LOCAL SEARCH ALGORITHMS

Chapter 4

Outline

- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms

Local search algorithms

- In many optimization problems
  - the path from the **start node** to the **goal** **is irrelevant**
  - the **goal state** itself is the **solution**

- **State space** = set of **"complete" configurations**
- Find **configuration** satisfying constraints

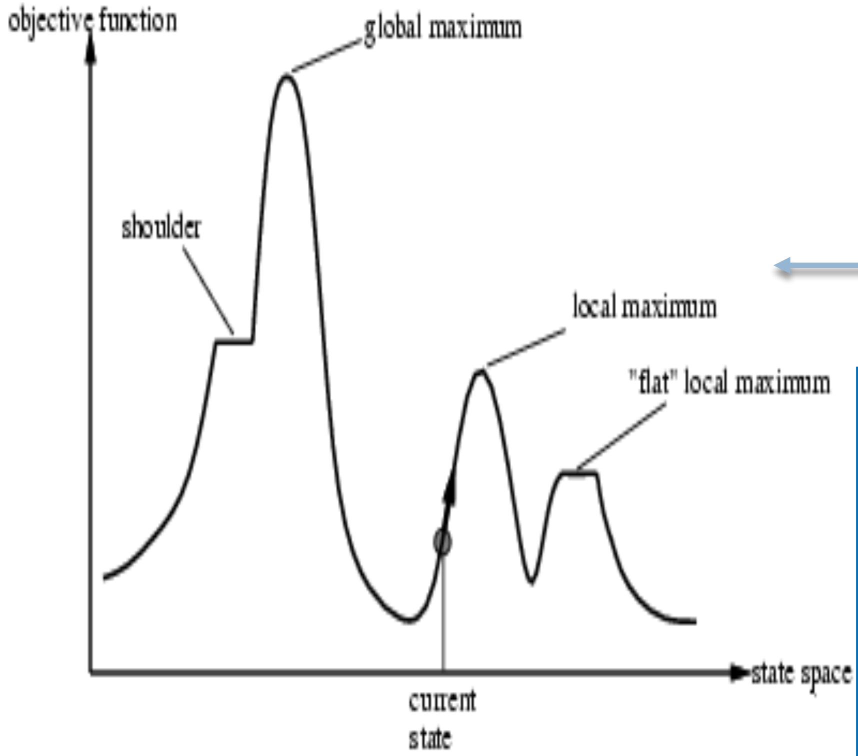
- In such cases, we can use local search algorithms
  - keep a **single "current" state**
  - try to **improve it**

Example: *n*-queens

- **Put** ***n* queens** on an $n \times n$ board with **no two queens** on the **same** row, column, or diagonal
  - **Each state** has **n queens** on the board, **one per column**
  - **Successors** of a state: **all possible states** generated by moving a single queen to another square in the same column

State-space landscape

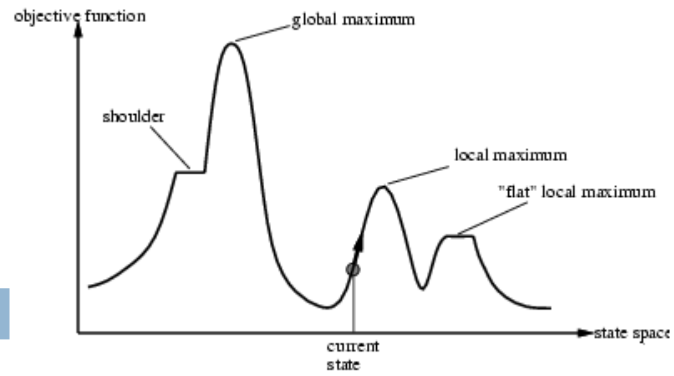
Local search algorithms explore the **state-space landscape**



A **landscape** has
- **location** (defined by the **state**)
- **elevation** (defined by the **value** of heuristic cost function **or** objective function)

The aim is to find:
- **a global minimum** (lowest valley)
  if **elevation** corresponds to cost
- **a global maximum** (highest peak)
  if **elevation** corresponds to objective function

Hill-climbing search

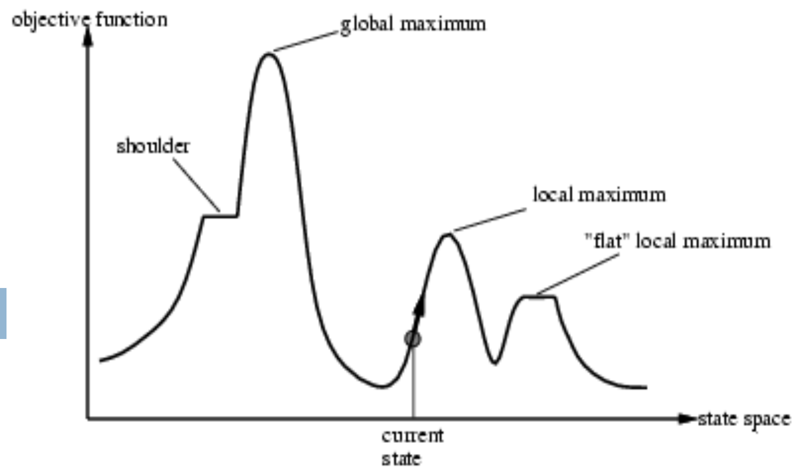


- Assume the **elevation** corresponds to the **objective function**
- **Hill-climbing search** modifies the **current state** to try to **improve it**

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    current ← MAKE-NODE(problem.INITIAL-STATE)
    loop do
        neighbor ← a highest-valued successor of current
        if neighbor.VALUE ≤ current.VALUE then return current.STATE
        current ← neighbor
```

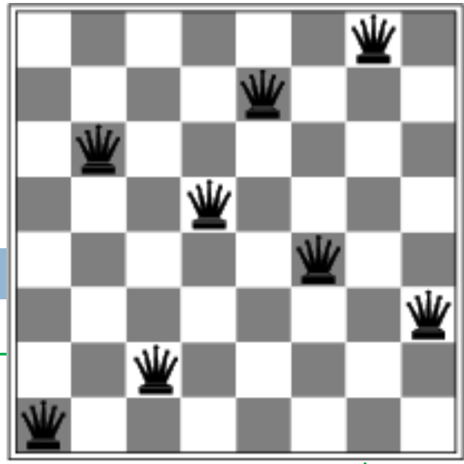
- Picks **a neighbor** with the highest value
- Usually **chooses at random** among neighbors with maximum value
- Terminates when it reaches a "peak" where no neighbor has a higher value

Hill-climbing search



- **Hill climbing** **often gets stuck** for the following reasons:
  - **Local maxima**: a **peak** that is
    - **higher** than each of its neighboring states
    - **lower** than the global maximum
  - **Plateaux**: a **flat area** of the state-space landscape
    - **flat local maximum**, from which no progress is possible
    - **shoulder**, from which progress is possible

Hill-climbing search



Eg.: 8-queens

Starting from a randomly generated 8-queens state

- **Hill climbing** **gets stuck 86%** of the time
- **Solving only 14%** of problem instances
- It **works quickly**, taking just **4 steps** on average when it succeeds
  and **3 steps** when it gets stuck,
  even if the **state space** with ≈ **17 million states**

Allowing up to**100** consecutive **sideways moves**:

- **Solving 94%** of problem instances
- The algorithm averages **21 steps** for each successful instance and
  64 steps for each failure

Hill climbing variants

- **Stochastic hill climbing**
  - chooses **at random** from the set of **all improving neighbors**
- **First-choice hill climbing**
  - jumps to the **first improving neighbor** found
- **Random-restart hill climbing**
  - **series of hill climbing** runs until a goal is found
  - It will find a good solution very quickly

    Eg., For three million queens, it can find solutions in under a minute

Hill climbing and Random Walk

- **Hill-climbing algorithm** that ***never*** makes "downhill" moves toward states with lower value is incomplete
  - because it can get stuck on a local maximum

- **Purely random walk**: moving to a **successor** chosen **uniformly at random** from the set of successors is complete but extremely inefficient

- **Idea**: to **combine** hill climbing with a random walk that yields both **efficiency** and **completeness**

Simulated annealing search

- **Simulated annealing:** a version of stochastic hill climbing where some downhill moves are allowed
  - **If** the **move improves** the situation → **always accepted**
  - **Otherwise** → **accepted** with **some probability** less than 1

- Downhill moves
  - accepted early in the annealing schedule
  - then accepted less often as time goes on

Simulated annealing search

Idea: **escape local maxima** by allowing some "bad" moves but gradually decrease their frequency

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs:   problem, a problem
             schedule, a mapping from time to "temperature"
   current ← MAKE-NODE(problem.INITIAL-STATE)
   for t =1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0  then   current ← next
      else              current ← next only with probability e^(ΔE/T)
```

The **schedule input** determines the value of the temperature T as a function of time

Simulated annealing search

Idea: **escape local maxima** by allowing some "bad" moves but gradually decrease their frequency

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
   inputs:  problem, a problem
            schedule, a mapping from time to "temperature"

   current ← MAKE-NODE(problem.INITIAL-STATE)
   for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0  then  current ← next
      else  current ← next only with probability e^(ΔE/T)
```

The **higher the temperature** the **higher the probability** of making a **non-improving move**

Properties of simulated annealing search

- One can prove: If *T* decreases slowly enough,

  then **simulated annealing search** will find a global optimum with probability approaching 1

- Simulated annealing widely used in
  - airline scheduling
  - large-scale optimization tasks etc

Local beam search

- **Keep track** of ***k*** **states** rather than just one
- **Start** with ***k*** **randomly** generated **states**
- **At each iteration** **all** **the successors of** **all** ***k*** **states** are generated
  - **If** any one is a **goal state**, algorithm **stops**
  - **Else select** the ***k*** **best** **successors** from the complete list and **repeat** (they could be all successors of the same node)

Genetic algorithms

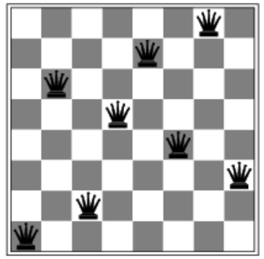
- A **successor state** is generated **by combining two parent states** rather than by modifying a single state

- **Start** with ***k*** **randomly** generated **states** (population)
  - A **state** is represented as a string over a finite alphabet (often a string of 0s and 1s or digits)

- **Each state** is associated to **a value** via an **evaluation function** (fitness function)
  - **Returns higher** values for **better** states

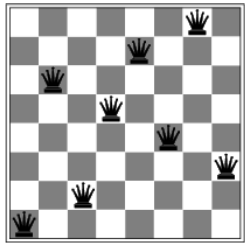
- **The next generation of states** is produced by selection, crossover, and mutation

Genetic algorithms



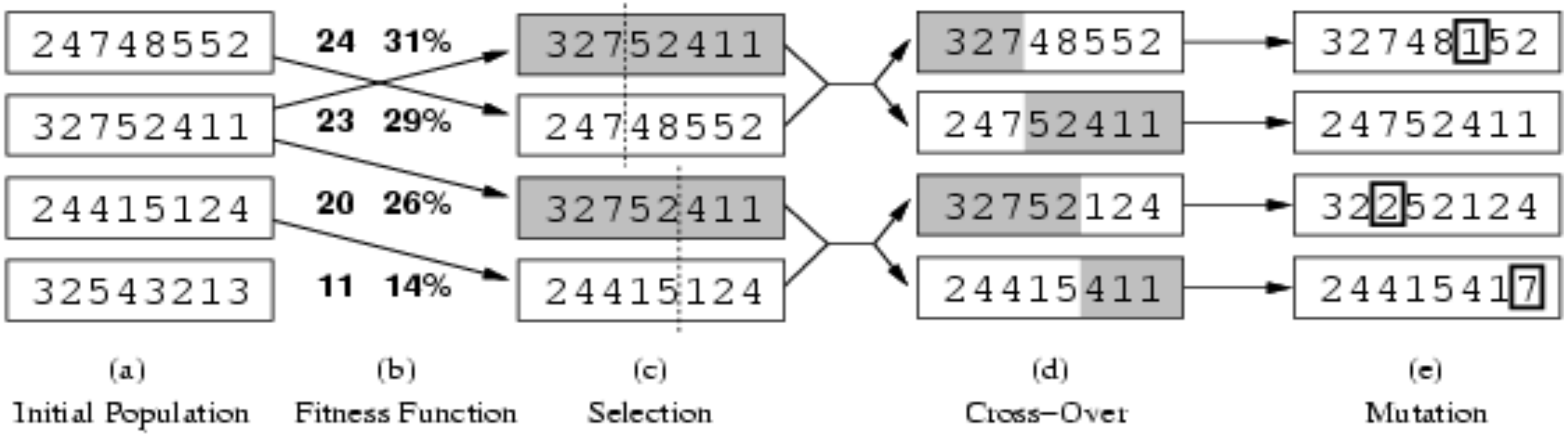
- States: strings of 8 digits representing **8-queens states**
- Fitness function:
  - High values for better states
  - We use the **number of non-attacking pairs of queens**
  - The higher the fitness the more likely the node is to be selected for reproductions

Genetic algorithms



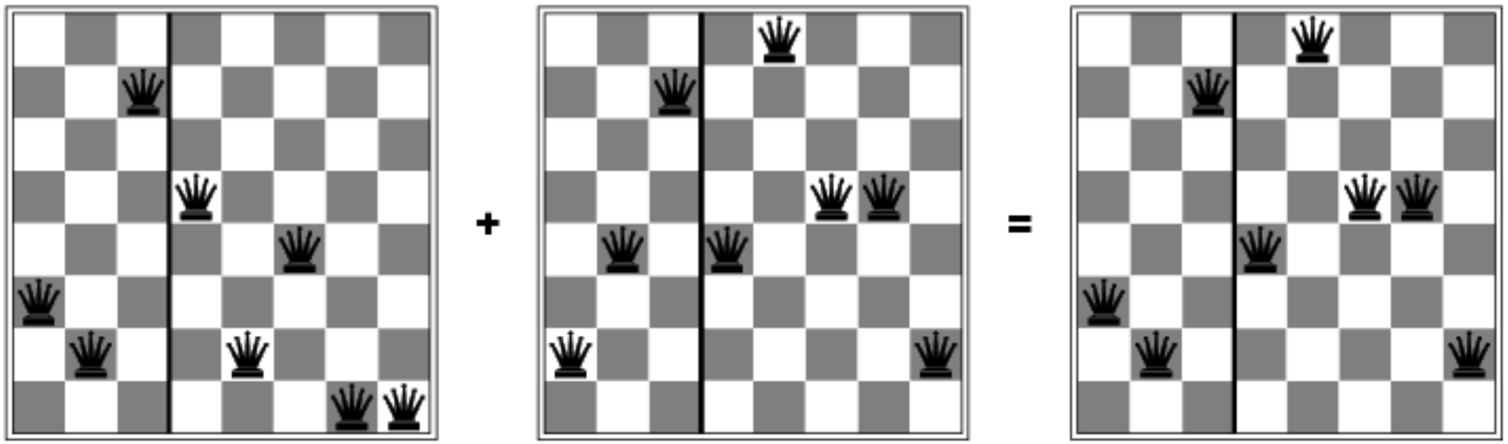
- For each pair to be mated, a crossover point is chosen randomly from the positions in the string

- Each location is subject to random mutation with a small independent probability



The 8-queens **states** corresponding to:

the first two parents in (c) the first child in (d)



3 2 7 | 5 2 4 1 1

2 4 7 | 4 8 5 5 2

3 2 7 | 4 8 5 5 2