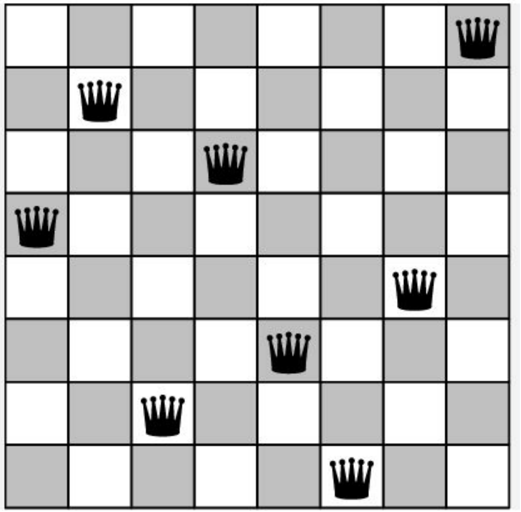
Artificial Intelligence

Lec 7: Local Search (contd.)

Pratik Mazumder

Content taken from various sources including
CS188 Intro to AI at UC Berkeley

Local Search: Hill-climbing (Greedy Local Search)



- Suppose you are at a state with multiple neighbors.
- For each neighbor, you have the objective function value.
- Strategy:
    - You can pick the neighbor with the best objective function and repeat
    - But then, when will you stop?
        - If **none** of the neighbours have a **lower objective function value** then stop [Case: when objective function has to be minimized]
        - If **none** of the neighbours have a **higher objective function value** then stop [Case: when objective function has to be maximized]
- Hill-climbing or Greedy Local Search

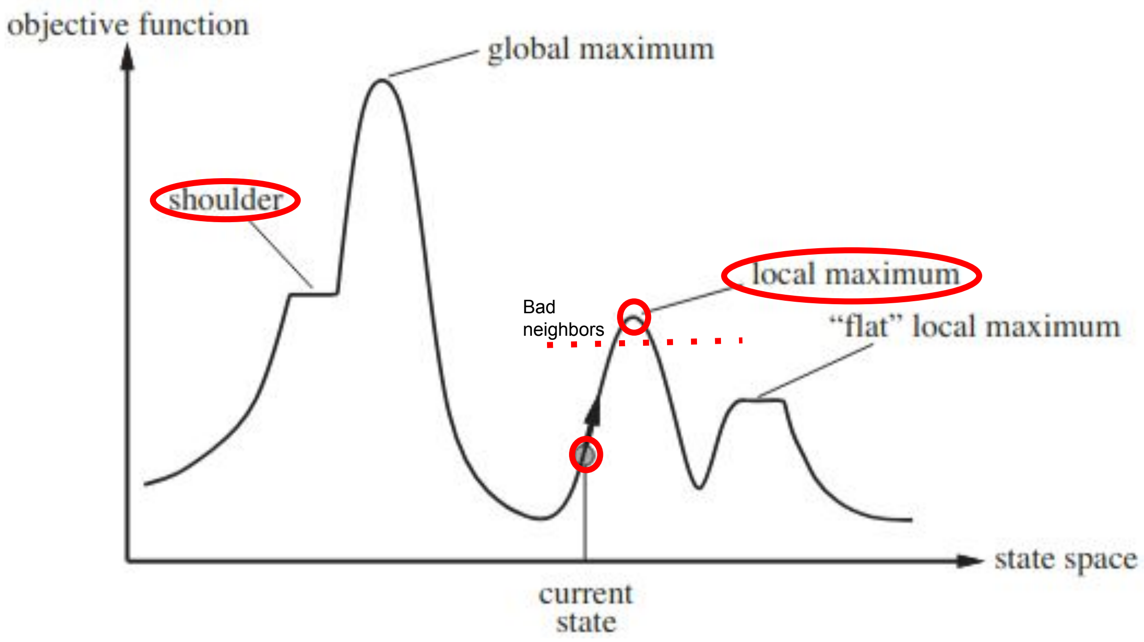
Hill-climbing (Greedy Local Search)

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

$current \leftarrow$ MAKE-NODE($problem$.INITIAL-STATE)
loop do
  $neighbor \leftarrow$ a highest-valued successor of $current$
  if neighbor.VALUE $\leq$ current.VALUE **then return** $current$.STATE
  $current \leftarrow neighbor$

Hill-climbing (Greedy Local Search)

- Is it always possible to reach the global maximum?
  - Depends on where you start.



Hill Climbing gets stuck at a local maxima (or local minima)

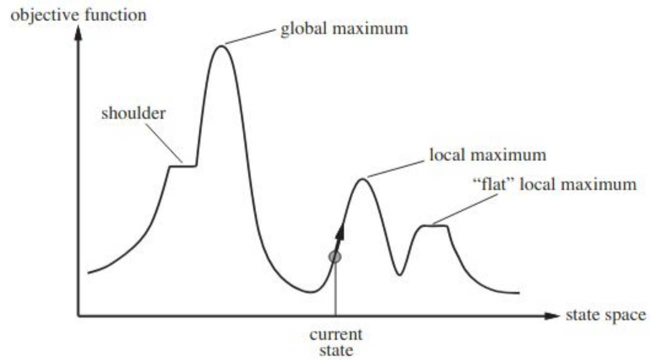
Escaping Shoulders: Sideways Move

- If no downhill (or uphill) moves, **allow sideways moves, hoping that the algorithm escapes.**

- Allow movement to **the next state** even if that has **the same objective function value.**

- Need to place a **limit** on the possible **number** of **sideways moves** to avoid infinite **loops**.

- For 8-queens,
  - Allow sideways moves with a limit of say 100
  - Raises the percentage of problem instances solved from 14% to 94%.
  - However, with this approach, an average of 21 steps are needed for successful solutions and 64 steps for failures.

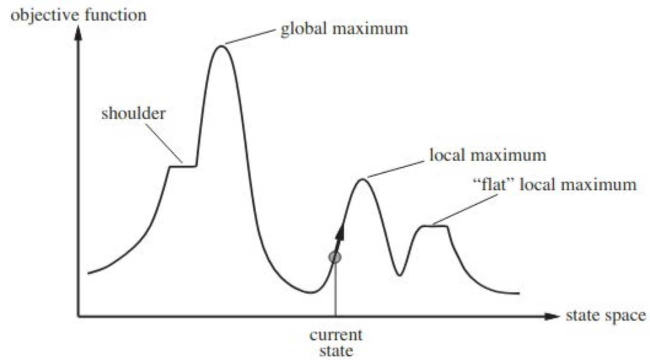


function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

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*

neighbor ← a highest-valued successor of *current*
if neighbor.VALUE < current.VALUE **then return** *current*.STATE
current ← *neighbor*

Escaping Shoulders: Sideways Move



- Allow movement to the next state, even if that has the same objective function value.

- Problem with allowing Sideways move:

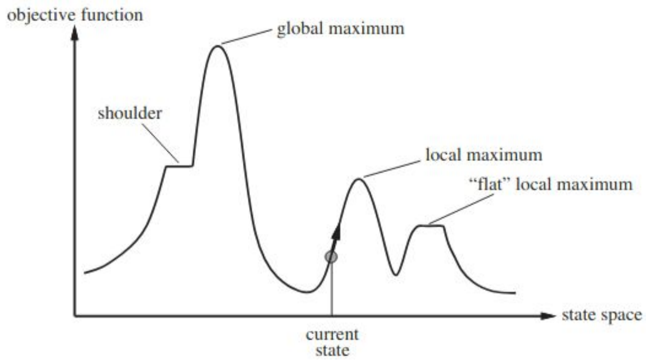
  - May keep oscillating
    - A to B then back to A then B
  - Or oscillate between a set of states

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

$\quad$ *current* ← MAKE-NODE(*problem*.INITIAL-STATE)
$\quad$ **loop do**
$\quad\quad$ *neighbor* ← a highest-valued successor of *current*
$\quad\quad$ **if** neighbor.VALUE ≤ current.VALUE **then return** *current*.STATE
$\quad\quad$ *current* ← *neighbor*

$\quad\quad$ *neighbor* ← a highest-valued successor of *current*
$\quad\quad$ **if** neighbor.VALUE < current.VALUE **then return** *current*.STATE
$\quad\quad$ *current* ← *neighbor*

Escaping Shoulders: Sideways Move



- Allow movement to the next state, even if that has the same objective function value.

- Problem with allowing Sideways move:
    - May keep oscillating
        - A to B then back to A then B
    - Or oscillate between a set of states

- Solution: Reduce the "Amnesia" a little bit and use some memory

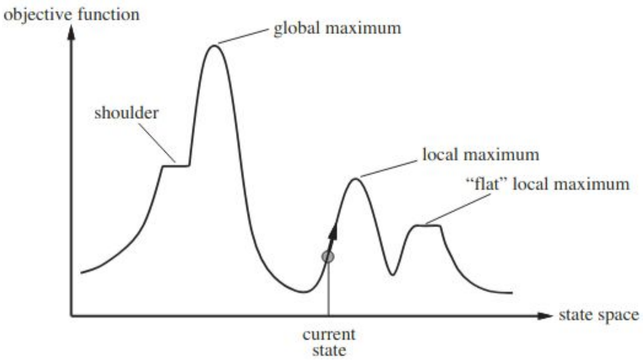
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

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*

  neighbor ← a highest-valued successor of *current*
  if neighbor.VALUE < current.VALUE **then return** *current*.STATE
  current ← *neighbor*

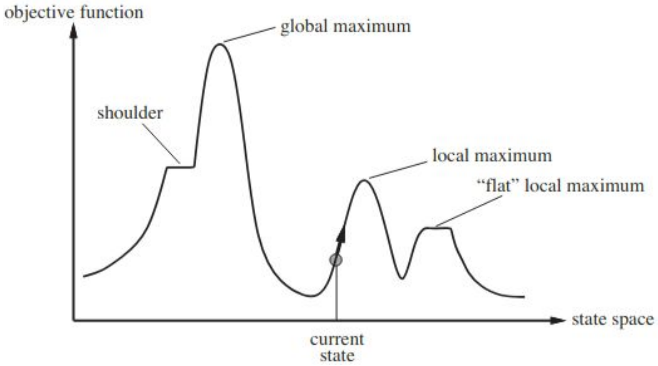
Escaping Shoulders: Tabu Search

- Prevent returning quickly to the same state.
- Maintain a fixed length queue (“tabu list").
- Add the most recent state to a queue.
    - and drop the oldest from the queue in case of full memory.
- **Never make the step that is currently tabu'ed.**
- Properties
    - As the size of the **tabu list grows**, **hill-climbing** will become **like** a **systematic search** algorithm (basically, don’t repeat any state).
    - In practice, a reasonable sized tabu list (around 100) improves the performance of hill climbing in many problems.



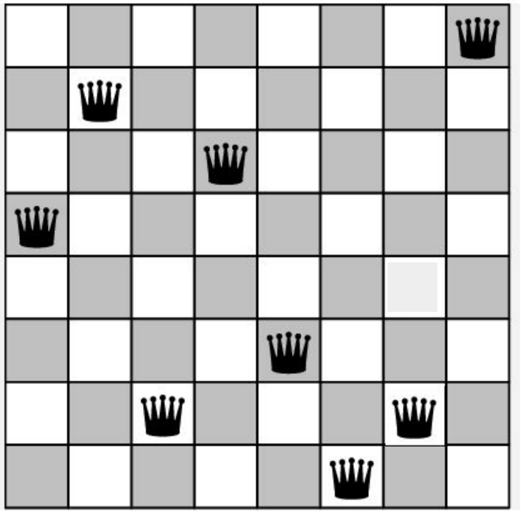
Escaping Shoulders: Enforced Hill Climbing

- Perform **BFS from a local optima**
  - To find the **nearest state** with a **better** objective function **value**
  - Effectively adding slightly larger memory
- Typically,
  - Prolonged periods of exhaustive search
  - Bridged by relatively, quick periods of hill-climbing
- Middle ground b/w local and systematic search
- Termination can be tricky -
  - So can limit with time of search



Trivial Algorithms

- Random Sampling
  - Generate a state randomly.
- Random Walk
  - Randomly pick a neighbor of the current state.
- Ignoring memory and time restrictions, these are complete.
- Greedy Local Search/Hill climbing will not necessarily find a solution.
- Can we combine these approaches?



Stochastic Hill-climbing

- Goal: Avoid getting stuck in local minima.
- Strategies:
    - Random selection among uphill moves
    - The selection probability can vary with the steepness of the uphill move.
- Random-walk hill-climbing
- Random-restart hill-climbing
- Hill-climbing with both

Hill-climbing with Random Walk

- Idea: Combine Hill-climbing with Random Walk

- At each step, perform **one of the two**
  - **Greedy** - With probability p move to **neighbour** with the **largest** evaluation function **value** (in case of maximising).

  - **Random** - With probability 1-p move to a **random neighbour.**

- Which is more ideal? p constant or variable

  - **Initially**, it may be okay to take **more random** steps. Why?
    - May have started very **close** to **local maxima** (if the goal is maximization).

  - **As time goes on**, we should take more and **more greedy** steps.

  - So **p** should **increase with time**. What happens then?

Hill-climbing with Random Restarts

- Approach:
    - Perform hill-climbing.
    - If you get stuck at a local optima, randomly jump to a new state and perform hill-climbing.
    - Keep the best solution found so far.
- Variations
    - For each restart: run till the greedy optimum and restart, or run for a fixed time and restart.
    - Run a fixed number of restarts or run indefinitely.

Hill-climbing with Both

- At each step, perform one of the three with some probability.
  - Greedy - move to neighbour with the largest evaluation function value.
  - Random Walk - Move to a random neighbour.
  - Random Restart - Resample a new current state.

Simulated Annealing

- Idea
  - **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 value is δ (= newobjval - oldobjval)

- If δ is positive, then move to that state (when we want to maximize the objective value).

- Otherwise:
  - Move to this state with a probability proportional to δ

  - Therefore, worse moves (very large negative δ) are executed less often.

- However, there is always a chance of escaping local maxima.

  - Therefore, overtime, make it less likely to accept locally bad moves.

Simulated Annealing

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**
    $T \leftarrow schedule(t)$
    if T = 0 **then return** *current*
    next ← a randomly selected successor of *current*
    $\Delta E \leftarrow next.$VALUE – $current.$VALUE
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $e^{\Delta E/T}$

The algorithm shows that the simulated annealing algorithm is **a version of stochastic hill climbing** where **some downhill moves are allowed (in case the objective is to maximize the objective function value)**.

- **Downhill moves** are **accepted** readily **early** in the annealing schedule and then **less often as time goes on.**
- The **schedule input** determines the **value of the temperature T as a function of time**.
- Symbolic meaning of T:
  - If the **temperature** is **low**, we are stable and can decide properly based only on **greedy** steps.
  - If the **temperature** is **too low**, we are frozen and can't move, then the **current state is the solution.**
  - If the **temperature** is **high**, we are unstable and may make bad decisions, so **allow downhill moves**.

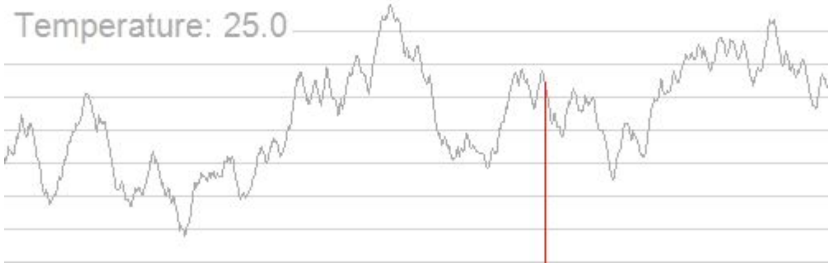
Simulated Annealing

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

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

Temperature T

- 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



Physical Interpretation of Simulated Annealing

- A Physical Analogy:
    - imagine letting a ball roll downhill on the function surface
        - this is like hill-climbing (for minimization)
    - now imagine shaking the surface, while the ball rolls, gradually reducing the amount of shaking
        - this is like simulated annealing

Local beam search

- Idea: Keeping only one node in memory is an extreme reaction to memory problems
- Keep track of k states instead of one [Beam of size k]
    - Initially: k randomly selected states
    - Next: determine all successors of k states
    - If any of the successors is a goal then finish
    - Else select k best from successors and repeat

Local beam search

- Not the same as k random-start searches run in parallel!
- Searches that find good states recruit other searches to join them

- Problem: quite often, all k states end up on same local hill

- Idea: Stochastic beam search

    - Choose k successors randomly, biased towards good ones
        - better valued states have higher probability of being selected
        - lesser valued states may also get selected but with a lower probability

    - Improves local beam search by introducing a degree of randomness into the selection process, maintaining diversity among candidate solutions, and reducing the risk of premature convergence to local optima.

- Observe the close analogy to natural selection!