Search (II)

Mingsheng Long

Tsinghua University

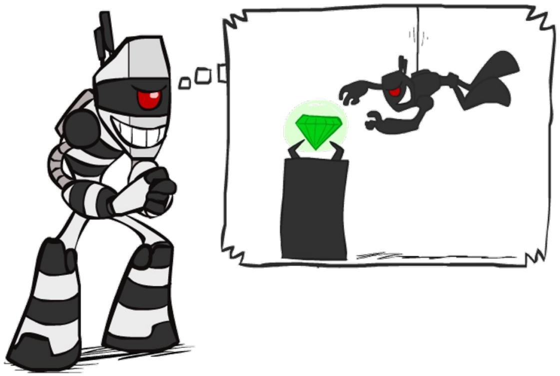
Spring 2023

Outline

- **Constraint Satisfaction Problems**
  - Backtracking Search
  - Dynamic Ordering
  - Arc Consistency
  - Problem Structure
- Local Search
  - Hill Climbing
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithm

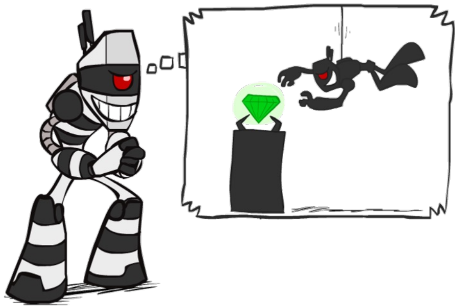
Search Problems

- Planning: sequences of actions
  - The path to the goal is important
  - Paths have various costs and depths
  - Heuristics give problem-specific guidance



- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths generally at the same depth
  - Constraint Satisfaction Problems (CSPs) are a special class of identification problems

Constraint Satisfaction Problems



- Standard search problems
  - **State** is a black box: arbitrary data structure
  - **Goal test** can be any function over states
  - **Successor function** can also be anything

- Constraint satisfaction problems (CSPs)
  - A special subset of search problems
  - **State** is structured:
    - An assignment of variables X_i with values from a domain D_i
  - **Goal test** is a set of constraints specifying allowable combinations of values for subsets of variables

Constraint Satisfaction Problems

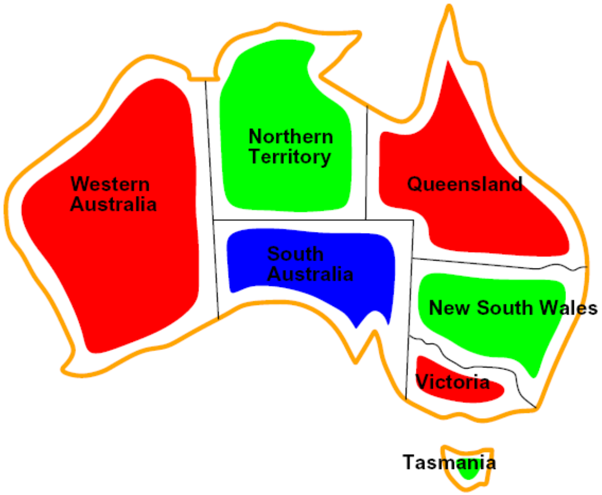
- Constraint satisfaction problem $P = (X, D, C)$
  - Variables: $X = \{X_1, \cdots, X_n\}$
  - Domains: $D = \{D_1, \cdots, D_n\}$
    - Each domain $D_i = \{v_1, \cdots, v_k\}$ for variable X_i
  - Constraints: C, specifying allowable combinations of values

- An assignment of values to some or all of the variables:

$$\{X_i = v_i, X_j = v_j, \cdots\}$$

- A complete assignment is one in which every variable is assigned
- A solution to a CSP is a consistent and complete assignment

Example: Map Coloring



- Variables:

$$\mathrm{WA, NT, Q, NSW, V, SA, T}$$

- Domains:

$$D = \{\mathrm{R, G, B}\}$$

- Constraints: adjacent regions must have different colors, e.g.:

$$\mathrm{WA} \neq \mathrm{NT}, \cdots$$

- Solutions: assignments satisfying all constraints, e.g.:

$$\{\mathrm{WA} = \mathrm{R}, \mathrm{NT} = \mathrm{G}, \mathrm{Q} = \mathrm{R},$$
$$\mathrm{NSW} = \mathrm{G}, \mathrm{V} = \mathrm{R}, \mathrm{SA} = \mathrm{B}, \mathrm{T} = \mathrm{G}\}$$

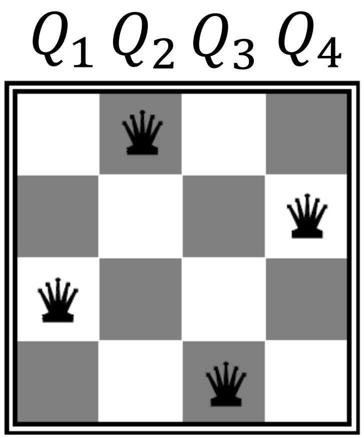
Example: N-Queens

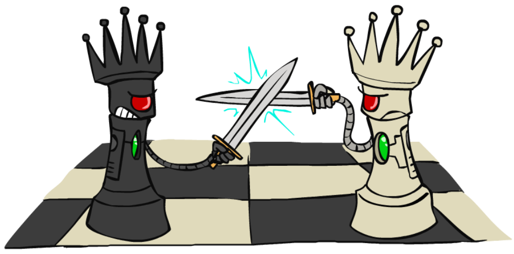
- Variables: Q_k, the row coordinate for each column
- Domains: $\{1, 2, 3, \cdots, N\}$
- Constraints:
  - Implicit:

$$\forall i, j, \text{non-threatening}(Q_i, Q_j)$$

  - Explicit:

$$(Q_1, Q_2) \in \{(1,3), (1,4), \cdots\}, \cdots$$





- Solutions: complete assignments for each column, e.g.:

$$Q_1 = 3, Q_2 = 1, Q_3 = 4, Q_4 = 2$$

Example: Sudoku

- Constraints:

$$\text{Alldiff}(A1, A2, A3, A4, A5, A6, A7, A8, A9)$$
$$\text{Alldiff}(B1, B2, B3, B4, B5, B6, B7, B8, B9)$$
$$\ldots$$
$$\text{Alldiff}(A1, B1, C1, D1, E1, F1, G1, H1, I1)$$
$$\text{Alldiff}(A2, B2, C2, D2, E2, F2, G2, H2, I2)$$
$$\ldots$$

Beyond binary constraints:

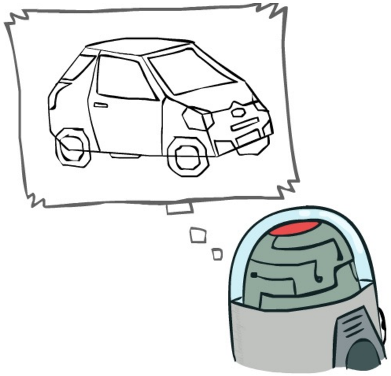
A global constraint is one involving an arbitrary number of variables (but not necessarily all the variables)

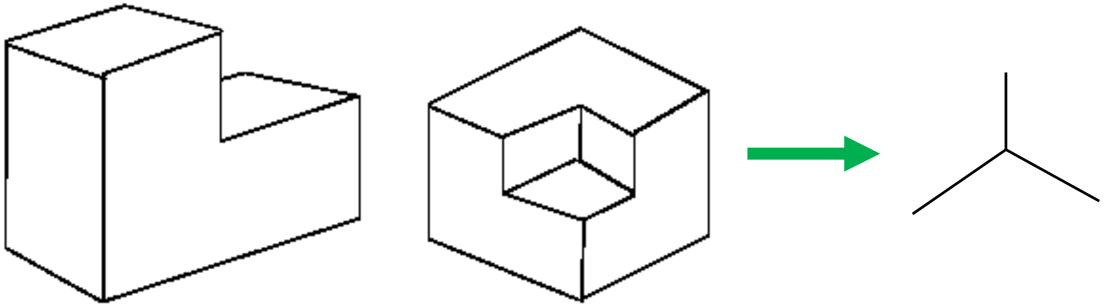
	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

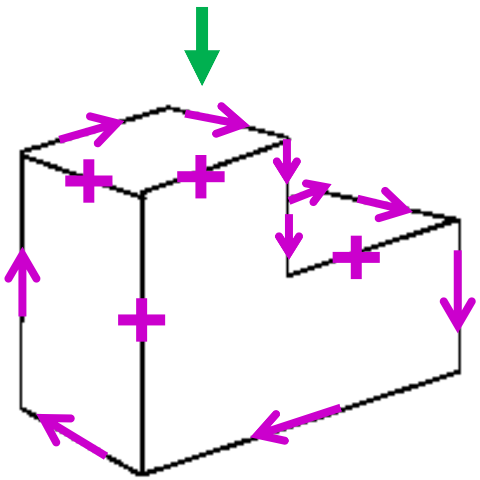
Example: The Waltz Algorithm

- Interpreting line drawings of solid polyhedra
  - One of the earliest example of an AI computation posed as a CSP





What kind of intersection?
Concave or convex?



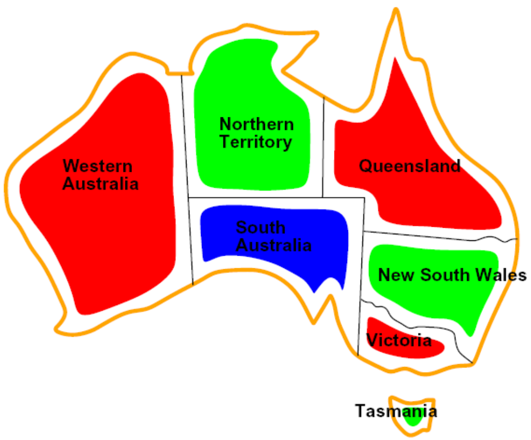
Variables: intersections

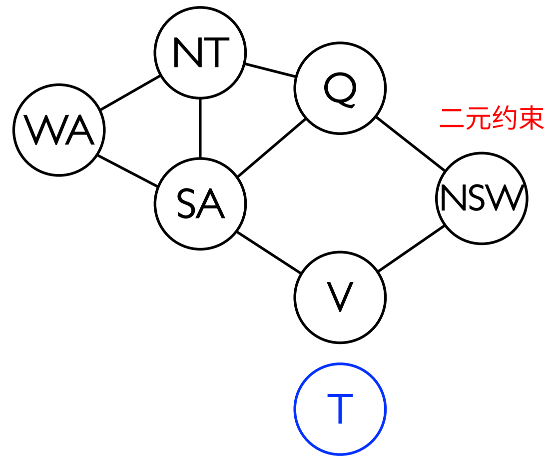
Constraints: adjacent intersections

Solutions: physically realizable 3D interpretations

Constraint Graphs

- Binary CSP: each constraint relates (at most) two variables
- Binary constraint graph: nodes are variables, arcs show constraints

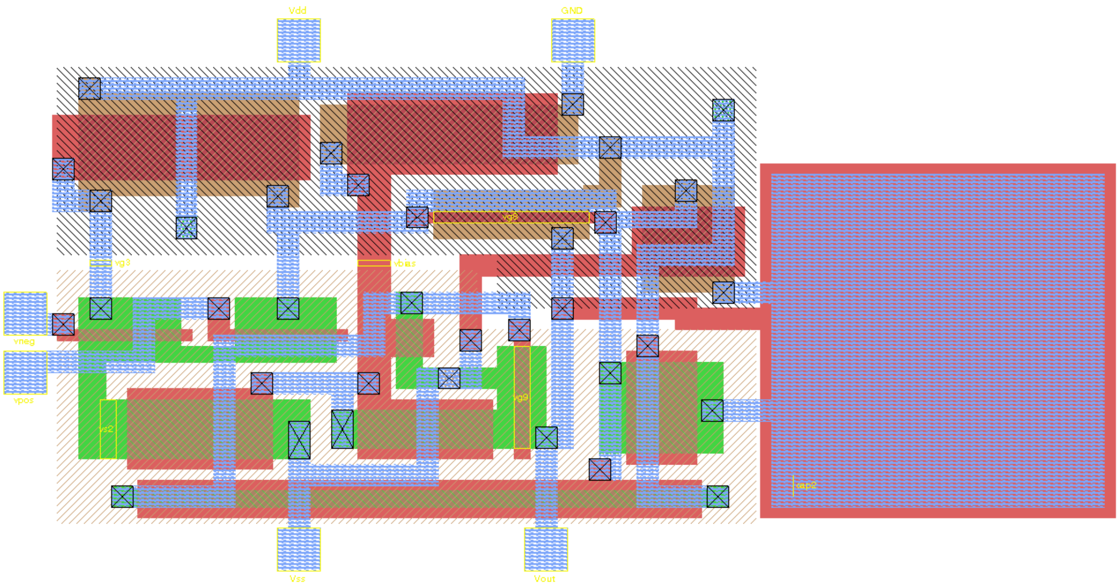




- General-purpose CSP algorithms:
  - We use the graph structure to speed up search (more later)
  - E.g., Tasmania (T) is an independent subproblem

Applications of CSPs

- Assignment problems: e.g., who teaches what class
- Timetabling problems: e.g., which class is offered when and where
- Transportation scheduling
- Factory scheduling
- Hardware configuration
- Circuit layout
- Fault diagnosis
- And lots more…



- Many real-world problems involve real-valued variables and difficult…

Outline

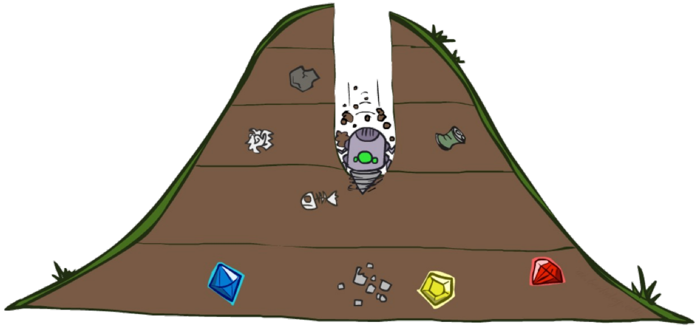
- **Constraint Satisfaction Problems**
  - **Backtracking Search**
  - Dynamic Ordering
  - Arc Consistency
  - Problem Structure
- Local Search
  - Hill Climbing
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithm

Standard Search Formulation

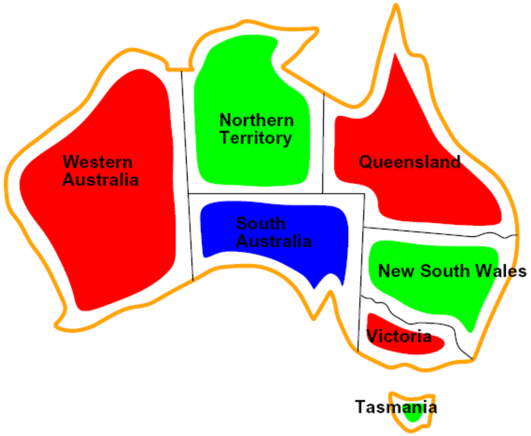
Standard search formulation of CSPs:

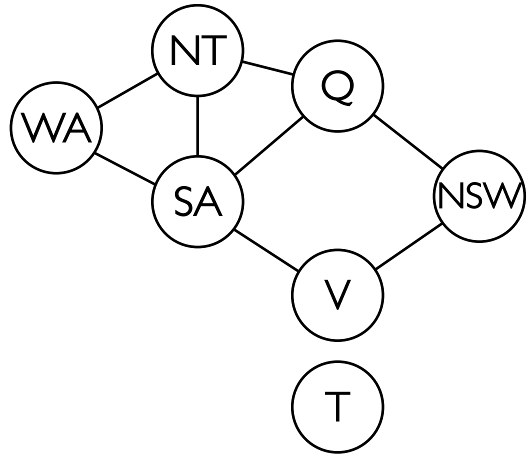
- States: the values assigned so far (partial assignments)
- Initial state: the empty assignment, {}
- Successor function: assign a value to an unassigned variable
- Goal test: complete and satisfy all constraints

- Standard search methods: DFS, BFS, …
- What problems does naive search have?

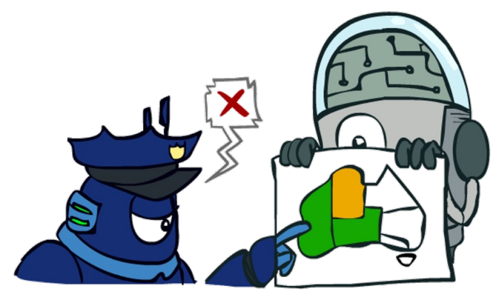


Map Coloring: Depth First Search





DFS: Duplicate states and paths



Backtracking Search

- Idea I: One variable at a time
  - Variable assignments are commutative, so fix ordering
  - I.e., $[\mathrm{WA} = \mathrm{R}$ then $\mathrm{NT} = \mathrm{G}]$ same as $[\mathrm{NT} = \mathrm{G}$ then $\mathrm{WA} = \mathrm{R}]$
- Idea II: Check constraints as you go
  - Incremental goal test: consider only values which do not conflict with previous assignments
  - Might have to do some computation to check the constraints

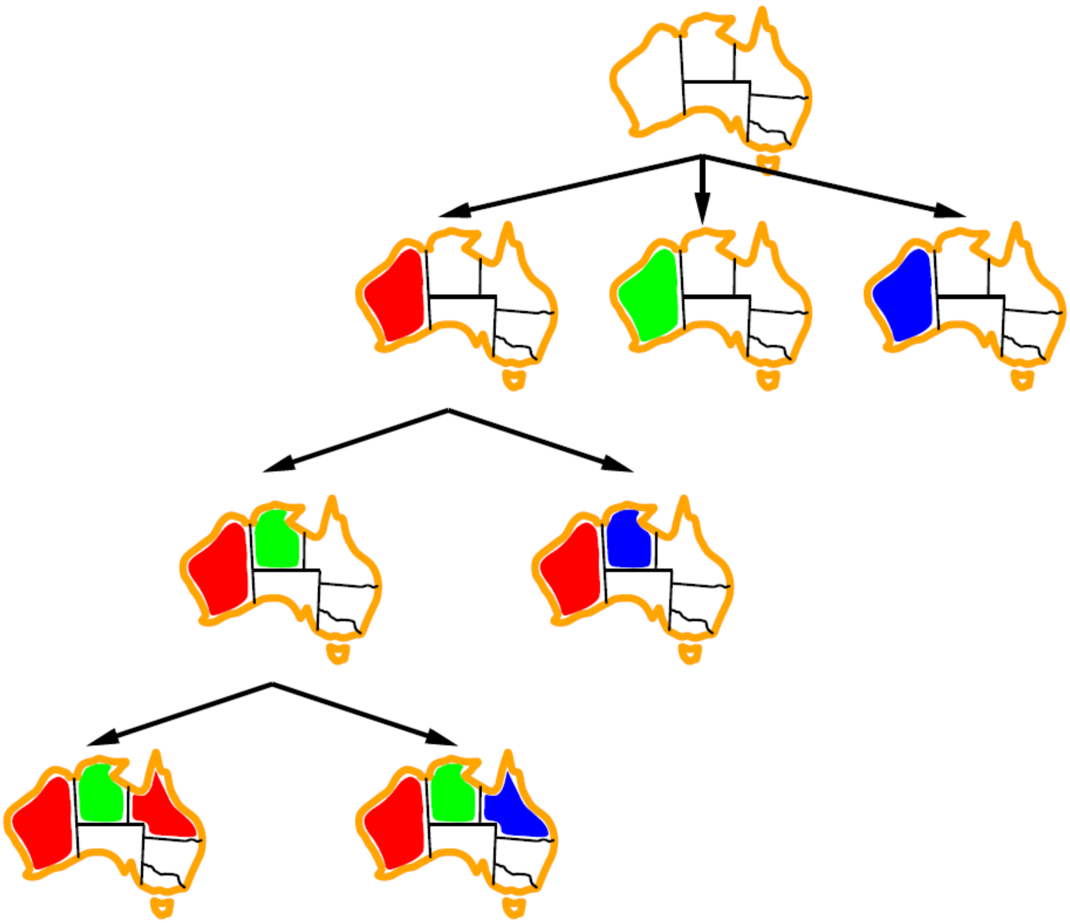
Backtracking Search = DFS + variable-ordering + fail-on-violation

Backtracking Search

- Backtracking Search = DFS + variable-ordering + fail-on-violation
  变量排序

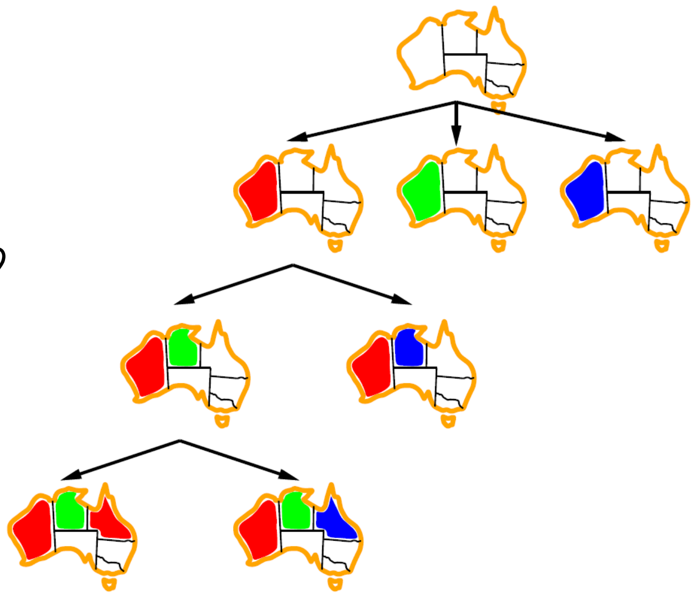
function BACKTRACK(*assignment, Domains*) **returns** a solution, or *failure*
  if *assignment* is complete **then return** *assignment* ← General Search checks consistency on full assignment
  choose unassigned variable X_i Variable-ordering
  order domain values $Domains_i$ of chosen X_i 对其他值排序
  for each *value* **in** that order **do**:
    if *value* is consistent with *assignment* **then** ← Fail-on-violation; Backtracking Search checks consistency at each assignment
      add { $X_i = value$ } to *assignment*
      result = BACKTRACK(*assignment, Domains*)
      if *result* ≠ *failure* **then return** *result*
      remove { $X_i = value$ } from *assignment* ← Backtracking (回溯)
  return *failure*

Backtracking Example



How to Improve Backtracking?

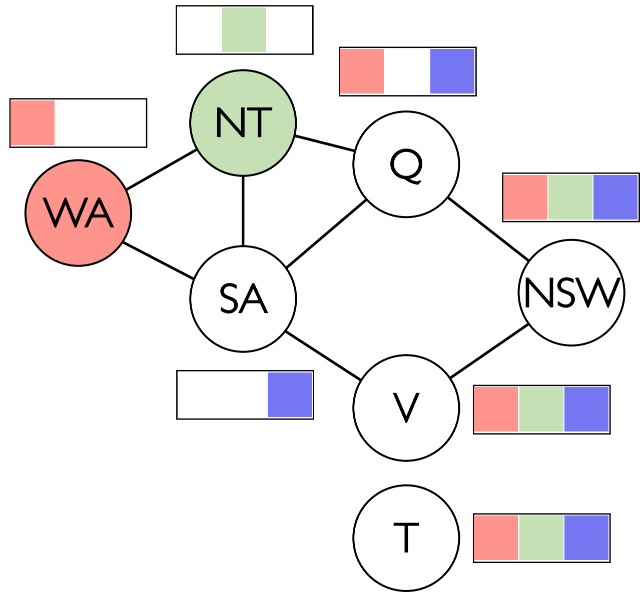
- General-purpose ideas give huge gains in speed
  - Heuristics
- Ordering
  - Which variable should be assigned next?
  - In what order should its values be tried?
- Filtering
  - Can we detect inevitable failure early?
- Structure
  - Can we exploit the problem structure?



Outline

- **Constraint Satisfaction Problems**
  - Backtracking Search
  - **Dynamic Ordering**
  - Arc Consistency
  - Problem Structure
- Local Search Methods
  - Hill Climbing
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithm

Choose an Unassigned Variable



- Which variable to assign next?

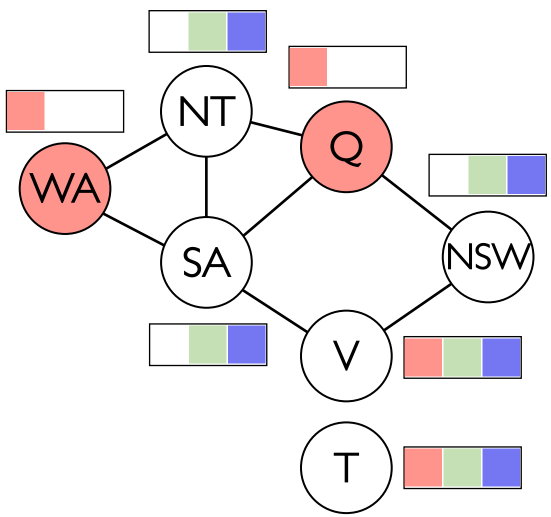
Smaller branching factors

> **Key idea: most constrained variable**
> Choose variable that has the fewest consistent values.

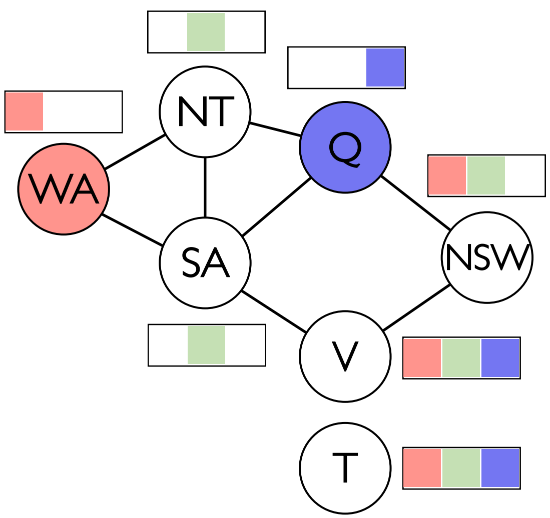
- This example: SA (has only one value)

Order Values of a Selected Variable

- What values to try for Q?



$2 + 2 + 2 = 6$ consistent values



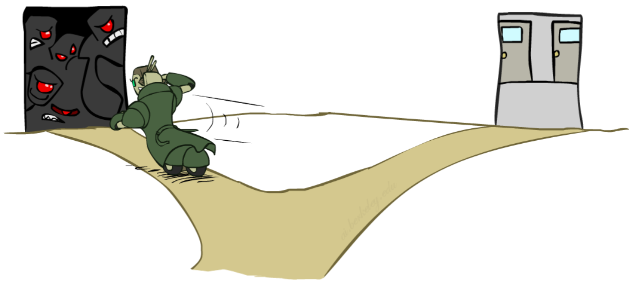
$1 + 1 + 2 = 4$ consistent values

Key idea: least constrained value

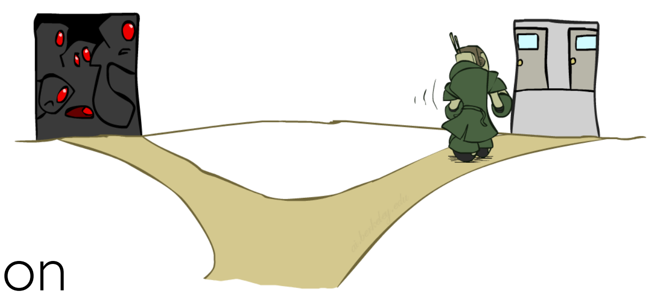
Order values of selected X_i by decreasing number of consistent values of neighboring variables.

Resolve Conflicting Heuristics

- Most constrained variable (MCV):
  - Must assign every variable
  - If going to fail, fail early → more pruning
  - For some problem, improve by 1000×



- Least constrained value (LCV):
  - Need to choose some value
  - Choosing value most likely to lead to solution



- Combining these ordering ideas makes 1000 queens feasible

Outline

- **Constraint Satisfaction Problems**
  - Backtracking Search
  - Dynamic Ordering
  - **Arc Consistency**
  - Problem Structure
- Local Search
  - Hill Climbing
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithm

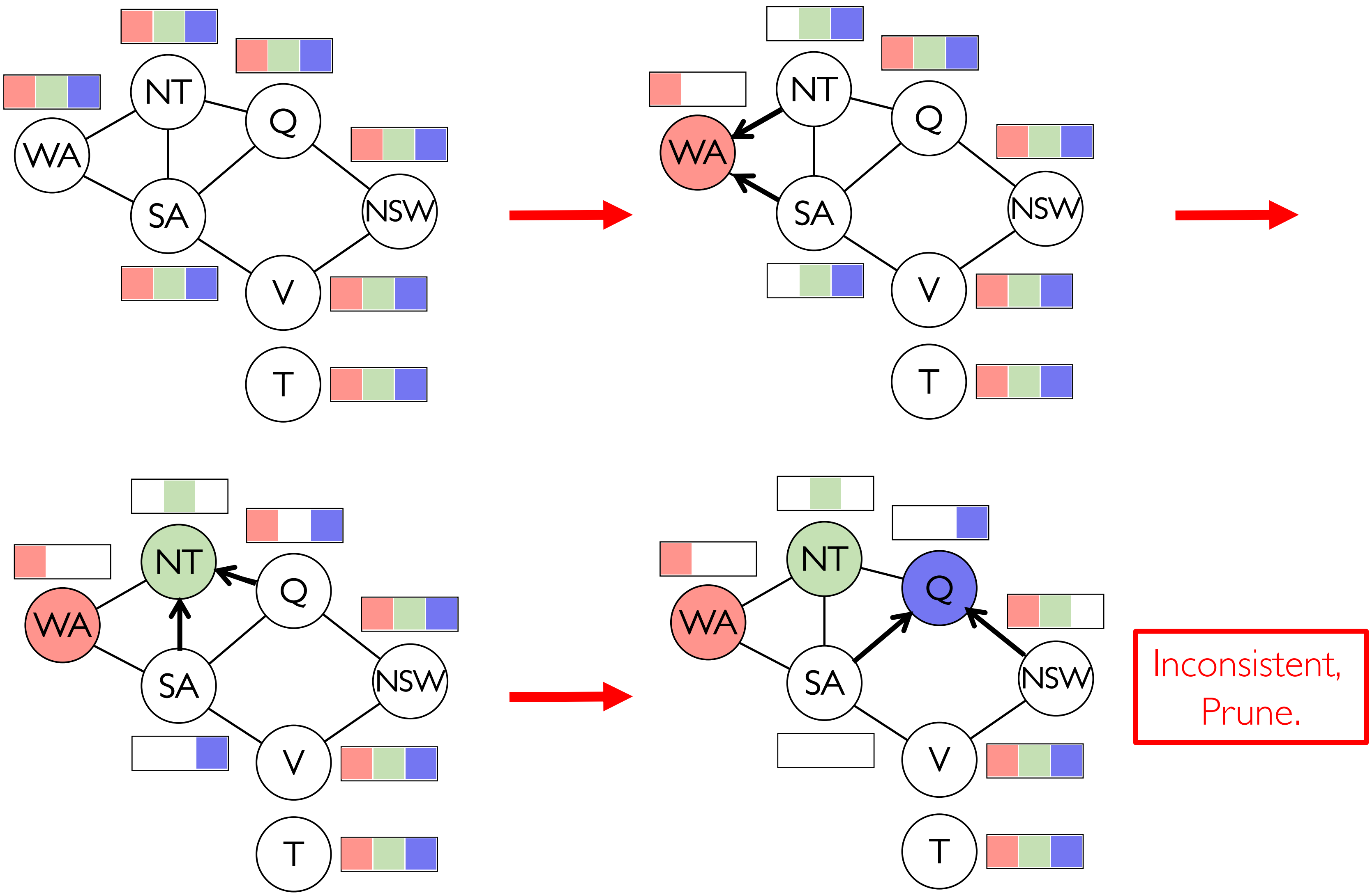
Interleave Search and Inference

- Filtering: Keep track of domains for unassigned variables and infer new domain reductions

function BACKTRACK(*assignment, Domains*) **returns** a solution, or *failure*
  if *assignment* is complete **then return** *assignment*
  choose unassigned variable X_i
  order domain values $Domains_i$ of chosen X_i
  for each *value* **in** that order **do**:
    if *value* is consistent with *assignment* **then**
      add { X_i = *value* } to *assignment*
      $Domains_i = \{value\}$
      if FILTER(X_i, *Domains*) $\neq$ *failure* **then**
        result = BACKTRACK(*assignment, Domains*)
        if *result* $\neq$ *failure* **then return** *result*
      remove { X_i = *value* } from *assignment* and restore *Domains*
  return *failure*

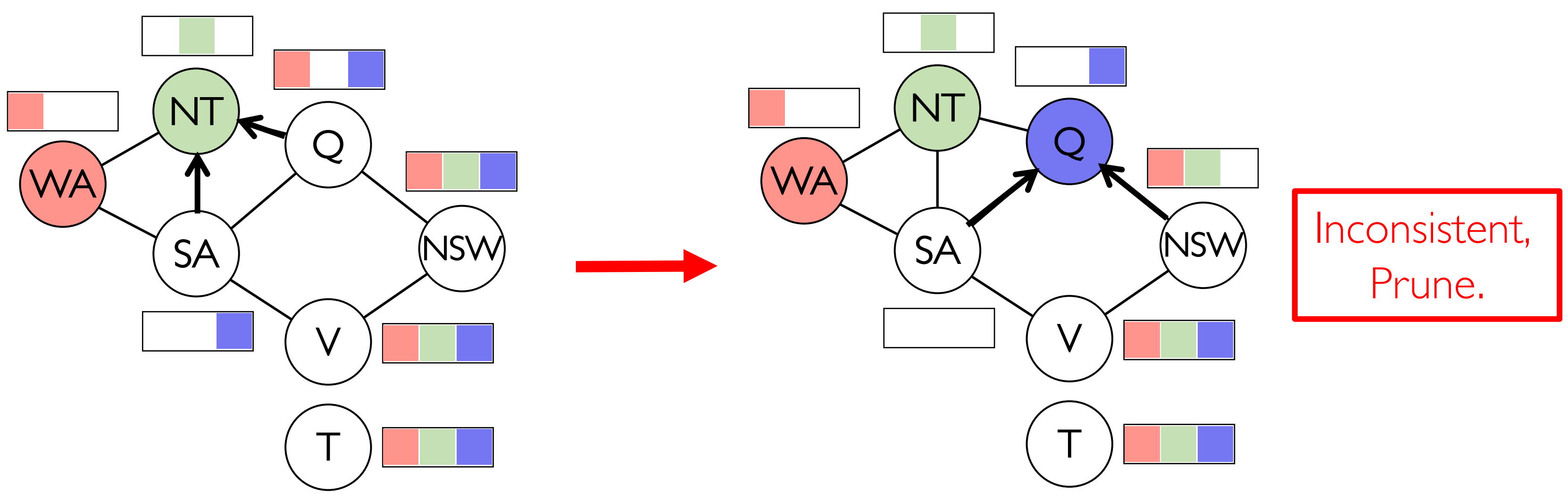
Filtering (Forward checking or AC-3) removes bad values from domains.
If one of the domains is reduced to empty, prune.

Filtering: Forward Checking



Filtering: Forward Checking

- Forward checking (one-step lookahead)
  - After assigning a variable X_i, eliminate inconsistent values from the domains of X_i's neighbors
  - If any domain becomes empty, don't recurse.
  - When unassign X_i, restore neighbors' domains.



Arc Consistency

- An arc $X_i \rightarrow X_j$ is consistent iff for every $x_i \in \mathbf{Domains}_i$, there exists $x_j \in \mathbf{Domains}_j$ which can be assigned without violating a constraint
- Enforce arc consistency: Remove values from $\mathbf{Domains}_i$ to make arc consistent

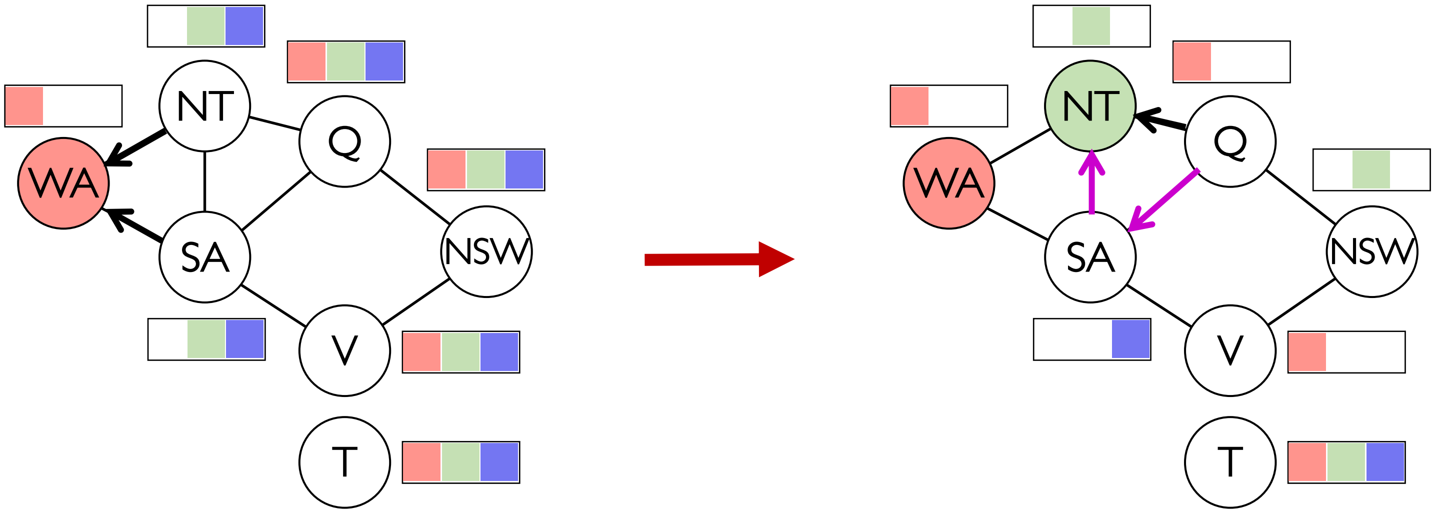
Remember: Delete from $\mathbf{Domains}_i$

```
function EnforceArcConsistency(Domains, X_i, X_j) returns true iff succeeds
    removed = false
    for each x in Domains_i do
        if no value y in Domains_j allows (x, y) to satisfy the constraint between X_i and X_j
            then delete x from Domains_i; removed = true
    return removed
```

- Forward checking: Enforcing consistency of arcs pointing to each new assignment

Filtering: Constraint Propagation

- Forward checking doesn't provide early detection for all failures
- Constraint propagation: reasoning from constraint to constraint
  - Algorithms: AC-3 (the most popular), PC-2, *etc.*
  - Detects failure earlier than forward checking



Constraint propagation:
further remove Blue for Q

AC-3

- AC-3: repeatedly enforce arc consistency on constraint chains
- Important: If X loses a value, neighbors of X need to be rechecked

Called when the domain of X_t is reduced.

function AC-3(X_t, *Domains*) **returns** *false* if an inconsistency is found and *true* otherwise
    initialize *queue* with all arcs (X_s, X_t) for X_s in Neighbours(X_t)
    while *queue* is not empty **do**
        $(X_i, X_j) \leftarrow$ RemoveFirst(*queue*)
        if EnforceArcConsistency(*Domains*, X_i, X_j) **then**
            if size of $Domains_i = 0$ **then return** *false*
            for each X_k **in** Neighbours(X_i) **do**
                add (X_k, X_i) to *queue*
    return *true*

The domain of X_i is reduced.

Constraint propagation:

$X_k \rightarrow X_i \rightarrow X_j$

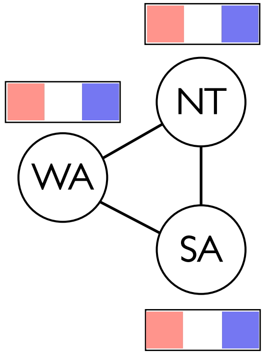
AC-3 Properties

- Assume a CSP with:
  - n variables each with domain size at most d, c binary constraints
- The complexity of AC-3: $O(\, c \cdot d \cdot d^2 \,)$

For each arc $X_k \rightarrow X_i$

Runtime for enforcing arc consistency

Inserted in the queue up to d times because X_i has at most d values to delete



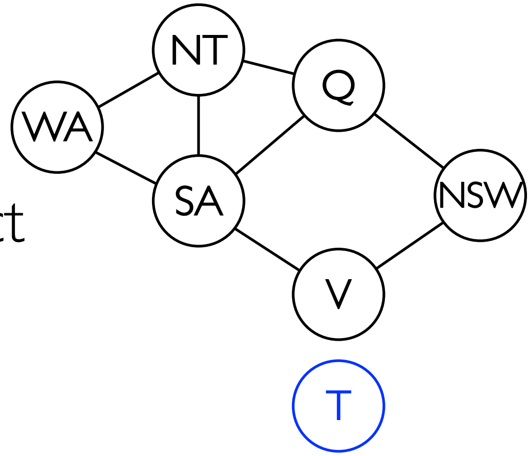
- AC-3 isn't always effective:
  - No consistent assignments, but AC-3 doesn't detect it…
  - Intuition: if we look locally at the graph, nothing blatantly wrong.

Outline

- **Constraint Satisfaction Problems**
  - Backtracking Search
  - Dynamic Ordering
  - Arc Consistency
  - **Problem Structure**
- Local Search
  - Hill Climbing
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithm

Problem Structure

- Extreme case: independent subproblems
  - Example: Tasmania and mainland do not interact
- Independent subproblems are identifiable as connected components of constraint graph

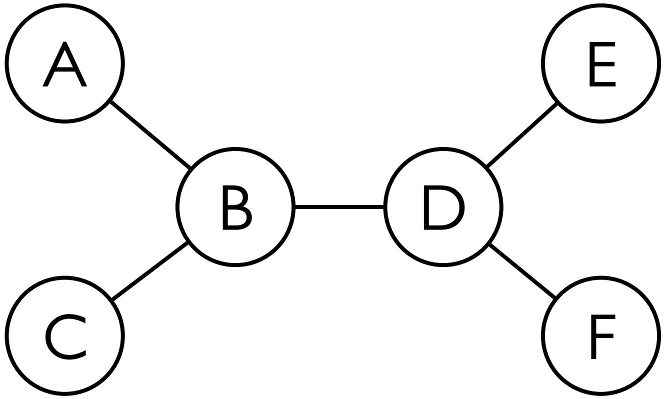


Example: Suppose a graph of n variables can be broken into subproblems of only c variables taking d values:

- Worst-case solution cost is $O(d^c n/c)$, linear in n
- E.g., $n = 80, d = 2, c = 20$, 10 million nodes/sec
- Without the decomposition: 2^{80} = 4 billion years
- With the decomposition: $4 \cdot 2^{20}$ = 0.4 seconds

Tree-Structured CSPs

- Theorem: if the constraint graph has no loops, the CSP can be solved in linear time: $O(nd^2)$
  - Compare to general CSPs, where worst-case time is $O(d^n)$

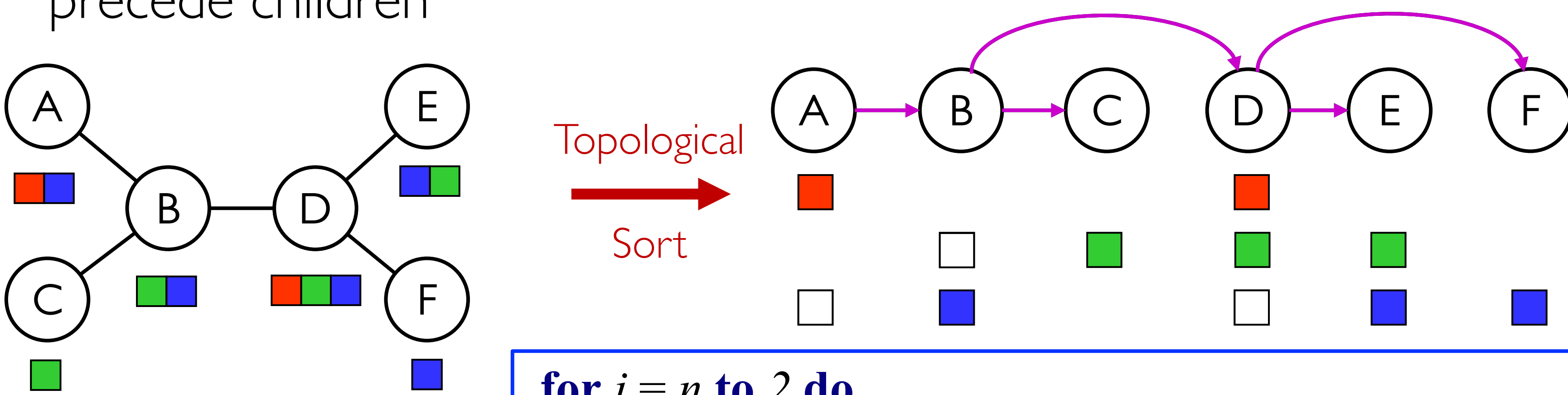


- This property also applies to probabilistic reasoning on graphs
  - An example of the relation between syntactic restrictions and the complexity of reasoning

Tree-Structured CSPs

- Algorithm for tree-structured CSPs:
  - Ordering: Choose a root variable, order variables so that parents precede children

Runtime: $O(nd^2)$



  - Remove backward:

for $j = n$ **to** 2 **do**
    EnforceArcConsistency(Parent(X_j), X_j)
    if it cannot be made consistent **then return** *failure*

  - Assign forward:

for $i = 1$ **to** n **do**
    assignment[X_i] ← any consistent value from D_i
    if there is no consistent value **then return** *failure*

Outline

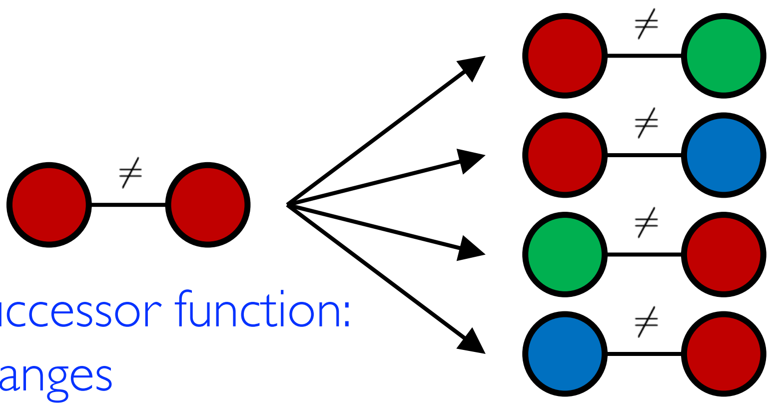
- Constraint Satisfaction Problems
  - Backtracking Search
  - Dynamic Ordering
  - Arc Consistency
  - Problem Structure
- **Local Search**
  - Hill Climbing
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithm

Local Search

- Solving CSPs is often NP-hard
  - Huge search space: exponential in the number of variables
  - All have cost $O((\max_i |\mathrm{Domains}_i|)^n)$, only useful for constants
- Alternative: Local search
  - Generally much faster and memory efficient (a constant amount)
  - But incomplete and suboptimal
- Useful method in practice
  - Best available method for many constraint satisfaction and constraint optimization problems
  - Online processing, e.g. fast flight reschedule due to weather change

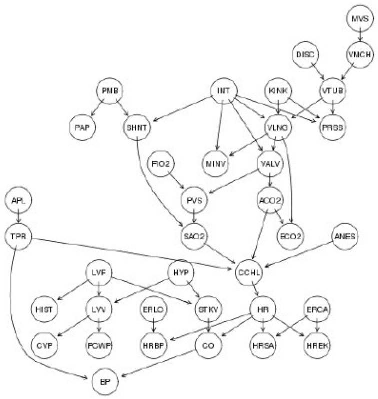
Solving CSPs: Local Search

- Tree search keeps unexplored alternatives on the frontier
  - Ensures completeness and optimality
  - Extends partial assignments
- Local search improves a single option until you can't make it better
- Apply local search to CSPs:
  - Take a complete assignment with unsatisfied constraints
  - Reassign variable values until solved
  - No frontier
  - Live on the edge

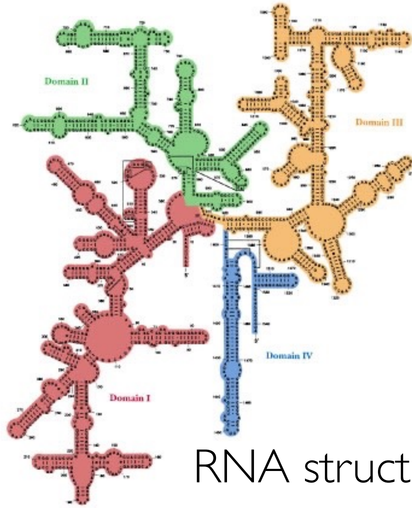


New successor function: local changes

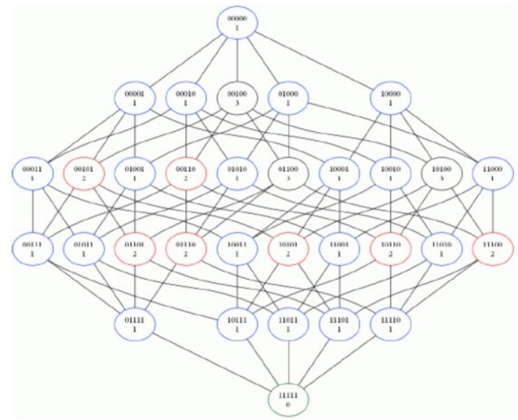
Applications of Local Search



Probabilistic Reasoning

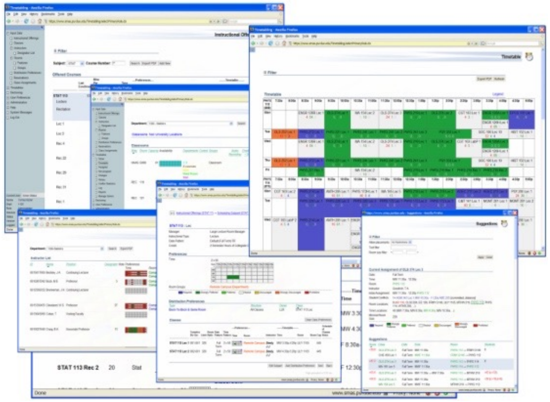


RNA structure design

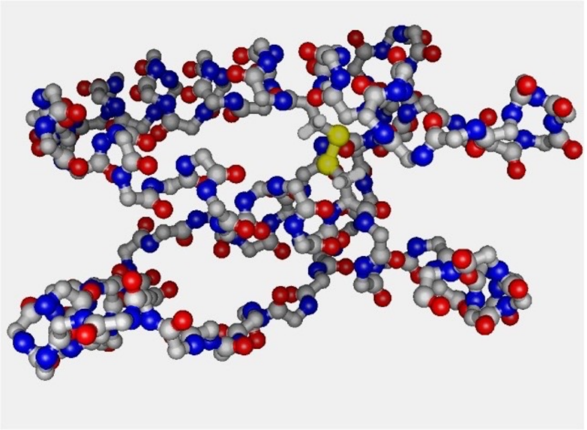


Propositional satisfiability (SAT)

Scheduling of Hubble Space Telescope: 1 week → 10 seconds



University Timetabling



Protein Folding

Outline

- Constraint Satisfaction Problems
  - Backtracking Search
  - Dynamic Ordering
  - Arc Consistency
  - Problem Structure
- **Local Search**
  - **Hill Climbing**
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithm

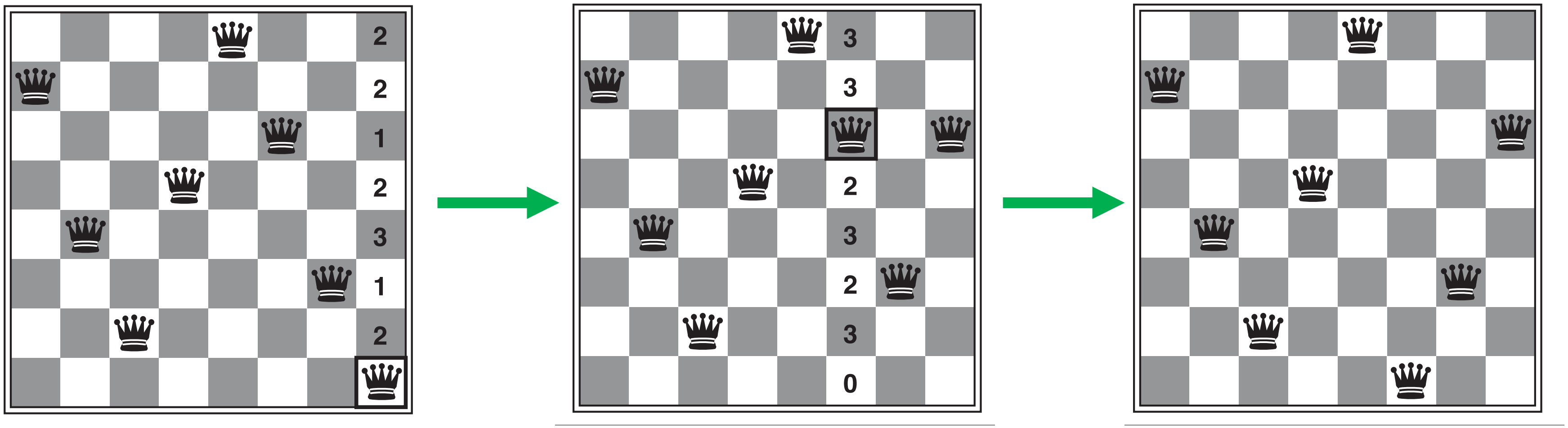
Hill Climbing

- Also called greedy local search
- Simple and general idea:
  - Start wherever
  - **Repeat:** move to the best neighboring state
  - If no neighbors better than current, quit

function HILL-CLIMBING($problem$) **returns** a state that is a local maximum
    $current \leftarrow problem$.INITIAL
    while $true$ **do**
        $neighbor \leftarrow$ a highest-valued successor state of $current$
        if VALUE($neighbor$) $\leq$ VALUE($current$) **then return** $current$
        $current \leftarrow neighbor$

Example: N-Queen

- At each stage, a queen is chosen for reassignment in its column.
- The number of conflicts (in this case, the number of attacking queens) is shown in each square.



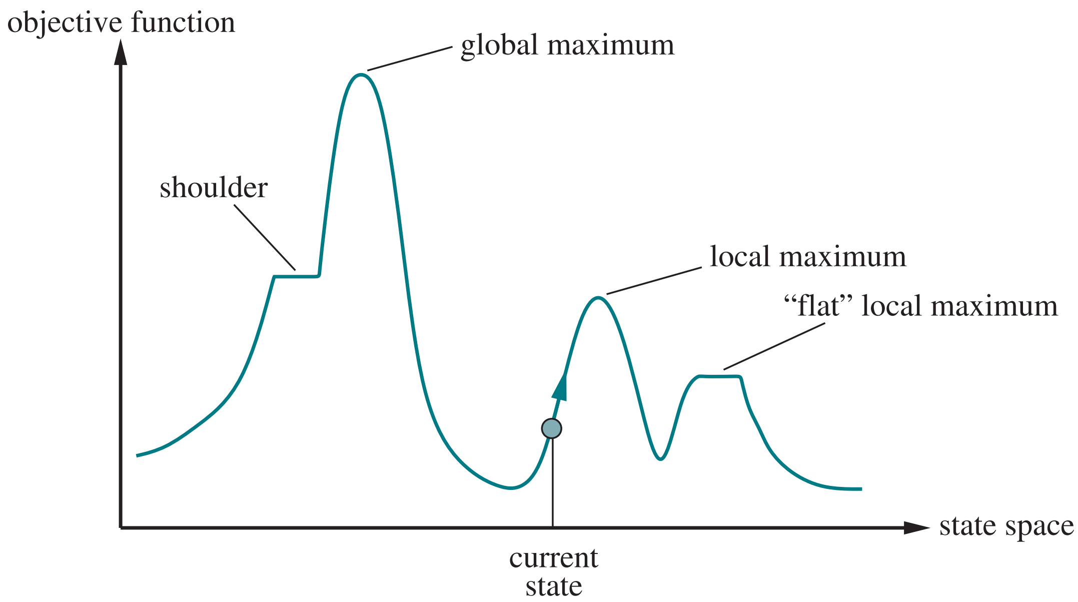
Random initialization

Reassignment for a column

Reassignment for another. Done

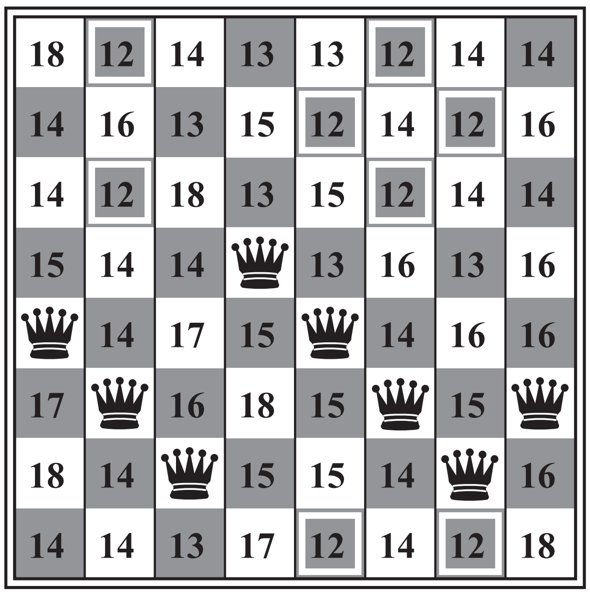
Hill Climbing Diagram

- Problems:
  - Local optima: a peak that is higher than each of its neighbors
    - Get stuck with nowhere else to go
  - Plateaus: a flat local maximum or a shoulder
    - Get lost: unable to determine in which direction it should step



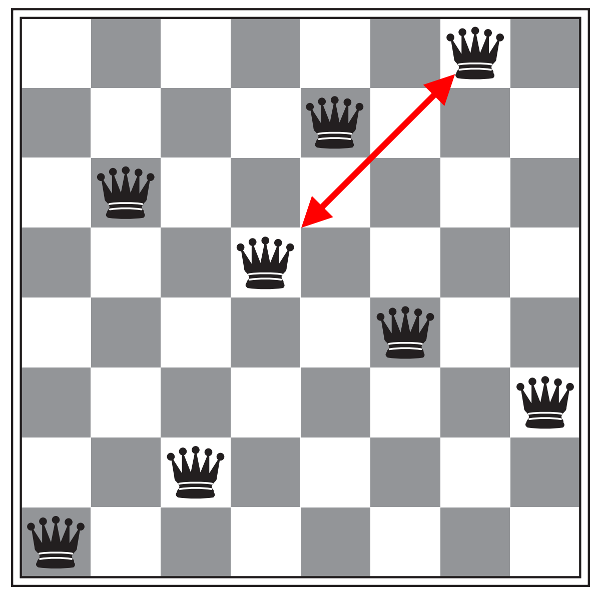
Local Optima

- Current cost: $h = 1$
- No single move can improve on this solution
  - In fact, every single move only makes things worse ($h \geq 2$)



$h = 17$

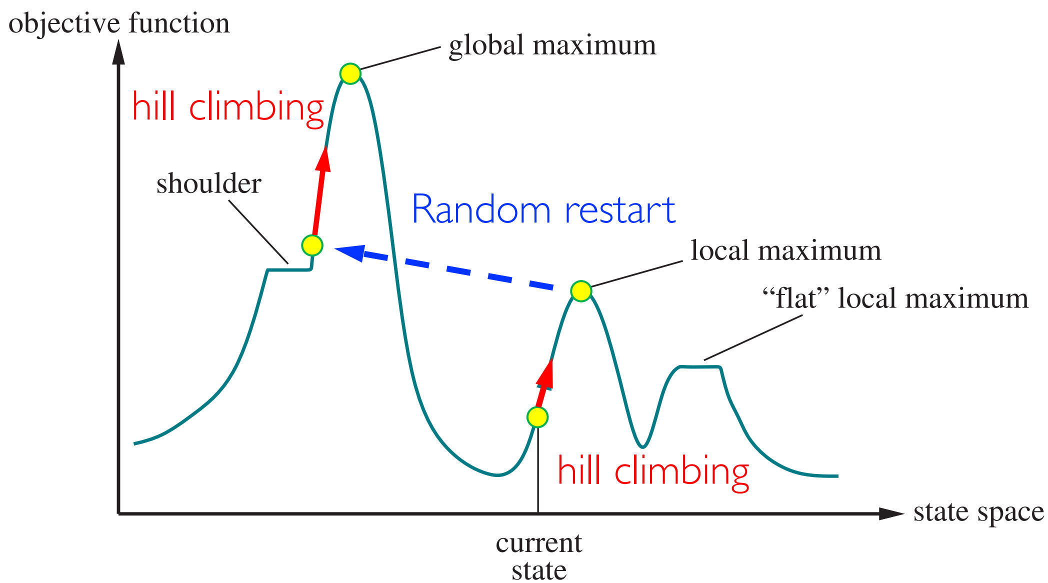
5 steps



$h = 1$

How to Improve Hill Climbing?

- Stochastic hill climbing: chooses at random from uphill moves
  - The probability of selection can vary with the steepness
- Random-restart hill climbing: a series of hill-climbing searches from randomly generated initial states, until a goal is found
  - Complete with probability approaching 1



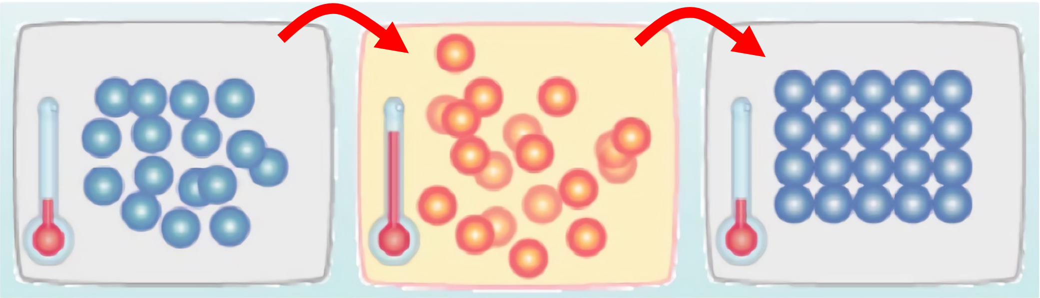
For 8-queens,
each hill-climbing search has a probability p of success
$p \approx 0.14$, meaning roughly 7 iterations for global optima

Outline

- Constraint Satisfaction Problems
  - Backtracking Search
  - Dynamic Ordering
  - Arc Consistency
  - Problem Structure
- **Local Search**
  - Hill Climbing
  - **Simulated Annealing**
  - Local Beam Search
  - Genetic Algorithm

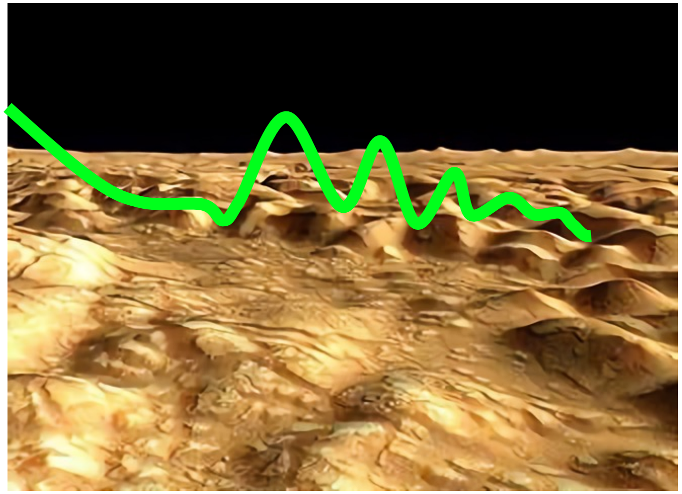
Simulated Annealing

- Simulated Annealing: physics inspired twist on random walk



- Basic idea:
  - Allow "bad" moves occasionally
  - With high temperature, more bad moves allowed, shake the system out of its local minimum
  - Gradually reduce temperature

Imagine a ping-pong

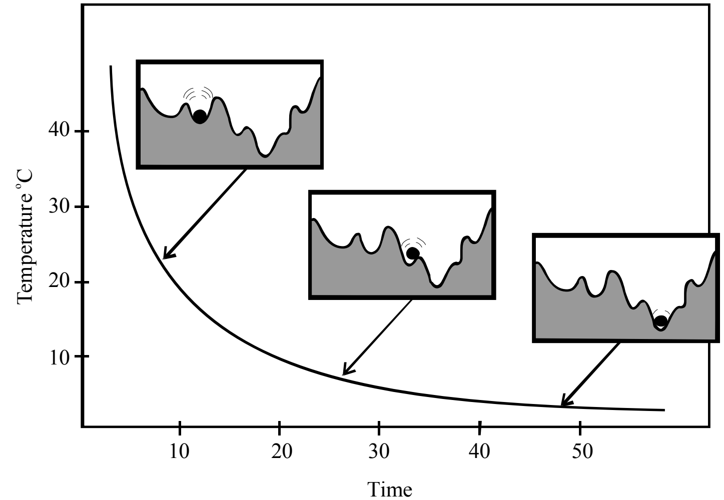


Simulated Annealing

function SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state
  current ← *problem*.INITIAL
  for t = 1 **to** ∞ **do**
    T ← *schedule*(t)
    if T = 0 **then return** *current*
    next ← a randomly selected successor of *current*
    ΔE ← VALUE(*current*) – VALUE(*next*)
    if $\Delta E > 0$ **then** *current* ← *next*
    else *current* ← *next* only with probability $\boxed{e^{\Delta E/T}}$

Acceptance Probability ($\Delta E < 0$): $e^{\Delta E/T}$

- Cooling schedule: a gradual reduction from a high value to 0
  - Exponential cooling often works best, typically at a rate of $0.7 \sim 0.9$
  - Complete with probability approaching 1

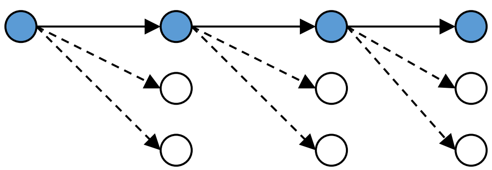


Outline

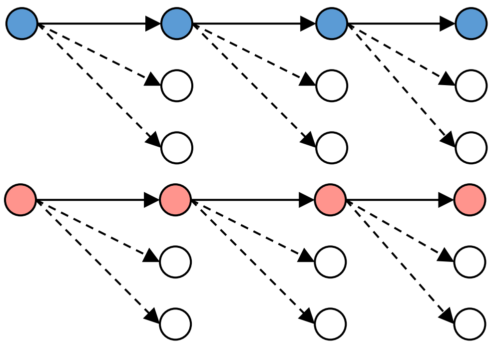
- Constraint Satisfaction Problems
  - Backtracking Search
  - Dynamic Ordering
  - Arc Consistency
  - Problem Structure
- **Local Search**
  - Hill Climbing
  - Simulated Annealing
  - **Local Beam Search**
  - Genetic Algorithm

Local Beam Search

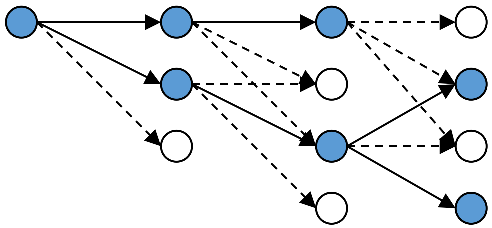
- Basic idea: greedily keep k states at all times
  - Begins with k randomly generated states
  - For each iteration
    - Generate all successors from k current states
    - Choose best k of these to be the new current states



Greedy Search



Random-restart Greedy Search

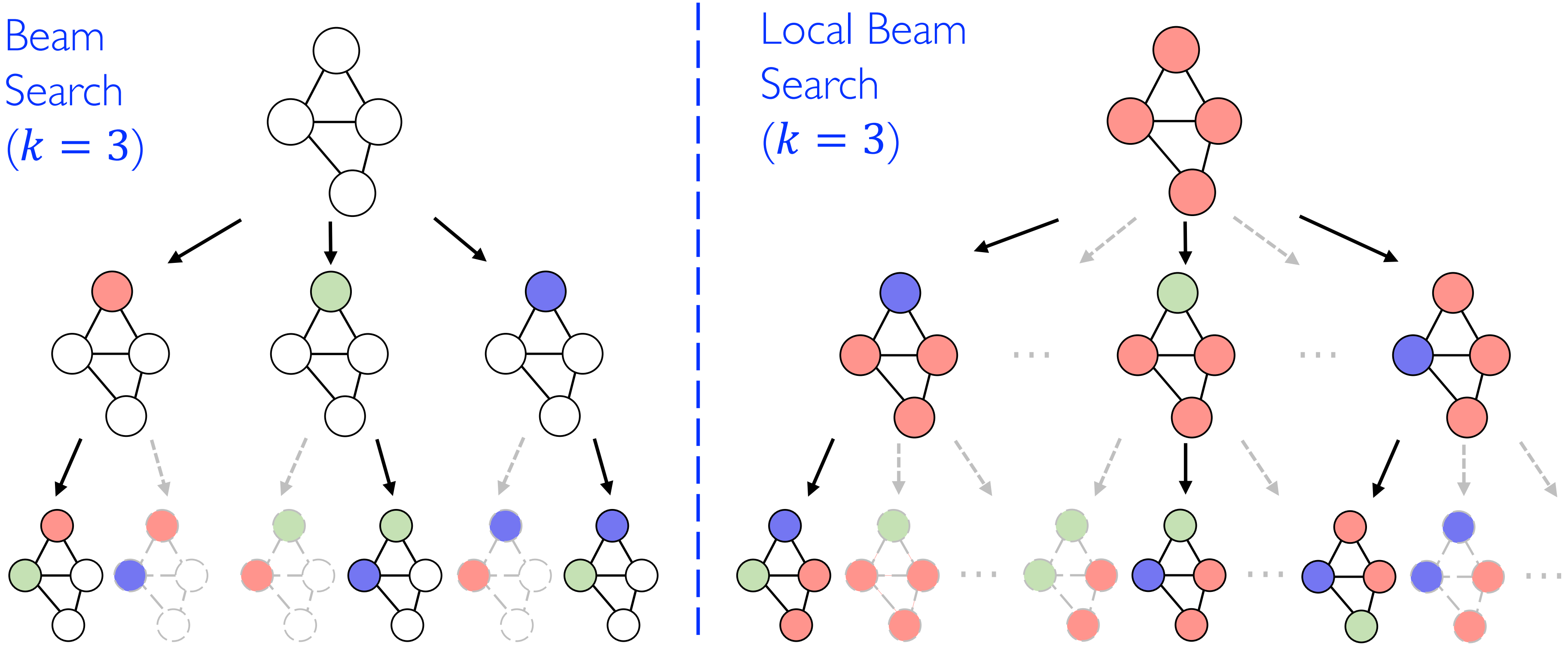


Local Beam Search ($k = 2$)

Information is shared among k search threads.

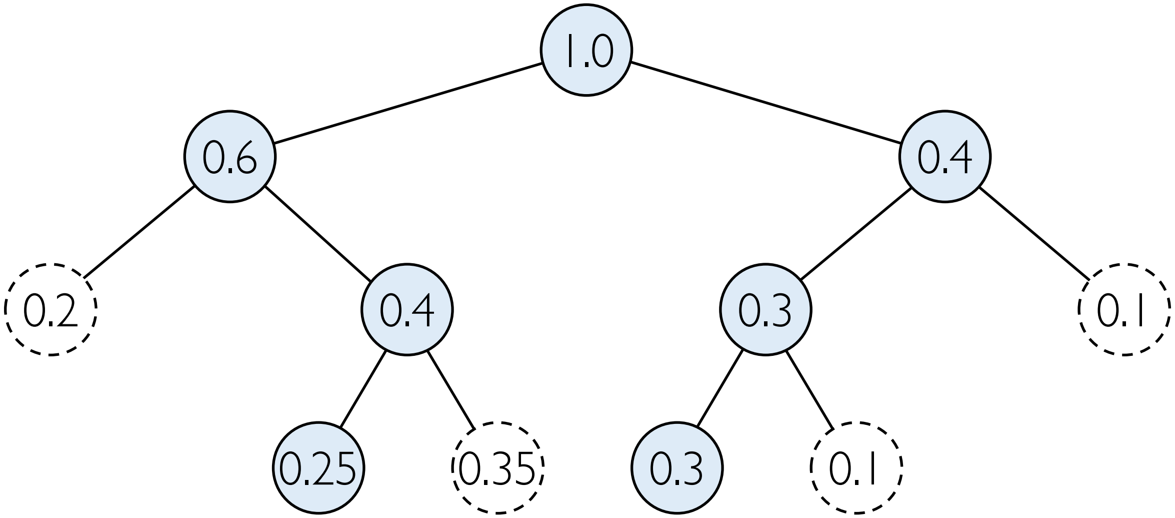
Beam Search vs. Local Beam Search

- Beam Search: a **path-based** algorithm
  - Not guaranteed to find optimal assignment
- Running time: $O(n(kb)\log(kb))$ with branching factor b



Stochastic Beam Search

- Local beam search can suffer from lack of diversity among k states
  - They can quickly become concentrated in a small region of the state space.
- Stochastic beam search
  - Chooses k from the the pool of candidate successors at random
  - Probability of chosen: an increasing function of its value

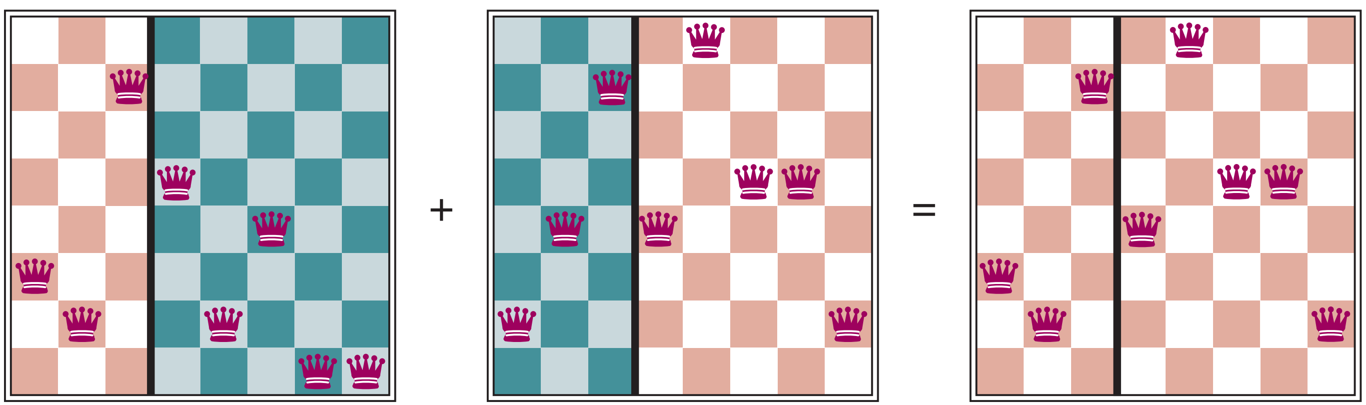


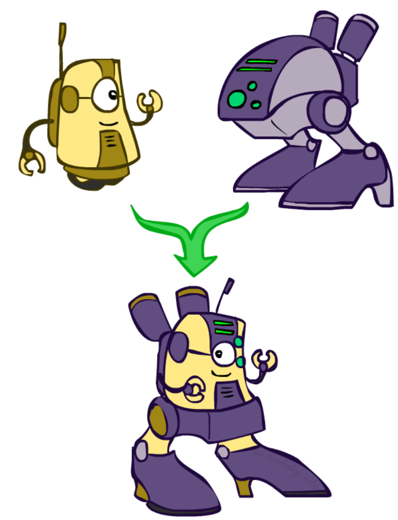
Outline

- Constraint Satisfaction Problems
  - Backtracking Search
  - Dynamic Ordering
  - Arc Consistency
  - Problem Structure
- **Local Search**
  - Hill Climbing
  - Simulated Annealing
  - Local Beam Search
  - **Genetic Algorithm**

Genetic Algorithm (GA)

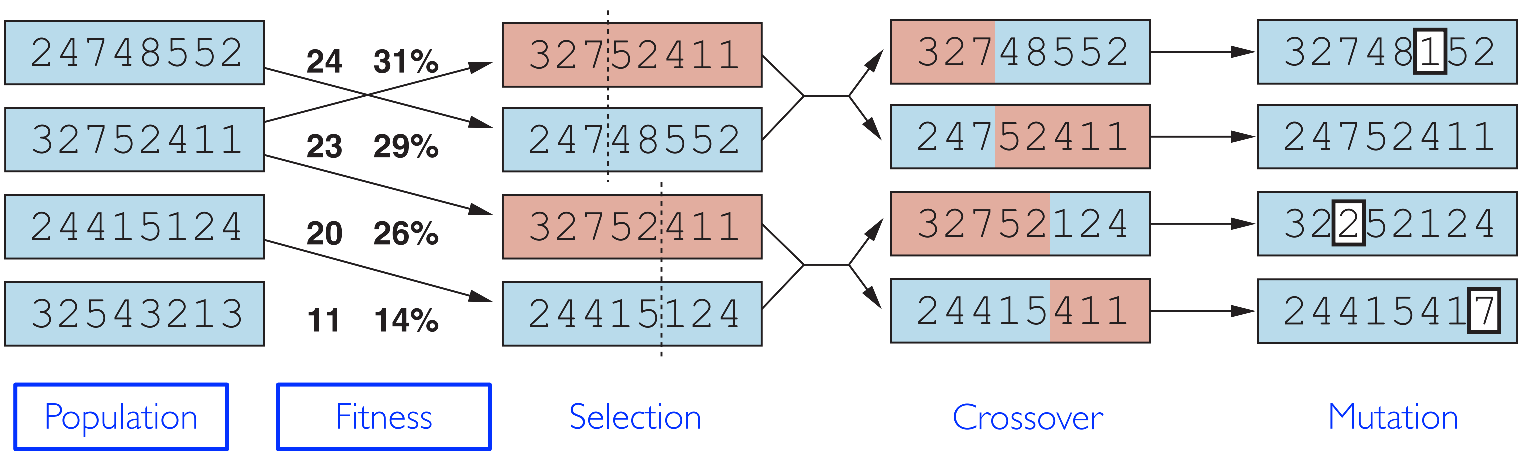
- Basic Idea:
  - A variant of stochastic beam search
  - Successor states are generated by combining two parent states rather than by modifying a single state
  - Hill Climbing + Stochastic Exploration + Parallel Communication
- Example: 8-queens





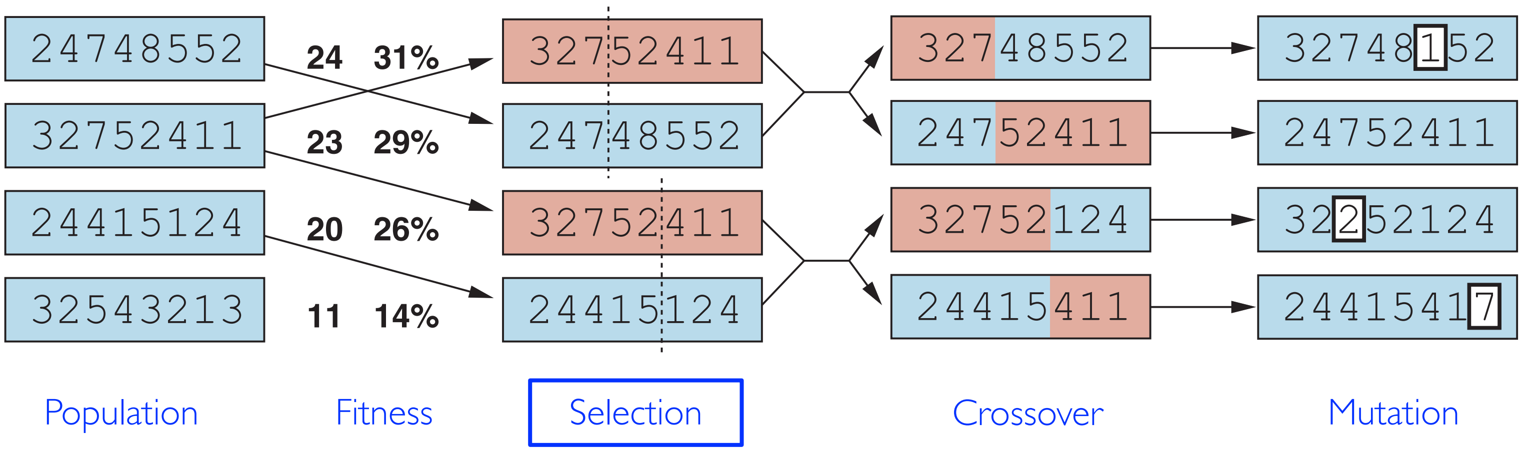
Genetic Algorithm: Population

- **Population:** initially a set of k randomly generated states
- **Individual (state):** represented as a **chromosomes** over a finite alphabet
  - 8-queens problem: $Q_1 Q_2 \cdots Q_8$
- **Fitness function:** the objective function
  - 8-queens problem: the number of nonattacking pairs of queens



Genetic Algorithm: Selection

- Selection: pairs are selected at random for reproduction
  - The probability of being chosen is directly proportional to the fitness score
  - There are also many variants of this selection rule
  - Knowledge engineering is important: should select modular parts

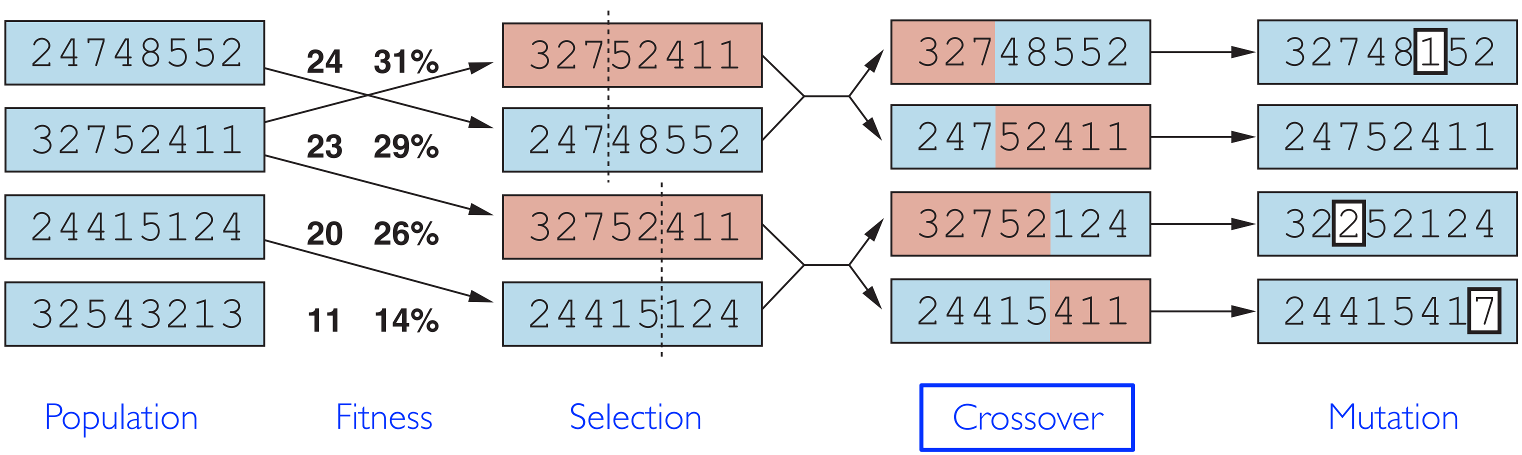


Genetic Algorithm: Crossover

- **Crossover**: offspring are created by crossing over the parent strings
  - Crossover point: chosen randomly from the positions in the string

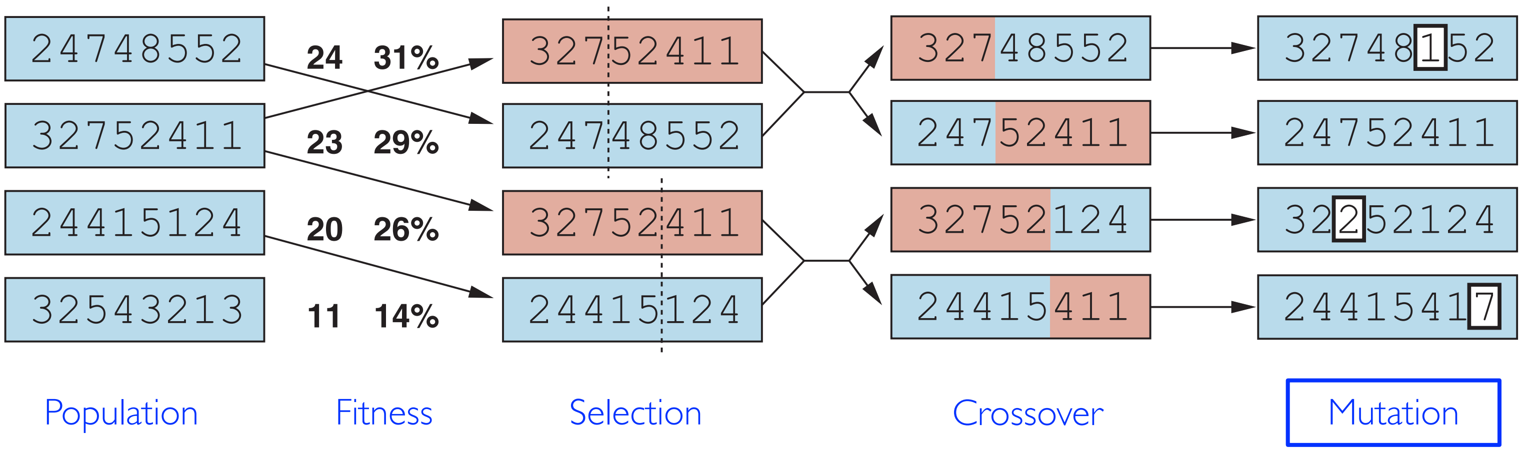
> **Intuition:** Like Simulated Annealing, crossover
> - **Early**: takes large steps with diverse population
> - **Later**: takes smaller steps with similar individuals

Simulated Annealing



Genetic Algorithm: Mutation

- Mutation: each location is subject to random mutation with a small independent probability
  - Mutation increases the diversity of individuals in the population
  - Good mutation with higher fitness score will gain more popularity
    - The fittest survive by natural selection. -- Darwinism

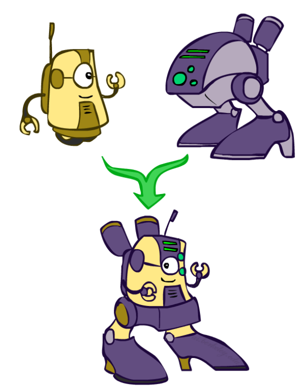


Genetic Algorithm

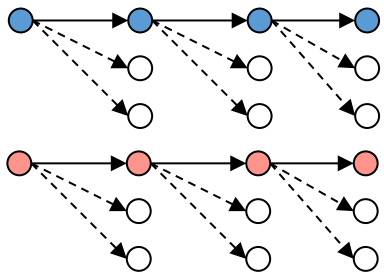
```
function GENETIC-ALGORITHM(population, fitness) returns an individual
  repeat
      weights ← WEIGHTED-BY(population, fitness)
      population2 ← empty list
      for i = 1 to SIZE(population) do
Selection   parent1, parent2 ← WEIGHTED-RANDOM-CHOICES(population, weights, 2)
Crossover   child ← REPRODUCE(parent1, parent2)
Mutation    if (small random probability) then child ← MUTATE(child)
          add child to population2
      population ← population2
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
  n ← LENGTH(parent1)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

Crossover details

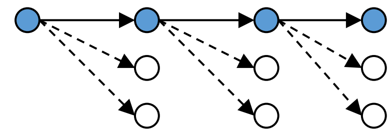


Small Picture of Local Search



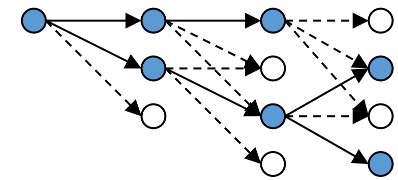
Random-restart Hill Climbing

Random Restart



Hill Climbing
(Greedy Local Search)

Keeping k states



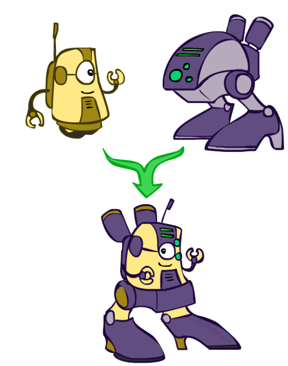
Local Beam Search

Randomization

Stochastic Beam Search

Evolutionism

Genetic Algorithm (GA)

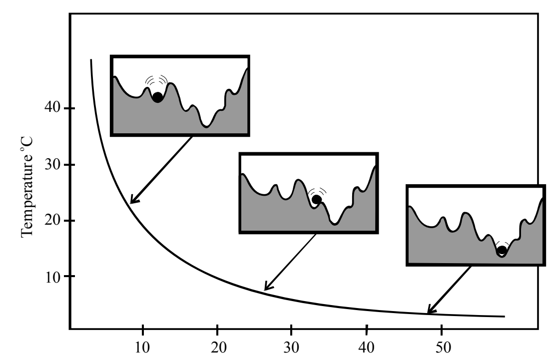


Randomization

Stochastic Hill Climbing

Allowing bad moves

Simulated Annealing



Thank You

Questions?

Mingsheng Long
mingsheng@tsinghua.edu.cn
http://ise.thss.tsinghua.edu.cn/∼mlong
答疑：东主楼11区413室

[Some slides adapted from Dan Klein and Pieter Abbeel]