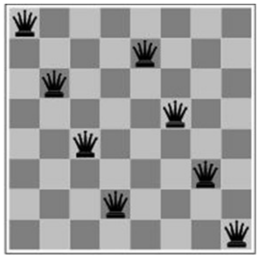
Local Search and Optimization

Outline

- Local search techniques and optimization
  - Hill-climbing

Local search and optimization

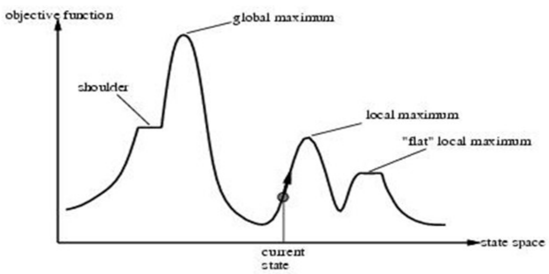
- Previously: systematic exploration of search space.
  - Backtrack search
  - Can solve n-queen problems for n = 200
- Different algorithms can be used
  - Local search
  - Can solve n-queen for n = 1,000,000



Local search and optimization

- Local search
  - Keep track of single current state
  - Move only to neighboring states
  - Ignore paths
- Advantages:
  - Use very little memory
  - Can often find reasonable solutions in large or infinite (continuous) state spaces.
- "Pure optimization" problems
  - All states have an objective function
  - Goal is to find state with max (or min) objective value
  - Does not quite fit into CSP formulation
  - Local search can do quite well on these problems.

"Landscape" of search for max value



Hill-climbing search

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*

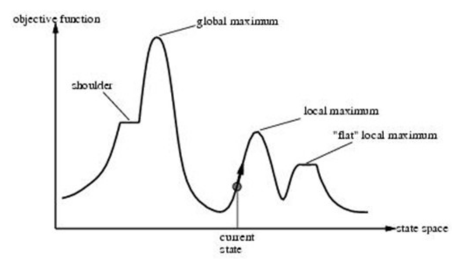
This version of HILL-CLIMBING found local maximum.

Hill-climbing search

- "a loop that continuously moves in the direction of increasing value"
  - terminates when a peak is reached
  - Aka greedy local search
- Value can be either
  - Objective function value
  - Heuristic function value (minimized)
- Hill climbing does not look ahead of the immediate neighbors of the current state.
- Can randomly choose among the set of best successors, if multiple have the best value
- Characterized as "trying to find the top of Mount Everest while in a thick fog"

Hill climbing and local maxima

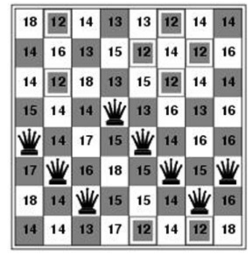
- When local maxima exist, hill climbing is suboptimal
- Simple (often effective) solution
  - Multiple random restarts



Hill-climbing example

- 8-queens problem, complete-state formulation
  - All 8 queens on the board in some configuration
- Successor function:
  - move a single queen to another square in the same column.
- Example of a heuristic function *h(n)*:
  - the number of pairs of queens that are attacking each other (directly or indirectly)
  - (so we want to minimize this)

Hill-climbing example

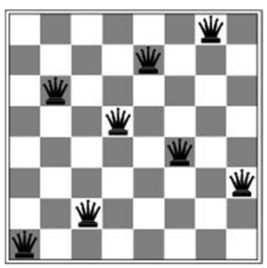


$(c_1\ c_2\ c_3\ c_4\ c_5\ c_6\ c_7\ c_8) = (5\ 6\ 7\ 4\ 5\ 6\ 7\ 6)$

Current state: h=17

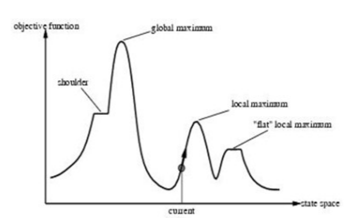
Shown is the h-value for each possible successor in each column

A local minimum for 8-queens



A local minimum in the 8-queens state space (h=1)

Other drawbacks



- Ridge = sequence of local maxima difficult for greedy algorithms to navigate
- Plateau = an area of the state space where the evaluation function is flat.

Performance of hill-climbing on 8-queens

- Randomly generated 8-queens starting states…
- 14% the time it solves the problem
- 86% of the time it get stuck at a local minimum
- However…
  - Takes only 4 steps on average when it succeeds
  - And 3 on average when it gets stuck
  - (for a state space with ~17 million states)

Possible solution…sideways moves

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
  - Need to place a limit on the possible number of sideways moves to avoid infinite loops
- For 8-queens
  - Now allow sideways moves with a limit of 100
  - Raises percentage of problem instances solved from 14 to 94%
  - However….
    - 21 steps for every successful solution
    - 64 for each failure

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
  - stochastic hill climbing by generating successors randomly until a better one is found
  - Useful when there are a very large number of successors
- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima.

Hill-climbing with random restarts

- Different variations
  - For each restart: run until termination v. run for a fixed time
  - Run a fixed number of restarts or run indefinitely
- Analysis
  - Say each search has probability p of success
    - E.g., for 8-queens, p = 0.14 with no sideways moves
  - Expected number of restarts?
  - Expected number of steps taken?

Search using Simulated Annealing

- Simulated Annealing = hill-climbing with non-deterministic search
- Basic ideas:
  - like hill-climbing identify the quality of the local improvements
  - instead of picking the best move, pick one randomly
  - say the change in objective function is δ
  - if δ is positive, then move to that state
  - otherwise:
    - move to this state with probability proportional to δ
    - thus: worse moves (very large negative δ) are executed less often
  - however, there is always a chance of escaping from local maxima
  - over time, make it less likely to accept locally bad moves
  - (Can also make the size of the move random as well, i.e., allow "large" steps in state space)

Physical Interpretation of Simulated Annealing

- Annealing = physical process of cooling a liquid or metal until particles achieve a certain frozen crystal state
  - simulated annealing:
    - free variables are like particles
    - seek "low energy" (high quality) configuration
    - get this by slowly reducing temperature T, which particles move around randomly

Simulated annealing

function SIMULATED-ANNEALING(*problem, schedule*) **return** a solution state
  input: *problem*, a problem
    schedule, a mapping from time to temperature
  local variables: *current***,** a node.
    *next***,** a node.
    *T***,** a "temperature" controlling the probability of downward steps

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  for t ← 1 to ∞ do
    T ← *schedule*[*t*]
    if *T* = *0* **then return** *current*
    next ← a randomly selected successor of *current*
    ΔE ← VALUE[*next*] - VALUE[*current*]
    if *ΔE* > 0 **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E / T}$

More Details on Simulated Annealing

- Lets say there are 3 moves available, with changes in the objective function of d1 = -0.1, d2 = 0.5, d3 = -5. (Let T = 1).
- pick a move randomly:
  - if d2 is picked, move there.
  - if d1 or d3 are picked, probability of move = exp(d/T)
  - move 1: prob1 = exp(-0.1) = 0.9,
    - i.e., 90% of the time we will accept this move
  - move 3: prob3 = exp(-5) = 0.05
    - i.e., 5% of the time we will accept this move
- T = "temperature" parameter
  - high T => probability of "locally bad" move is higher
  - low T => probability of "locally bad" move is lower
  - typically, T is decreased as the algorithm runs longer
    - i.e., there is a "temperature schedule"

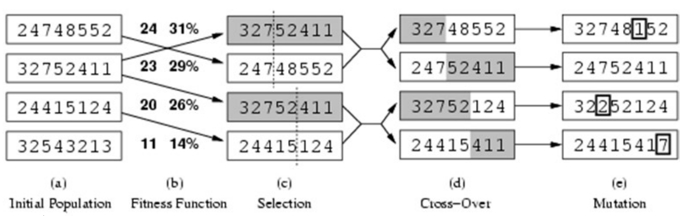
Simulated Annealing in Practice

- method proposed in 1983 by IBM researchers for solving VLSI layout problems (Kirkpatrick et al, *Science*, 220:671-680, 1983).
  - theoretically will always find the global optimum (the best solution)
- useful for some problems, but can be very slow
  - slowness comes about because T must be decreased very gradually to retain optimality
  - In practice how do we decide the rate at which to decrease T? (this is a practical problem with this method)

Genetic algorithms

- Different approach to other search algorithms
  - A successor state is generated by combining two parent states
  -
- A state is represented as a string over a finite alphabet (e.g. binary)
  - 8-queens
    - State = position of 8 queens each in a column
      => 8 x log(8) bits = 24 bits (for binary representation)
- Start with *k* randomly generated states (population)
- Evaluation function (fitness function).
  - Higher values for better states.
  -
  - Opposite to heuristic function, e.g., # non-attacking pairs in 8-queens
- Produce the next generation of states by "simulated evolution"
  - Random selection
  - Crossover
  - Random mutation
  -

Genetic algorithms



- 23/(24+23+20+11) = 29% etc
-

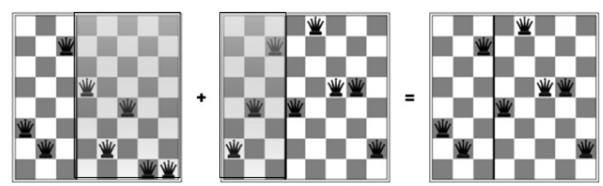
4 states for 8-queens problem

2 pairs of 2 states randomly selected based on fitness. Random crossover points selected

New states after crossover

Random mutation applied

Genetic algorithms



Has the effect of "jumping" to a completely different new part of the search space (quite non-local)

Genetic algorithm pseudocode

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 i **from** 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

Comments on genetic algorithms

- Positive points
  - Random exploration can find solutions that local search can't
    - (via crossover primarily)
  - Appealing connection to human evolution
    - E.g., see related area of genetic programming

- Negative points
  - Large number of "tunable" parameters
    - Difficult to replicate performance from one problem to another

  - Lack of good empirical studies comparing to simpler methods

  - Useful on some (small?) set of problems but no convincing evidence that GAs are better than hill-climbing w/random restarts in general