BFS CODE:

```
from collections import deque
def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
print("BFS traversal starting from node A:")
bfs(graph, 'A')
```

DFS:

```
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    print(node, end=" ")
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
print("DFS traversal starting from node A:")
dfs(graph, 'A')
```

A*:

```
def a_star(graph, start, goal, h):
    open_nodes = set([start])
    parent = {}
    g = {n: float('inf') for n in graph}
    g[start] = 0
    f = {n: float('inf') for n in graph}
    f[start] = h[start]
    while open_nodes:
        current = min(open_nodes, key=lambda x: f[x])
        if current == goal:
            path = []
            while current in parent:
                path.append(current)
                current = parent[current]
            path.append(start)
            path.reverse()
            return path
        open_nodes.remove(current)
        for neighbor, cost in graph[current].items():
            new_g = g[current] + cost
            if new_g < g[neighbor]:
                parent[neighbor] = current
                g[neighbor] = new_g
                f[neighbor] = g[neighbor] + h[neighbor]
                open_nodes.add(neighbor)
graph = {
    'A': {'B': 1, 'C': 3},
    'B': {'D': 1, 'E': 5},
    'C': {'E': 2},
    'D': {'F': 4},
    'E': {'F': 1},
    'F': {}
}
h = {'A': 7, 'B': 6, 'C': 2, 'D': 1, 'E': 0, 'F': 0}
print("Path from A to F:")
print(a_star(graph, 'A', 'F', h))
```

WUMPUS:

```
world = [
    ['', '', ''],
    ['', '', ''],
    ['A', 'G', '']]
start = (2, 0)
goal = (2, 1)
from collections import deque
def bfs(start, goal):
    queue = deque([[start]])
    visited = set()
    while queue:
        path = queue.popleft()
        x, y = path[-1]
        if (x, y) == goal:
            return path
        if (x, y) in visited:
            continue
        visited.add((x, y))
        for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
            nx, ny = x+dx, y+dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                if world[nx][ny] != 'P' and world[nx][ny] != 'W':
                    new_path = list(path)
                    new_path.append((nx, ny))
                    queue.append(new_path)
path_to_gold = bfs(start, goal)
print("Path to Gold:", path_to_gold)
```

Hill Climbing

import java.util.Random;

```java
public class SimpleNQueens {
    static Random rand = new Random();

    // Count attacking pairs (how many queens attack each other)
    static int attacks(int[] board) {
        int count = 0;
        for (int i = 0; i < board.length; i++) {
            for (int j = i + 1; j < board.length; j++) {
                if (board[i] == board[j]) count++; // same row
                if (Math.abs(board[i] - board[j]) == Math.abs(i - j)) count++; // diagonal
            }
        }
        return count;
    }

    // Create a random board (1 queen per column)
    static int[] randomBoard(int n) {
        int[] b = new int[n];
        for (int i = 0; i < n; i++) b[i] = rand.nextInt(n);
        return b;
    }

    // Print board
    static void printBoard(int[] b) {
        int n = b.length;
        for (int r = 0; r < n; r++) {
            for (int c = 0; c < n; c++)
                System.out.print(b[c] == r ? "Q " : ". ");
            System.out.println();
        }
        System.out.println("Attacks: " + attacks(b) + "\n");
    }

    // --- Hill Climbing ---
    static int[] hillClimb(int n) {
        int[] board = randomBoard(n);
        int best = attacks(board);

        for (int step = 0; step < 1000; step++) {
            int[] next = board.clone();
            int col = rand.nextInt(n);
            next[col] = rand.nextInt(n);

            int nextScore = attacks(next);
            if (nextScore < best) { // accept better move
                board = next;
                best = nextScore;
            }
            if (best == 0) break; // solved
        }
        return board;
    }

    // --- Simulated Annealing ---
    static int[] simulatedAnnealing(int n) {
        int[] board = randomBoard(n);
```

```
        int score = attacks(board);
        double T = 10.0; // start temperature

        for (int step = 0; step < 2000 && T > 0.001; step++) {
            int[] next = board.clone();
            int col = rand.nextInt(n);
            next[col] = rand.nextInt(n);

            int nextScore = attacks(next);
            int diff = nextScore - score;

            // accept if better, or sometimes if worse
            if (diff < 0 || Math.exp(-diff / T) > rand.nextDouble()) {
                board = next;
                score = nextScore;
            }
            T *= 0.99; // cool down
            if (score == 0) break; // solved
        }
        return board;
    }

    public static void main(String[] args) {
        int n = 8;

        System.out.println("=== Hill Climbing ===");
        int[] hill = hillClimb(n);
        printBoard(hill);

        System.out.println("=== Simulated Annealing ===");
        int[] anneal = simulatedAnnealing(n);
        printBoard(anneal);
    }
}
```

Alpha-Beta Pruning Algorithm

```
public class AlphaBetaSimple {

    // Alpha-Beta Pruning function
    static int alphaBeta(int depth, int nodeIndex, boolean isMax,
                         int[] values, int alpha, int beta, int maxDepth) {
        // If we reached a leaf node, return its value
        if (depth == maxDepth)
            return values[nodeIndex];

        if (isMax) { // Maximizer's move
            int best = Integer.MIN_VALUE;
            // Loop through left and right child
            for (int i = 0; i < 2; i++) {
                int val = alphaBeta(depth + 1, nodeIndex * 2 + i, false,
                        values, alpha, beta, maxDepth);
                best = Math.max(best, val);
                alpha = Math.max(alpha, best);
                if (beta <= alpha) break; // prune
            }
            return best;
```

```java
    } else { // Minimizer's move
      int best = Integer.MAX_VALUE;
      for (int i = 0; i < 2; i++) {
        int val = alphaBeta(depth + 1, nodeIndex * 2 + i, true,
            values, alpha, beta, maxDepth);
        best = Math.min(best, val);
        beta = Math.min(beta, best);
        if (beta <= alpha) break; // prune
      }
      return best;
    }
  }

  public static void main(String[] args) {
    // Example game tree leaf values
    int[] values = {3, 5, 6, 9, 1, 2, 0, -1};
    int maxDepth = 3;

    int result = alphaBeta(0, 0, true, values,
        Integer.MIN_VALUE, Integer.MAX_VALUE, maxDepth);

    System.out.println("Best value (with Alpha-Beta Pruning): " + result);
  }
}
```

CSP

```java
import java.util.*;

public class MapColoringCSP {
  static String[] colors = {"Red", "Green", "Blue"};
  static Map<String, List<String>> neighbors = new HashMap<>();
  static Map<String, String> assignment = new HashMap<>();

  // Forward Checking — checks if assignment is consistent
  static boolean isConsistent(String region, String color) {
    for (String n : neighbors.get(region)) {
      String assignedColor = assignment.get(n);
      if (color.equals(assignedColor)) return false; // same color not allowed
    }
    return true;
  }

  // Backtracking + Forward Checking
  static boolean backtrack(List<String> regions, int index) {
    if (index == regions.size()) return true; // all assigned

    String region = regions.get(index);

    for (String color : colors) {
      if (isConsistent(region, color)) {
        assignment.put(region, color); // assign color

        if (backtrack(regions, index + 1))
          return true; // success

        assignment.remove(region); // undo assignment (backtrack)
```

```
        }
      }
      return false; // no color worked
    }

    public static void main(String[] args) {
      // Define adjacency (constraints)
      neighbors.put("A", Arrays.asList("B", "C"));
      neighbors.put("B", Arrays.asList("A", "C", "D"));
      neighbors.put("C", Arrays.asList("A", "B", "D"));
      neighbors.put("D", Arrays.asList("B", "C"));

      List<String> regions = Arrays.asList("A", "B", "C", "D");

      if (backtrack(regions, 0)) {
        System.out.println("✅ Solution found:");
        for (String r : regions)
          System.out.println(r + " → " + assignment.get(r));
      } else {
        System.out.println("❌ No solution found.");
      }
    }
}
```

Parse First-Order Logic Sentences

```
import java.util.*;

public class MiniFOLParser {
    public static void main(String[] args) {
      parse("∀x (Human(x) → Mortal(x))");
      parse("∃y (Dog(y) ∧ Loves(y, John))");
    }

    static void parse(String s) {
      s = s.replaceAll("\\s+", ""); // remove spaces
      String quant = s.substring(0,1);
      String var = s.substring(1,2);
      String inner = s.substring(s.indexOf('(')+1, s.lastIndexOf(')'));

      String conn = inner.contains("→") ? "→" : inner.contains("∧") ? "∧" : inner.contains("∨") ? "∨" : "";
      String[] parts = conn.isEmpty() ? new String[]{inner, ""} : inner.split(conn);

      System.out.println("\nSentence: " + s);
      System.out.println("Quantifier: " + quant);
      System.out.println("Variable: " + var);
      System.out.println("Left Predicate: " + parts[0]);
      System.out.println("Connective: " + conn);
      System.out.println("Right Predicate: " + (parts.length>1?parts[1]:""));
    }
}
```

Bayesian Network

```
public class SimpleBayes {
    enum Weather { SUNNY, RAINY }
```

```
  enum Forecast { GOOD, BAD }

  // Prior P(Weather)
  static double pWeather(Weather w) {
    return w == Weather.SUNNY ? 0.7 : 0.3;
  }

  // P(Forecast | Weather)
  static double pForecastGivenWeather(Forecast f, Weather w) {
    if (w == Weather.SUNNY) return f == Forecast.GOOD ? 0.8 : 0.2;
    else              return f == Forecast.GOOD ? 0.3 : 0.7;
  }

  // P(Picnic = Yes | Weather)
  static double pPicnicGivenWeatherYes(Weather w) {
    return w == Weather.SUNNY ? 0.9 : 0.2;
  }

  // Bayes: P(Weather = w | Forecast = f)
  static double posteriorWeatherGivenForecast(Weather w, Forecast f) {
    // numerator = P(f|w) * P(w)
    double numer = pForecastGivenWeather(f, w) * pWeather(w);
    // denominator = sum_w P(f|w)P(w)
    double denom = 0.0;
    for (Weather ww : Weather.values()) denom += pForecastGivenWeather(f, ww) * pWeather(ww);
    return numer / denom;
  }

  // Marginalize to get P(Picnic = Yes | Forecast = f)
  static double pPicnicYesGivenForecast(Forecast f) {
    double sum = 0.0;
    for (Weather w : Weather.values()) {
      double pWgivenF = posteriorWeatherGivenForecast(w, f);
      sum += pPicnicGivenWeatherYes(w) * pWgivenF;
    }
    return sum;
  }

  public static void main(String[] args) {
    Forecast evidence = Forecast.GOOD;

    double pSunnyGivenGood = posteriorWeatherGivenForecast(Weather.SUNNY, evidence);
    double pPicnicYesGivenGood = pPicnicYesGivenForecast(evidence);

    System.out.printf("P(Weather=Sunny | Forecast=Good) = %.4f%n", pSunnyGivenGood);
    System.out.printf("P(Picnic=Yes | Forecast=Good)   = %.4f%n", pPicnicYesGivenGood);
  }
}
```