AI Week 5 Notes

Here's your formatted document with improved readability while keeping the original wording intact.

Week 5: Local Search Strategies

Dr. Ammar Masood
Department of Cyber Security,
Air University Islamabad
CS340 - Introduction to AI

Table of Contents

Problem Statement

In the search problems of **Chapter 3**, we aimed to find paths through the search space, such as from **Arad to Bucharest**. However, sometimes we are only interested in the **final state**, not the path taken.

Example

- **8-Queens Problem**: Finding a valid final configuration of 8 queens.
- **Other Applications**: Integrated-circuit design, factory floor layout, job shop scheduling, automatic programming, etc.

Solution: Local Search Algorithms

- Use a **single current state** and move to neighboring states.
- **Advantages**:
  - Uses very little memory.
  - Often finds reasonable solutions in large/infinite state spaces.
  - Useful for pure **optimization problems** (finding the best state).

Local Search / Optimization

- **Goal**: Find the best state based on an **objective function**.
- **Objective function**: Measures the “fitness” of a state.
- **Problem**: Find the **optimal state** that maximizes/minimizes the objective function.

Local Search Strategies

- Local search algorithms operate by **searching from a start state to neighboring states**, without tracking the path.
- They are **not systematic** and may fail to explore parts of the search space.
- **When is local search used?**
  - When the **state-space is too large** to explore completely.
  - Starts with an **initial solution** and iteratively improves it.
  - Unlike **uninformed search**, it **does not maintain a search tree**.
  - Solves **optimization problems** where the goal is to find the **best state**.

State-Space Landscape

- **If elevation = objective function**, the goal is to find the **highest peak** (global maximum) → **Hill Climbing**.
- **If elevation = cost**, the goal is to find the **lowest valley** (global minimum) → **Gradient Descent**.

Advantages

1. Uses **very little memory**.
2. Can often find **reasonable solutions** in **large/infinite state spaces**.

Hill Climbing

Idea

- Always move towards a **state with a higher value** (maximization) or **lower cost** (minimization).

Limitations

- **Gets stuck in local optima**.
- **Susceptible to plateaus and ridges**.

Examples

- **Robotics Pathfinding**: A robot moves toward higher ground to avoid obstacles.
- **Traveling Salesman Problem (TSP)**: Finding a shorter route step by step.

Hill Climbing Algorithm

1. **Start anywhere**.
2. Always choose the **best neighbor**.
3. If **no better neighbors**, quit.

4. **Random selection** among the best successors (if multiple exist).

Working

- Keeps track of **one current state**.
- Moves to the **neighboring state with the highest value**.
- Terminates when reaching a **"peak"** (no neighbor has a higher value).
- **Does not look beyond immediate neighbors**.

Example: 8-Queens Problem

- **Initial State**: 8 queens placed randomly on the board.
- **Successor Function**: Moving one queen per column ($8 \times 7 = 56$ successors).
- **Heuristic Function**: Number of **pairs of attacking queens**.

Drawbacks of Hill Climbing

Problems

- **Local Maxima**: Gets stuck in **suboptimal solutions**.
- **Plateaus**: Flat regions with no direction for movement.
- **Ridges**: Higher solutions exist, but cannot be reached directly.

Solutions

- **Introduce randomness** to escape local optima.

Variants of Hill Climbing

1. **Stochastic Hill Climbing**
    - Chooses **randomly** from uphill moves.
    - Slower convergence but may **find better solutions**.
2. **First-Choice Hill Climbing**

- Randomly generates successors until one is **better than the current state**.
- **Useful when many successors exist**.

Random Restart Hill Climbing

- **Idea**: "If at first, you don't succeed, try again."
- Conducts multiple hill-climbing searches from **randomly generated initial states**.
- **Complete with probability 1** (eventually finds a goal state).
- **Expected number of restarts** = **1/p** (where p is the probability of success).

Example

- Climbing a **mountain range**:
  - Start climbing randomly → may reach **local maxima**.
  - Restart from a **new location** to find a **higher peak**.

Simulated Annealing

- **Inspired by metal cooling** → Gradually reducing randomness.
- **Accepts worse solutions** with some probability → Avoids local optima.
- **Uses a cooling schedule** (temperature **T gradually decreases**).

Example Applications

- **Job Scheduling**: Finding the best sequence to reduce delays.
- **Neural Network Training**: Avoids poor convergence to bad local minima.

Working

1. **If move improves the state**, always accept.
2. **If move worsens the state**, accept with **some probability.**
3. **Probability decreases** as:

- **Move gets worse (ΔE increases)**.
- **Temperature (T) decreases**.

Cooling Schedule

- If **T is reduced too fast**, solution quality suffers.
- **T(t) = a * T(t-1)**, where **$0.8 \leq a \leq 0.99$**.

Local Beam Search

- **Expands only the k best nodes** at each level (unlike BFS/DFS).
- Tracks the **most promising paths**.
- **More efficient** than exhaustive search, but may **miss the global best solution**.

Working

- Tracks **k states instead of one**.
- At each step:
  - Generates **all successors** of k states.
  - Selects the **k best successors**.
  - If a goal state is found, return it.

Comparison with Random Restart

- **Random Restart**: Each search is independent.
- **Local Beam Search**: Best successors influence other searches.

Genetic Algorithms

- Inspired by **evolutionary biology**.
- **Faster, randomized search** for optimal solutions.

Operations

- **Crossover**: Combines two parents to create new children.
- **Mutation**: Introduces random changes.

Working

1. **Initialize population** (random solutions).
2. **Crossover & Mutation** on the fittest individuals.
3. **Keep only the best solutions**.

Gradient Descent

- **Optimization algorithm** to minimize a **cost function**.
- Iteratively **adjusts parameters** to reduce error.

Working

1. **Initialize parameters** (random values).
2. **Compute Gradient**: Find direction of steepest increase.
3. **Update Parameters**: Move **opposite to gradient**.

This formatting improves readability and ensures all the original details are maintained. Let me know if you need further adjustments! 🚀