Artificial Intelligence

BS (CS) _SUMMER_2024

Lab_07 Manual

Learning Objectives:

1. Local Search Algorithm
2. Hill Climbing
3. Stochastic Hill Climbing
4. First Choice Hill Climbing
5. Local Beam Search

Beyond Classical Search Algorithm

Local Search Algorithm

A local search algorithm in artificial intelligence is an optimization technique that seeks the optimal solution to a problem through iterative adjustments to an initial response. Its goal is to find the best possible solution within a specified search space. In contrast to global search methods, local search algorithms explore the entire solution space and concentrate on making incremental changes to enhance the current solution until they converge to a locally optimal solution.

This approach proves particularly valuable in scenarios where the solution space is extensive, as local search algorithms operate with minimal memory usage, typically a constant amount. They excel in exhaustive searches, efficiently navigating large or even infinite state spaces to identify a satisfactory solution to an optimization problem based on a predefined objective function.

Working:

The basic working principle is:

1. **Initialization**: Start with an initial solution, which can be generated randomly or through some heuristic method.
2. **Evaluation:** Evaluate the quality of the initial solution using an objective function or a fitness measure. This function quantifies how close the solution is to the desired outcome.
3. **Neighbor Generation:** Generate a set of neighboring solutions by making minor changes to the current solution. These changes are typically referred to as "moves."
4. **Selection:** Choose one of the neighboring solutions based on a criterion, such as the improvement in the objective function value. This step determines the direction in which the search proceeds.
5. **Termination:** Continue the process iteratively, moving to the selected neighboring solution, and repeating steps 2 to 4 until a termination condition is met. This condition could be a maximum number of iterations, reaching a predefined threshold, or finding a satisfactory solution.

Hill Climbing

Hill climbing is a straightforward local search algorithm that starts with an initial solution and iteratively moves to the best neighboring solution that improves the objective function. Here's how it works:

1. **Initialization:** Begin with an initial solution, often generated randomly or using a heuristic method.
2. **Evaluation:** Calculate the quality of the initial solution using an objective function or fitness measure.
3. **Neighbor Generation:** Generate neighboring solutions by making small changes (moves) to the current solution.
4. **Selection:** Choose the neighboring solution that results in the most significant improvement in the objective function.
5. **Termination:** Continue this process until a termination condition is met (e.g., reaching a maximum number of iterations or finding a satisfactory solution).

Algorithm 1: Hill Climbing Search Algorithm

```
Data: An arbitrary search space (ss)
Result: An optimal state (s)
Function hillClimbing(ss)
    s ← random state from ss
    while true do
        N ← neighbors(s)
        if N is empty then
            return s
        end
        for n in N do
            if f(n) ≥ f(s) then
                s ← n
            end
        end
    end
    return s
end
```

Pseudo Code:

```
import random

def objective_function(solution):
    # Define your objective function here
    # This function should evaluate the quality of a solution and return a value
    # The higher the value, the better the solution
    # Modify this function based on your specific optimization problem
    return sum(solution)

def generate_neighbor(current_solution):
```

```
    # Generate a neighboring solution by making a small modification to
the current solution
    # This function should implement the logic to generate a neighboring
solution
    # Modify this function based on your specific optimization problem
    neighbor = current_solution[:]
    index = random.randint(0, len(neighbor) - 1)
    neighbor[index] = 1 - neighbor[index]  # Flip the value at the
selected index
    return neighbor

def hill_climbing():
    # Initialization
    current_solution = [random.randint(0, 1) for _ in range(10)]  #
Generate an initial solution
    current_fitness = objective_function(current_solution)

    # Iterative process
    while True:
        # Neighbor generation
        neighbor = generate_neighbor(current_solution)
        neighbor_fitness = objective_function(neighbor)

        # Comparison
        if neighbor_fitness >= current_fitness:
            current_solution = neighbor
            current_fitness = neighbor_fitness
        else:
            break  # Terminate if no better solution is found

    return current_solution, current_fitness

# Usage example
best_solution, best_fitness = hill_climbing()
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Stochastic Hill Climbing

Stochastic hill climbing is a variant of the basic hill climbing method. While basic hill climbing always chooses the steepest uphill move, "stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.

Algorithm 2: Stochastic Hill Climbing Algorithm

current position = initial solution;
repeat
  for *All neighbours of current position* **do**
    Obtain a random neighbour;
    if *cost of neighbour* $\leq$ *cost of current position* **then**
      current position = neighbour position;
      break;
    end
  end
until *cost of current position* $\leq$ *cost of all its neighbours*;

Pseudo Code:

```
// Pseudo Code

function h(State s) {
  // Heuristic Evaluation Function
}

function List::ChooseRandom() {
    // return move with probability proportional to the improvement.
}

function HillClimbing(State s) {
    State best = s;
    State current;
    List betterMoves = List();

    while (true) {
        current = best;

        // Look for better moves
        for (State next : s.nextStates()) // foreach...in..
            if (h(best) < h(next))
                betterMoves.add(next);

        // Choose randomly, from better moves -- if any
        if (betterMoves.length() > 0)
            best = betterMoves.chooseRandom();

        // If current & best are STILL the same, then we reached a peak.
        if (best == current)
```

```
            return best;
    }
}
```

First Choice Hill Climbing

This simplifies the neighbor selection step by choosing the first option better than the current node and do not evaluate all neighbors to find the best one at every iteration. This reduces computation time per iteration.

Pseudo Code:

```
// Pseudo Code

function h(State s) {
  // Heuristic Evaluation Function
}

function generateRandomState() {
    // return a new randomly generated state.
}

function HillClimbing(State s) {
    State best = s;
    State current;

    while (true) {
        current = best;

        // Look for better moves.
        for(i = 0; i < THRESH_HOLD; i++) // foreach...in..
        {
            generated = generateRandomState();
            if (h(best) < h(generated)) {
                best = generated
                break;
            }
        }

        // If current & best are STILL the same, then we reached a peak.
        if (best == current)
            return best;
    }
}
```

Local Beam Search

Local beam search represents a parallelized adaptation of hill climbing, designed specifically to counteract the challenge of becoming ensnared in local optima. Instead of starting with a single initial solution, local beam search begins with multiple solutions, maintaining a fixed number (the "beam width") simultaneously. The algorithm explores the neighbors of all these solutions and selects the best solutions among them.

1. **Initialization:** Start with multiple initial solutions.
2. **Evaluation:** Evaluate the quality of each initial solution.
3. **Neighbor Generation:** Generate neighboring solutions for all the current solutions.
4. **Selection:** Choose the top solutions based on the improvement in the objective function.
5. **Termination:** Continue iterating until a termination condition is met.

Algorithm 1: Beam Search Algorithm

Data: Graph (G), start node (s), goal node (g), beam width (β)
Result: Path with lowest cost

```
Function beamSearch(G, s, g, β)
    openList ← s
    closedList ← empy list
    path ← empy list
    while open list is not empty do
        b ← best node from openList
        openList.remove(b)
        closedList.add(b)
        if b is g then
            path.add(b)
            return path
        end
        N ← neighbors(b)
        for n in N do
            if n is in neither closedList nor openList then
                openList.add(n)
            else if n is in openList then
                if path with current parent ≤ path with old parent
                 then
                    Replace parents of n
                end
            else if n is not in closedList then
                openList.add(n)
            end
        end
        if number of nodes in openList > β then
            openList ← best β nodes in openList
        end
    end
    return path
end
```