DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- For example, in the 8-queens problem, what matters isthe final configuration of queens, not the order in which they are added.
- In such cases, we can **use local search algorithms.** They operate using a **single current state** (rather than multiple paths) and generally move only to neighbors of that state.
- The important applications of these class of problems are (a) integrated-circuit design, (b) Factory-floor layout, (c) job-shop scheduling, (d) automatic programming, (e) telecommunications network optimization, (f) Vehicle routing, and (g) portfolio management.

Key advantages of Local Search Algorithms

(1) They use very little memory – usually a constant amount; and

(2) they can often find reasonable solutions in large or infinite(continuous) state spaces for which systematic algorithms are unsuitable.

2.18 OPTIMIZATION PROBLEMS

In addition to finding goals, local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the **best state** according to an **objective function**.

State Space Landscape

To understand local search, it is better explained using **state space landscape** as shown in Figure.

A landscape has both "**location**" (defined by the state) and "**elevation**" (defined by the value of the heuristic cost function or objective function).

If elevation corresponds to **cost**, then the aim is to find the **lowest valley** – a **global minimum**; if elevation corresponds to an **objective function**, then the aim isto find the **highest peak** – a **global maximum.**

Local search algorithms explore this landscape. A complete local search algorithm always finds a **goal** if one exists; an **optimal** algorithm always finds a **global minimum/maximum**.

Hill-climbing search

The **hill-climbing** search algorithm as shown in figure, is simply a loop that continually moves in the direction of increasing value – that is, **uphill**. It terminates when it reaches a "**peak**" where no neighbor has a higher value.

```
function HILL-CLIMBING( problem) return a state that is a local
        maximum input: problem, a problem
 local variables: current, a
                          node.
           neighbor, a node.

        current ←MAKE-NODE(INITIAL-STATE[problem])
        loop do
              neighbor ← a highest valued successor of current
              if VALUE [neighbor] ≤ VALUE[current] then return
              STATE[current] current ←neighbor
```

Figure 2.24 The hill-climbing search algorithm (steepest ascent version), which is the most basic local search technique. At each step the current node is replaced by the best neighbor; the neighbor with the highest VALUE. If the heuristic cost estimate h is used, we could find the neighbor with the lowest h.

Hill-climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Greedy algorithms often perform quite well. **Problems with hill-climbing**

Hill-climbing often gets stuck for the following reasons :

· **Local maxima**: a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go

· **Ridges**: A ridge is shown in Figure 2.10. Ridges results in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

· **Plateaux**: A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which it is possible to make progress.

Hill-climbing variations

- Stochastic hill-climbing
  - Random selection among the uphill moves.

- The selection probability can vary with the steepness of the uphill move.

- First-choice hill-climbing
  - cfr. stochastic hill climbing by generating successors randomly until a better one is found.
- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima.

Simulated annealing search

A hill-climbing algorithm that never makes "downhill" moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor choosen uniformly at random from the set of successors – is complete, but extremely inefficient.

Simulated annealing is an algorithm that combines hill-climbing with a random walk in someway that yields both efficiency and completeness.

Figure shows simulated annealing algorithm. It is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move – the amount E by which the evaluation is worsened.
Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

Genetic algorithms

A Genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states, rather than by modifying a single state

Like beam search, Gas begin with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet – most commonly, a string of 0s and 1s. For example, an 8 8-quuens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8 x log2 8 = 24 bits.

Figure shows a population of four 8-digit strings representing 8-queen states. The production of the next generation of states is shown in Figure

In (b) each state is rated by the evaluation function or the **fitness function**.

In (c),a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b).

Figure describes the algorithm that implements all these steps.

```
function GENETIC_ALGORITHM( population, FITNESS-FN) return an
        individual input: population, a set of individuals
                FITNESS-FN, a function which determines the quality of the
        individual repeat
                new_population←empty set
                loop for ifrom 1 to SIZE(population) do
               x ←RANDOM_SELECTION(population, FITNESS_FN)
         y ←RANDOM_SELECTION(population,
                            FITNESS_FN)
                  child ←REPRODUCE(x,y)
                        if (small random probability) then child
                          MUTATE(child ) add child to new_population
                 population ←new_population
        until some individual is fit enough or enough time has elapsed
        return the best individual
```

Figure 2.28 A genetic algorithm.