CS 1571 Introduction to AI
Lecture 6

Search for optimal configurations

Milos Hauskrecht
milos@cs.pitt.edu
5329 Sennott Square

Topics

- **Review of a CSP problem:**
  – Formulation
  – Search
  – Constraint propagation
- **Heuristics for CSP problems**
- **Search for optimal configurations:**
  – Examples
  – Hill climbing
  – Simulated annealing
  – Genetic algorithms
- **Search for optimal configurations with continuous variables**

Search problem

A search problem:

- **Search space (or state space):** a set of objects among which we conduct the search;
- **Initial state:** an object we start to search from;
- **Operators (actions):** transform one state in the search space to the other;
- **Goal condition:** describes the object we search for

- **Possible metric on a search space:**
  – measures the quality of the object with regard to the goal

Search problems occur in planning, optimizations, learning

Constraint satisfaction problem (CSP)

Two types of search:

- **path search** (a path from the initial state to a state satisfying the goal condition)
- **configuration search (**a configuration satisfying goal conditions)

Constraint satisfaction problem (CSP) is **a configuration search problem** where**:**

- A state is defined by a set of variables
- Goal condition is represented by a set constraints on possible variable values

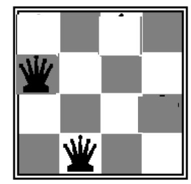
Special properties of the CSP allow more specific procedures to be designed and applied for solving them

Example of a CSP: N-queens

Goal: n queens placed in non-attacking positions on the board

Variables:

- Represent queens, one for each column:
  - Q_1, Q_2, Q_3, Q_4
- Values:
  - Row placement of each queen on the board {1, 2, 3, 4}



$Q_1 = 2, Q_2 = 4$

Constraints: $Q_i \neq Q_j$ Two queens not in the same row

$|Q_i - Q_j| \neq |i - j|$ Two queens not on the same diagonal

Satisfiability (SAT) problem

Determine whether a sentence in the conjunctive normal form (CNF) is satisfiable (can evaluate to true)

- Used in the propositional logic (covered later)

$$(P \vee Q \vee \neg R) \wedge (\neg P \vee \neg R \vee S) \wedge (\neg P \vee Q \vee \neg T) \ldots$$

Variables:

- Propositional symbols (P, R, T, S)
- Values: *True, False*

Constraints:

- Every conjunct must evaluate to true, at least one of the literals must evaluate to true

$$(P \vee Q \vee \neg R) \equiv True, (\neg P \vee \neg R \vee S) \equiv True, \ldots$$

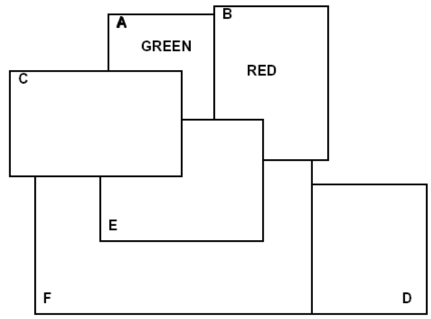
Map coloring

Color a map using k different colors such that no adjacent countries have the same color

Variables:

- Represent countries
  - A, B, C, D, E
- Values:
  - K -different colors

  {Red, Blue, Green,..}



Constraints: $A \neq B, A \neq C, C \neq E$, etc

Other real world CSP problems

Scheduling problems:

- E.g. telescope scheduling
- High-school class schedule

Design problems:

- Hardware configurations
- VLSI design

More complex problems may involve:

- **real-valued variables**
- **additional preferences on variable assignments –** the optimal configuration is sought

Constraint satisfaction as a search problem

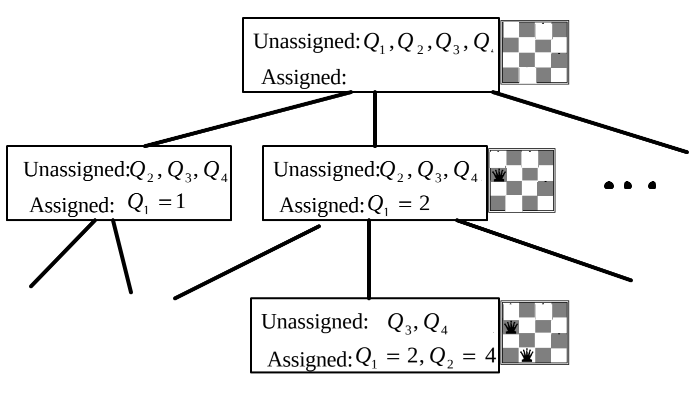
Formulation of a CSP as a search problem:

- **States.** Assignment (partial, complete) of values to variables.
- **Initial state.** No variable is assigned a value.
- **Operators.** Assign a value to one of the unassigned variables.
- **Goal condition.** All variables are assigned, no constraints are violated.

- **Constraints** can be **represented**:
  - **Explicitly** by a set of allowable values
  - **Implicitly** by a function that tests for the satisfaction of constraints

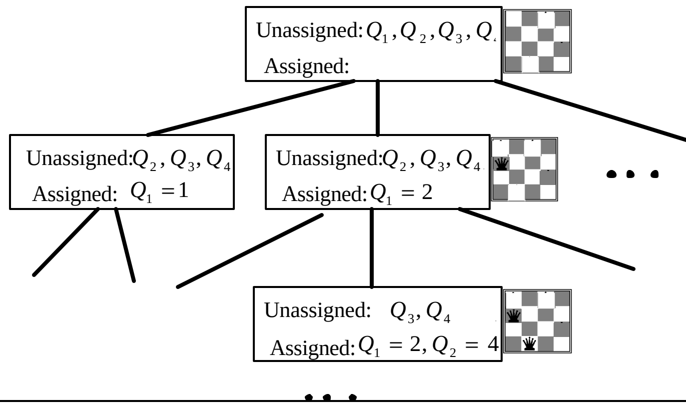
Solving a CSP through standard search

- **Maximum depth of the tree:** Number of variables of the CSP
- **Depth of the solution:** Number of variables of the CSP
- **Branching factor:** if we fix the order of variable assignments the branch factor depends on the number of their values



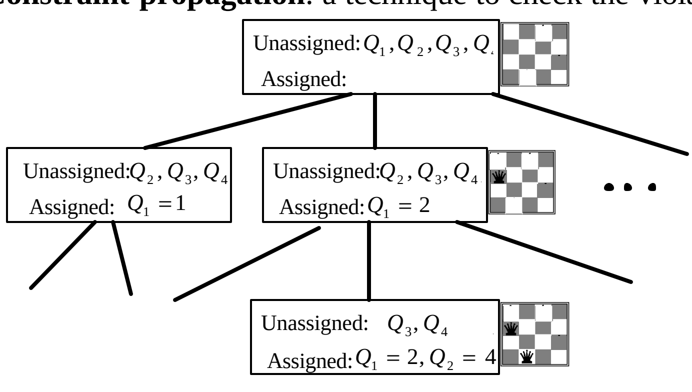
Solving a CSP through standard search

- **What search algorithm to use: Depth first search !!!**
  - Since we know the depth of the solution
  - We do not have to keep large number of nodes in queues



Solving a CSP through standard search

- **When to stop the expansion of the node?**
  - No valid assignment of values to variables exists for the branch of the tree rooted at that node
  - **Constraint propagation**: a technique to check the violations



Constraint propagation

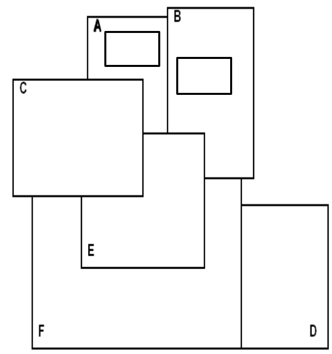
A **state** (more broadly) is defined by a set variables and their legal and illegal assignments

Legal and illegal assignments can be represented through variable **equations** and variable **disequations**

Example: **map coloring**

Equation $A = \text{Red}$

Disequation $C \neq \text{Red}$



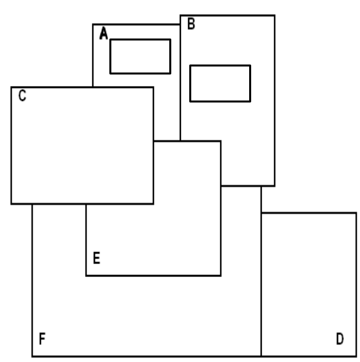
Constraints + assignments can entail new equations and disequations

$A = \text{Red} \rightarrow B \neq \text{Red}$

Constraint propagation

- Assign A=Red

	Red	Blue	Green
A	✓		
B	✗		
C	✗		
D			
E	✗		
F			

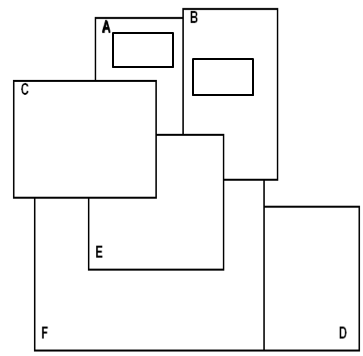


✓ - equations ✗ - disequations

Constraint propagation

- Assign E=Blue

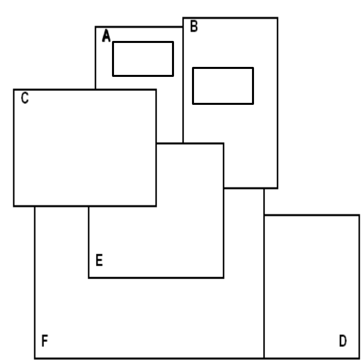
	Red	Blue	Green
A	✓	✗	
B	✗	✗	
C	✗	✗	
D			
E	✗	✓	
F		✗	



Constraint propagation

- Assign F=Green

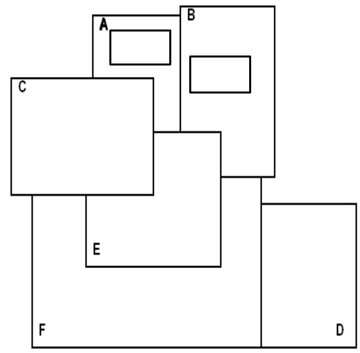
	Red	Blue	Green
A	✓	✗	
B	✗	✗	✗
C	✗	✗	✗
D			✗
E	✗	✓	✗
F		✗	✓



Constraint propagation

- Assign F=Green

	Red	Blue	Green
A	✓	✗	
B	✗	✗	✗
C	✗	✗	✗
D			✗
E	✗	✓	✗
F		✗	✓

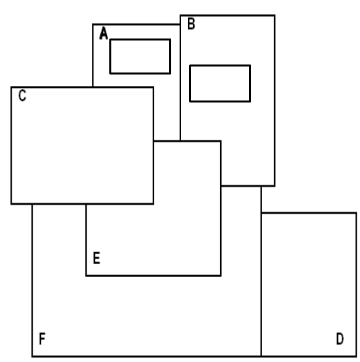


Conflict !!! No legal assignments available for B and C

Constraint propagation

- We can derive remaining legal values through propagation

	Red	Blue	Green
A	✓	✗	
B	✗	✗	✓
C	✗	✗	✓
D			
E	✗	✓	
F		✗	



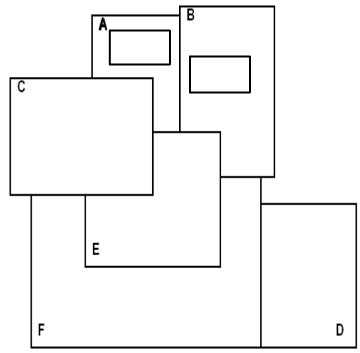
B=Green

C=Green

Constraint propagation

- We can derive remaining legal values through propagation

	Red	Blue	Green
A	✓	✗	✗
B	✗	✗	✓
C	✗	✗	✓
D	✗		
E	✗	✓	✗
F	✓	✗	✗



B=Green
C=Green → **F=Red**

Constraint propagation

Three known techniques for propagating the effects of past assignments and constraints:

- **Value propagation**
- **Arc consistency**
- **Forward checking**

- **Difference:**
  - Completeness of inferences
  - Time complexity of inferences.

Constraint propagation

1. **Value propagation. Infers:**
   - **equations from** the set of **equations** defining the partial assignment, **and constraints**
2. **Arc consistency. Infers:**
   - **disequations from** the set of **equations and disequations** defining the partial assignment, and **constraints**
   - **equations through the exhaustion of alternatives**
3. **Forward checking. Infers:**
   - **disequations from** a set of **equations** defining the partial assignment, and **constraints**
   - **Equations through the exhaustion of alternatives**

   Restricted forward checking:
   - uses only active constraints (active constraint – only one variable unassigned in the constraint)

Heuristics for CSP

Backtracking searches the space in the depth-first manner.

But we can choose:

- **Which variable to assign next?**
- **Which value to choose first?**

Heuristics

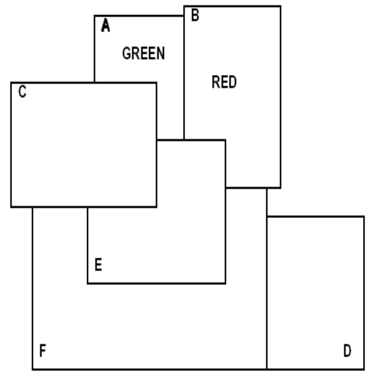
- **Most constrained variable**
  - Which variable is likely to become a bottleneck?
- **Least constraining value**
  - Which value gives us more flexibility later?

Heuristics for CSP

Examples: **map coloring**

Heuristics

- **Most constrained variable**
  - Country E is the most constrained one (cannot use Red, Green)
- **Least constraining value**
  - Assume we have chosen variable C
  - Red is the least constraining valid color for the future



Configuration search for optimal solutions

Search for the optimal configuration

Configuration-search problems:

- Are often enhanced with some **quality measure**

Quality measure

- reflects our preference towards each configuration (or state)

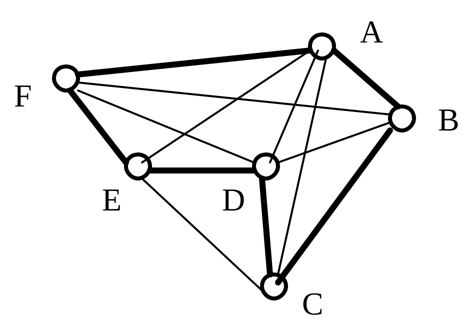
Goal

- find the configuration with the optimal quality

Example: Traveling salesman problem

Problem:

- A graph with distances

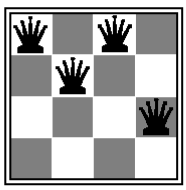


- **Goal:** find the shortest tour which visits every city once and returns to the start

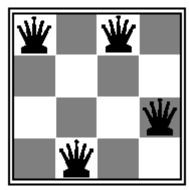
An example of a valid tour: ABCDEF

Example: N queens

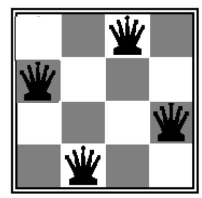
- **Some CSP problems do not have a quality measure**
- **The quality of a configuration in a CSP** can be measured by the number of constraints violated
- Solving corresponds to the minimization of the number of constraint violations



# of violations =3



# of violations =1



# of violations =0

Iterative improvement algorithms

- Give solutions to the configuration-search with the optimality measure

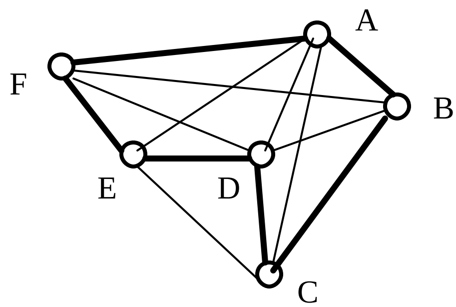
Properties of iterative improvement algorithms:

- Search the space of “complete” configurations
- Operators make “local” changes to “complete” configurations
- **Keep track of just one state (the current state), not a memory of past states**
  - **!!! No search tree is necessary !!!**

Example: Traveling salesman problem

Problem:

- A graph with distances



- **Goal:** find the shortest tour which visits every city once and returns to the start

An example of a valid tour: ABCDEF

Example: Traveling salesman problem

"Local" operator for generating the next state:

- divide the existing tour into two parts,
- reconnect the two parts in the opposite order

Example:

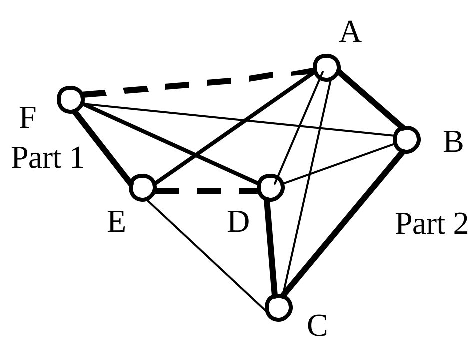
ABCDEF

⇩

ABCD | EF |

⇩

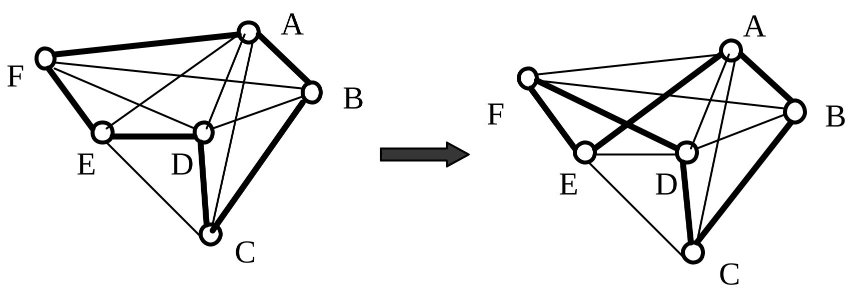
ABCDFE



Example: Traveling salesman problem

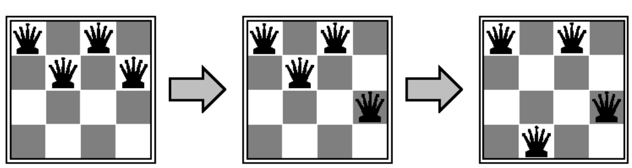
"Local" operator:

– generates the next configuration (state)



Example: N-queens

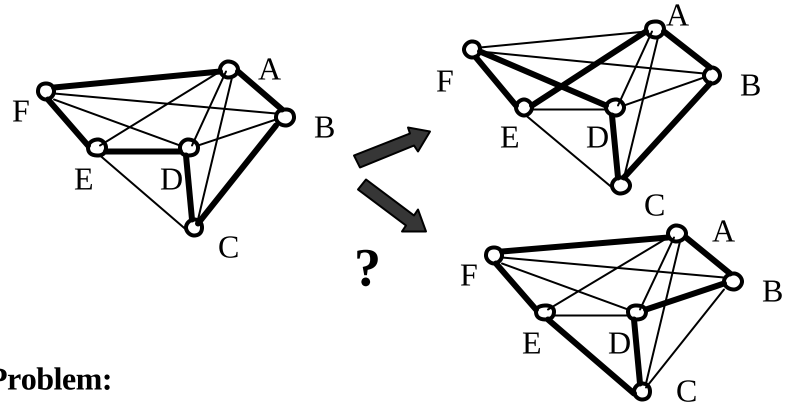
- **"Local" operators for generating the next state:**
  – Select a variable (a queen)
  – Reallocate its position



Searching configuration space

Iterative improvement algorithms

- keep only one configuration (the current configuration) active



Problem:

- How to decide about which operator to apply?

Iterative improvement algorithms

Two strategies to choose the configuration (state) to be visited next:

- **Hill climbing**
- **Simulated annealing**

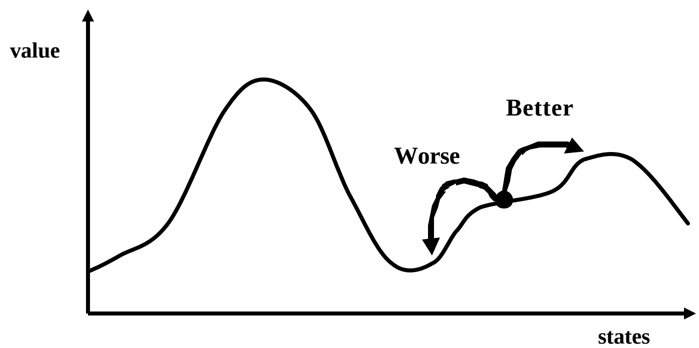
- Later: Extensions to multiple current states:
  - **Genetic algorithms**

- **Note:** Maximization is inverse of the minimization

$$\min \; f(X) \Leftrightarrow \max \left[-f(X) \right]$$

Hill climbing

- **Local improvement algorithm**
- Look around at states in the local neighborhood and choose the one with the best value
- Assume: we want to maximize the



Hill climbing

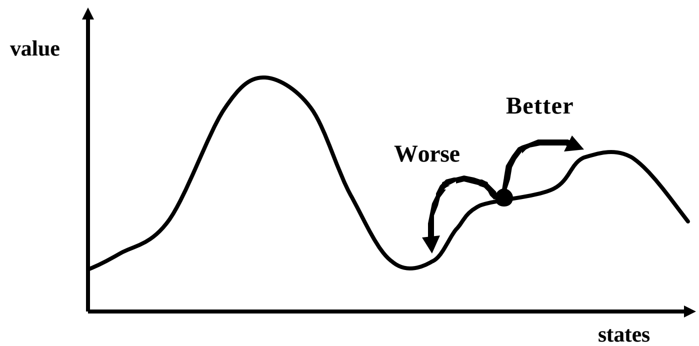
- Always choose the next best successor state
- Stop when no improvement possible

```
function HILL-CLIMBING(problem) returns a solution state
  inputs: problem, a problem
  static: current, a node
          next, a node

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

Hill climbing

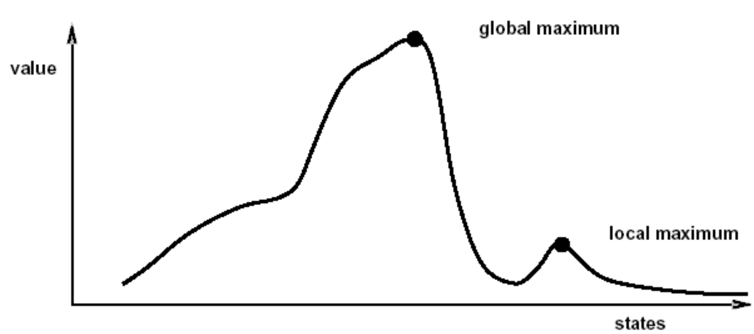
- Local improvement algorithm
- Look around at states in the local neighborhood and choose the one with the best value



- **What can go wrong?**

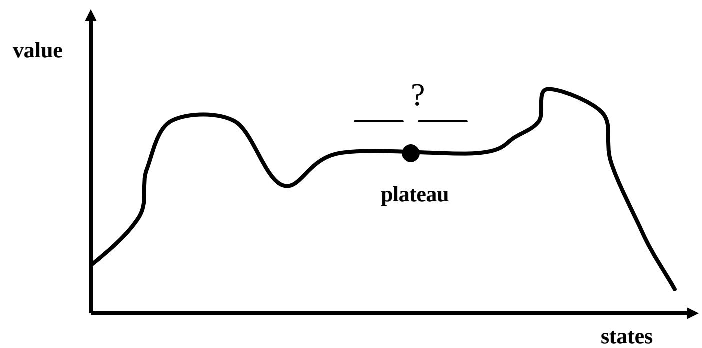
Hill climbing

- Hill climbing can get trapped in the local optimum



Hill climbing

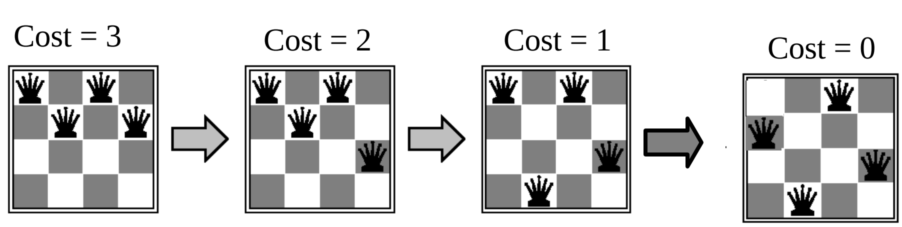
- Hill climbing can get clueless on plateaus



CS 1571 Intro to AI

Hill climbing and n-queens

- The quality of a configuration given by the number of constraints violated
- **Then: Hill climbing** reduces the number of violated constraints
- **Min-conflict strategy (heuristic):**
  - Choose randomly a variable with conflicts
  - Choose its value such that it violates the fewest constraints

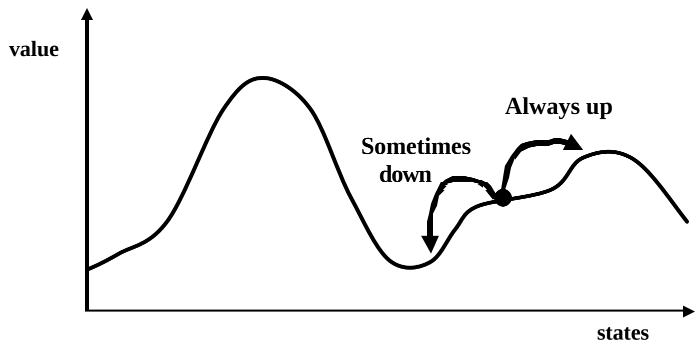


Success !! But not always!!! The local optima problem!!!

CS 1571 Intro to AI

Simulated annealing

- Permits "bad" moves to states with lower values, thus escape the local optima
- **Gradually decreases** the frequency of such moves and their size (parameter controlling it – **temperature**)



Simulated annealing

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

Simulated annealing algorithm

- The probability of moving into a state with lower value

$$e^{\Delta E / T}$$

- T is a **temperature parameter**:

  for $T \to 0$ the probability that a state with smaller value is selected goes down and approaches 0

- Algorithm was originally developed for modeling physical processes (Metropolis et al, 53)
- **If T is decreased slowly enough the best state is always reached**

- **Applications:** VLSI design, airline scheduling