APPLIED ALGORITHMS

SOLVING PROBLEM BY SEARCHING

Lecture 5: Local Search

T.S Ha Minh Hoang
Th.S Nguyen Minh Anh

Phenikaa University

Last Update: 6th March 2023

Recap

- CSP
- Backtracking
- Forward Checking and Heuristics

Backtracking search in the worst case performs an exhaustive DFS of the entire search tree, which can take a very very long time. How do we avoid this?

Outline

Search for the optimal configuration

Constraint satisfaction problem

- **Objective**: find a configuration that satisfies all constraints

Optimization problem

- **Objective**: find the best configuration that optimize (min/max) an **objective function**
- The **objective function**: reflects our preference towards each configuration (or state)

The search space

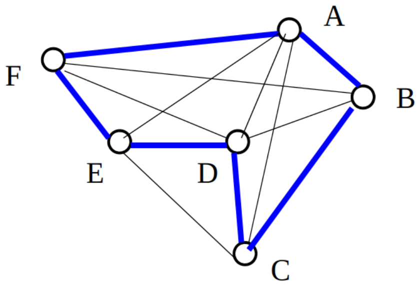
If the search space of configurations is

- **Discrete** or **finite**
  - then it is a **combinatorial optimization problem**
- **Continuous**
  - then it is a **parametric/continous optimization problem**

Example: Traveling salesman problem

Problem:

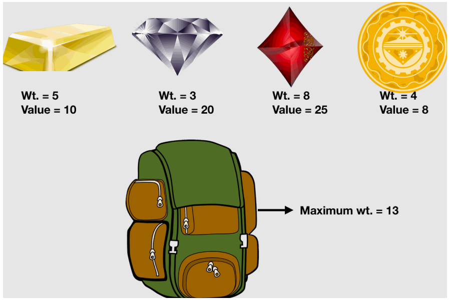
- A graph with distances
- Find tour - a path that visits every city once and returns to the start (e.g. ABCDEF)
- Goal: the shortest tour



Example: Knapsack Problem

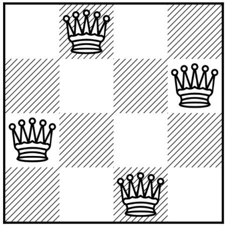
Problem:

- Given precious *n* items, each associated with a value
- Pick the items into your bag which has a capacity limitation
- Goal: maximizes the overall value of items in your bag



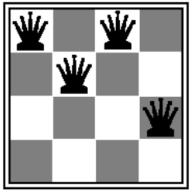
The N-Queens

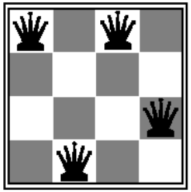
- A CSP problem
- Is it possible to formulate the problem as an **optimization problem**?

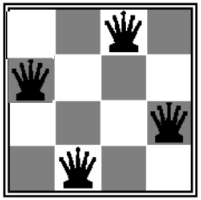


The N-Queens

- A CSP problem
- Is it possible to formulate the problem as an **optimization problem**? **Yes**
- The **quality of a configuration in a CSP** can be measured by **the number of violated constraints**
- **Solving**: **minimize** the *number of constraint violations*


# of violations =3


# of violations =1


# of violations =0

Outline

Motivation

Some issues of the search algorithms we have discussed so far:

- The search algorithms we have seen so far include systematic search (breadth-first, depth-first, iterative deepening, etc.) where we look at the entire search space in a systematic manner till we have found a goal (or all goals, if we have to).
- We also have seen heuristic search (best-first, A*-search) where we compute an evaluation function for each candidate node and choose the one that has the best heuristic value (the lowest heuristic value is usually the best).
- In both systematic search and heuristic search, we have to keep track of what's called the frontier (some books call it open or agenda) data structure that keeps track of all the candidate nodes for traversal in the next move.
- We make a choice from this frontier data structure to choose the next node in the search space we go to.
- We usually have to construct the path from the start node to the goal node to solve a problem.

Motivation

- In many optimization problems, we need to find an optimal path to the goal from a start state (e.g., find the shortest path from a start city to an end city), but in many other optimization problems, the path is irrelevant.
- The goal state itself is the solution.
- Here, it is not important how we get to the goal state, we just need to know the goal state.
- **Idea:** iterative improvement algorithms: keep a single "current" state, and try to improve it.

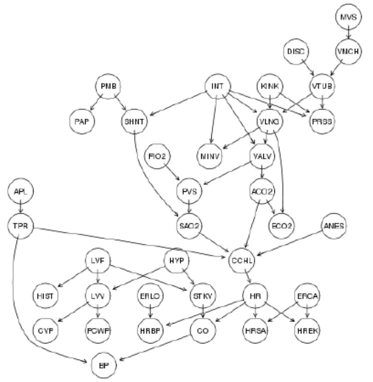
Local Search: Motivation

- Solving CSPs is NP-hard
  - Search space for many CSPs is huge
  - Exponential in the number of variables
  - Even arc consistency with domain splitting is often not enough

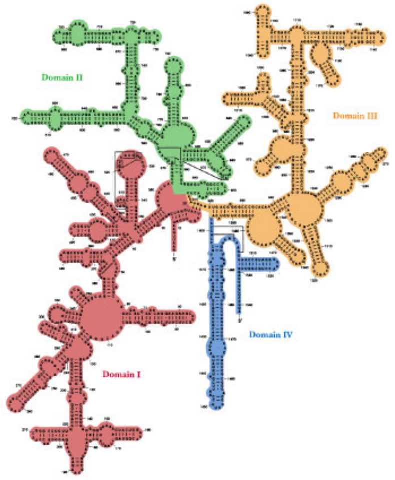
- Alternative: local search
  - Often finds a solution quickly
  - But cannot prove that there is no solution

- Useful method in practice
  - Best available method for many constraint satisfaction and constraint optimization problems
  - Extremely general!
    - Works for problems other than CSPs
    - E.g. arc consistency only works for CSPs

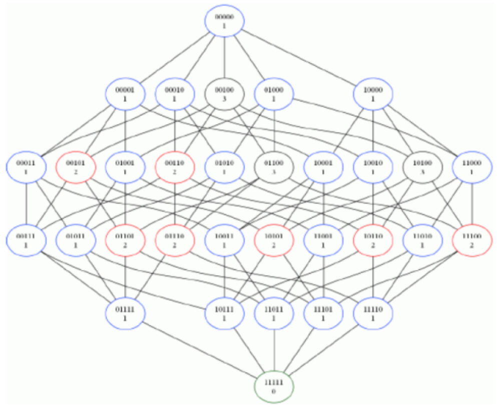
Some Successful Application Areas for Local Search



Probabilistic Reasoning

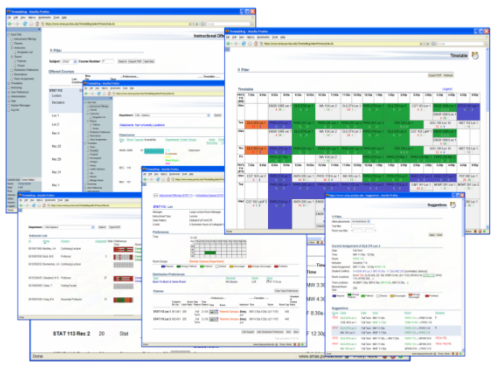


RNA structure design

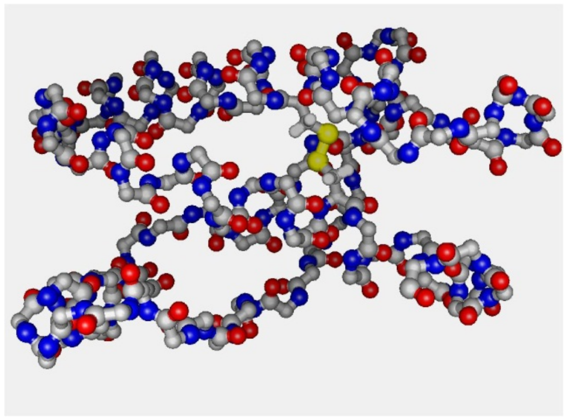


Propositional satisfiability (SAT)

Scheduling of Hubble Space Telescope:
1 week → 10 seconds



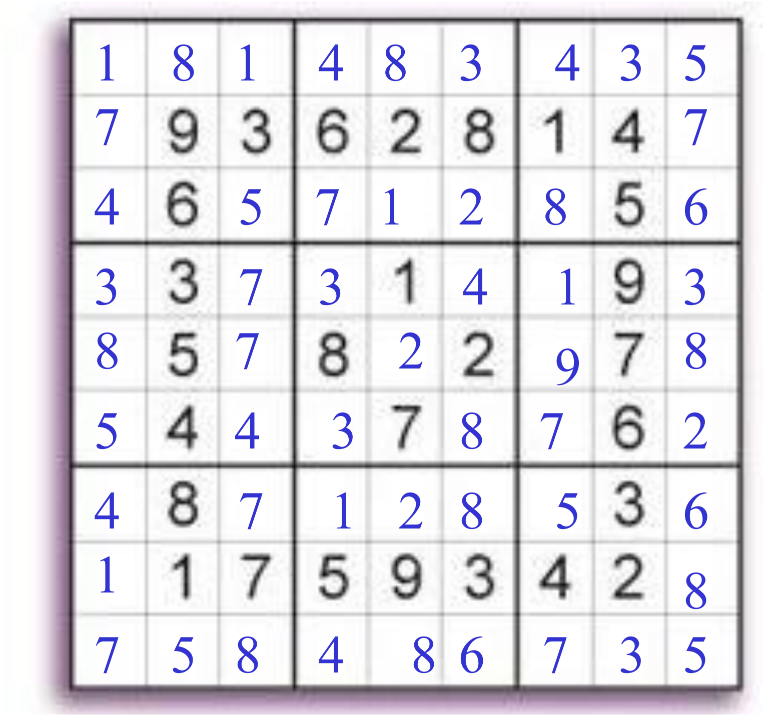
University Timetabling



Protein Folding

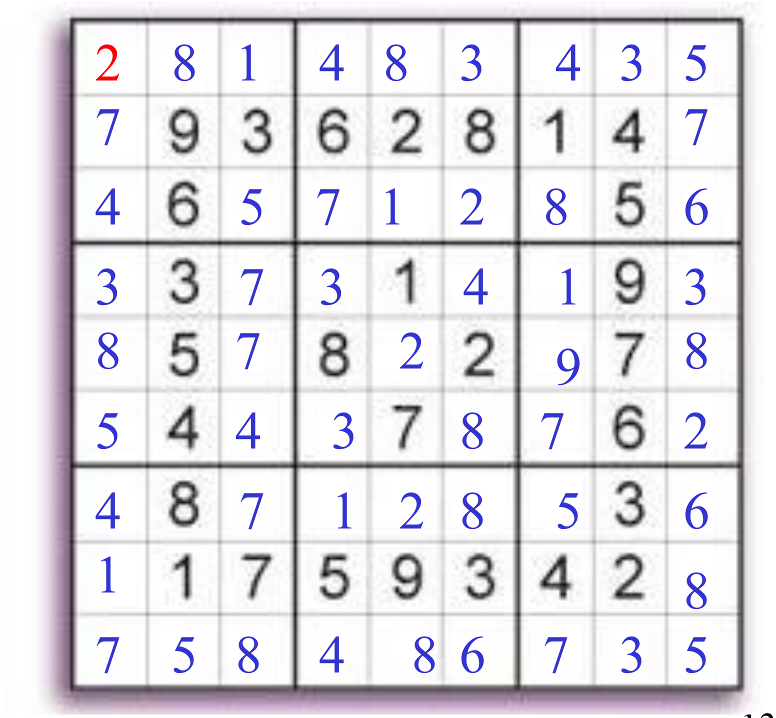
Local Search

- Idea:
  - Consider the space of complete assignments of values to variables (all possible worlds)
  - Neighbours of a current node are similar variable assignments
  - Move from one node to another according to a function that scores how good each assignment is



1	8	1	4	8	3	4	3	5
7	9	3	6	2	8	1	4	7
4	6	5	7	1	2	8	5	6
3	3	7	3	1	4	1	9	3
8	5	7	8	2	2	9	7	8
5	4	4	3	7	8	7	6	2
4	8	7	1	2	8	5	3	6
1	1	7	5	9	3	4	2	8
7	5	8	4	8	6	7	3	5

$\longrightarrow$



2	8	1	4	8	3	4	3	5
7	9	3	6	2	8	1	4	7
4	6	5	7	1	2	8	5	6
3	3	7	3	1	4	1	9	3
8	5	7	8	2	2	9	7	8
5	4	4	3	7	8	7	6	2
4	8	7	1	2	8	5	3	6
1	1	7	5	9	3	4	2	8
7	5	8	4	8	6	7	3	5

Local Search Problem: Definition

Definition: A local search problem consists of a:

CSP: a set of variables, domains for these variables, and constraints on their joint values. A node in the search space will be a complete assignment to all of the variables.

Neighbour relation: an edge in the search space will exist when the neighbour relation holds between a pair of nodes.

Scoring function: h(n), judges cost of a node (want to minimize)

- E.g. the number of constraints violated in node n.
- E.g. the cost of a state in an optimization context.

Backtracking vs Local Search

Backtracking: extend partial assignments

Local search: modify complete assignments

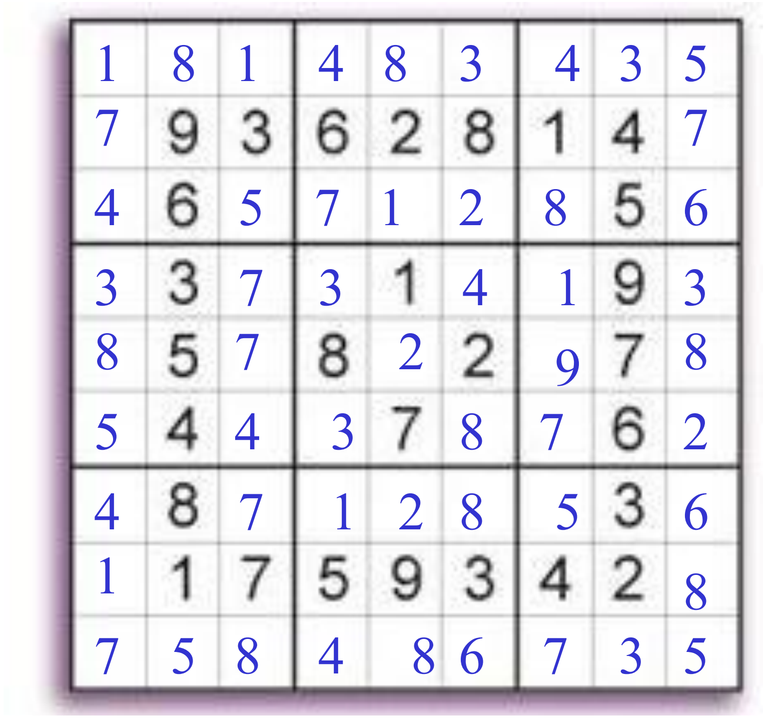
Example: Sudoku as a local search problem

CSP: usual Sudoku CSP

- One variable per cell; domains {1,…,9};
- Constraints:
  each number occurs once per row, per column, and per 3x3 box

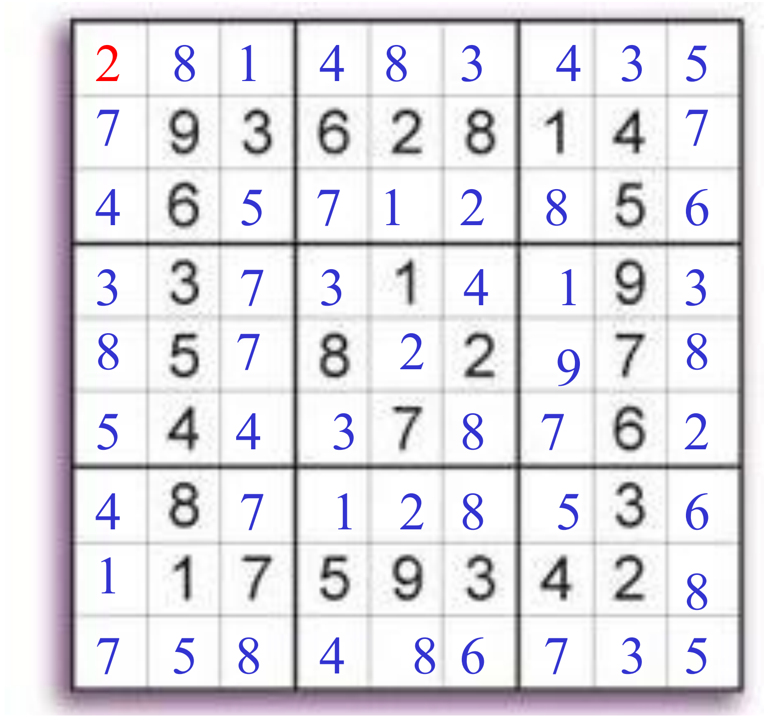
Neighbour relation: value of a single cell differs

Scoring function: number of constraint violations



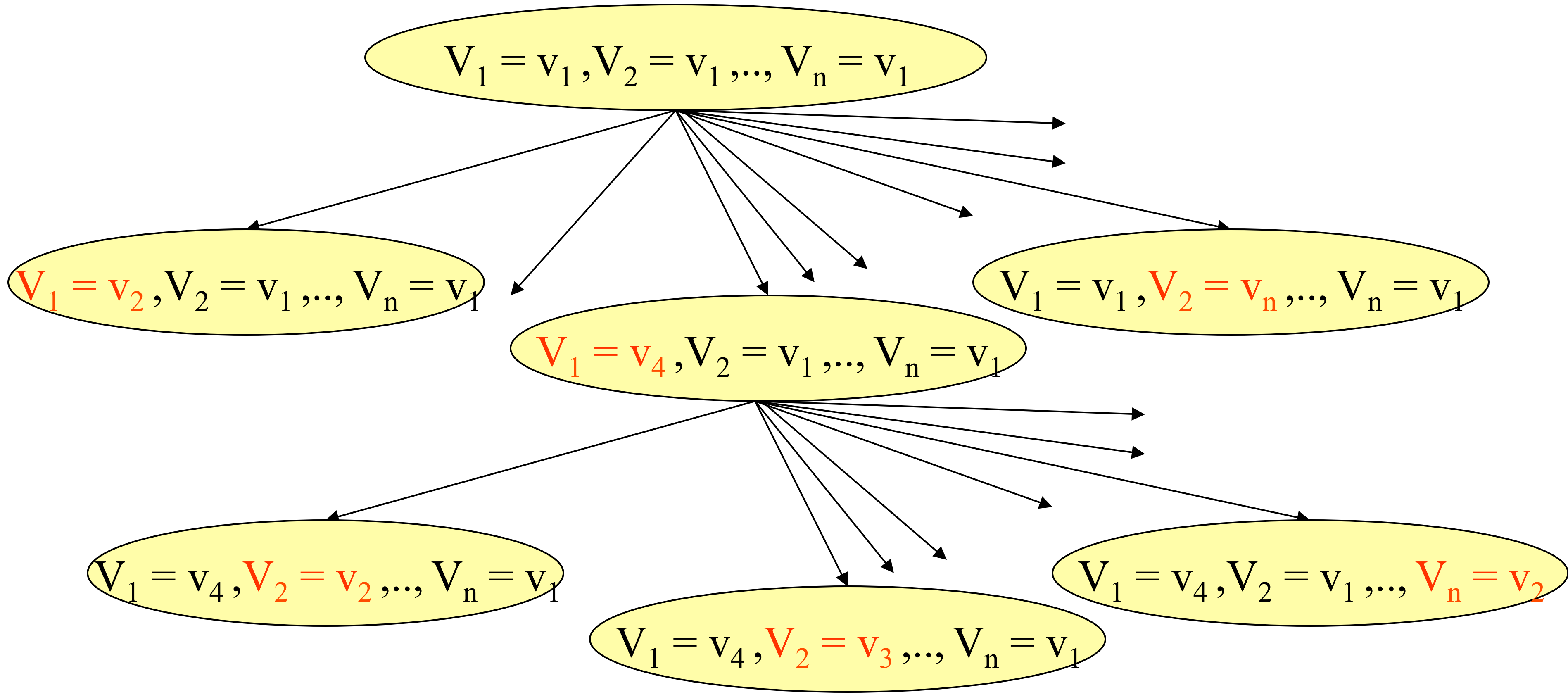
1	8	1	4	8	3	4	3	5
7	9	3	6	2	8	1	4	7
4	6	5	7	1	2	8	5	6
3	3	7	3	1	4	1	9	3
8	5	7	8	2	2	9	7	8
5	4	4	3	7	8	7	6	2
4	8	7	1	2	8	5	3	6
1	1	7	5	9	3	4	2	8
7	5	8	4	8	6	7	3	5

$\longrightarrow$



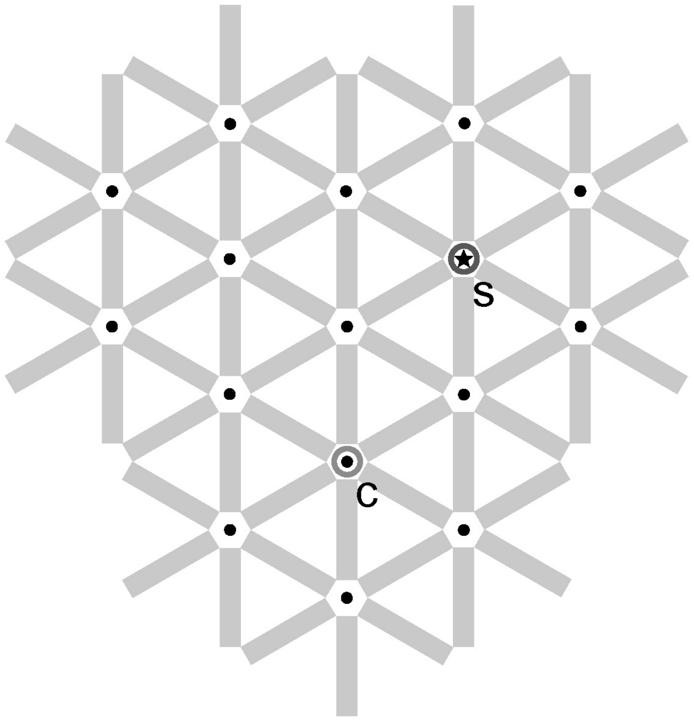
2	8	1	4	8	3	4	3	5
7	9	3	6	2	8	1	4	7
4	6	5	7	1	2	8	5	6
3	3	7	3	1	4	1	9	3
8	5	7	8	2	2	9	7	8
5	4	4	3	7	8	7	6	2
4	8	7	1	2	8	5	3	6
1	1	7	5	9	3	4	2	8
7	5	8	4	8	6	7	3	5

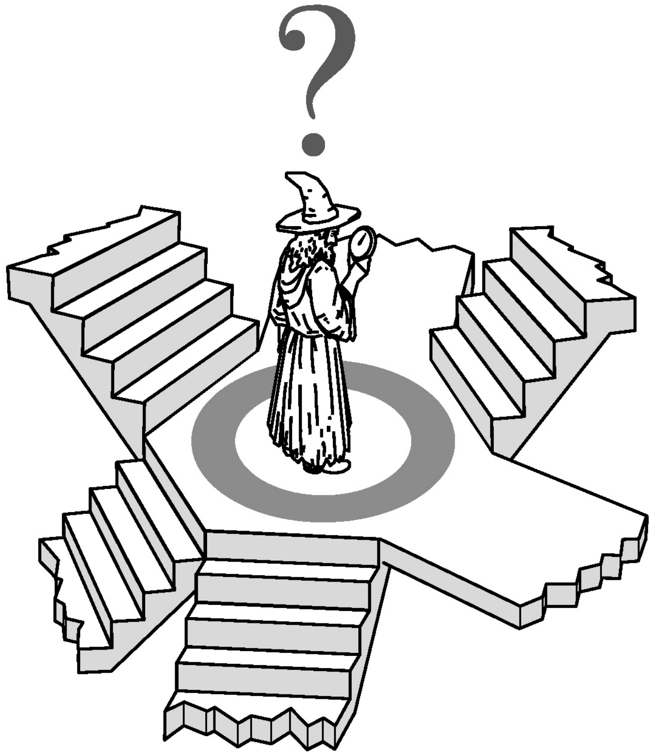
Search Space for Local Search


$V_1 = v_1, V_2 = v_1, \ldots, V_n = v_1$
$V_1 = v_2, V_2 = v_1, \ldots, V_n = v_1$
$V_1 = v_4, V_2 = v_1, \ldots, V_n = v_1$
$V_1 = v_1, V_2 = v_n, \ldots, V_n = v_1$
$V_1 = v_4, V_2 = v_2, \ldots, V_n = v_1$
$V_1 = v_4, V_2 = v_3, \ldots, V_n = v_1$
$V_1 = v_4, V_2 = v_1, \ldots, V_n = v_2$

Only the current node is kept in memory at each step.
Very different from the systematic tree search approaches we have seen so far! Local search does NOT backtrack!

Local search: only use local information



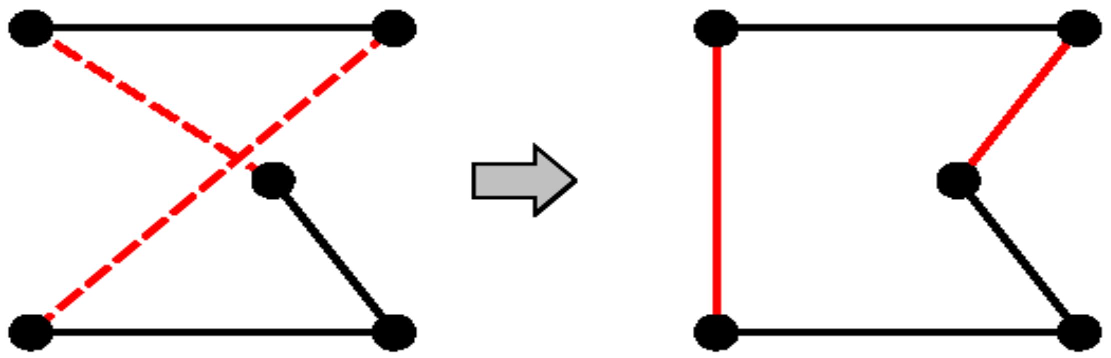


Iterative improvement

- The evaluation can be the length of the tour in the TSP problem.

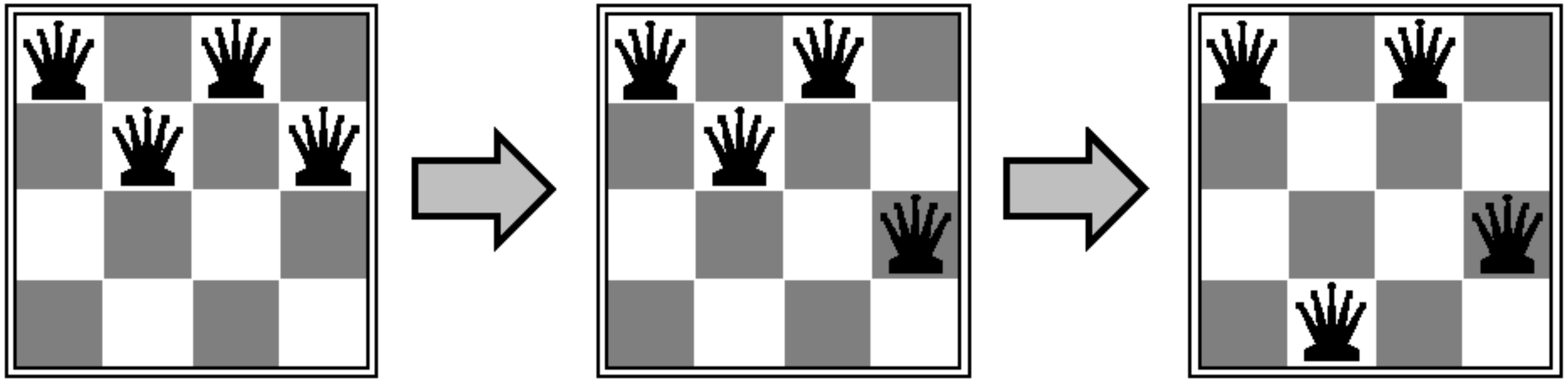
Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges



Iterative improvement example: n-queens

- Goal: Put n chess queens on an n x n board, with no two queens on the same row, column, or diagonal.



- Here, goal state is initially unknown but is specified by constraints that it must satisfy.

An 8-queens state and its successor

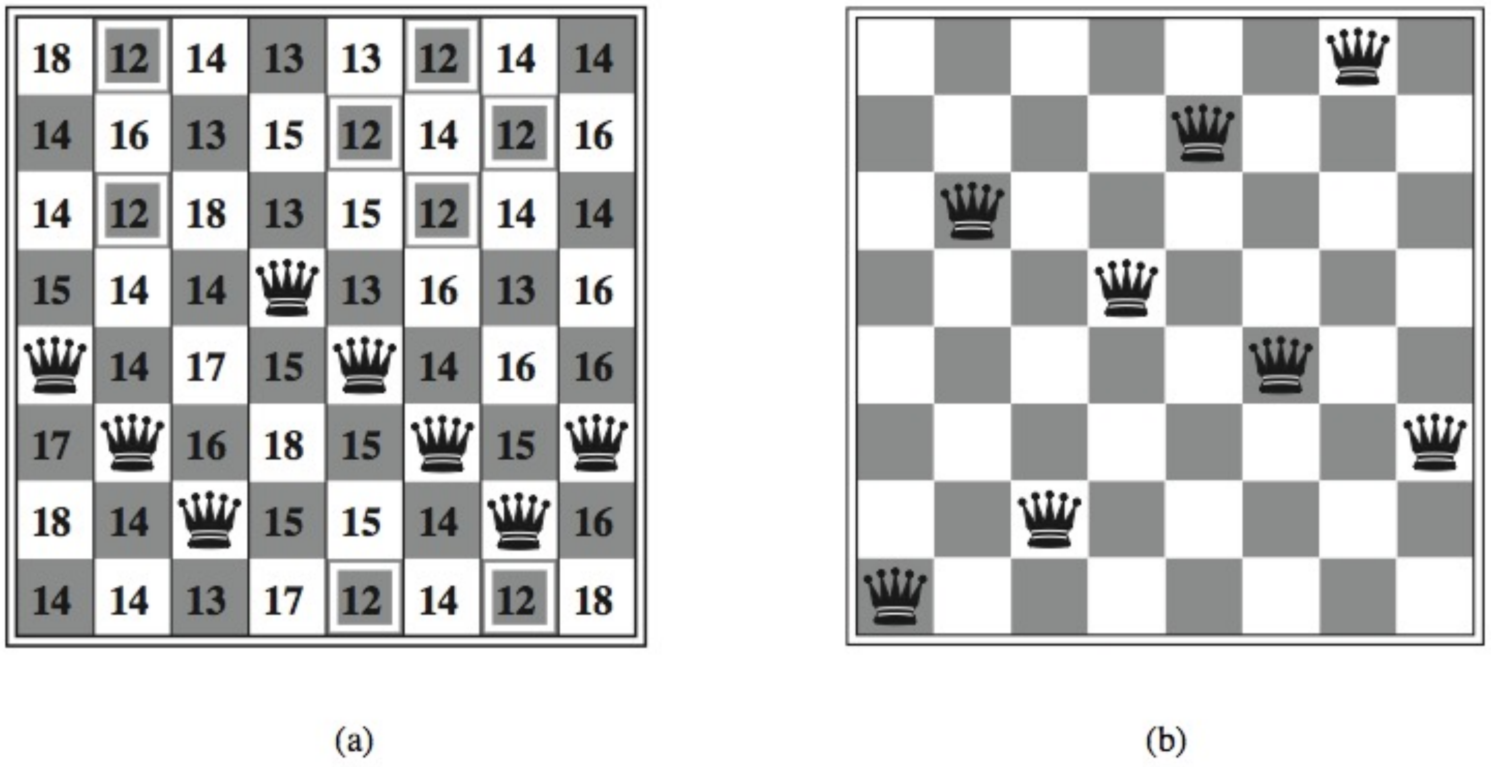
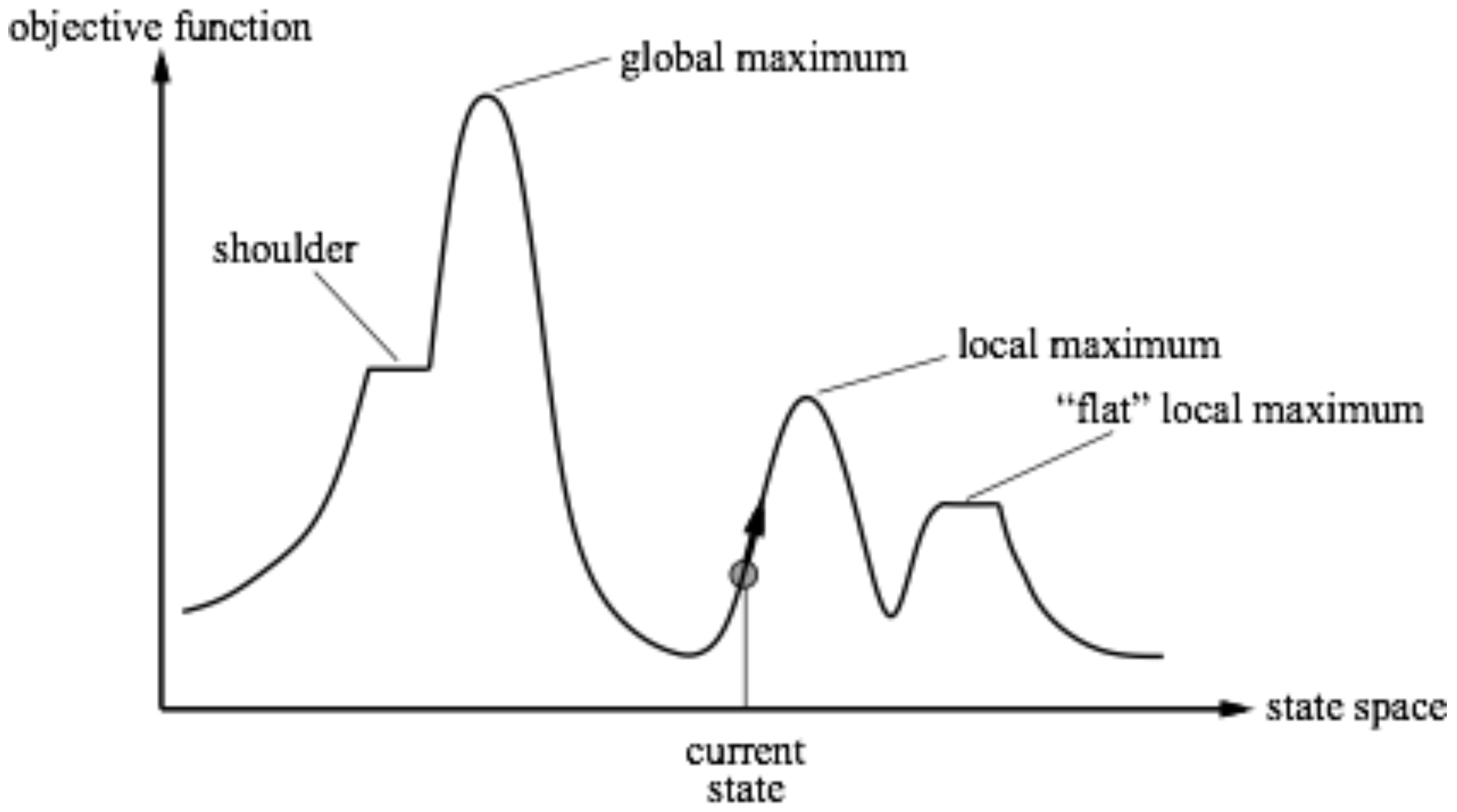


Figure 4.3 **FILES: figures/8queens-successors.eps (Wed Nov 4 16:23:55 2009) figures/8queens-local-minimum.eps (Wed Nov 4 16:14:15 2009).** (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Plotting evaluations of a space

- A search space can look like the graph below. Here we assume state space has only one dimension.



Example: N-queens

Generate possible neighbors for the following state: 1 relocate

Example: N-queens

Generate possible neighbors for the following state: 2 relocate

Components of a local search

Some issues of the search algorithms we have discussed so far

Properties of a local search

Some issues of the search algorithms we have discussed so far

Outline

Hill climbing (or gradient ascent/descent)

- This is a very simple algorithm for finding a local maximum
- Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.
- In other words, we start from an initial position (possibly random), the move along the direction of steepest ascent/gradient (descent/negative of gradient, for minimization), go along the direction for a little while; and repeat the process.

Some ideas on making Hill-climbing better

- Allowing sideways moves
  - When stuck on a ridge or plateau (i.e., all successors have the same value), allow it to move anyway hoping it is a shoulder and after some time, there will be a way up.
- How many sideways moves?
  - If we always allow sideways moves when there are no uphill moves, an infinite loop may occur if it's not a shoulder.
- Solution
  - Put a limit on the number of consecutive sideways moves allowed
- Experience with a real problem
  - The authors allowed 100 consecutive sideways moves in the 8-queens problem.
  - This raises the percentage of problems solved from 14% to 94%
  - However, now it takes on average 21 steps when successful and 64 steps when it fails.

Outline

Outline

Local search algorithms

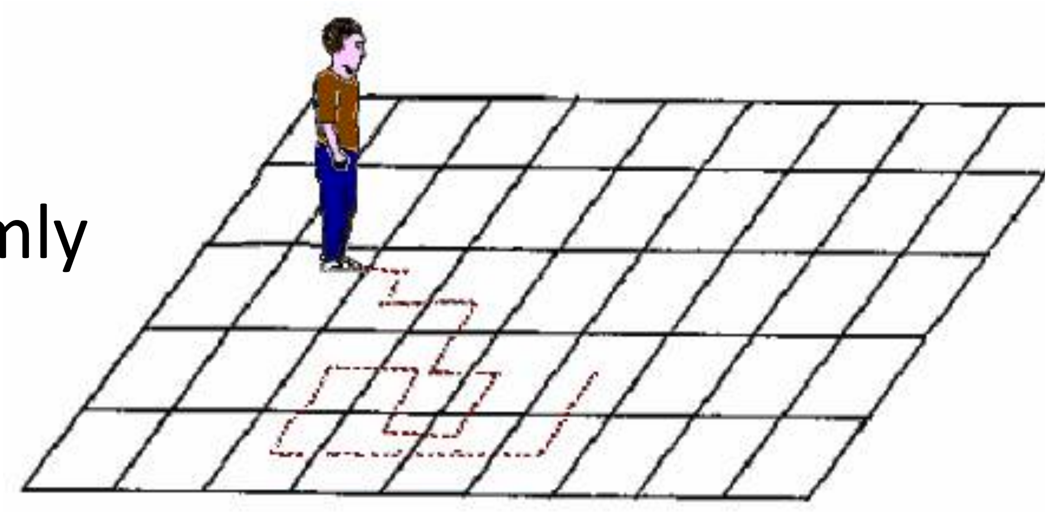
- State space = set of "complete" configurations
- Find configuration satisfying constraints,
  - e.g., all n-queens on board, no attacks
- In such cases, we can use local search algorithms
- Keep a single "current" state, try to improve it.
- Very memory efficient
  - *duh* - only remember current state

Local Search and Optimization

- Local search
  - Keep track of single current state
  - Move only to “neighboring” state

    Defined by operators
  - Ignore previous states, path taken
- 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 path-cost/goal-state formulation
  - Local search can do quite well on these problems.

Trivial Algorithms



- Random Sampling
  - Generate a state randomly
- Random Walk
  - Randomly pick a neighbor of the current state
- Why even mention these?
  - Both algorithms asymptotically complete.
  - http://projecteuclid.org/download/pdf_1/euclid.aop/1176996718 for Random Walk

Hill-climbing search

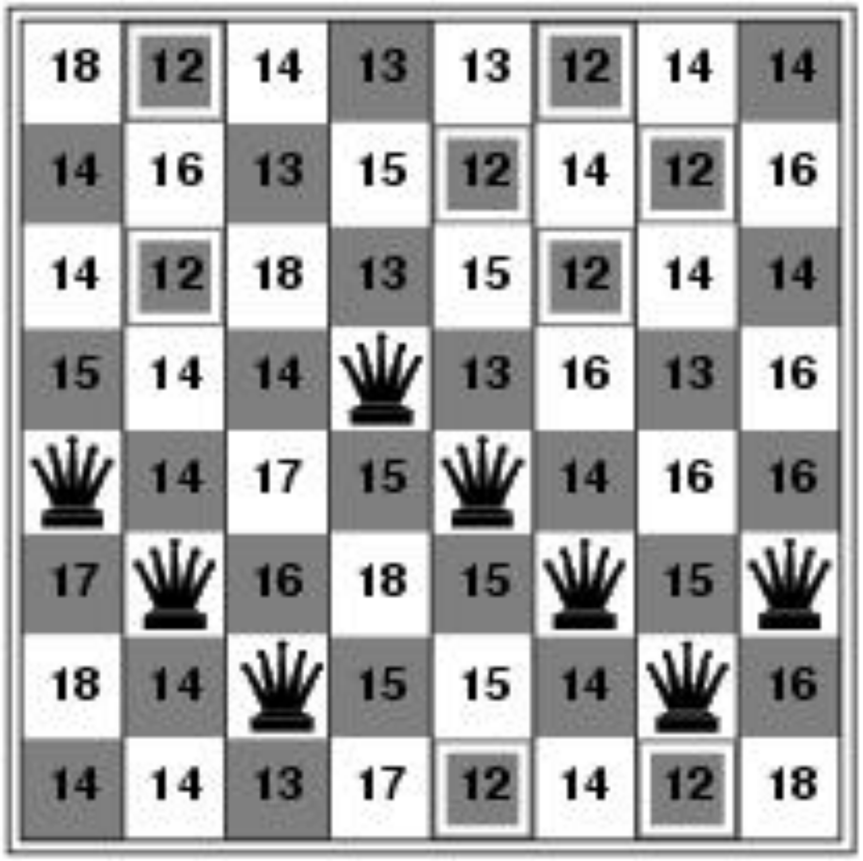
- "a loop that continuously moves towards increasing value"
  - terminates when a peak is reached
  - Aka greedy local search
- Value can be either
  - Objective function value
  - Heuristic function value (minimized)

- Hill climbing does not look ahead of the immediate neighbors
- Can randomly choose among the set of best successors
  - if multiple have the best value

- "climbing Mount Everest in a thick fog with amnesia"

Need Heuristic Function

Convert to Optimization Problem



- *h* = number of ***pairs*** of queens attacking each other
- *h = 17* for the above state

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 8^8 =~17 million states)

Escaping Shoulders: Sideways Move

- If no downhill (uphill) moves, allow sideways moves in hope that algorithm can escape
  - Must limit the number of possible sideways moves to avoid infinite loops
- For 8-queens
  - Allow sideways moves with limit of 100
  - Raises percentage of problems solved from 14 to 94%

  - However….
    - 21 steps for every successful solution
    - 64 for each failure

Escaping Local Optima - Enforced Hill Climbing

- Perform breadth first search from a local optima
  - to find the next state with better h function
- Typically,
  - prolonged periods of exhaustive search
  - bridged by relatively quick periods of hill-climbing
- Middle ground b/w local and systematic search

Hill Climbing: Stochastic Variations

→When the state-space landscape has local minima, any search that moves only in the greedy direction cannot be complete

→Random walk, on the other hand, *is* asymptotically complete

Idea: Combine random walk & greedy hill-climbing

At each step do one of the following:

- Greedy: With prob p move to the neighbor with largest value
- Random: With prob 1-p move to a random neighbor

Hill-climbing with random restarts ☆

- If at first you don't succeed, try, try again!
- Different variations
  - For each restart: run until termination vs. run for a fixed time
  - Run a fixed number of restarts or run indefinitely

Use this algorithm!

- 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?

Restarts	0	2	4	8	16	32	64
Success?	14%	36%	53%	74%	92%	99%	99.994%

  - Expected number of steps taken?

Hill-Climbing with Both Random Walk & Random Sampling

At each step do one of the three

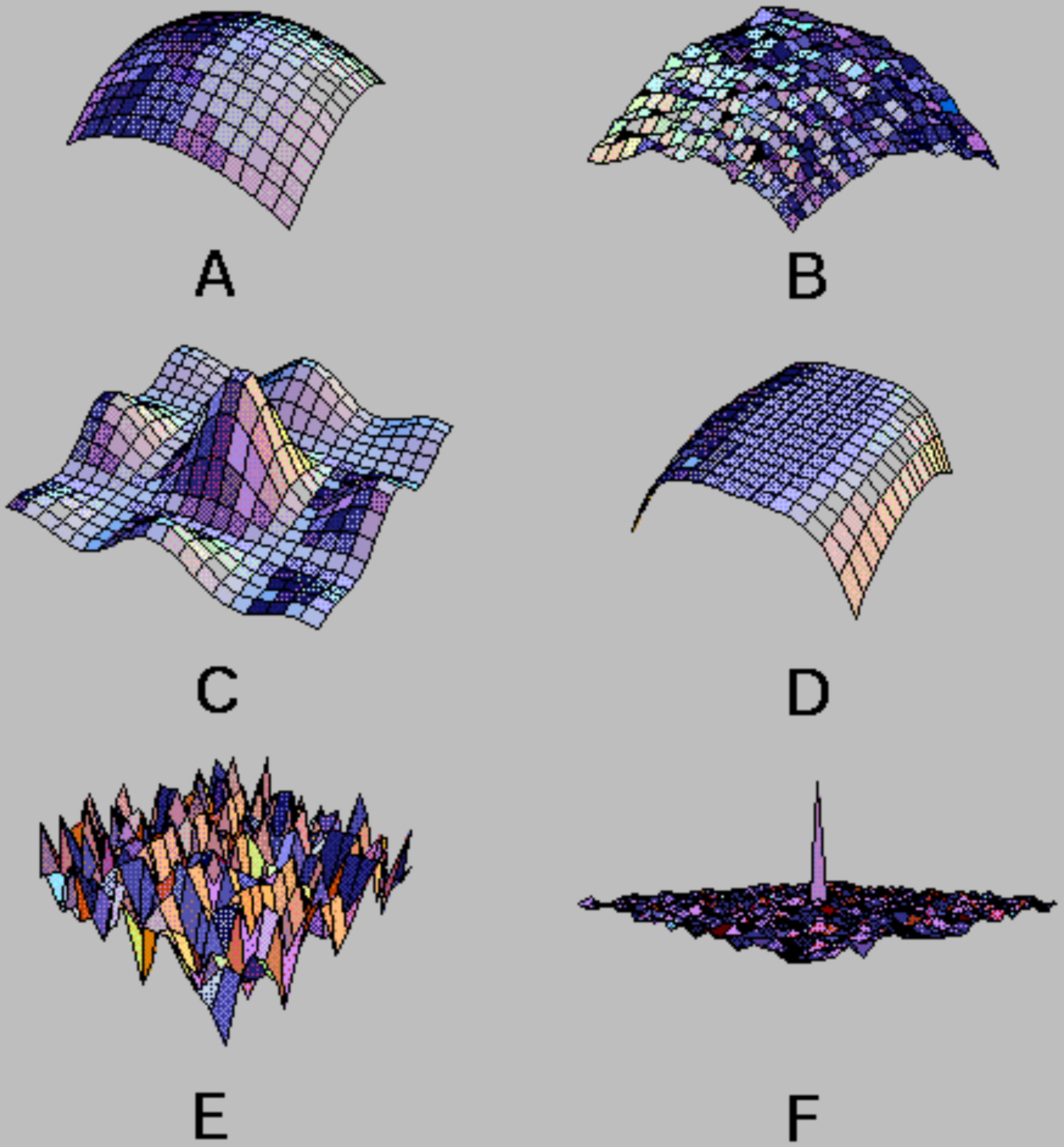
- Greedy: move to the neighbor with largest value
- Random Walk: move to a random neighbor
- Random Restart: Start over from a new, random state

Hill climbing (or gradient ascent/descent)

- This is a very simple algorithm for finding a local maximum.
- Iteratively maximize "value" of current state, by replacing it by successor state that has highest value, as long as possible.
- In other words, we start from an initial position (possibly random), the move along the direction of steepest ascent/gradient (descent/negative of gradient, for minimization), go along the direction for a little while; and repeat the process.

We usually don’t know the underlying search space when we start solving a problem.

How would Hill-Climbing do on the following problems?



What we think hill-climbing looks like

What we learn hill-climbing is usually like

In other words, when we perform hill-climbing, we are short-sighted. We can't really see far. Thus, we make local best decisions with the hope of finding a global best solution. This may be an impossible task.

Statistics from a real problem: 8-queen problem

- Starting with a randomly generated 8-queens state, hill-climbing gets stuck 86% of the time at a local maximum or a ridge or a plateau.
- That means, it can solve only 14% of the 8-queens problems given to it.
- However, hill-climbing is very efficient.
  - It takes only 4 steps on an average when it succeeds.
  - It takes only 3 steps when getting stuck.
- This is very good, in some ways, in a search space that is 8^8 ≈ 17 million states (Each queen can be in 8 places, independent of each other. First queen can be in 8 places, second can be in 8 places, ….).
- What we want: An algorithm that compromises on efficiency, but can solve more problems.

Some ideas about making Hill-climbing better

- Allowing sideways moves
  - When stuck on a ridge or plateau (i.e., all successors have the same value), allow it to move anyway hoping it is a shoulder and after some time, there will be a way up.
- How many sideways moves?
  - If we always allow sideways moves when there are no uphill moves, an infinite loop may occur if it's not a shoulder.
- Solution
  - Put a limit on the number of consecutive sideways moves allowed.
- Experience with a real problem
  - The authors allowed 100 consecutive sideways moves in the 8-queens problem.
  - This raises the percentage of problems solved from 14% to 94%
  - However, now it takes on average 21 steps when successful and 64 steps when it fails.

Variants of Hill-climbing

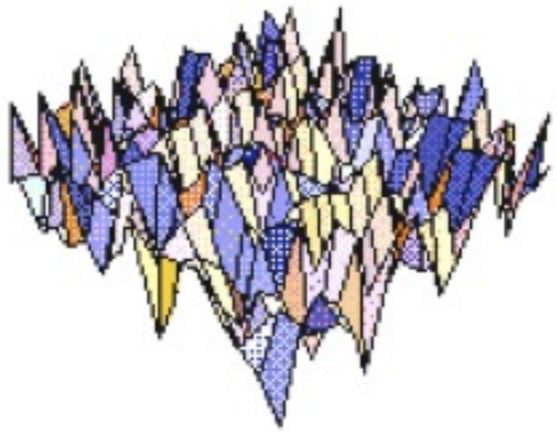
- Stochastic hill-climbing
    - Choose at random from among the uphill moves, assuming several uphill moves are possible. Not the steepest.
    - It usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
- First-choice hill climbing
    - Generates successors randomly until one is generated that is better than current state.
    - This is a good strategy when a state may have hundreds or thousands of successor states.
- Steepest ascent, hill-climbing with limited sideways moves, stochastic hill-climbing, first-choice hill-climbing are all incomplete.
    - Complete: A local search algorithm is complete if it always finds a goal if one exists.
    - Optimal: A local search algorithm is complete if it always finds the global maximum/minimum.

Variants of Hill-climbing

- Random-restart hill-climbing
    - If you don’t succeed the first time, try, try again.
- If the first hill-climbing attempt doesn’t work, try again and again and again!
- That is, generate random initial states and perform hill-climbing again and again.
- The number of attempts needs to be limited, this number depends on the problem.
- With the 8-queens problem, if the number of restarts is limited to about 25, it works very well.
- For a 3-million queen problem with restarts (not sure how many attempts were allowed) with sideways moves, hill-climbing finds a solution very quickly.
- Luby et al. (1993) has showed that in many cases, to restart a randomized search after a particular fixed amount of time is much more efficient than letting each search continue indefinitely.

Final Words on Hill-climbing

- Success of hill-climbing depends on the shape of the state space landscape.
- If there are few local maxima and plateaus, random-start hill-climbing with sideways moves works well.
- However, for many real problems, the state space landscape is much more rugged.
- NP-complete problems are hard because they have exponential number of local maxima to get stuck on.
- In spite of all the problems, random-hill climbing with sideways moves works and other approximation techniques work reasonably well on such problems.



Iterative Best Improvement

- How to determine the neighbor node to be selected?
- Iterative Best Improvement:
  - select the neighbor that optimizes some evaluation function
- Which strategy would make sense? Select neighbour with …

Maximal number of constraint violations

Similar number of constraint violations as current state

No constraint violations

Minimal number of constraint violations

- Evaluation function:
  h(n): number of constraint violations in state n
- Greedy descent: evaluate h(n) for each neighbour, pick the neighbour n with minimal h(n)
- Hill climbing: equivalent algorithm for maximization problems
  - Minimizing h(n) is identical to maximizing –h(n)

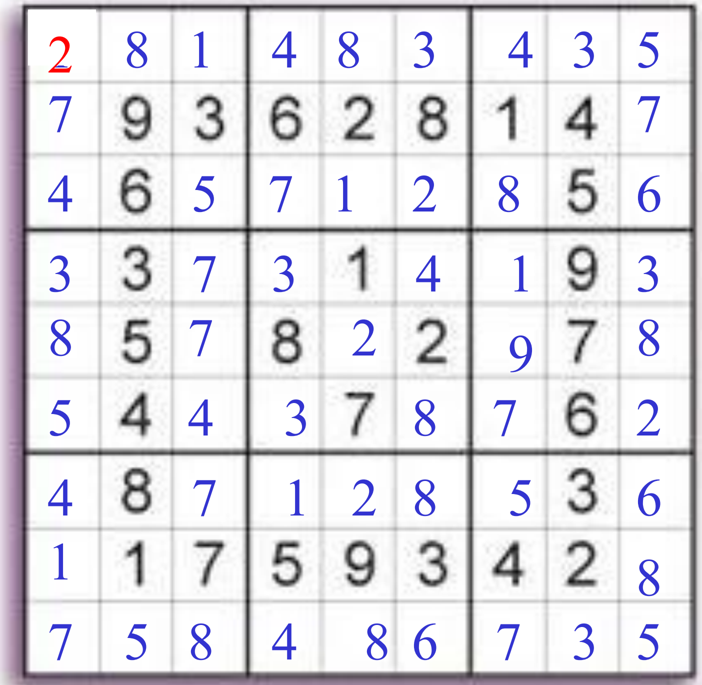
Example: Greedy descent for Sudoku

Assign random numbers between 1 and 9 to blank fields

Repeat

- For each cell & each number: Evaluate how many constraint violations changing the assignment would yield
- Choose the cell and number that leads to the fewest violated constraints; change it

Until solved



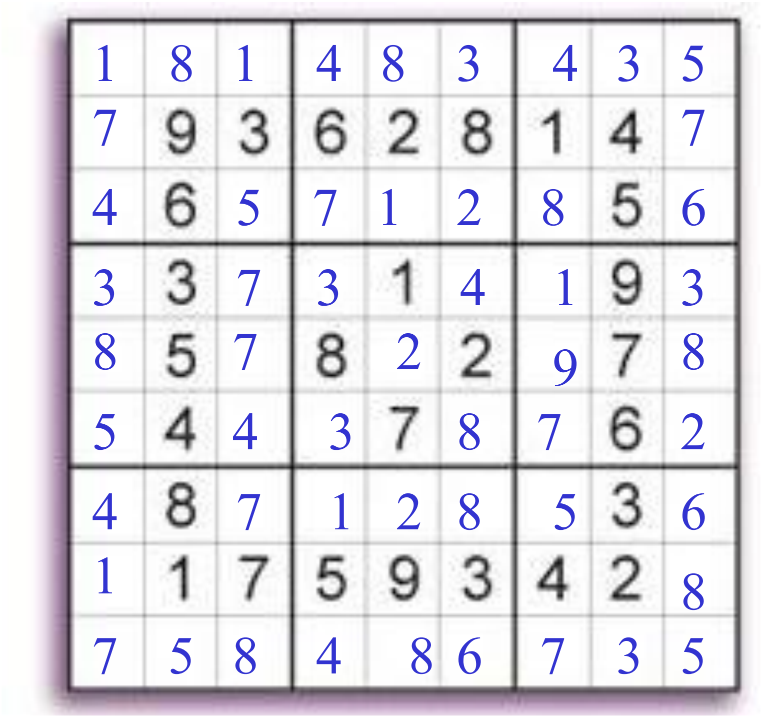
2	8	1	4	8	3	4	3	5
7	9	3	6	2	8	1	4	7
4	6	5	7	1	2	8	5	6
3	3	7	3	1	4	1	9	3
8	5	7	8	2	2	9	7	8
5	4	4	3	7	8	7	6	2
4	8	7	1	2	8	5	3	6
1	1	7	5	9	3	4	2	8
7	5	8	4	8	6	7	3	5

Example: Greedy descent for Sudoku

Example for one local search step:

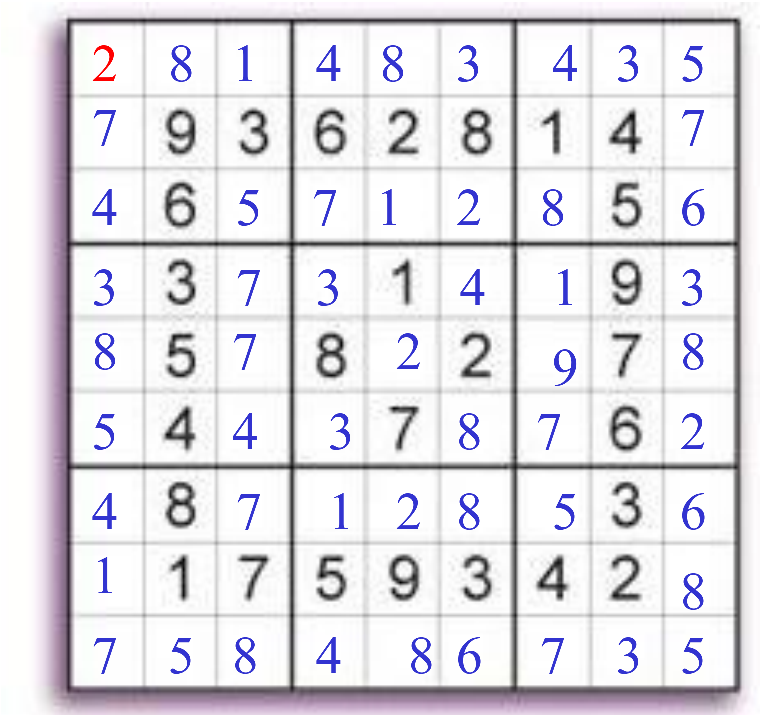
Reduces #constraint violations by 3:

- Two 1s in the first column
- Two 1s in the first row
- Two 1s in the top-left box



1	8	1	4	8	3	4	3	5
7	9	3	6	2	8	1	4	7
4	6	5	7	1	2	8	5	6
3	3	7	3	1	4	1	9	3
8	5	7	8	2	2	9	7	8
5	4	4	3	7	8	7	6	2
4	8	7	1	2	8	5	3	6
1	1	7	5	9	3	4	2	8
7	5	8	4	8	6	7	3	5

$\longrightarrow$



2	8	1	4	8	3	4	3	5
7	9	3	6	2	8	1	4	7
4	6	5	7	1	2	8	5	6
3	3	7	3	1	4	1	9	3
8	5	7	8	2	2	9	7	8
5	4	4	3	7	8	7	6	2
4	8	7	1	2	8	5	3	6
1	1	7	5	9	3	4	2	8
7	5	8	4	8	6	7	3	5

General Local Search Algorithm

```
Procedure Local-Search(V,dom,C)
    Inputs
        V: a set of variables
        dom: a function such that dom(X) is the domain of variable X
        C: set of constraints to be satisfied
    Output   complete assignment that satisfies the constraints
    Local
        A[V] an array of values indexed by V
    repeat
        for each variable X do
            A[X] ← a random value in dom(X);

        while (stopping criterion not met & A is not a satisfying assignment)
            Select a variable Y and a value V ∈ dom(Y)
            Set A[Y] ← V

        if (A is a satisfying assignment) then
            return A

    until termination
```

Random initialization

Local search step

General Local Search Algorithm

```
Procedure Local-Search(V,dom,C)
    Inputs
        V: a set of variables
        dom: a function such that dom(X) is the domain of variable X
        C: set of constraints to be satisfied
    Output    complete assignment that satisfies the constraints
    Local
        A[V] an array of values indexed by V
    repeat
        for each variable X do
            A[X] ← a random value in dom(X);

        while (stopping criterion not met & A is not a satisfying assignment)
            Select a variable Y and a value V ∈ dom(Y)
            Set A[Y] ← V

        if (A is a satisfying assignment) then
            return A

    until termination
```

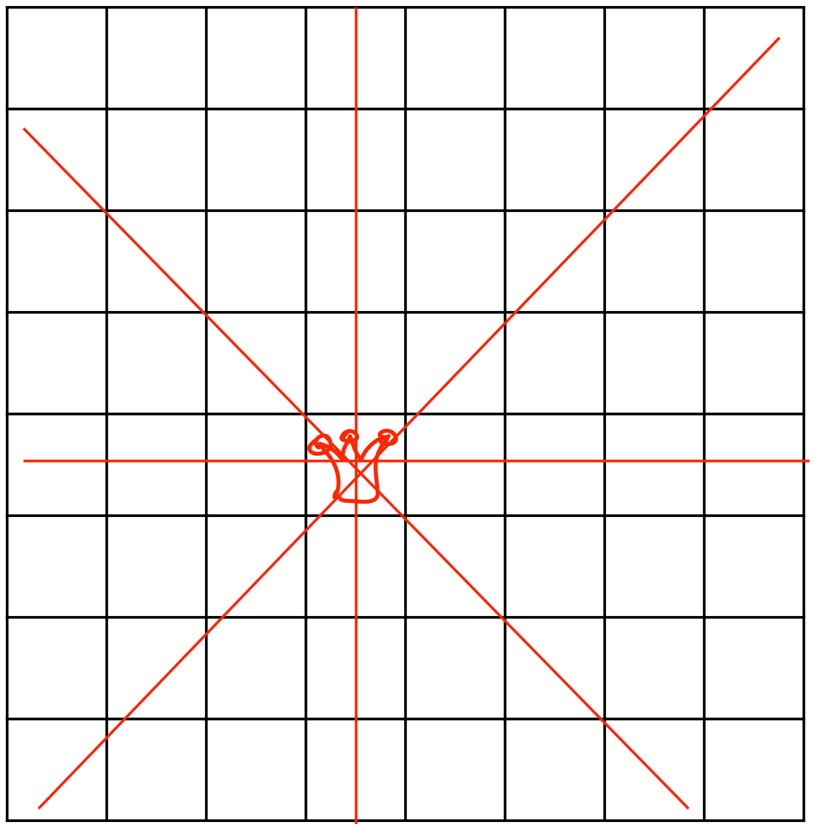
Based on local information.
E.g., for each neighbour evaluate how many constraints are unsatisfied.

Greedy descent: select Y and V to minimize #unsatisfied constraints at each step

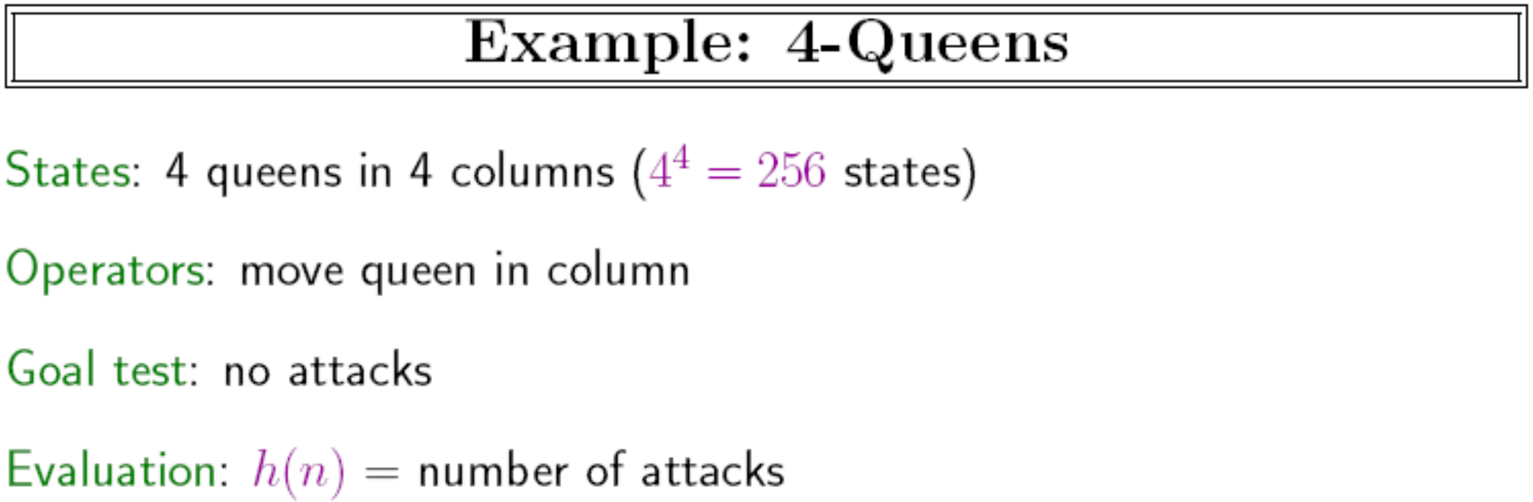
Another example: N-Queens

- Put n queens on an n × n board with no two queens on the same row, column, or diagonal (i.e attacking each other)

- Positions a queen can attack



Example: N-queens



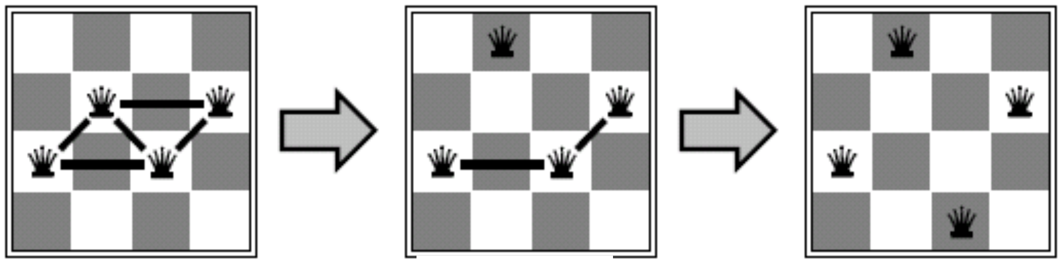
Example: 4-Queens

States: 4 queens in 4 columns ($4^4 = 256$ states)

Operators: move queen in column

Goal test: no attacks

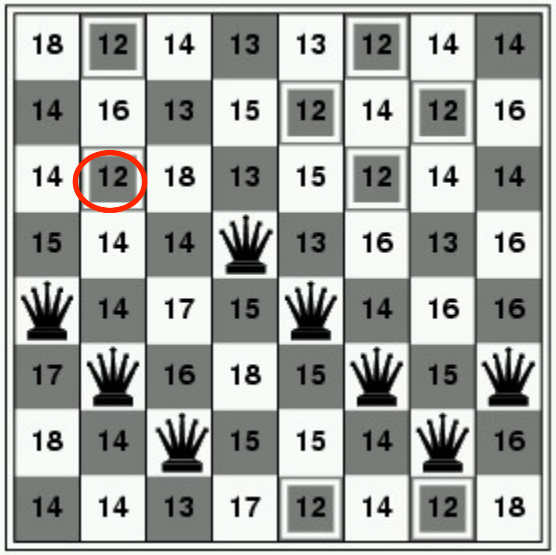
Evaluation: $h(n)$ = number of attacks



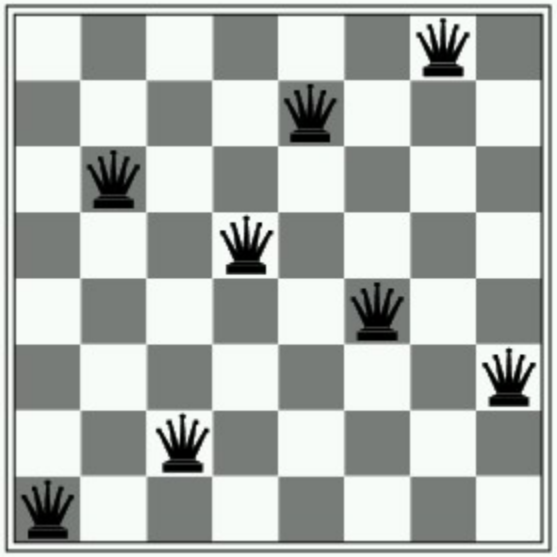
h = 5 h = ? h = ?

1 0 2 3

Example: N-Queens



5 steps



h = 17

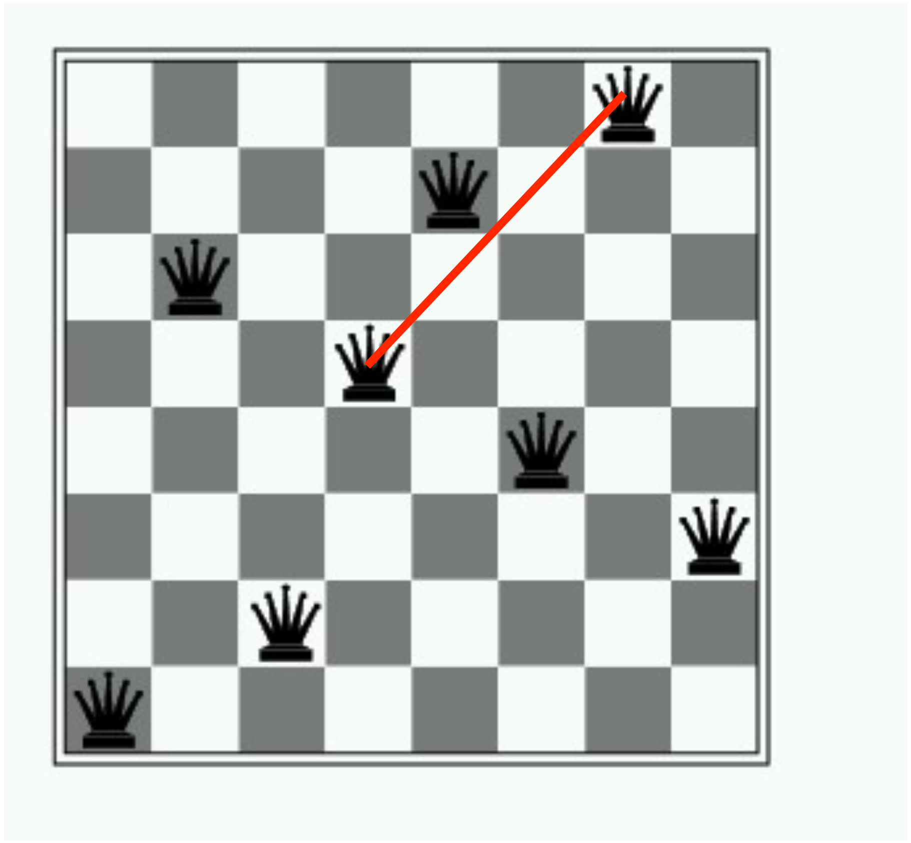
h = 1

Each cell lists h (i.e. #constraints unsatisfied) if you move the queen from that column into the cell

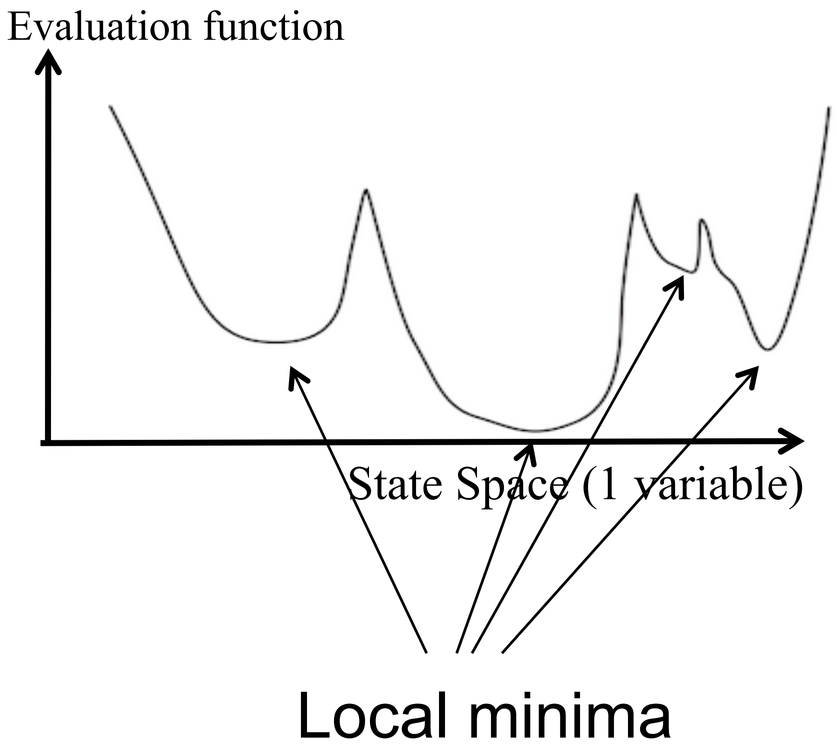
The problem of local minima

- Which move should we pick in this situation?
  - Current cost: h=1
  - No single move can improve on this
  - In fact, every single move only makes things worse (h ≥ 2)

- Locally optimal solution
  - Since we are minimizing: local minimum



Local minima



- Most research in local search concerns effective mechanisms for escaping from local minima
- Want to quickly explore many local minima: global minimum is a local minimum, too

Different neighbourhoods

- Local minima are defined with respect to a neighbourhood.
- Neighbourhood: states resulting from some small incremental change to current variable assignment
- 1-exchange neighbourhood
  - One stage selection: all assignments that differ in exactly one variable.
    How many of those are there for N variables and domain size d?

    $O(Nd)$ $O(d^N)$ $O(N^d)$ $O(N+d)$
  - O(dN). N variables, for each of them need to check d-1 values
  - Two stage selection: first choose a variable (e.g. the one in the most conflicts), then best value
    - Lower computational complexity: O(N+d). But less progress per step
- 2-exchange neighbourhood
  - All variable assignments that differ in exactly two variables. $O(N^2d^2)$
  - More powerful: local optimum for 1-exchange neighbourhood might not be local optimum for 2-exchange neighbourhood

Lecture Overview

- Domain splitting: recap, more details & pseudocode
- Local Search
- Time-permitting: Stochastic Local Search (start)

Stochastic Local Search

- We will use greedy steps to find local minima
  - Move to neighbour with best evaluation function value

- We will use randomness to avoid getting trapped in local minima

General Local Search Algorithm

```
Procedure Local-Search(V,dom,C)
    Inputs
        V: a set of variables
        dom: a function such that dom(X) is the domain of variable X
        C: set of constraints to be satisfied
    Output    complete assignment that satisfies the constraints
    Local
        A[V] an array of values indexed by V
    repeat
        for each variable X do
            A[X] ← a random value in dom(X);

        while (stopping criterion not met & A is not a satisfying assignment)
            Select a variable Y and a value V ∈ dom(Y)
            Set A[Y] ← V

        if (A is a satisfying assignment) then
            return A

    until termination
```

Random restart

Extreme case 1: random sampling.
Restart at every step:
Stopping criterion is "true"

General Local Search Algorithm

```
Procedure Local-Search(V,dom,C)
    Inputs
        V: a set of variables
        dom: a function such that dom(X) is the domain of variable X
        C: set of constraints to be satisfied
    Output    complete assignment that satisfies the constraints
    Local
        A[V] an array of values indexed by V
    repeat
        for each variable X do
            A[X] ←a random value in dom(X);

        while (stopping criterion not met & A is not a satisfying assignment)
            Select a variable Y and a value V ∈dom(Y)
            Set A[Y] ←V

        if (A is a satisfying assignment) then
            return A

    until termination
```

Extreme case 2: greedy descent
Only restart in local minima:
Stopping criterion is “no more improvement in eval. function h”

Greedy Descent + Randomness

- Greedy steps
  - Move to neighbour with best evaluation function value

- Next to greedy steps, we can allow for:

  1. Random restart:
     reassign random values to all variables (i.e. start fresh)

  2. Random steps:
     move to a random neighbour

Stochastic Local Search for CSPs

- Start node: random assignment
- Goal: assignment with zero unsatisfied constraints
- Heuristic function h: number of unsatisfied constraints
  - Lower values of the function are better
- Stochastic local search is a mix of:
  - Greedy descent: move to neighbor with lowest h
  - Random walk: take some random steps
  - Random restart: reassigning values to all variables

Stochastic Local Search for CSPs: details

- Examples of ways to add randomness to local search for a CSP

- In one stage selection of variable and value:
  - instead choose a random variable-value pair

- In two stage selection (first select variable V, then new value for V):
  - Selecting variables:
    - Sometimes choose the variable which participates in the largest number of conflicts
    - Sometimes choose a random variable that participates in some conflict
    - Sometimes choose a random variable
  - Selecting values
    - Sometimes choose the best value for the chosen variable
    - Sometimes choose a random value for the chosen variable

Simulated Annealing

- A hill-climbing algorithm that never makes a “downhill” move toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck in a local maximum.
- In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient.
- Therefore, it is reasonable to try to combine hill-climbing with with a random walk in some way to get both efficiency and completeness.
- **Simulated annealing** is one such algorithm.

Simulated Annealing

- To discuss simulated annealing, we need to switch our point of view from hill climbing to **gradient descent** (i.e., minimizing cost; simply place a negative sign in front of the cost function in hill climbing).

Simulated Annealing

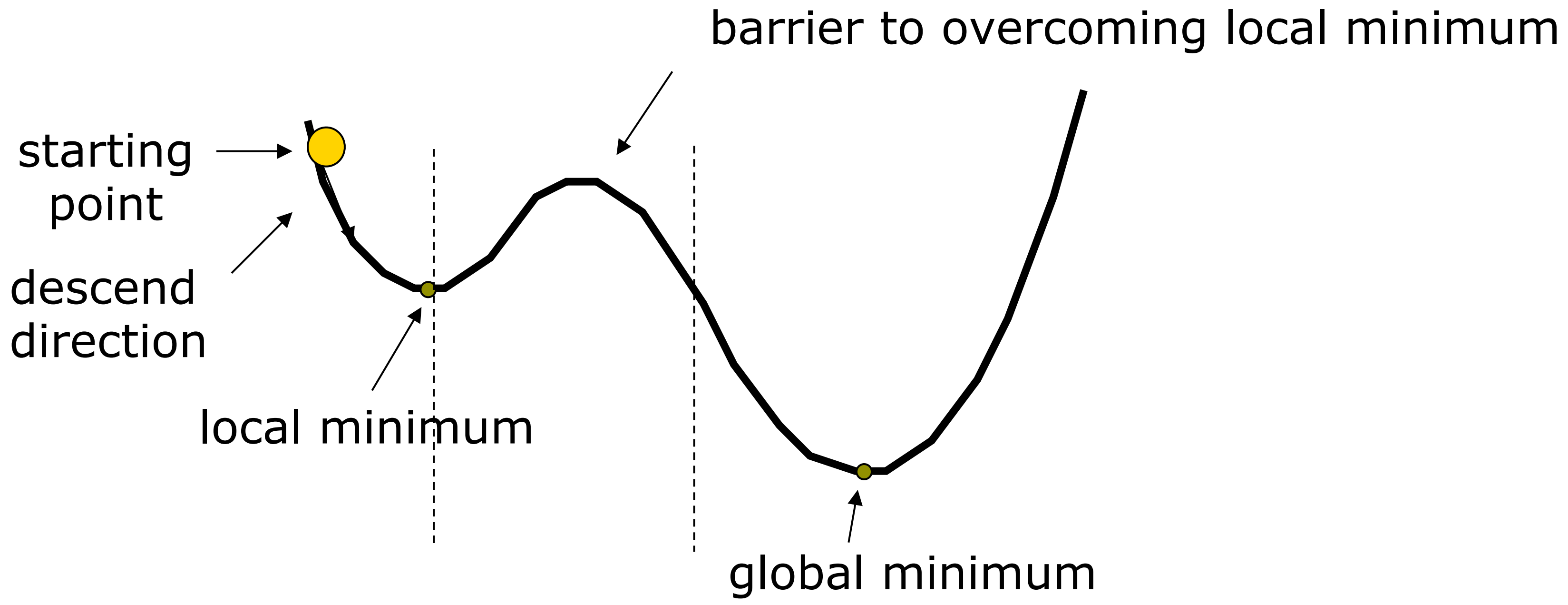
- Analogy: Imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface.
- If we let the ball roll, it will come to rest at a local minimum.
- If we shake the surface, we can bounce the ball out of the local minimum.
- The trick is to shake just hard enough to bounce the ball out of the local minima, but not hard enough to bounce the ball out of the global minimum.
- The simulated annealing algorithm starts by shaking hard (i.e., at a high temperature) and then gradually reduces the intensity of shaking (i.e., lower the temperature)

Minimizing energy

- In this new formulation of the problem
  - We compare our state space to that of a physical system (a bouncing ping-pong ball in a bumpy surface) that is subject to natural laws
  - We compare our value function to the overall potential energy E of the system.
- On every update, we want $\Delta E \leq 0$, i.e., the energy of the system should decrease.
- We start with a high energy state and reduce the energy of the system slowly letting it randomly walk/jump around/search at that energy level or temperature it is at, before reducing the energy level or temperature.

Local Minima Problem

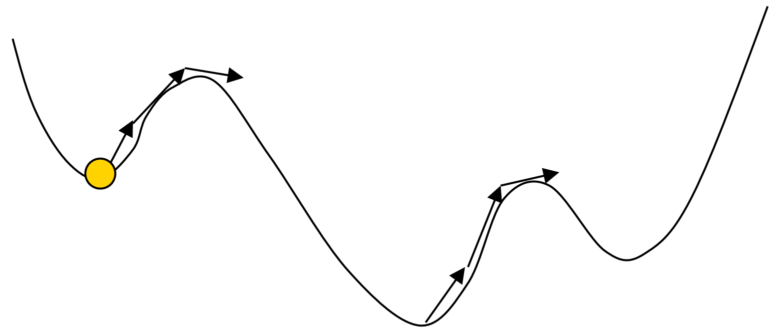
- Question: How do you avoid this local minimum?



Consequences of the Occasional Ascents

desired effect

Help escaping the local optima.



adverse effect

Might pass global optima after reaching it

(easy to avoid by keeping track of best-ever state)

Simulated annealing: basic idea

- From current state, pick a random successor state
- If it has better value than current state, then “accept the transition,” that is, use successor state as current state (standard hill-climbing, in this case hill-descending).
- Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is accept successors that do make the state better with a certain probability).
- In other words, although we want our next step to be a good step, if necessary we allow bad steps that take us in the opposite direction with a non-zero probability.

Simulated annealing algorithm

- Idea: Escape local extrema by allowing “bad moves,” but gradually decrease their size and frequency.
- Even though it says ∞ number of times, it means a large no of times.
- We are working with a situation where a state with a lower energy is a better state.

```
for t=1 to $\infty$ do
   T ← schedule (t)
   if T=0 then return current
   next ← a randomly selected successor of current
   $\Delta E$=next.value – current.value
   if $\Delta E < 0$ then current ← next
   else current ← next only with probability $e^{-\Delta E/T}$
endfor
```

Note on simulated annealing: limit cases

- The **for** loop of the simulated annealing algorithm is similar to hill-climbing (but we are working with a situation where lower energy is better.
- However, instead of picking the best move, it picks a random move (like stochastic hill climbing).
- If the random move improves the situation, it is always accepted.
- Otherwise, the the algorithm accepts the move with some probability less than 1.
- The probability decreases exponentially with the "badness" of the move—the amount ΔE by which the evaluation is worsened. I.e., a less bad move is more likely.
- The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- If the "schedule" lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated Annealing
written to find minimum value solutions

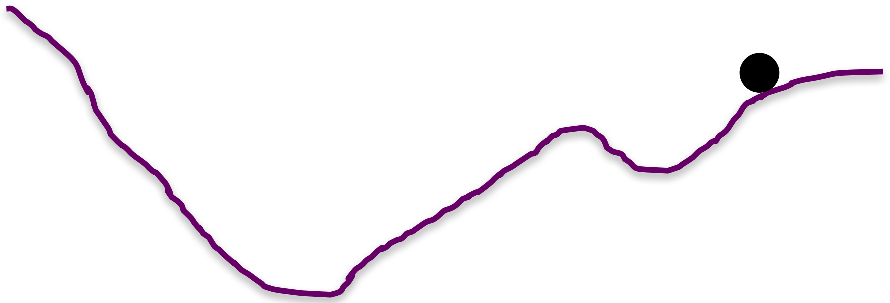
```
function SIMULATED-ANNEALING( problem, schedule) return a solution state
    input: problem, a problem
        schedule, a mapping from time to temperature
    local variables: current, a node.
                     next, a node.
                     T, a "temperature" controlling the prob. of downward 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 $e^{-\Delta E/T}$
```

Physical Interpretation of Simulated Annealing

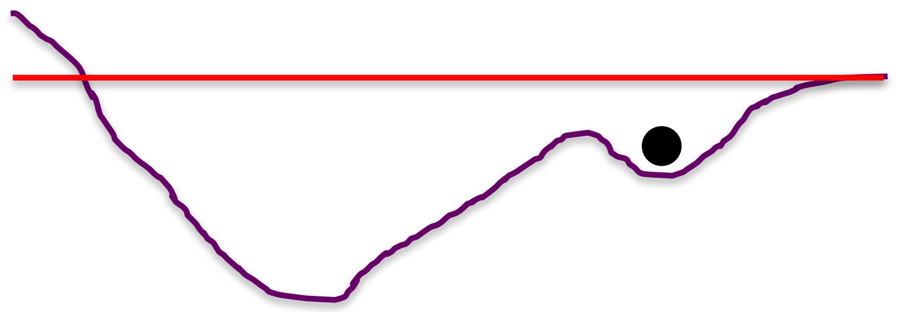
Minimization (not max)

- A Physical Analogy:
  - Imagine letting a ball roll downhill on the function surface
  - Now shake the surface, while the ball rolls,
  - Gradually reducing the amount of shaking



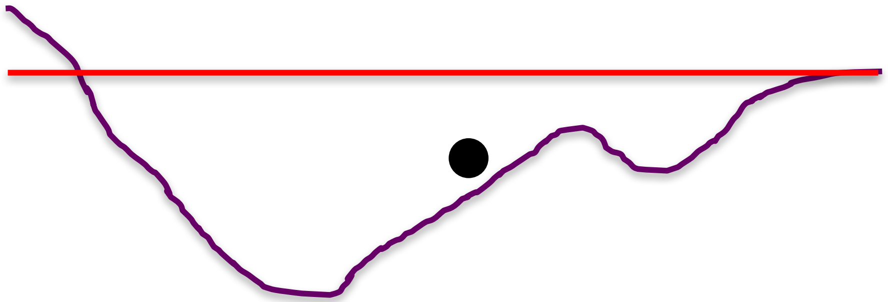
Physical Interpretation of Simulated Annealing

- A Physical Analogy:
  - Imagine letting a ball roll downhill on the function surface
  - Now shake the surface, while the ball rolls,
  - Gradually reducing the amount of shaking



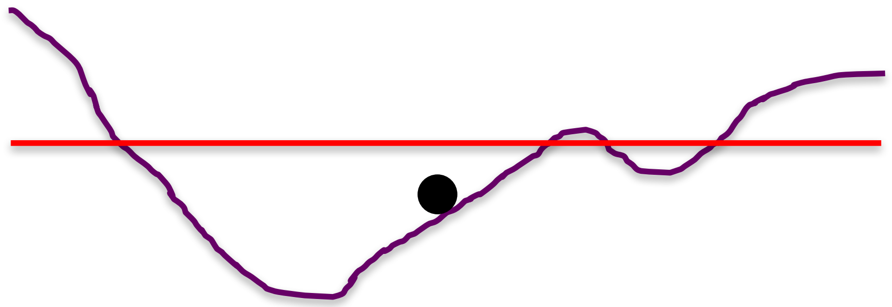
Physical Interpretation of Simulated Annealing

- A Physical Analogy:
  - Imagine letting a ball roll downhill on the function surface
  - Now shake the surface, while the ball rolls,
  - Gradually reducing the amount of shaking



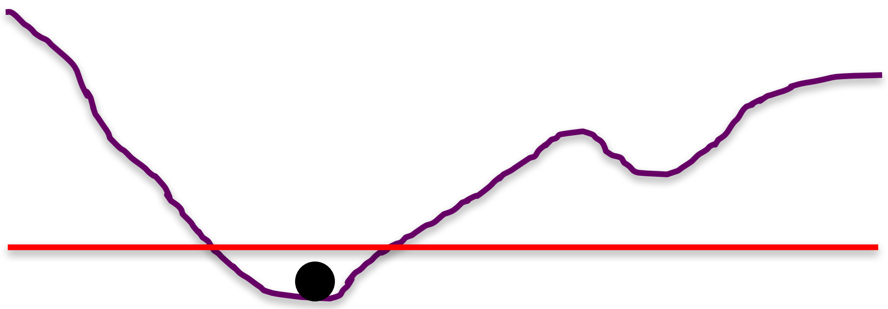
Physical Interpretation of Simulated Annealing

- A Physical Analogy:
    - Imagine letting a ball roll downhill on the function surface
    - Now shake the surface, while the ball rolls,
    - Gradually reducing the amount of shaking



Physical Interpretation of Simulated Annealing

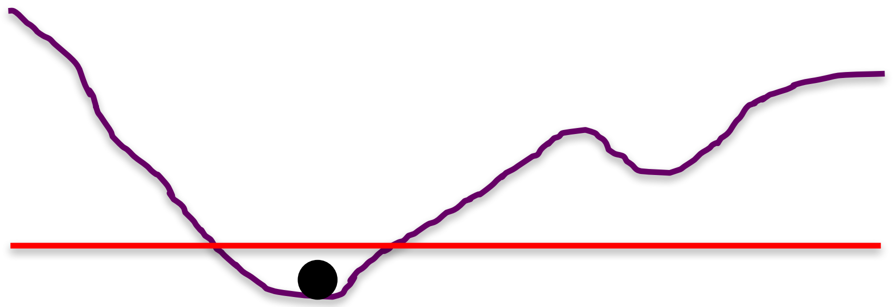
- A Physical Analogy:
    - Imagine letting a ball roll downhill on the function surface
    - Now shake the surface, while the ball rolls,
    - Gradually reducing the amount of shaking



- Annealing = physical process of cooling a liquid → frozen
    - simulated annealing:
        - free variables are like particles
        - seek "low energy" (high quality) configuration
        - slowly reducing temp. T with particles moving around randomly

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"

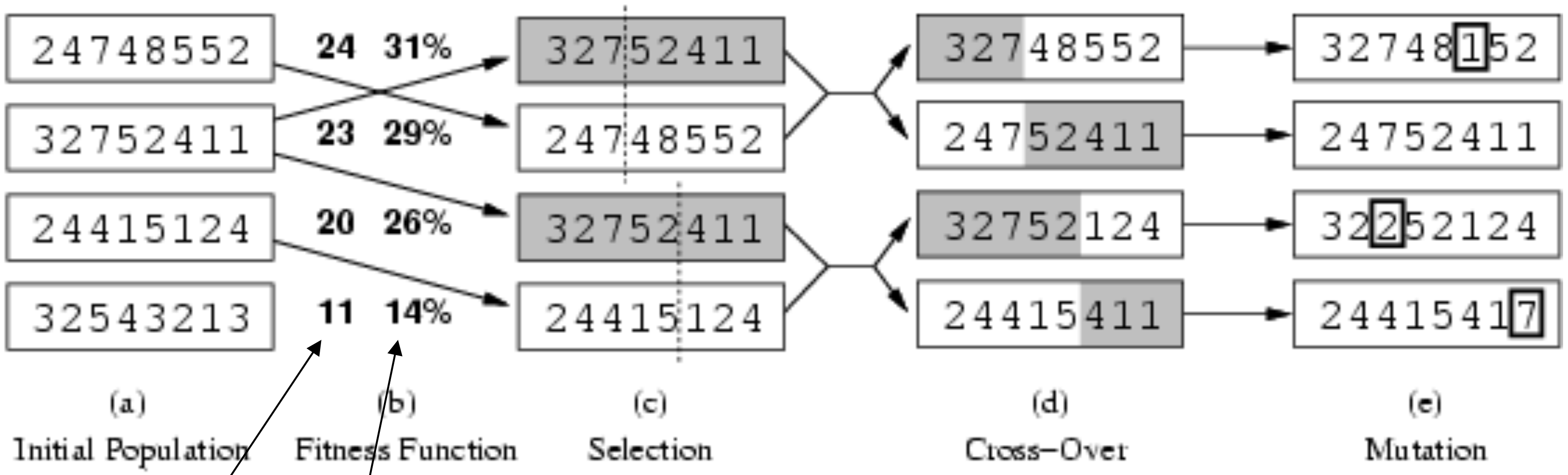


Simulated Annealing in Practice

- 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
- Other applications: Traveling salesman, Graph partitioning, Graph coloring, Scheduling, Facility Layout, Image Processing, …
- useful for some problems, but can be very slow
  - slowness comes about because T must be decreased very gradually to retain optimality

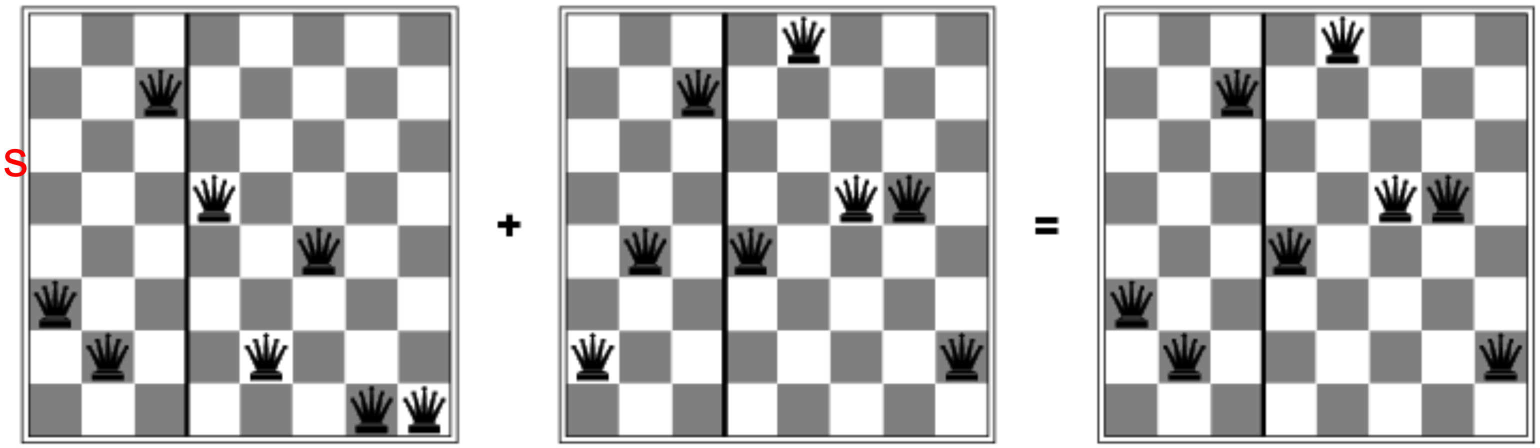
Genetic algorithms

- Local beam search, but…
  - A successor state is generated by ***combining two parent states***
- Start with *k* randomly generated states (population)
- A state is represented as a ***string*** over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher = better
- Produce the next generation of states by selection, crossover, and mutation



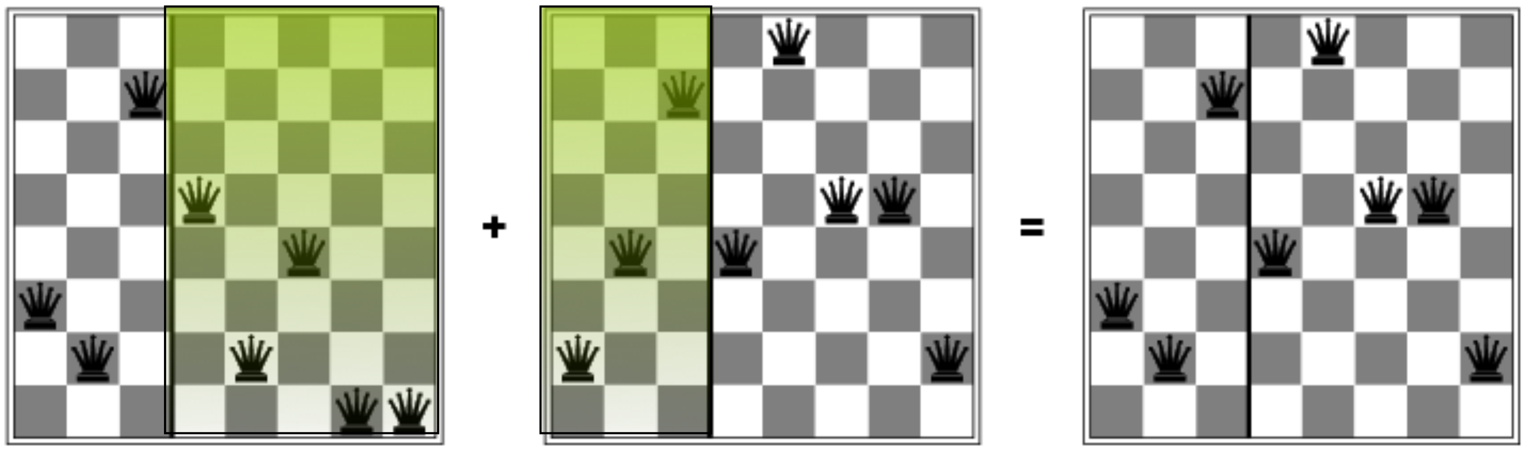
fitness:
#non-attacking queens

probability of being
regenerated
in next generation



- Fitness function: number of non-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
- 24/(24+23+20+11) = 31%
- 23/(24+23+20+11) = 29% etc

Genetic algorithms



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