**A\* Procedure:**

1. **Define the Graph** – Represented using edge(Node1, Node2, Cost) facts.
2. **Define Heuristic Function** – heuristic(Node, H) provides estimated cost to goal.
3. *A Algorithm Initialization*\* – Starts with astar(Start, Goal, Path, Cost), initializing open list.
4. **Node Expansion in A**\* – Extracts best node, generates successors, calculates G and F values.
5. **Sorting in A**\* – Open list is sorted based on F = G + H (ascending order).
6. **Goal Check in A**\* – If the first node in the list is the goal, return the path and cost.
7. **Recursive Search in A**\* – Continues expanding until the goal is reached.
8. *Memory Bounded A Initialization*\* – Calls mba\_star(Start, Goal, Path, Cost, MemoryLimit).
9. **Memory Check in MBA**\* – If open list exceeds MemoryLimit, it is trimmed.
10. **Trimming Function** – Reduces open list size to half (trim(Open, TrimmedOpen)).
11. **Sorted Expansion in MBA**\* – After trimming, continues like A\* with sorted open list.
12. **Recursive Search in MBA**\* – Expands nodes while ensuring memory constraints.
13. **Final Path and Cost Calculation** – Returns optimal or near-optimal path within memory limits.
14. **Query for A**\* – Example: ?- astar(a, f, Path, Cost).
15. **Query for MBA**\* – Example: ?- mba\_star(a, f, Path, Cost, 3).

**Genetic Algorithm**

1. Define the Initial Population – The population consists of values [0,1,2,3,4,5,6,7].
2. Define Fitness Function – fitness(X, F) calculates F = X^2 to evaluate each individual’s fitness.
3. Selection Process –

* Compute fitness for all population members.
* Pair fitness values with individuals.
* Sort population in descending order based on fitness.
* Select the top 4 individuals.

1. Crossover Operation –

* Takes two selected parents.
* Generates two offspring using averaging techniques:
  + C1 = (P1 + P2) // 2
  + C2 = (P2 + 1) // 2

1. Mutation Process –

* Randomly changes an offspring’s value.
* Ensures the new value is different from the original.

1. Main Genetic Algorithm Execution –

* Initialize population.
* Perform selection to get top individuals.
* Apply crossover to create offspring.
* Apply mutation to introduce randomness.

1. Stopping Condition – The algorithm runs once (single generation), returning the best solution.

**Propositional Logic**

1. **Define Facts** – Establish weather conditions:

* rainy(weather). (It is rainy)
* cold(weather). (It is cold)

1. **Define Rules for Actions** –

* wear\_jacket if it is rainy or cold.
* carry\_umbrella if it is rainy.
* stay\_home if it is both rainy and cold.

1. **Define Inference Rule (inference/1)** –

* Checks if wear\_jacket is true → Decision = 'Wear a jacket'.
* Checks if carry\_umbrella is true → Decision = 'Carry an umbrella'.
* Checks if stay\_home is true → Decision = 'Stay home'.

**First Order Logic**

1. **Define Parent Relationships** –

* parent(john, mary). (John is Mary’s parent)
* parent(mary, susan). (Mary is Susan’s parent)

1. **Define Ancestor Rule** –

* Direct parent-child relationship: ancestor(X, Y) :- parent(X, Y).
* Recursive transitive relation: ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

1. **Define Inference Mechanism (infer/1)** –

* Uses call(Query) to check if the query is true.
* If true, prints: 'True: Query'.
* If false, prints: 'False: Query'.

**Classical Planning**

1. **Define Initial State** –

* state([at(package, depot), at(truck, depot)]).
* The package and truck are initially at the depot.

2. **Define Goal State** –

* goal([at(package, warehouse)]).
* The package must be at the warehouse.

3. **Define Action (move/3)** –

* Preconditions: package and truck must be at depot.
* Delete List: Removes at(package, depot).
* Add List: Adds at(package, warehouse).

4. **Plan Solving (solve\_plan/2)** –

* If the goal is a subset of the current state, return an empty plan ([]).
* Otherwise, select an applicable action (action/4).
* Check if its preconditions are satisfied in the current state.
* Apply delete effects (subtract/3).
* Apply add effects (union/3).
* Recursively solve the plan for the new state.

**Backtracking of CSP**

1. Define the Main Predicate (solve/2)

* solve(N, Solution): Finds a valid placement of N queens on an N×N chessboard.
* length(Solution, N): Ensures the list Solution has N elements (one queen per row).
* maplist(between(1, N), Solution): Assigns each queen a column position (between 1 and N).
* Calls safe(Solution) to verify no two queens threaten each other.

1. Define the Safety Constraint (safe/1)

* Ensures all queens in Solution are placed without attacking each other.
* Calls safe/3 recursively to check diagonal and column constraints.

1. Define Recursive Safety Check (safe/3)

* safe([], \_, \_): Base case, no conflicts in an empty list.
* safe([Q|Qs], Q1, D): Checks if the current queen Q1:
  + Is not in the same column (Q1 =\= Q).
  + Is not on the same diagonal (abs(Q1 - Q) =\= D).
  + Increments the diagonal counter (D1 is D + 1) and checks the next queen.

**Simulated Annealing**

1. Define city distances using distance/3 predicate for direct connections.
2. Handle undirected travel with dist/3 to allow both-way travel.
3. Generate possible routes using generate\_route/2 with city permutations.
4. Calculate route distance using total\_distance/2 recursively.
5. Initialize Simulated Annealing with simulated\_annealing/3.
6. Set initial temperature using initial\_temperature/1 (e.g., 1000).
7. Optimize route iteratively with anneal/4 by modifying the route.
8. Modify the route randomly in anneal\_step/5 (currently missing swap logic).
9. Compare new vs. old routes and accept better ones.
10. Use probability-based acceptance for worse routes (not implemented in current code).
11. Continue annealing process until convergence or temperature is low.
12. Return the best route with minimum distance.
13. Run the algorithm using ?- simulated\_annealing([a, b, c, d], BestRoute, BestDistance).
14. Expected output: A near-optimal route with its distance.
15. Improve by adding proper random swapping for effective optimization.