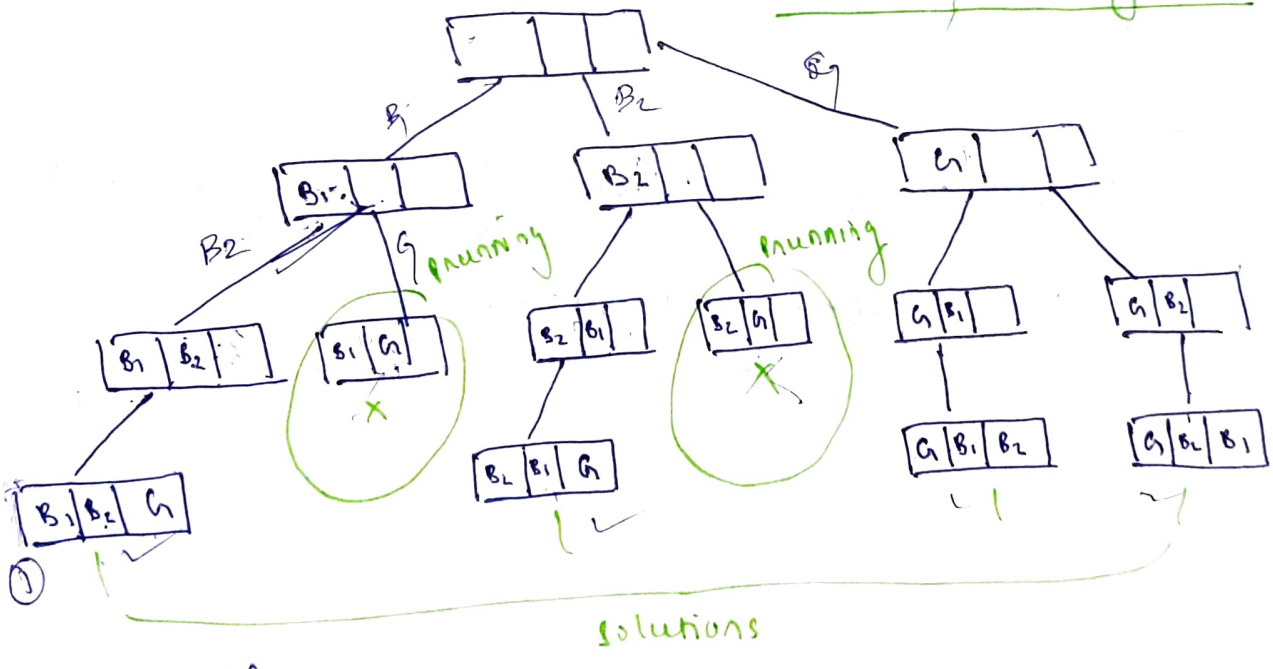
→ If we find a potential candidate which will lead us to goal of problem which will become part of partial soln and that will be used as a part of final soln.

Problem 2 (B_1, B_2, G)

We want to find all combination 2 boys, 1 girls in 3 seat

Constrains :- girls should not be in middle of 2 boys

State Space Diagram



Strategy

Step 1

Identify all the constrains (implicit, explicit) of given problem

Step 2

define an appropriate bounding func^n based on given constrains.

Step 3

Construct a state space diagram on tree by using all bounding func^n

Step 4

Compute optimal soln from the constructed state space diagram.

```
Backtracking (a, k) {
    if ( a is a possible soln of the prob.) then
        print (a);
    else {
        k ← k+1
        compute Sk
        while (Sk ≠ φ) {
            Sk ← Sk - ak ;
        }
        Backtracking(a, k);
    }
}
```

```
Backtracking ( a, k) {
    if ( is_a_solution (a, k, input))
        process_solution (a, k);
    else {
        k ← k+1
        construct_candidate (a, k, input, c, ncandidate);
        for (i←0, i < ncandidate, i++) {
            a[k] = c[i];
            # make_move (a, k, input);
            Backtrack (a, k, input);
            # unmark_move (a, k, input);
        }
        if (finished)
            return true;
    }
}
```

not needed

```
void Backtracking (int a[], int k, int input)   int d[]   {
    int c [MAX_CANDIDATE]   2
    int nCandidate 2        // no. of candidates in c[]
    int i
    if( IS_A_SOLUTION (a[], k, input)) {
            Process_Solution ( a[] , k, input);   int d[]
    }
    else {
        k++;
        Construct_Candide (a[], k, input, c, nCandidate);
        for (i=0; i<nCandidate; i++) {
            a[k] = c[i];
            Backtracking (a[], k, input)
            if (finished) {
                return;
            }
        }
    }
}
```

Construct All Subsets

How many subsets are there of an 'n' element set

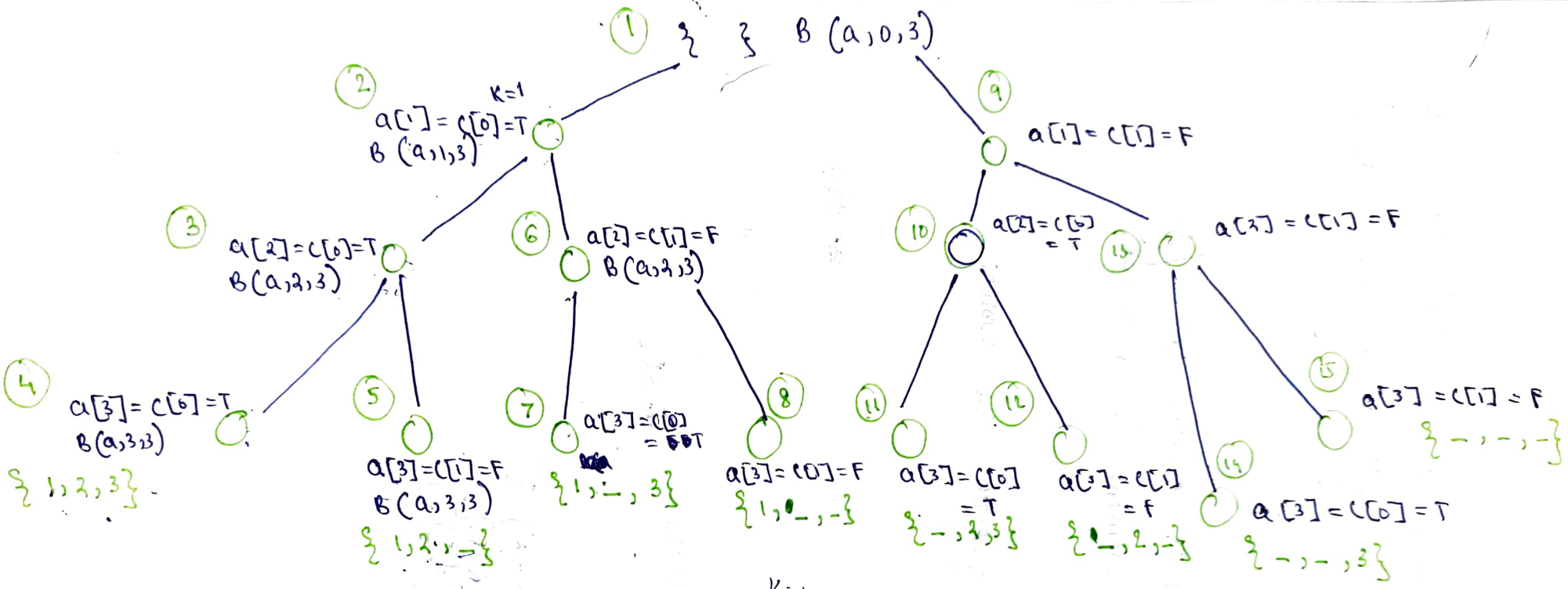
to construct all 2^n subsets of array size n / vectors of size n.

→ when the value of a[i] is either true / false

→ it signifies whether ith element is on is not in the subset.

→ what order will this generate the subsets

```
boolean Is_A_Solution (int a[], int k, int input) {
    if (k == input)
        then return true;
}
void Process_Solution (int a[], int k, int d[]) {
    int i;
    for (i = 1; i <= k; i++) {
        if (a[i] == TRUE)
            printf (d[i]);
    }
}
Construct_Candidate (int a[], int k, int input, int c[], int *nCandidate)
{
    c[0] ← TRUE
    c[1] ← FALSE
    *nCandidate ← 2
}
```


1
{ } B (a, 0, 3)
2
K=1
a[1] = C[0] = T
B (a, 1, 3)
9
a[1] = C[1] = F
3
a[2] = C[0] = T
B (a, 2, 3)
6
a[2] = C[1] = F
B (a, 2, 3)
10
a[2] = C[0] = T
13
a[2] = C[1] = F
4
a[3] = C[0] = T
B (a, 3, 3)
{ 1, 2, 3 }
5
a[3] = C[1] = F
B (a, 3, 3)
{ 1, 2, - }
7
a[3] = C[0] = BOT
{ 1, -, 3 }
8
a[3] = C[0] = F
{ 1, -, - }
11
a[3] = C[0] = T
{ -, 2, 3 }
12
a[3] = C[1] = F
{ -, 2, - }
14
a[3] = C[0] = T
{ -, -, 3 }
15
a[3] = C[1] = F
{ -, -, - }

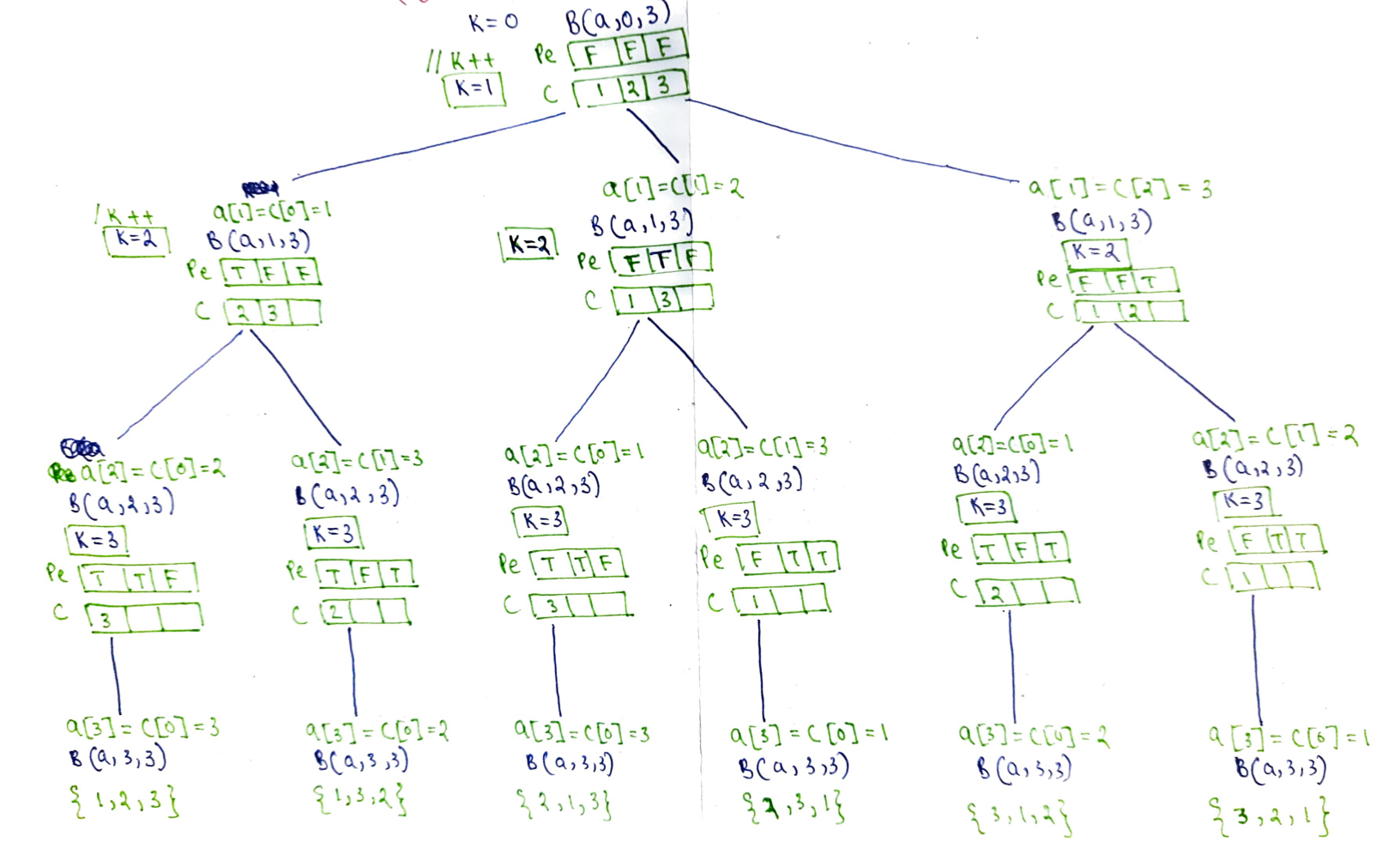
bool Is_A_Solution (int [], int k, int n)
{
    if (k == n) then return T;
}

Permutations

void process_solution (int a[], int k) {
    int i;
    for (i=1; i<=k; i++)
        printf ("%d", a[i]);
}

void construct_candidate (int a[], int k, int n, int c[], int *ncandidate)
{
    Bool Pe_sol [MAX]
    INT i
    for (i=1; i<=n; i++) {
        Pe_sol [i] ← F
    }
    *ncandidate ← 0
    for (i=1; i<=n; i++) {
        if (Pe_sol [i] == F) {
            c [*ncandidate] ← i
            *ncandidate ++
        }
    }
}

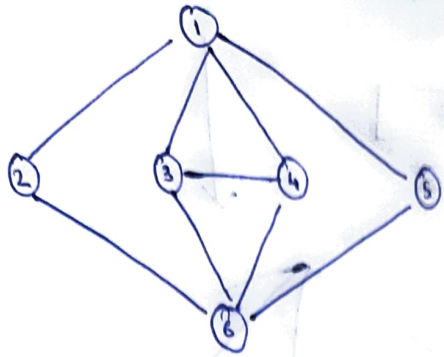
for (i=1; i<k; i++){
    Pe_Sol [a[i]] ← T
}


K=0
B(a,0,3)
Pe F F F
C 1 2 3
// K++
K=1
a[1]=C[0]=1
B(a,1,3)
Pe T F F
C 2 3
// K++
K=2
a[1]=C[1]=2
B(a,1,3)
K=2
Pe F F F
C 1 3
a[1]=C[2]=3
B(a,1,3)
K=2
Pe F F T
C 1 2
a[2]=C[0]=2
B(a,2,3)
K=3
Pe T T F
C 3
a[2]=C[1]=3
B(a,2,3)
K=3
Pe T F T
C 2
a[2]=C[0]=1
B(a,2,3)
K=3
Pe T T F
C 3
a[2]=C[1]=3
B(a,2,3)
K=3
Pe F T T
C 1
a[2]=C[0]=1
B(a,2,3)
K=3
Pe T F T
C 2
a[2]=C[1]=2
B(a,2,3)
K=3
Pe F T T
C 1
a[3]=C[0]=3
B(a,3,3)
{1,2,3}
a[3]=C[0]=2
B(a,3,3)
{1,3,2}
a[3]=C[0]=3
B(a,3,3)
{2,1,3}
a[3]=C[0]=1
B(a,3,3)
{2,3,1}
a[3]=C[0]=2
B(a,3,3)
{3,1,2}
a[3]=C[0]=1
B(a,3,3)
{3,2,1}

N-Queen Problem

Constructing All Paths

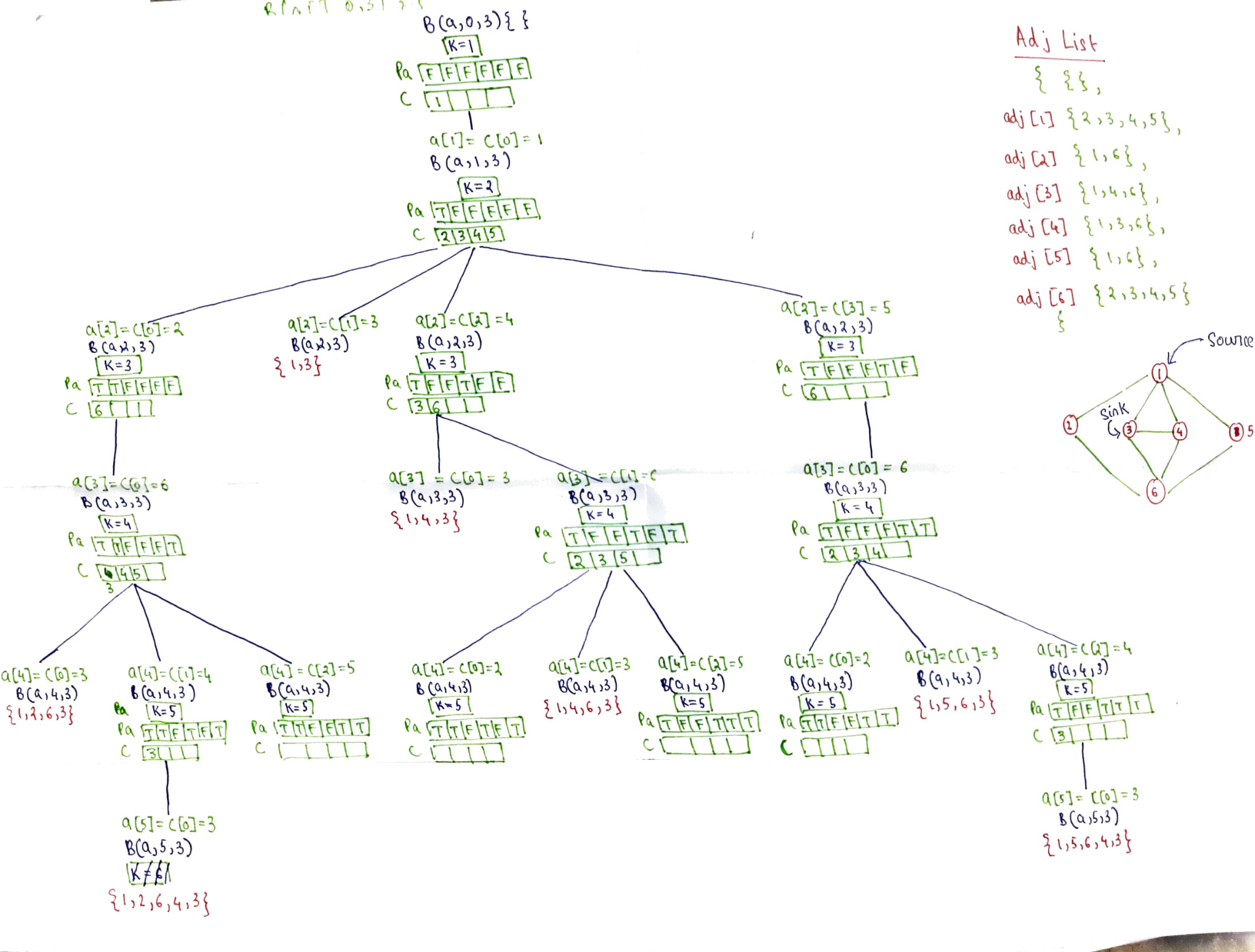
```
bool  Is_A_Solution ( int a[] , int k , int t)        destination
{
    if (a[k] == t) then
        return T
}
void  Process_Solution ( int a[] , int k )
{
    int i;
    for (i=1 ; i<= k ; i++)
        printf ("%d ",
}
void  Construct_candidate (int a[], int k , int t , int c[],
                                                    int *ncandidate)
{
    bool  Pa_Sol [MAX]
    for (int i=1 ;  i<=MAX ; i++)
        Pa_Sol [i] <- F
    for ( i = 1 ;  i< k  ; i++)
        Pa_sol [a[i]] <- T
    if ( k == 1 ) {
        c[0] <- 1
        * nCandidate <- 1
    }
    else {
        * nCandidate = 0
        for (i<-0 ; i < adjList [a[k-1]].size() ; i++ ) {
            if (! Pa_Sol [AdjList [a[k-1]][i]] ) {
                c[* nCandidate] <- AdjList [a[k-1]][i];
                * nCandidate ++ ;
            }
        }
    }
}
```



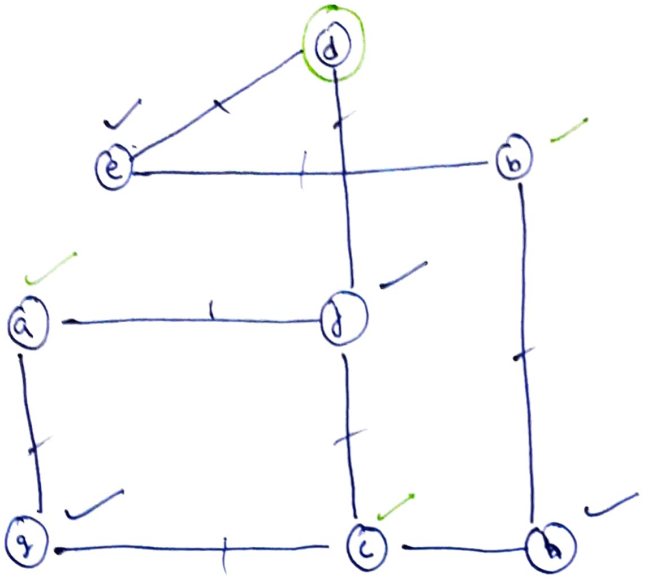
1 → source
3 → destination

AdjList

```
{{};
 a[1] {2,3,4,5};
 a[2] {1,6},
 a[3] {1,4,6},
 a[4] {1,3,6},
 a[5] {1,6},
 a[6] {2,3,4,5}
}
```


B(a,0,3){ }
K=1
Pa F F F F F F
C 1
a[1]= C[0]= 1
B(a,1,3)
K=2
Pa T F F F F F
C 2 3 4 5
a[2]=C[0]=2
B(a,2,3)
K=3
Pa T T F F F F
C 6
a[3]=C[0]=6
B(a,3,3)
K=4
Pa T T F F F T
C 4 5
3
a[4]=C[0]=3
B(a,4,3)
{1,2,6,3}
a[4]=C[1]=4
B(a,4,3)
K=5
Pa T T F T F T
C 3
a[5]=C[0]=3
B(a,5,3)
K≠6
{1,2,6,4,3}
a[4]=C[2]=5
B(a,4,3)
K=5
Pa T T F F T T
C
a[3]=C[1]=4
B(a,3,3)
K=4
Pa T F F T F T
C 2 3 5
a[4]=C[0]=2
B(a,4,3)
K=5
Pa T T F T F T
C
a[4]=C[1]=3
B(a,4,3)
{1,4,6,3}
a[4]=C[2]=5
B(a,4,3)
K=5
Pa T F F T T T
C
a[3]=C[0]=3
B(a,3,3)
{1,4,3}
a[2]=C[1]=3
B(a,2,3)
{1,3}
a[2]=C[2]=4
B(a,2,3)
K=3
Pa T F F T F F
C 3 6
a[3]=C[0]=3
B(a,3,3)
{1,4,3}
a[3]=C[1]=4
B(a,3,3)
K=4
Pa T F F T F T
C 2 3 5
a[4]=C[0]=2
B(a,4,3)
K=5
Pa T T F T F T
C
a[4]=C[1]=3
B(a,4,3)
{1,4,6,3}
a[4]=C[2]=5
B(a,4,3)
K=5
Pa T F F T T T
C
a[2]=C[3]=5
B(a,2,3)
K=3
Pa T F F F T F
C 6
a[3]=C[0]=6
B(a,3,3)
K=4
Pa T F F F T T
C 2 3 4
a[4]=C[0]=2
B(a,4,3)
K=5
Pa T T F F T T
C
a[4]=C[1]=3
B(a,4,3)
{1,5,6,3}
a[4]=C[2]=4
B(a,4,3)
K=5
Pa T F F T T T
C 3
a[5]=C[0]=3
B(a,5,3)
{1,5,6,4,3}
Adj List
{ {},
adj[1] {2,3,4,5},
adj[2] {1,6},
adj[3] {1,4,6},
adj[4] {1,3,6},
adj[5] {1,6},
adj[6] {2,3,4,5}
{
Source
sink
1
2
3
4
5
6

Q1

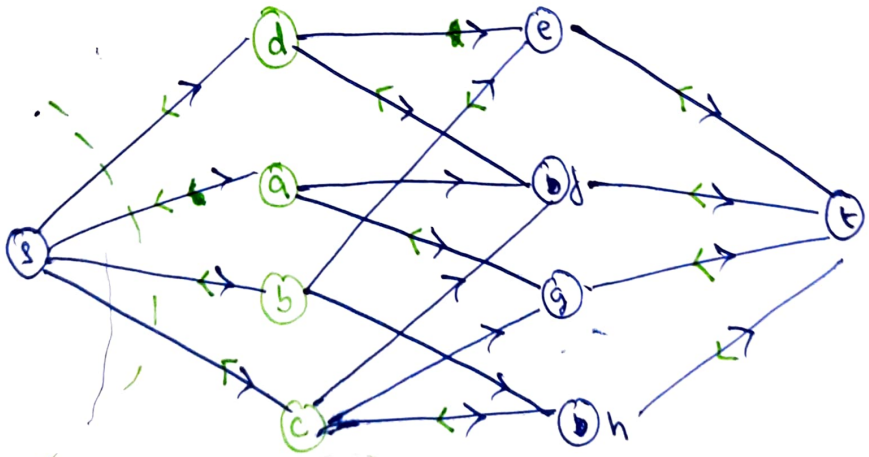


(i) Is graph bipartite

(ii) If bipartite find max matching

(iii) Find the cut where max flow = min cut.

(iv) Is the matching a perfect matching



P_1 : s → b → e → t $c_f = 1$

P_2 : s → c → h → t $c_f = 1$

P_3 : s → a → g → t $c_f = 1$

P_4 : s → d → f → t $c_f = 1$

max flow = 4

Intractable Problem and Approximation Algo. and P SPACE

Algorithm

(worst case)

Efficient Algo can be represented

$P(n)$

P is called tractable problem

→ If it can be solved in polynomial time

→ called easiest problem

→ Its solvable by efficient algo

P is called Intractable problem

→ If it can solved is reasonable time for small input time

→ Or it cannot be solved at all

→ It can be classified into different categories based on there degree of difficulty

Categories :-

(i) NP

(ii) NP complete

(iii) NP hard

Intractibility

Problem Reduct°

Approximation Algo
(i) AA for vertex cover problem
(ii) AA for TSP problem
(iii) AA for Set cover problem.

Example for simple Reduction

(i) Max Bipartite Matching → FN
(ii) closest pair → sorting
(iii) LIS → Max. Edit distance
(iv) GCD → LCM
(v) ~~X~~ Sorting → Conver Hull

Example for elementary Harder Reduction

(i) SAT → clique
(ii) Clique → VC
(iii) VC → IS
(iv) HC → TSP
(v) IS → clique
(vi) 3 SAT - IS

VC - vertex cover
IS - Independent Set
SAT - Satisfiability
HC - Hamiltonian cycle
TSP - Travelling Salesman Problem.

~~STA~~ SAT (Mother of NP complete problem)
(Satisfiability)

circuit Satisfiability

Formula Satisfiability

3 - CNF (3 - SAT)

CNF ⇒ Conjuctive Normal Form

Problem

- Abstract Problem
- Decision Problem
- Optimization problem

(i) Subset Sum Problem

(i) TSP

→ covers all vertex and $s = t$
→ no vertex repeat

→ covers all edge and $s = t$
→ not edge repeat

→ given a undirected

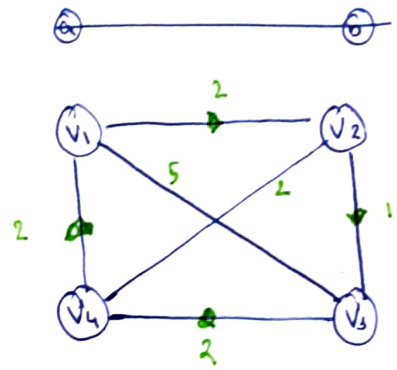
TSP - optimization prob no. of paths = $n!$ → $O(n!)$

↓↓

TSP - Decision Problem

Given a complete weighted undirected graph $G(V, E)$ and target T, determine if the graph has a simple cycle that covers all vertices of G and has a cost $\leq T$

TSP Decision Problem $\in$ NP



$T = 7$

Candidate input

$\langle V_1, V_2, V_4, V_5, V_1 \rangle$

$\langle V_1, V_2, V_3, V_4, V_1 \rangle$

(i) ~~NO.~~ all vertices belongs to G or not

(ii) 1st and last vertex are same or not.

(iii)

(iv)

$\Rightarrow \in NP$

Subset Sum Problem $\in$ NP

$S = \{2, 3, 4\}$

$T = 5$

(i) The nos ~~are the~~ $\in$ to S

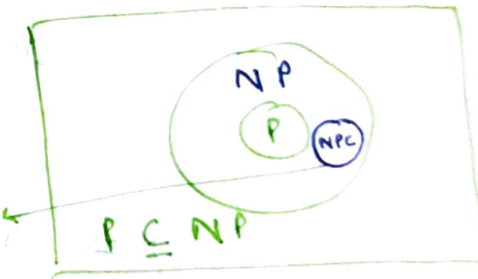
(ii) The sum = T

$\Rightarrow \in NP$

Class P

A problem ~~denoted by~~ x (x) is in class P if that can be solvable in polynomial time or if there is a deterministic algo to solve in polynomial time

ex: all sorting, searching algo.



Class NP

A problem x is in class of NP if for which the candidate soln for this problem can be verified in polynomial time

ex: all sorting, search.

- → TSP
- → subset sum
- → vertex cover
- → set cover
- → Independt set

NP Complete class

A problem x is in NP-C class if all problems in NP are polynomially reducible to x

Reducibility

Y is reducible to X if we can transform any instance of Y into an instance of X

Notation

$Y \leq X$ - Big

(Y is ~~polynomially~~ reducible ~~to~~ X)

(Y is reducible to X)

small - $Y \leq_p X$ - Big

(Y is polynomially reducible to X)

↓

Any instance of Y into instance of X polynomially (in polynomial time)

Y = BMM (Bipartite Max Matching)

X = FN (Flow network)

$BMM \leq_p FN$

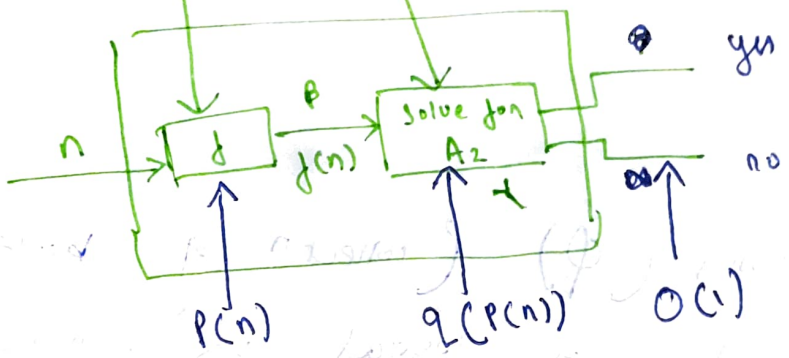
$Y \leq_p X$

Property of Reducibility

1) If $X \leq_p Y$ and Y has a polynomial time algo
$\Rightarrow$ X has a polynomial time algo

2) If $X \leq_p Y$ and X has no polynomial time algo
$\Rightarrow$ Y has no polynomial time algo

Proof



$T(n) = P(n) + q(P(n)) + O(1)$ (because Y is harder)
$= q(P(n))$

NP - Complete

A problem X is in NP-C if

(i) $X \in NP$ → verify

(ii) $\forall X' \in NP$, where $X' \leq_p X$
(X' ↓ small, X ↓ Big)

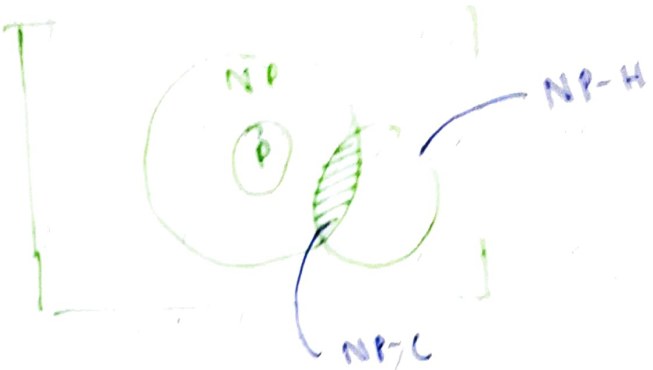
To prove

(i) Choose an appropriate problem from NP-complete class (L')

(ii) $L' \leq_p X$

$\Rightarrow$ X is NP-C problem.

NP - Hard



(i) $x \notin NP$

(ii) $\forall\ x' \in NPC,\ x' \leq_p x$

SAT (Satisfiability Problem)

A boolean expression (ϕ) (collection of boolean variable, connected with logical OR, AND) is a satisfiable.

i.e. assign the value (True or false) for the boolean variable x_i to which the boolean expression ϕ returns true

$\phi = x_1 \vee x_2 \vee \neg x_2 \wedge x_4$

OR, AND

boolean variable