Local Search

Lecture 6, CMSC 170

John Roy Daradal / Instructor

Previously on CMSC 170

Constraint Satisfaction Problems:

- Variables, Domains, Constraints
- Backtracking Search
- Forward Checking
- Variable, Value Ordering
- Avoiding Thrashing

Today's Topics

- Local Search
- Hill Climbing
- Simulated Annealing
- Tabu Search

Constrained Problems

Satisfaction

- **all** constraints must be **satisfied**

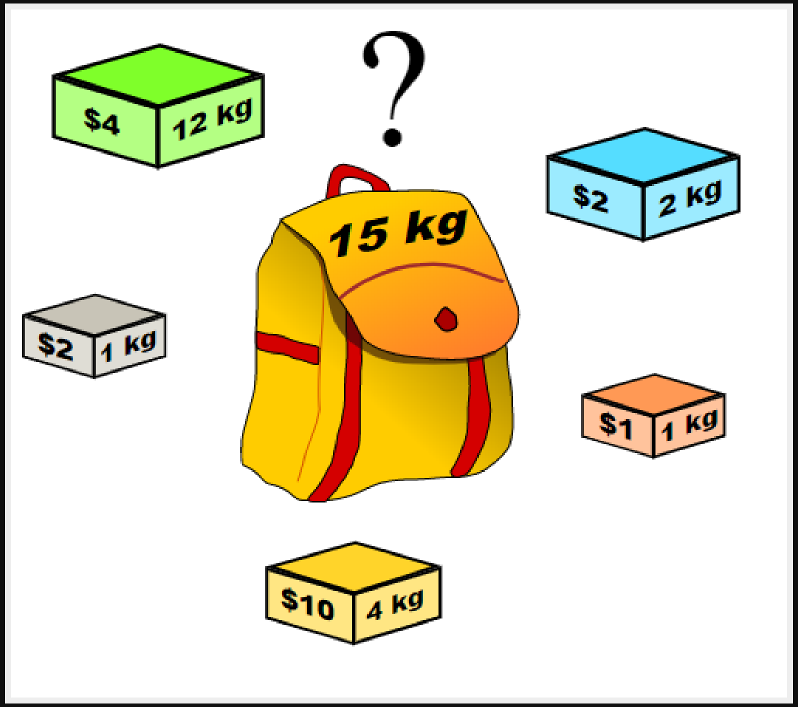
Optimization

- *not all* constraints might be satisfied
- find solution that **minimizes** *penalty*

CSP & COP Examples

Knapsack Problem

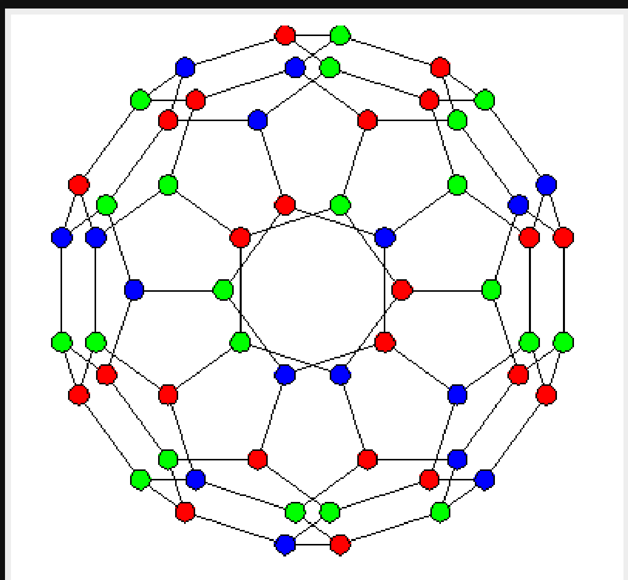
- choose items that will **fit** in knapsack
- **maximize value**



CSP & COP Examples

Graph Coloring

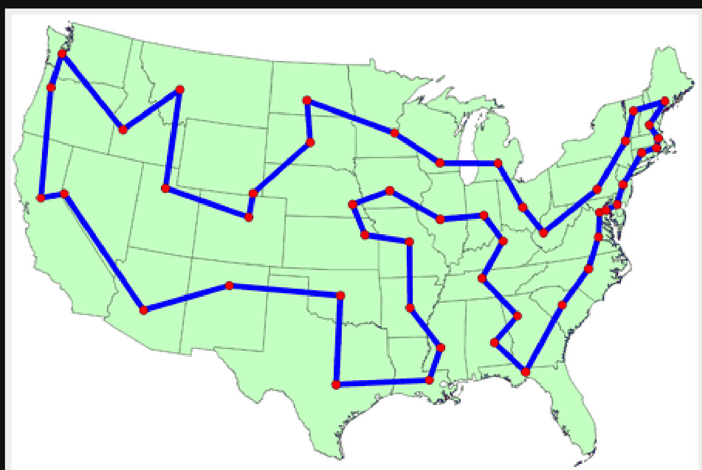
- no two adjacent vertices **share** same color
- **minimize** number of colors



CSP & COP Examples

Traveling Salesman

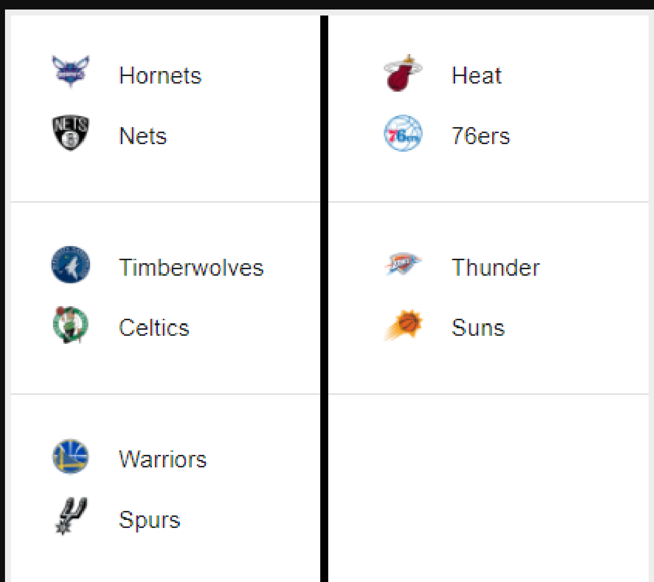
- find **shortest route** that visits each city and goes back to starting city



CSP & COP Examples

Traveling Tournament

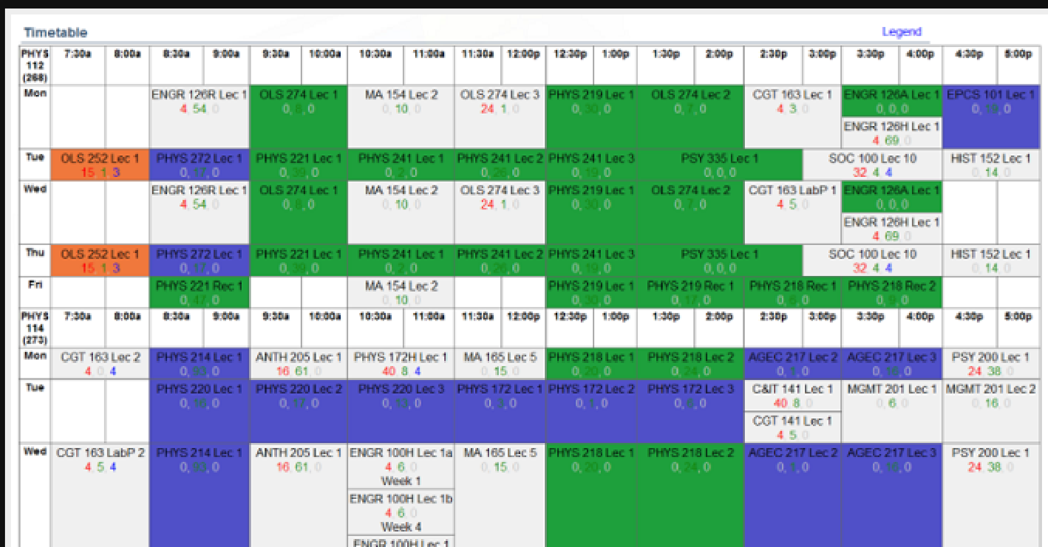
- schedule home & away games
- **minimize** total travel distance



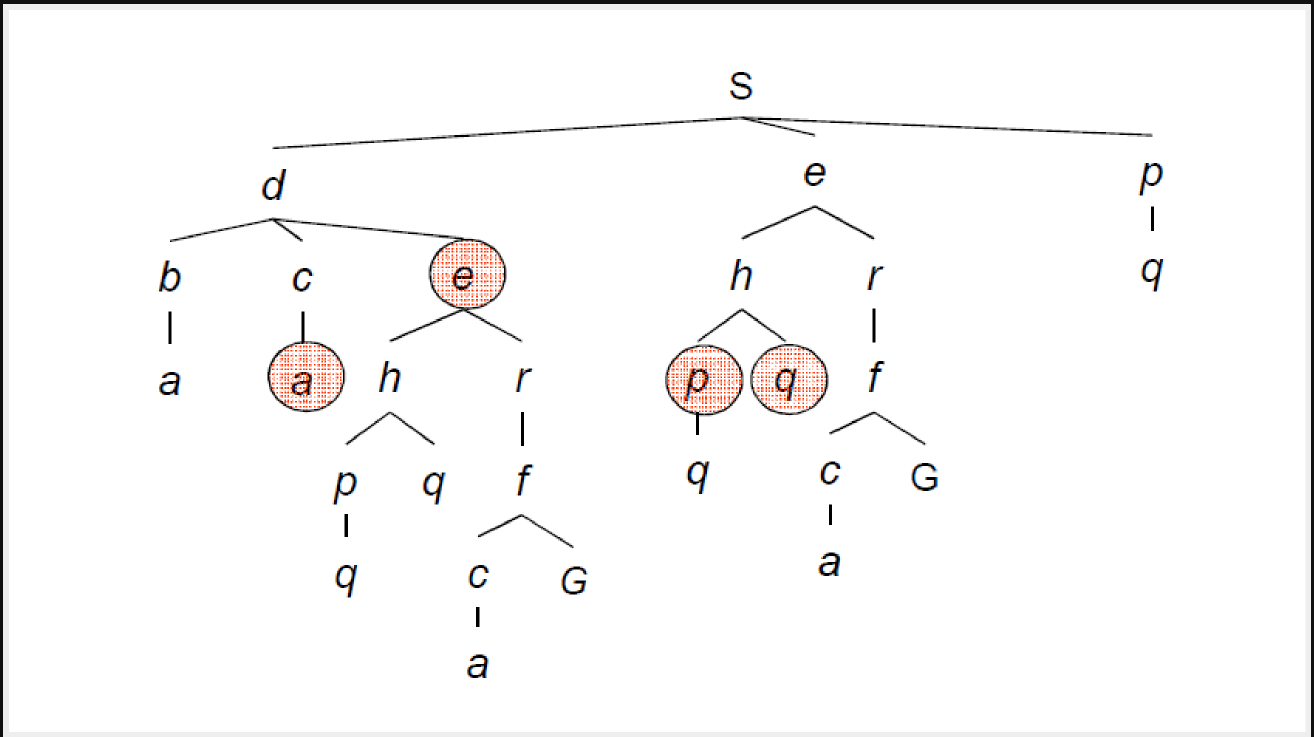
CSP & COP Examples

Timetabling

- assign **classes** to room & timeslot
- accommodate schedule **preference**



Tree Search



Tree Search

- **Fringe**: keeps *unexplored* states to ensure completeness
- *Examples*: DFS, BFS, UCS, Greedy, A*, **Backtracking**

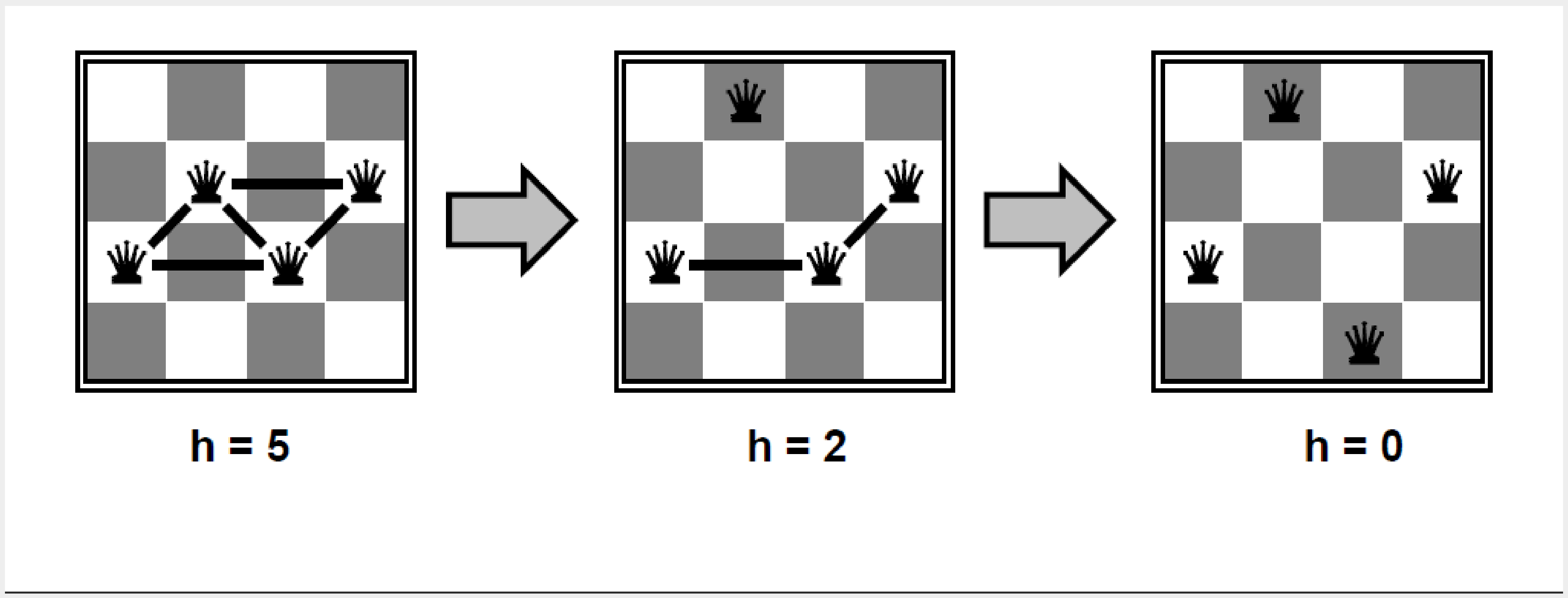
Backtracking Search

- **Partial** assignment
- **Extend** solution: add new *var=value*
- **Backtrack**: change *var=value* if it **fails**
- *Stop*: complete & correct solution found, or search tree exhausted
- For constraint **satisfaction** problems

Local Search

- **Complete** (invalid) assignment
- **Iteratively improve** (modify)
- **No fringe**: only keeps track of *one solution* (no *fallback*)
- *Stop*: can't improve current solution
- For constraint **satisfaction** & **optimization**

Example: N-Queens

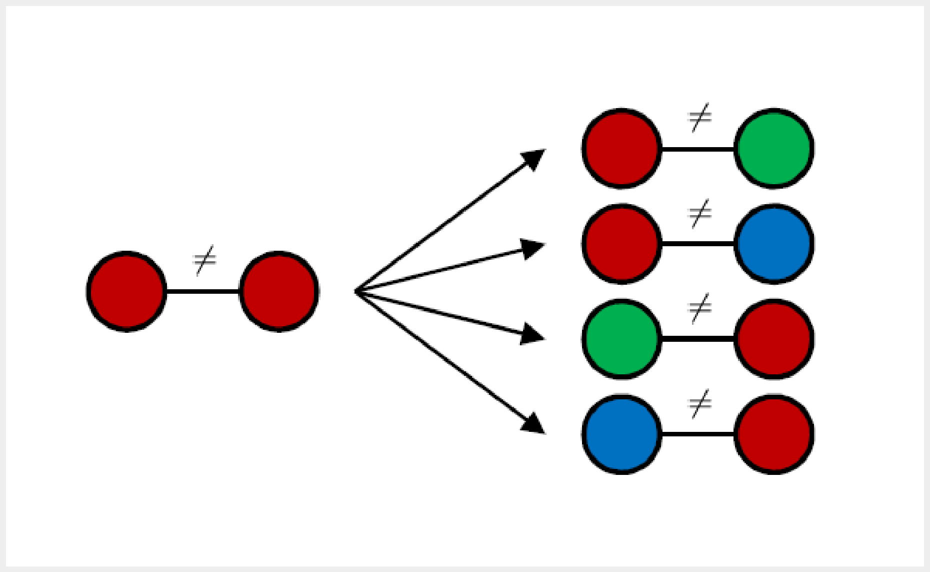


Local Search

- **Successor**: local changes
- **State**: solution
- **Neighbor**: current state + few *local* moves
- **Neighborhood**: candidate solutions
- *Modify* state, not *extend*

Local Moves

Describes how *state* will **change** per *iteration*



Local Search

- **Neighborhood** function
- **Objective** function
- **Legal neighbor** function
- **Selection** function

Local Search

Neighborhood function

- **N**(state)
- describe **local move**
- define **neighborhood** of state

Neighborhood

- **Change 1** variable $\longrightarrow$ different value
- **Change 2** variables $\longrightarrow$ different values
- **Swap** values of 2 variables
- **bigger** local move = **bigger** neighborhood

Local Search

Objective function

- **f**(state)
- state **grader**
- gives *state* a **score** based on your *goal*
- *minimize* or *maximize*

Objective Function

- **CSP**: no. of violations (*goal* = 0)
- **COP**: total constraint penalty (*minimize*)
- **Knapsack**: total value of items (*maximize*)
- Problem-dependent

Local Search

Legal neighbor function

- **L**(states)
- which neighbors are **legal**?
- **filter** out unwanted neighbors
- usually based on *objective function* score

Legal Neighbors

- Select all
- f(nbor) $>$ f(state): *strictly increasing*
- f(nbor) $<$ f(state): *strictly decreasing*
- f(nbor) $\geq$ f(state): *non-decreasing*
- f(nbor) $\leq$ f(state): *non-increasing*
- No degradation

Local Search

Selection function

- **S**(states)
- **select** a legal neighbor
- sometimes *merged* with legal neighbor fn
- may be *offline* or *online*

Offline vs Online Selection

- **Offline**: generate the *whole* neighborhood before selecting
- **Online**: generate neighbors *one-by-one*; if legal neighbor found, select that
- *Tradeoff*: optimality vs time

Selection Function

- **Best**, based on *heuristics*
- **Multi-stage** heuristics
- **First**: no need to expore whole nhood
- **Random**: effective if nhood is *too big*

Local Search

```
function LocalSearch(f, N, L, S) {
    s := generateInitialSolution();
    s* := s;
    for k := 1 to MaxTrials do
        if satisfiable(s) ∧ f(s) < f(s*) then
            s* := s;
        s := S(L(N(s), s), s);
    return s*;
}
```

Initial Solution

- **Random** solution
- From **another method** (greedy, BT search)

Local Search Usage

1. **Create** solution from scratch
2. **Improve** solution from other approach

- *Example*: *BT* to solve **hard** constraints, *LS* to *lower* **soft** constraint penalty

SAT vs OPT

- **Satisfiability**: infeasible → feasible (*violations*)
- **Optimization**: suboptimal → optimal (*objective function*)

SAT vs OPT

- **SAT** as **OPT**: set constraint penalty = ∞
- **OPT** as **SAT**: repeated SAT, keep best

Example: Graph Coloring

- Find solution with k colors
- Remove one color, **C**
- Reassign **C** vertices some other color
- Found a solution with k-1 colors
- Repeat

Local Search Efficiency

- *Not guaranteed* to finish quickly
- In practice, **acceptable** running time
- Can set **max iterations**
- **Anytime algorithm**: *stop anytime* and get a solution

Local Search

In *general*:

- **Faster**, better **memory** than tree search
- **Incomplete** and **suboptimal**
- *Guaranteed* to find **local optimum**
- *Not guaranteed* to find **global optimum**

Suboptimality

- Making *locally optimal* move **does not guarantee** *globally optimal* move
- Performance depends on **initial solution** & selecting **neighbors**
- *Improvement*: **random restarts**

Local Search Techniques

Min-Conflict

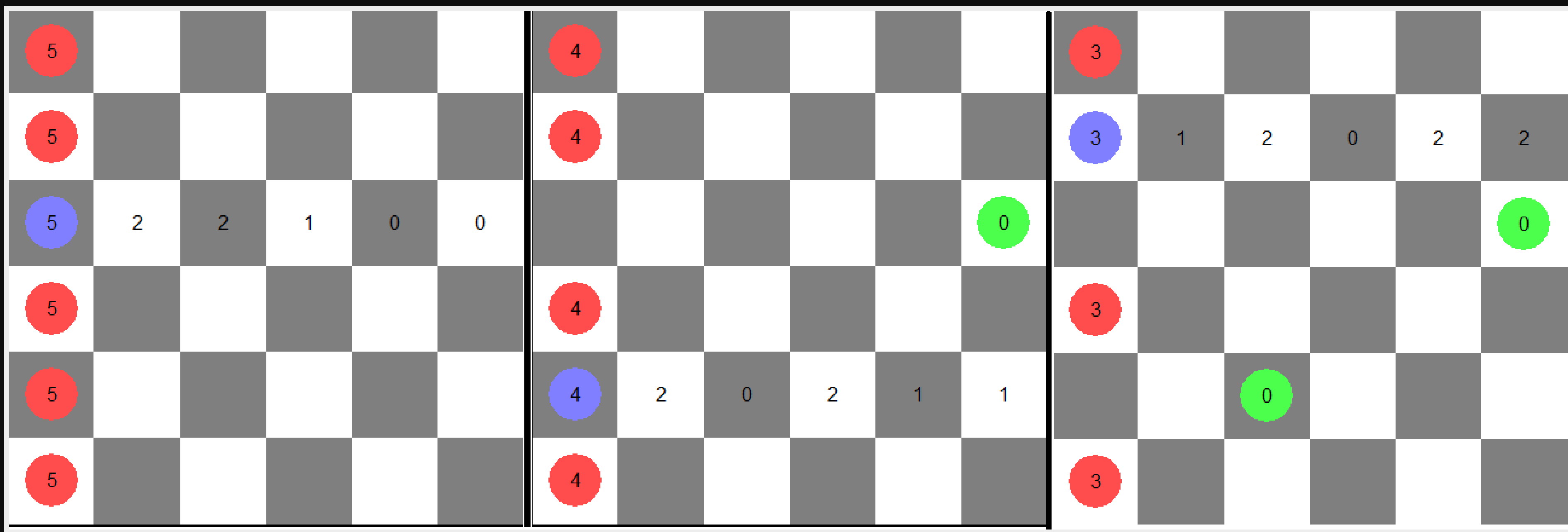
- Used for **CSP**
- **Neighborhood**: change 1 var = value
- **Objective fn**: count violations
- **Legal neighbors**: select all

Min-Conflict Heuristic

Selection = **multi-stage** heuristic:

- **Variable**: choose *random* variable
- **Value**: assign value that *minimizes* var's violations (**min-conflict**)

Min-Conflict Heuristic

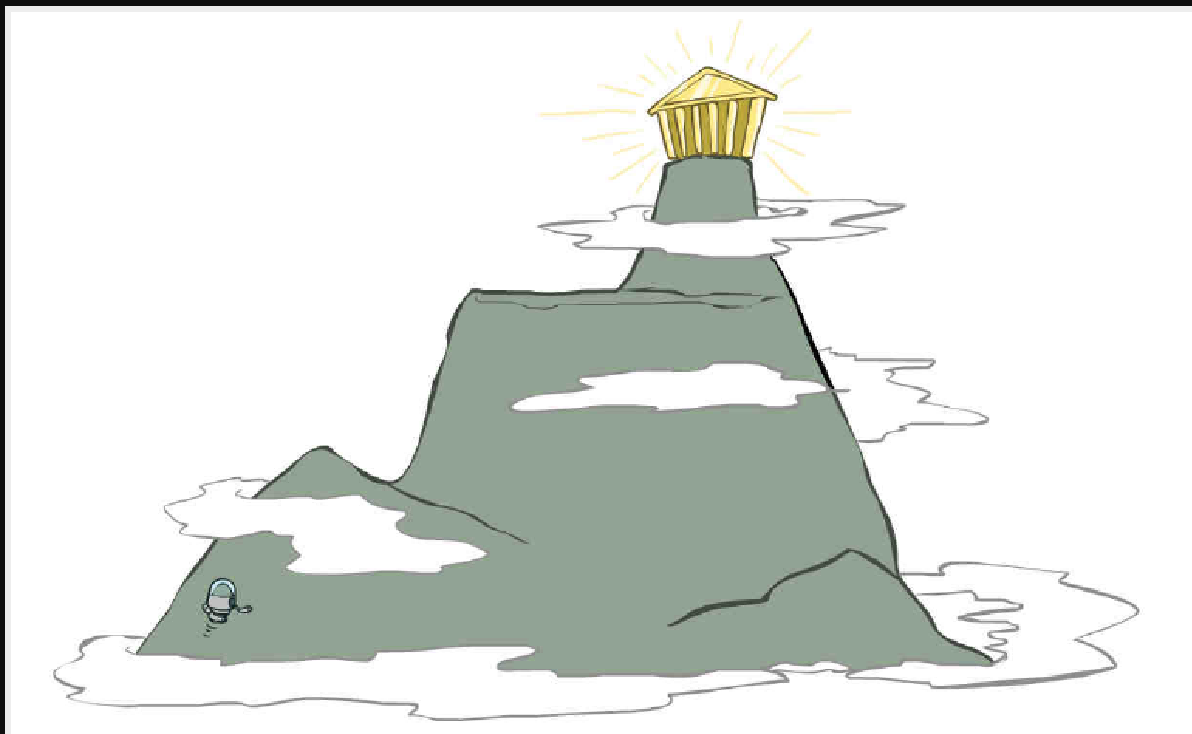


Max-Min Conflict

Selection = **multi-stage** heuristic:

- **Variable**: choose variable that appears in *most* violations (**max-conflict**)
- **Value**: assign value that *minimizes* var's violations (**min-conflict**)

Hill Climbing



Hill Climbing

- Start with *initial* state
- Repeatedly **select best** neighbor
- Choose **biggest improvement**
- If all neighbors worse, *stop*

Hill Climbing

- *Legal neighbors*: **non-degradation** (≥, ≤)
- *Selection*: **best** legal neighbor

Hill Climbing

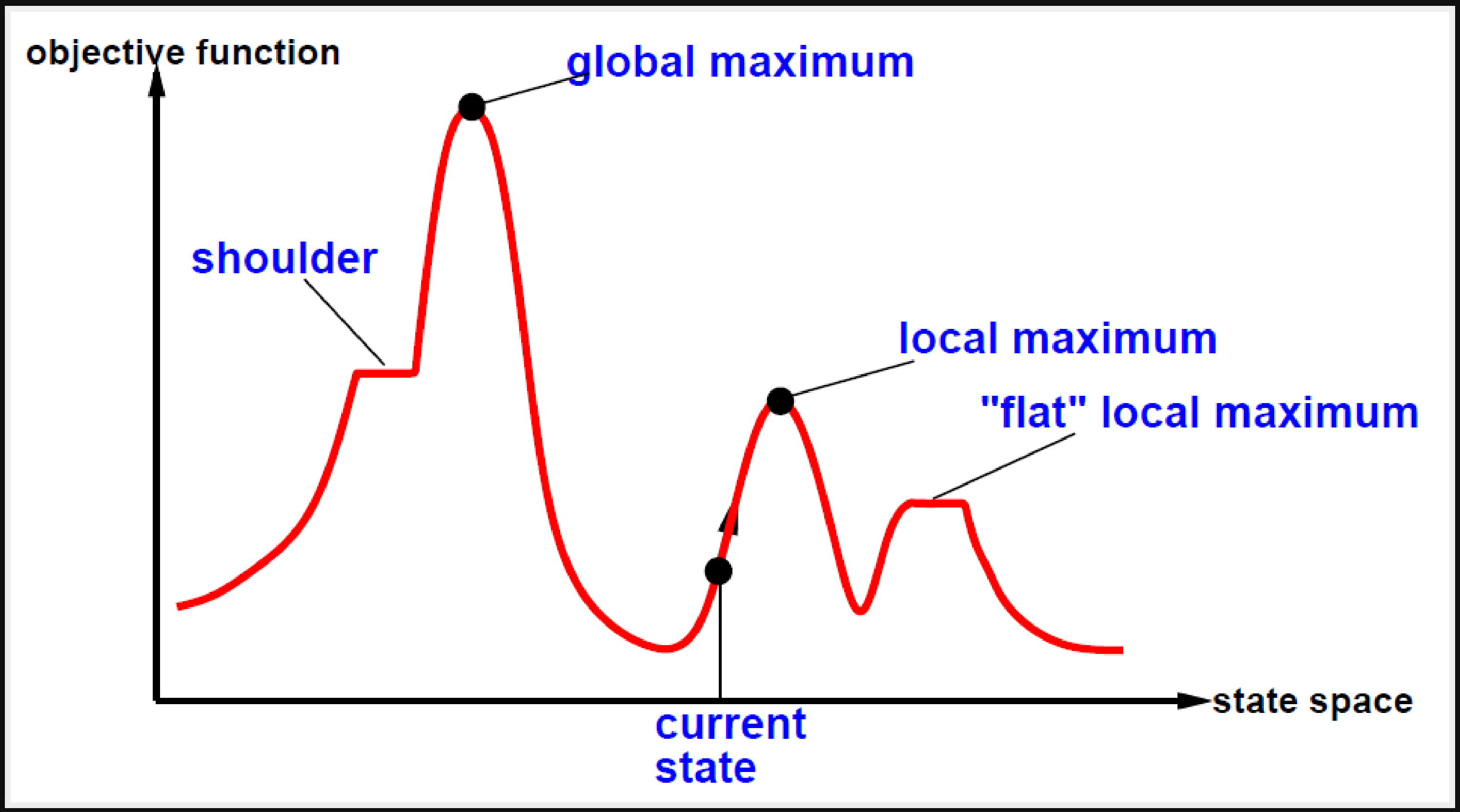
```
function HILL-CLIMBING( problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   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
   end
```

Hill Climbing

- **Complete**? No
- **Optimal**? No
- **Simplest** local search technique

Hill Climbing

Local max is not necessarily **global max**



Hill Climbing Problems

- *Problem*: Getting stuck on **local optimum**
- *Solution*: run local search **many times**
- **Random-restart HC**: *repeatedly* start with *random solution*, keep *best* solution

Iterated Local Search

```
function IteratedLocalSearch(f, N, L, S) {
    s := generateInitialSolution();
    s* := s;
    for k := 1 to MaxSearches do
        s := LocalSearch(f, N, L, S, s);
        if f(s) < f(s*) then
            s* := s;
        s := generateNewSolution(s);
    return s*;
}
```

Hill Climbing Problems

- *Problem*: Getting stuck on a **plateau**
- *Solution*: count no. of **iterations** in plateau (*no improvement*)
- *Solution*: **random sideways** moves

Stochastic Hill Climbing

- Choose **random** neighbor
- If neighbor **improves** objective fn, select as next state
- Does *not examine* all neighbors

Simulated Annealing

- *Idea*: **allow downhill / bad moves**, but make them **rare** as *time* goes on
- Allows you to *escape* local maxima
- **Annealing**: heating metal, cool slowly

Simulated Annealing

Temperature

- when T is **high**, can *bounce around* (go to neighbor even if it is not better)
- when T is **low**, less chance of going to bad neighbors
- T **cools down** with time
- set **temperature schedule**

Simulated Annealing

"Bounce around"

Simulated Annealing

Legal Neighbors + **Selection**:

- Compute **temperature**
- Choose **random** neighbor
- If *better*: select neighbor
- Else: select neighbor with **probability** *proportional* to temperature

Simulated Annealing

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

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

Simulated Annealing

- Used in VLSI layout, airline scheduling, etc.
- *Theoretical guarantee*: if T **decreased slowly** enough, will *converge* to **optimal** state!
- *Reality*: the **more downhill steps** you need to *escape* local optimum, the **less likely** you are to make them all **in a row**

Tabu Search

Tabu Search

- Local search with **memory**
- Keep track of *states* **already visited**
- Select **best legal** neighbor that is not *tabu*

Tabu Search

Short-term Memory

- **tabu list**: recently visited states
- cannot revisit until **tabu tenure** expired
- may increase / decrease size dynamically

Example: N-Queens

- **Local move**: assign var = value
- **Tabu list**: variable cannot be assigned to its old value
- Can also make **variable** *tabu* - don't change variable for some number of iterations

Tabu Search

Long-term Memory

- learn properties of bad solutions
- **diversify**: drive search into new regions
- avoid revisiting plateaus, dead-ends

Aspiration

- What if a move is tabu but really good?
- **Aspiration**: overrides tabu status
- Select neighbor if **not tabu** or **really good**

Metaheuristics

- *Examples*: Simulated Annealing, Tabu Search
- Aim to **escape local optima**
- Drive search towards **global optimum**

Metaheuristics

- *Efficiently explore* search space to find near-optimal solutions
- Typically includes *memory* or *learning*

Heuristics vs Metaheuristics

- **Heuristics**: problem-dependent
- **Metaheuristics**: problem-independent; general technique

Metaheuristics

- Variable Neighborhood Search
- Guided Local Search
- Genetic Algorithms
- Memetic Algorithms
- Evolutionary Algorithms

Variable Neighborhood Search

- **Different neighborhoods** per *iteration*
- Explores more of search space

Guided Local Search

- **Penalizes** states that reach *plateaus* and *deadends*
- *Modified* **objective fn**, penalties included

Summary

Local Search:

- Neighbors, Legal Neighbors, Selection, Objective function
- Hill Climbing
- Simulated Annealing
- Tabu Search

Next Meeting

Population-Based Search:

- Genetic Algorithms
- Swarm Algorithms
- **Assignment 4**: Local Search & PBS
- **MP #2**: Constrained Problems

References

- *Artificial Intelligence: A Modern Approach, 3rd Edition*, S. Russell and P. Norvig, 2010
- *Clever Algorithms*, J. Brownlee, 2011
- CS 188 Lec 5 slides, Dan Klein, UC Berkeley
- Introduction to Theoretical CS, Udacity
- Discrete Optimization, Coursera

Questions?