Q1)
Input: N = No. of Nodes; A = Adjacency Matrix;
Output: I : independent set;
Begin
Initialize the parameters of Simulated Annealing (SA) \( \Rightarrow \) (Tmax, Tmin,L,alpha);
Generate random solution \( \Rightarrow \) S;
Evaluate the fitness (which is depend on the adjacency matrix values);
T=Tmax;
While (T>Tmin) do
Begin
    For
    Generate random neighbor of S \( \Rightarrow \) S';
    E=f(S) – f(S');
    If E<0 then
        S=S';
    Else
        Accept S' with prob(exp(-E/T));
    End if;
    Update T depends on alpha;
End while;
End algorithm.

F(X): Fitness Function
Begin
    Check X with A matrix;
    If no adjacency nodes then it is true
    Else it is false;
End fitness;

Q2)

<table>
<thead>
<tr>
<th>Item1</th>
<th>Freq1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>Item2</td>
<td>Freq2</td>
</tr>
<tr>
<td>AB</td>
<td>4</td>
</tr>
<tr>
<td>AC</td>
<td>3</td>
</tr>
<tr>
<td>AD</td>
<td>4</td>
</tr>
<tr>
<td>BC</td>
<td>5</td>
</tr>
<tr>
<td>BD</td>
<td>4</td>
</tr>
<tr>
<td>CD</td>
<td>4</td>
</tr>
<tr>
<td>Item3</td>
<td>Freq3</td>
</tr>
<tr>
<td>ABC</td>
<td>3</td>
</tr>
<tr>
<td>ABD</td>
<td>4</td>
</tr>
<tr>
<td>BCD</td>
<td>3</td>
</tr>
<tr>
<td>Item4</td>
<td>Freq4</td>
</tr>
<tr>
<td>ABCD</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>A→B</td>
<td>100%</td>
</tr>
<tr>
<td>B→A</td>
<td>66%</td>
</tr>
<tr>
<td>A→C</td>
<td>75%</td>
</tr>
<tr>
<td>C→A</td>
<td>50%</td>
</tr>
<tr>
<td>A→D</td>
<td>100%</td>
</tr>
<tr>
<td>D→A</td>
<td>80%</td>
</tr>
<tr>
<td>B→C</td>
<td>80%</td>
</tr>
<tr>
<td>C→B</td>
<td>83%</td>
</tr>
<tr>
<td>B→D</td>
<td>66%</td>
</tr>
<tr>
<td>D→B</td>
<td>80%</td>
</tr>
<tr>
<td>C→D</td>
<td>66%</td>
</tr>
<tr>
<td>D→C</td>
<td>80%</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A→B</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>B→A</td>
<td>66%</td>
<td></td>
</tr>
<tr>
<td>A→C</td>
<td>75%</td>
<td></td>
</tr>
<tr>
<td>C→A</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>A→D</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>D→A</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>B→C</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>C→B</td>
<td>83%</td>
<td></td>
</tr>
<tr>
<td>B→D</td>
<td>66%</td>
<td></td>
</tr>
<tr>
<td>D→B</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>C→D</td>
<td>66%</td>
<td></td>
</tr>
<tr>
<td>D→C</td>
<td>80%</td>
<td></td>
</tr>
</tbody>
</table>

Q3)
Input: n=scout bee; m=selected site; e; nep;
Output: Best solution;
Begin
Initialize population (n) randomly;
Evaluate fitness of n solutions;
While not(stopping criteria) do
Begin
Select m sites for neighbors;
Recruit bees for selected sites (nsp);
Evaluate fitness;
Select the fitness bee form each patch;
Assign remaining bees (n-m) to search randomly;
Evaluate fitness;
End while;
End algorithm.
$m = 2$

\[
\frac{f(x)}{2} = \frac{2}{2} \Rightarrow \text{select } BB XY \Rightarrow \text{generate neighbors} \Rightarrow \text{RB XY} \quad 0
\]

\[
RB \quad 0
\]

\[
B \quad 0
\]

\[
RB Xy \quad 0
\]

\[
RB Xy \quad 0
\]
Q4) A)

Begin
Initialize S to the first positive training instance:
N is the set of all negative instances seen so far:
For each positive instance P
    Begin
    For every s ∈ S if s does not match P replace s
    with most specific generalizations that match P:
    Delete from S all hypotheses more general than
    some other hypotheses in S:
    Delete from S all hypotheses that match
    a previously observed negative instance in N:
    End:
For every negative instance n
    Begin
    Delete all members of S that match n:
        Add n to check future hypotheses for
        overgeneralization:
    End:
End

\[
\begin{array}{c}
\text{large blue cube} \\
\text{small blue ball} \\
\text{small blue cube} \\
\text{large blue ball} \\
\text{small blue cube} \\
\text{large blue ball}
\end{array}
\]
B)
STACK(X, Y)
P: CLEAR(Y) ^ HOLDING(X) ^ Y="Triangular".
D: CLEAR(Y) ^ HOLDING(X)
A: ARMEMPTY ^ ON(X, Y)

UNSTACK(X, Y)
P: ON(X, Y) ^ CLEAR(X) ^ ARMEMPTY
D: ON(X, Y) ^ ARMEMPTY
A: HOLDING(X) ^ CLEAR(Y)

PICKUP(X)
P: CLEAR(X) ^ O N T A B L E(X) ^ ARMEMPTY
D: O N T A B L E(X) ^ ARMEMPTY
A: HOLDING(X)

PUTDOWN(X)
P: HOLDING(X)
D: HOLDING(X)
A: O N T A B L E(X) ^ ARMEMPTY

Path : unstack(C,A), putdown(C), pickup(B), stack(B,A), pickup(C), stack(C,B)

Q5)
/* Initial phase */
Initialize the population Pop using a diversification generation method ;
Apply the improvement method to the population ;
Reference set Update Method ;
/* Scatter search iteration */
Repeat
Subset generation method ;
Repeat
Solution Combination Method ;
Generate Neighbors GN and use these GN as an improved stage in Scatter Search technique;
Check these neighbors if there are updated solutions in scatter search ;
If true then accepted these solutions
 Else generate other neighbors ;
Until Stopping criteria I
Reference Set Update Method ;
Until Stopping criteria
Output: Best found solution or set of solutions.
Q6)

1- GRASP

Input: Number of iterations.

Repeat

- $s = \text{Random-Greedy(seed)}$ /* apply a randomized greedy heuristic */
- $s' = \text{Local - Search}(s)$ /* apply a local search algorithm to the solution */

Until Stopping criteria /* e.g. a given number of iterations */

Output: Best solution found.

$s = \emptyset$ /* Initial solution (null) */

Evaluate the incremental costs of all candidate elements:

Repeat

Build the restricted candidate list $RCL$:

/* select a random element from the list $RCL$ */

$e_i = \text{Random-Selection}(RCL)$:

If $s \cup e_i \notin F$ Then /* Test the feasibility of the solution */

$s = s \cup e_i$:

Reevaluate the incremental costs of candidate elements:

Until Complete solution found.

- Single Population.
- Good diversity.
- Need memory.
- Keep of previous solutions.
- Need local search.

2- VNS

Input: a set of neighborhood structures $N_i$ for $k = 1, \ldots, k_{\text{max}}$ for shaking.

- a set of neighborhood structures $N_i$ for $k = 1, \ldots, l_{\text{max}}$ for local search.

$x = x_0$ /* Generate the initial solution */

Repeat

For $k=1$ To $k_{\text{max}}$ Do

Shaking: pick a random solution $x'$ from the $k^{th}$ neighborhood $N_k(x)$ of $x$:

Local search by VND:

For $l=1$ To $l_{\text{max}}$ Do

- Find the best neighbor $x''$ of $x'$ in $N_l(x')$:
- If $f(x'' < f(x'))$ Then $x' = x''$; $l=1$:
- Otherwise $l=l+1$:

Move or not:

If local optimum is better than $x$ Then

$x = x'$:

Continue to search with $N_l(k = 1)$:

Otherwise $k=k+1$:

Until Stopping criteria

Output: Best found solution.
- Single Population.
- Depend on neighbors in high degree.
- Low memory.
- Good diversity.
- Depend on shaking method.

3- Scatter Search
Initialize the population Pop using a diversification generation method;
Apply the improvement method to the population;
Reference set Update Method;
/* Scatter search iteration */
Repeat
    Subset generation method;
    Repeat
        Solution Combination Method;
        Improvement Method;
        Until Stopping criteria 1
        Reference Set Update Method;
    Until Stopping criteria
Output: Best found solution or set of solutions.

- Multi Population.
- Good diversity.
- Keep of some previous solutions.
- Depend on improvement strategy.

4- Tabu Search
Initialize
Identify initial Solution, Create empty TabuList, Set BestSolution=Solution
Define TerminationConditions
Done=FALSE
Repeat
if value of Solution > value of BestSolution then BestSolution=Solution
if no TerminationConditions have been met then
begin
    add Solution to TabuList
    if TabuList is full then delete oldest entry from TabuList
    find NewSolution by some transformation on Solution
    if no NewSolution was found or
    if no improved NewSolution was found for a long time then
    generate NewSolution at random
    if NewSolution not on TabuList then
    Solution = NewSolution
end
else
Done=TRUE
until done=TRUE

- Single Population.
- Accepted diversity.
- Keep of all previous solutions.
- Depend on size of list to stop or continues.

5- Iterated Local Search

\[ s_n = \text{local search}(s_0) : /\text{ Apply a given local search algorithm }/ \]

Repeat
\[ s' = \text{Perturb}(s_n, \text{search history}) : /\text{ Perturb the obtained local optima }/ \]
\[ s'_{\text{new}} = \text{Local search}(s') : /\text{ Apply local search on the perturbed solution }/ \]
\[ s_n = \text{Accept}(s_n, s'_{\text{new}}, \text{search memory}) : /\text{ Accepting criteria }/ \]
Until Stopping criteria
Output: Best solution found.

- Single Population.
- Accepted diversity.
- Depend on type of local search & perturbation strategy.
- Need low memory.