Tab 1

**SET – A:**
**Q1. Write a program to implement simple facts and queries using logic.**

```
likes(alice,reading).
likes(alice,music).
likes(bob,football).
likes(charlie,painting).
likes(david,music).
likes(ella,reading).

lives_in(alice,delhi).
lives_in(bob,mumbai).
lives_in(charlie,chennai).
lives_in(david,delhi).
lives_in(ella,kolkata).

age(alice,25).
age(bob,30).
age(charlie,22).
age(david,35).
age(ella,25).

same_hobby(X,Y):-
likes(X,H),
likes(Y,H),
X\=Y.

same_city(X,Y):-
lives_in(X,C),
lives_in(Y,C),
X\=Y.

same_age(X,Y):-
age(X,A),
age(Y,A),
X\=Y.


same_hobby(X,Y).
same_city(X,Y).
same_age(X,Y).
same_hobby(X,Y),same_city(X,Y).
```

**Q2. Solve a Sudoku puzzle using a constraint satisfaction problem (CSP) approach.**

```
:-use_module(library(clpfd)).

sudoku(Rows):-
Template=[_,_,_,_,_,_,_,_,_],
length(Rows,9),
maplist(same_length(Template),Rows),
append(Rows,Vs),Vs ins 1..9,
maplist(all_distinct,Rows),
transpose(Rows,Columns),maplist(all_distinct,Columns),
Rows=[A,B,C,D,E,F,G,H,I],
blocks(A,B,C),blocks(D,E,F),blocks(G,H,I),
maplist(label,Rows),
maplist(portray_row,Rows).
blocks([],[],[]).

blocks([A,B,C|Bs1],[D,E,F|Bs2],[G,H,I|Bs3]):-
all_distinct([A,B,C,D,E,F,G,H,I]),
blocks(Bs1,Bs2,Bs3).

portray_row(Row):-
write(Row),nl.


sudoku([
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_]
]).
```

**SET – B:**
**Q1. Implement uninformed search algorithms (BFS, DFS) to solve a maze.**

```
edge(a,b).
edge(a,c).
edge(b,d).
edge(b,g).
edge(c,e).
edge(d,f).
edge(e,f).
edge(f,g).

neighbor(X,Y):-edge(X,Y).
neighbor(X,Y):-edge(Y,X).

dfs(Start,Goal,Path):-
dfs_helper(Start,Goal,[Start],Rev), reverse(Rev,Path).
dfs_helper(Goal,Goal,Visited,Visited).

dfs_helper(Current,Goal,Visited,Path):-
neighbor(Current,Next), \+ member(Next,Visited), dfs_helper(Next,Goal,[Next|Visited],Path).

bfs(Start,Goal,Path):-
bfs_helper([[Start]],Goal,Rev), reverse(Rev,Path).
bfs_helper([[Goal|T]|_],Goal,[Goal|T]).

bfs_helper([[Current|T]|Others],Goal,Path):-
findall([Next,Current|T], (neighbor(Current,Next), \+ member(Next,[Current|T])), NewPaths),
append(Others,NewPaths,UpdatedQueue), bfs_helper(UpdatedQueue,Goal,Path).


dfs(a,g,Path).
bfs(a,g,Path).
```

**Q2. Evaluate and compare their efficiency in finding the optimal path from the above two algorithms**

- **BFS** finds the optimal (fewest-edges) path in an unweighted graph.
- **DFS** does **not** guarantee an optimal path; it may return a longer path depending on neighbor order.

### Correctness & optimality

- **BFS** — **Optimal** for unweighted graphs: it explores by increasing path length, so the first time it reaches the goal it has used the minimum number of edges.
- **DFS** — **Not optimal**: it follows one branch deeply and may encounter a longer path before the shortest one.

**Applied to your graph:**

- BFS produced `[a,b,g]` (2 edges) — optimal.
- DFS produced `[a,b,d,f,g]` (4 edges) — not optimal.

### Completeness

- **BFS:** Complete for finite graphs (will find a solution if one exists).
- **DFS:** Not necessarily complete in infinite/deep graphs unless you track visited nodes and/or limit depth.

### Time & space complexity (using branching factor `b`, shallowest goal depth `d`, maximum depth `m`)

- **BFS**
  - Time: `O(b^d)` — explores all nodes up to depth `d`.
  - Space: `O(b^d)` — must store the entire frontier (wide memory use).
- **DFS**
  - Time: worst-case `O(b^m)` — can explore much deeper parts of the tree.
  - Space: `O(b*m)` (stack depth) — memory efficient compared to BFS.

### Practical tradeoffs

- Use **BFS** when:
  - You need the **shortest** path in an unweighted graph.
  - Memory cost `O(b^d)` is acceptable.
- Use **DFS** when:
  - Memory is very limited and any path suffices.
  - You expect solutions to be very deep and branching small, or you can apply heuristics / pruning.

**SET – C:**

**Q1. Develop an AI program to solve the 8 Puzzle problem using heuristic search algorithms (A or Greedy Best-First) and analyze and optimize the solution.**

```
goal([1,2,3,4,5,6,7,8,0]).

move(S,N,up):-
nth0(Z,S,0),
Z>2,
NP is Z-3,
swap(S,Z,NP,N).

move(S,N,down):-
nth0(Z,S,0),
Z<6,
NP is Z+3,
swap(S,Z,NP,N).

move(S,N,left):-
nth0(Z,S,0),
Z mod 3=\=0,
NP is Z-1,
swap(S,Z,NP,N).

move(S,N,right):-
nth0(Z,S,0),
Z mod 3=\=2,
NP is Z+1,
swap(S,Z,NP,N).

swap(S,I,J,R):-
nth0(I,S,A),
nth0(J,S,B),
set(S,I,B,T),
set(T,J,A,R).

set([_|T],0,X,[X|T]).

set([H|T],N,X,[H|R]):-
N>0,
N1 is N-1,
set(T,N1,X,R).
```

```
bfs(Start,Sol):-bfs_q([[Start]],[],Sol).

bfs_q([[S|P]|_],_,Moves):-
goal(S),
reverse(P,M),
Moves=M.

bfs_q([[S|P]|Rest],V,Sol):-
findall([N,M|P],
(move(S,N,M),\+member(N,V)),NP),
append(Rest,NP,Q),
bfs_q(Q,[S|V],Sol).


bfs([1,2,3,4,5,6,7,0,8], M).
```

**Q2. Write a program to implement simple facts and queries using logic.**

**SET – D:**
**Q1. Solve a Sudoku puzzle using a constraint satisfaction problem (CSP) approach.**

```
:-use_module(library(clpfd)).

sudoku(Rows):-
Template=[_,_,_,_,_,_,_,_,_],
length(Rows,9),
maplist(same_length(Template),Rows),
append(Rows,Vs),Vs ins 1..9,
maplist(all_distinct,Rows),
transpose(Rows,Columns),maplist(all_distinct,Columns),
Rows=[A,B,C,D,E,F,G,H,I],
blocks(A,B,C),blocks(D,E,F),blocks(G,H,I),
maplist(label,Rows),
maplist(portray_row,Rows).
blocks([],[],[]).

blocks([A,B,C|Bs1],[D,E,F|Bs2],[G,H,I|Bs3]):-
all_distinct([A,B,C,D,E,F,G,H,I]),
blocks(Bs1,Bs2,Bs3).

portray_row(Row):-
write(Row),nl.


sudoku([
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_],
[_,_,_,_,_,_,_,_,_]
]).
```

**Q2. Write a program to solve the Towers of Hanoi problem using AI.**

```
move(1,S,D,_):-
write('Move disk from '),
write(S),
write(' to '),
write(D),
nl.

move(N,S,D,A):-
N>1,
M is N-1,
move(M,S,A,D),
move(1,S,D,_),
move(M,A,D,S).


move(3,left,right,middle).
```

**SET – E:**
**Q1. Create an AI player for Tic-Tac-Toe using the Minimax algorithm.**

```
win([A,A,A,_,_,_,_,_,_],A).
win([_,_,_,A,A,A,_,_,_],A).
win([_,_,_,_,_,_,A,A,A],A).
win([A,_,_,A,_,_,A,_,_],A).
win([_,A,_,_,A,_,_,A,_],A).
win([_,_,A,_,_,A,_,_,A],A).
win([A,_,_,_,A,_,_,_,A],A).
win([_,_,A,_,A,_,A,_,_],A).

full(Board):-
\+ member(e,Board).

move(Board,Pos,Player,New):-
nth0(Pos,Board,e,T),
nth0(Pos,New,Player,T).

score(Board,1):-win(Board,x),!.
score(Board,-1):-win(Board,o),!.

score(_,0).

minimax(Board,x,Score):-
score(Board,Score),!.

minimax(Board,o,Score):-
score(Board,Score),!.

minimax(Board,x,Best):-
findall(S,(move(Board,P,x,B1),minimax(B1,o,S)),L),max_list(L,Best).

minimax(Board,o,Best):-
findall(S,(move(Board,P,o,B1),minimax(B1,x,S)),L),min_list(L,Best).

best_move(Board,Pos):-
findall(Sc-P,(move(Board,P,x,B1),minimax(B1,o,Sc)),L),max_member(_-Pos,L).


best_move([x,o,x,o,e,e,e,e,e], P).
```

**Q2. Implement alpha-beta pruning to optimize the decision-making process.**

```
win([A,A,A,_,_,_,_,_,_],A).
win([_,_,_,A,A,A,_,_,_],A).
win([_,_,_,_,_,_,A,A,A],A).
win([A,_,_,A,_,_,A,_,_],A).
win([_,A,_,_,A,_,_,A,_],A).
win([_,_,A,_,_,A,_,_,A],A).
win([A,_,_,_,A,_,_,_,A],A).
win([_,_,A,_,A,_,A,_,_],A).

move(B,P,Pl,N):-nth0(P,B,e,T),nth0(P,N,Pl,T).
score(B,1):-win(B,x),!.
score(B,-1):-win(B,o),!.
score(_,0).

ab(B,_,_,_,x,S):-score(B,S),!.
ab(B,_,_,_,o,S):-score(B,S),!.
ab(B,A,Beta,Player,Best):-
   Player=x, findall(P-N,(move(B,P,x,N1),ab(N1,A,Beta,o,S),N=S),L),
   best_alpha(L,A,Beta,Best).
ab(B,A,Beta,Player,Best):-
   Player=o, findall(P-N,(move(B,P,o,N1),ab(N1,A,Beta,x,S),N=S),L),
   best_beta(L,A,Beta,Best).

best_alpha([_-S|_],A,_,A):-S=<A,!.
best_alpha([_-S|_],_,B,S):-S>=B,!.
best_alpha([_-S|T],A,B,Best):-A1 is max(A,S),best_alpha(T,A1,B,Best).
best_alpha([],A,_,A).

best_beta([_-S|_],_,B,B):-S>=B,!.
best_beta([_-S|_],A,_,S):-S=<A,!.
best_beta([_-S|T],A,B,Best):-B1 is min(B,S),best_beta(T,A,B1,Best).
best_beta([],_,B,B).

best_move_ab(Board,Pos):-findall(S-P,(move(Board,P,x,B1),ab(B1,-99,99,o,S)),L),max_member
(_-Pos,L).


best_move_ab([x,o,x,o,e,e,e,e,e], P).
```

**SET – F:**

**Q1. Implement an AI program to solve the 8 Puzzle problems.**

```
goal([1,2,3,4,5,6,7,8,0]).
move(S,N,up):-
nth0(Z,S,0),
Z>2,
NP is Z-3,
swap(S,Z,NP,N).
move(S,N,down):-
nth0(Z,S,0),
Z<6,
NP is Z+3,
swap(S,Z,NP,N).
move(S,N,left):-
nth0(Z,S,0),
Z mod 3=\=0,
NP is Z-1,
swap(S,Z,NP,N).
move(S,N,right):-
nth0(Z,S,0),
Z mod 3=\=2,
NP is Z+1,
swap(S,Z,NP,N).
swap(S,I,J,R):-
nth0(I,S,A),
nth0(J,S,B),
set(S,I,B,T),
set(T,J,A,R).
set([_|T],0,X,[X|T]).
set([H|T],N,X,[H|R]):-
N>0,
N1 is N-1,
set(T,N1,X,R).
bfs(Start,Sol):-bfs_q([[Start]],[],Sol).
bfs_q([[S|P]|_],_,Moves):-
goal(S),
reverse(P,M),
Moves=M.
bfs_q([[S|P]|Rest],V,Sol):-
findall([N,M|P],
(move(S,N,M),\+member(N,V)),NP),
append(Rest,NP,Q),
bfs_q(Q,[S|V],Sol).
bfs([1,2,3,4,5,6,7,0,8], M).
```

**Q2. Evaluate and compare different search algorithms for efficiency.**

## 1. Breadth-First Search (BFS)

- Expands nodes **level by level**.
- **Time Complexity:** O(b^d)
- **Space Complexity:** O(b^d)
- **Complete:** Yes
- **Optimal:** Yes (for equal step costs)
- **Use:** When shortest path (in number of steps) is required.

## 2. Depth-First Search (DFS)

- Expands nodes **deep into the tree** before backtracking.
- **Time Complexity:** O(b^m) (m = maximum depth)
- **Space Complexity:** O(b·m) (very low memory use)
- **Complete:** No (may go down infinite path)
- **Optimal:** No
- **Use:** When memory is limited or depth is small.

## 3. Greedy Best-First Search

- Uses only heuristic **h(n)** to choose the most promising node.
- **Time Complexity:** Depends on heuristic, usually much faster than BFS/DFS.
- **Space Complexity:** Can still be large because of open list.
- **Complete:** No
- **Optimal:** No
- **Use:** When you want a fast solution and optimality is NOT required (e.g., large search spaces).

## 4. A* Search

- Uses **f(n) = g(n) + h(n)** (cost so far + heuristic).
- **Time Complexity:** Exponential in worst case, but much fewer nodes with a good heuristic.
- **Space Complexity:** O(b^d) (stores many states).
- **Complete:** Yes (if heuristic is admissible).
- **Optimal:** Yes (if heuristic is admissible).
- **Use:** When best solution is required and a good heuristic (e.g., Manhattan distance) is available.

## 5. Minimax (for game playing)

- Explores the entire game tree assuming **optimal opponent**.

- **Time Complexity:** O(b^d).
- **Space Complexity:** O(b·d).
- **Optimal:** Yes (against perfect opponent).
- **Use:** Two-player games like Tic-Tac-Toe, Chess at small depth.

## 6. Alpha–Beta Pruning (Optimized Minimax)

- Eliminates branches that cannot affect the final Minimax decision.
- **Time Complexity:** Best case O(b^(d/2)) (much faster than Minimax).
- **Space Complexity:** O(b·d).
- **Optimal:** Same optimal result as Minimax.
- **Use:** Practical game-AI algorithms; deeper lookahead than plain Minimax.

## ✅ Comparison Summary

| Algorithm | Speed | Memory | Complete | Optimal |
|---|---|---|---|---|
| **BFS** | Slow for deep trees | Very high | ✔ Yes | ✔ Yes |
| **DFS** | Fast but risky | Very low | ✖ No | ✖ No |
| **Greedy** | Fast (depends on h) | Medium–high | ✖ No | ✖ No |
| **A*** | Efficient if h is good | High | ✔ Yes | ✔ Yes |
| **Minimax** | Slow for deep games | Medium | – | ✔ Optimal for adversarial games |
| **Alpha-Beta** | Much faster than Minimax | Medium | – | ✔ Same optimal result as Minimax |

**SET – G:**
**Q1. Develop a program to solve the 4-Queens problem using AI.**

```
:-use_module(library(clpfd)).

n_queens(N,Sol):-
length(Sol,N),
Sol ins 1..N,
all_distinct(Sol),
safe(Sol),
label(Sol).

safe([]).
safe([Q|T]):-
safe(T),
no_attack(Q,T,1).

no_attack(_,[],_).
no_attack(Q,[Q1|T],D):-
Q#\=Q1,
abs(Q-Q1)#\=D,
D1#=D+1,
no_attack(Q,T,D1).


n_queens(6,Sol).
```

**Q2. Optimize the solution and discuss the N-Queens problem.**

The N-Queens problem is a generalization of the 4-Queens problem. The task is to place **N queens on an N×N chessboard** so that **no two queens attack each other** — meaning no two queens share the same:

- row
- column
- diagonal (both left and right)

The simple solution uses **permutations** of columns (1..N) and checks diagonal safety.
Its time complexity is **O(N!)**, which becomes very slow as N increases.

## Optimized solution

To optimize the N-Queens program, we use the following techniques:

1. **Constraint propagation (CLP(FD))**
   Limit the domain of each queen to 1..N and enforce constraints directly:
   - all queens must be in different columns
   - all queens must be in different diagonals
     This prunes impossible positions early.
2. **All-distinct constraints**
   - `all_distinct(Qs)` for columns
   - `all_distinct(Qs+Row)` for one diagonal
   - `all_distinct(Qs-Row)` for the other diagonal
     These eliminate many invalid combinations before searching.
3. **Labeling heuristics**
   Using `labeling([ff],Qs)` (first-fail heuristic) selects the most constrained queen first, which reduces backtracking.
4. **Symmetry reduction** (optional)
   For large N, we avoid symmetric solutions by fixing the first queen to a limited range (e.g., first half of the board).

## Discussion

- The N-Queens problem has solutions for all **N ≥ 4** (except N=2 and N=3 which have none).
- The number of solutions grows rapidly, making brute-force methods inefficient.
- Using **constraint solving** or **heuristic search** drastically reduces the search space.
- Optimized algorithms can solve N=20, N=30, or even bigger boards quickly.