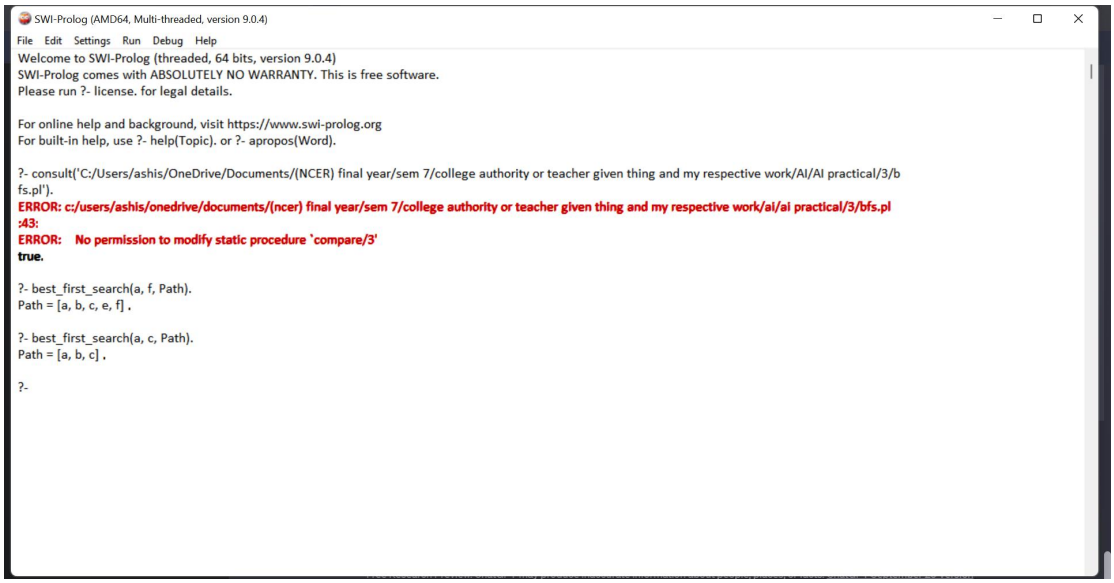
1. Write a Program in Prolog to solve any problem using Best First Search.
Answer:

```
% Define your graph with weighted edges and heuristics.
% Replace these with your actual graph and heuristics.
edge(a, b, 2).
edge(b, c, 3).
edge(b, d, 4).
edge(c, e, 5).
edge(d, e, 1).
edge(e, f, 2).

% Define heuristic values (straight-line distances to the goal).
heuristic(a, 6). % Replace with your specific values.
heuristic(b, 5).
heuristic(c, 4).
heuristic(d, 3).
heuristic(e, 2).
heuristic(f, 0).

% Define a predicate to calculate the total estimated cost.
total_cost(Node, Path, Cost) :-
    path_cost(Path, PCost),
    heuristic(Node, HCost),
    Cost is PCost + HCost.

path_cost([_], 0).
path_cost([A, B | Tail], Cost) :-
    edge(A, B, EdgeCost),
    path_cost([B | Tail], RestCost),
    Cost is EdgeCost + RestCost.

% Define Best-First Search algorithm.
best_first_search(Start, Goal, Path) :-
    best_first_search_internal([ [Start] ], Goal, RevPath),
    reverse(RevPath, Path).

best_first_search_internal([ [Goal | Path] | _ ], Goal, [Goal | Path]).
best_first_search_internal([ [Node | Path] | Rest ], Goal, Result) :-
    findall([Next, Node | Path], (edge(Node, Next, _), not(member(Next, Path))),
NewPaths),
    append(Rest, NewPaths, AllPaths),
    predsort(compare, AllPaths, SortedPaths),
    best_first_search_internal(SortedPaths, Goal, Result).

% Comparison function for sorting paths based on total cost.
compare(Result, [_, _, Path1], [_, _, Path2]) :-
    total_cost(Path1, Path1, Cost1),
    total_cost(Path2, Path2, Cost2),
    compare_paths(Cost1, Cost2, Result).
```

```
compare_paths(Cost1, Cost2, Result) :-
    (Cost1 < Cost2 -> Result = (<);
    Cost1 > Cost2 -> Result = (>);
    Result = (=)).

% Example usage:
% To find the path, call best_first_search(StartNode, GoalNode, Path).
% Replace StartNode and GoalNode with your problem's specific values.
```

Output:



2. Write a Program in Python to solve any problem using Best First Search.
Answer:

```
from queue import PriorityQueue

def best_first_search(graph, start, goal):
    frontier = PriorityQueue()
    frontier.put(start)  # Use a priority queue with the initial node
    came_from = {}  # Dictionary to store the best path
    came_from[start] = None

    while not frontier.empty():
        current = frontier.get()
        if current == goal:
            return reconstruct_path(came_from, current)

        for neighbor, weight in graph.get(current, []):
            if neighbor not in came_from:
                came_from[neighbor] = current
                frontier.put(neighbor)

    return None

def reconstruct_path(came_from, current):
```

```
    path = []
    while current:
        path.insert(0, current)
        current = came_from[current]
    return path

# Example usage:
graph = {
    'a': [('b', 2)],
    'b': [('c', 3), ('d', 4)],
    'c': [('e', 5)],
    'd': [('e', 1)],
    'e': [('f', 2)],
}

start_node = 'a'
goal_node = 'f'

path = best_first_search(graph, start_node, goal_node)
if path:
    print(f"Shortest path from {start_node} to {goal_node}: {path}")
else:
    print(f"No path found from {start_node} to {goal_node}")
```

Output: