# Local Search for CSPs

Alan Mackworth

UBC CS 322 – CSP 5

February 4, 2013

Textbook §4.8

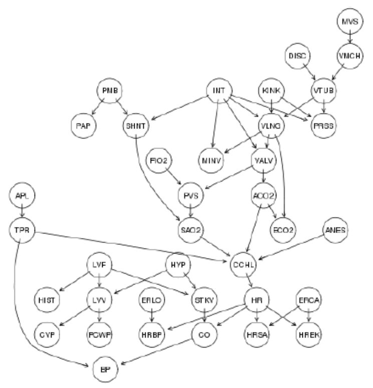
# Local Search: Motivation

- 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

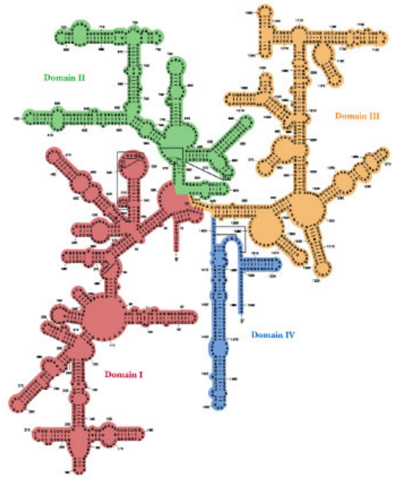
- Alternative: local search
  - Often finds a solution quickly
  - But cannot prove that there is no solution

- 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

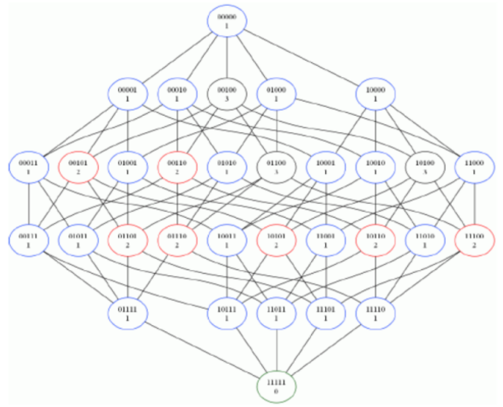
## Some Successful Application Areas for Local Search



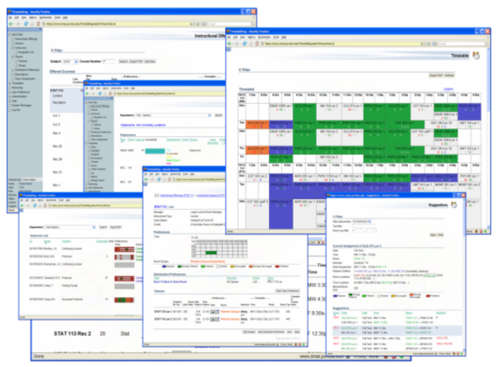
Probabilistic Reasoning



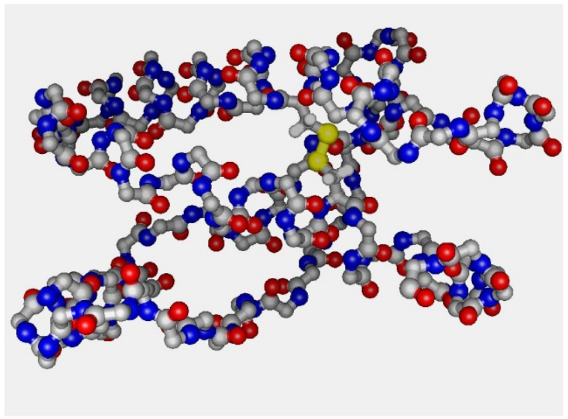
RNA structure design



Propositional satisfiability (SAT)



University Timetabling

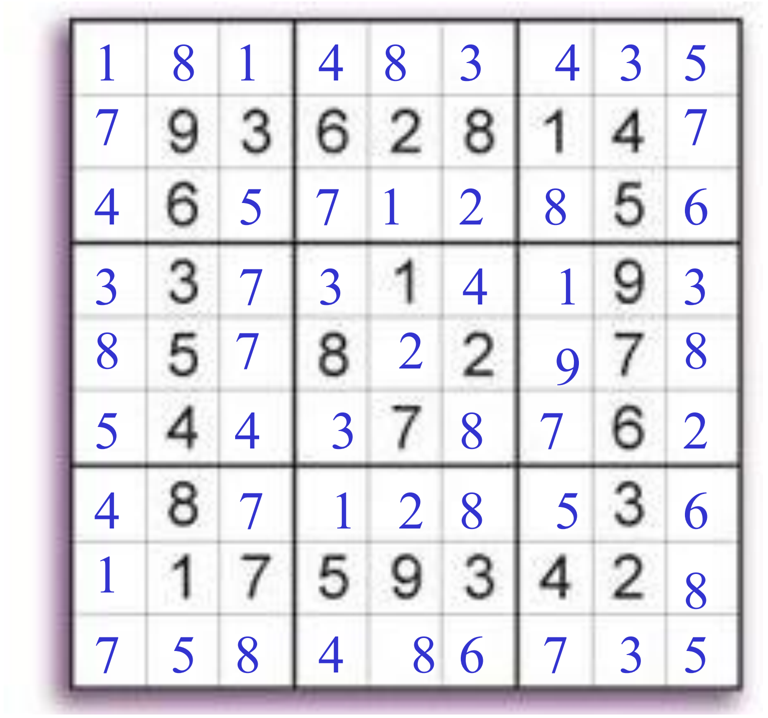


Protein Folding

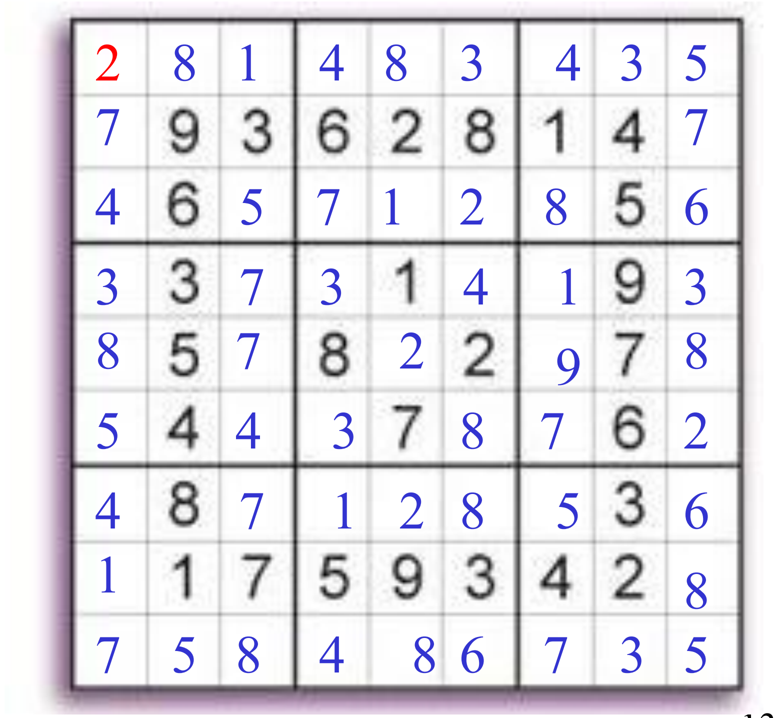
Scheduling of Hubble Space Telescope:
1 week → 10 seconds

# 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 |



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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.

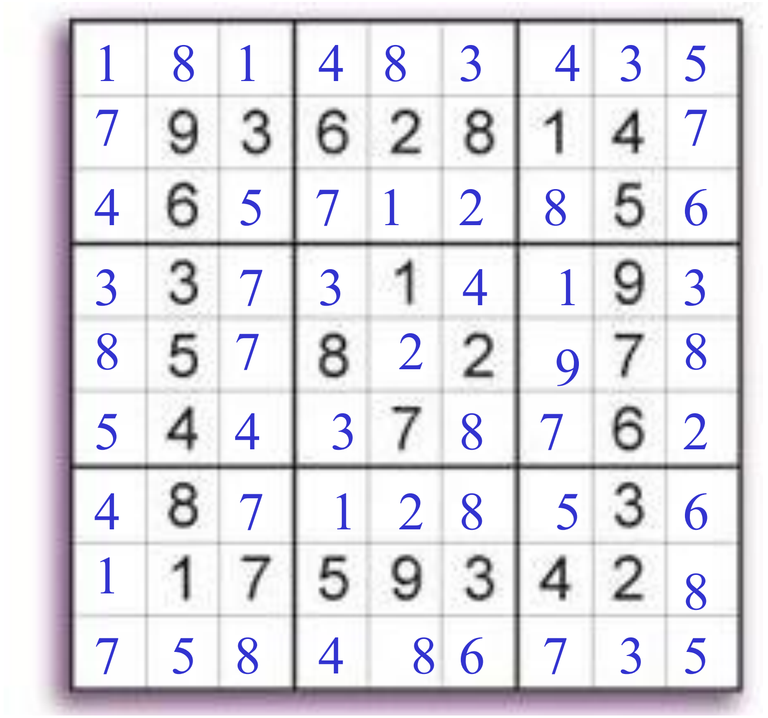
# Example: Sudoku as a local search problem

CSP: usual Sudoku CSP

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

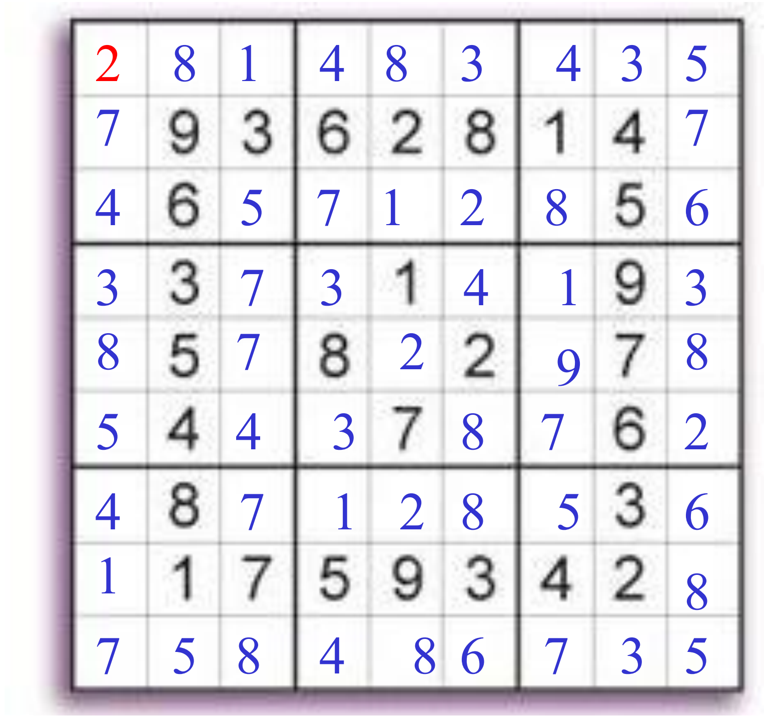
Neighbour relation: value of a single cell differs

Scoring function: number of constraint violations



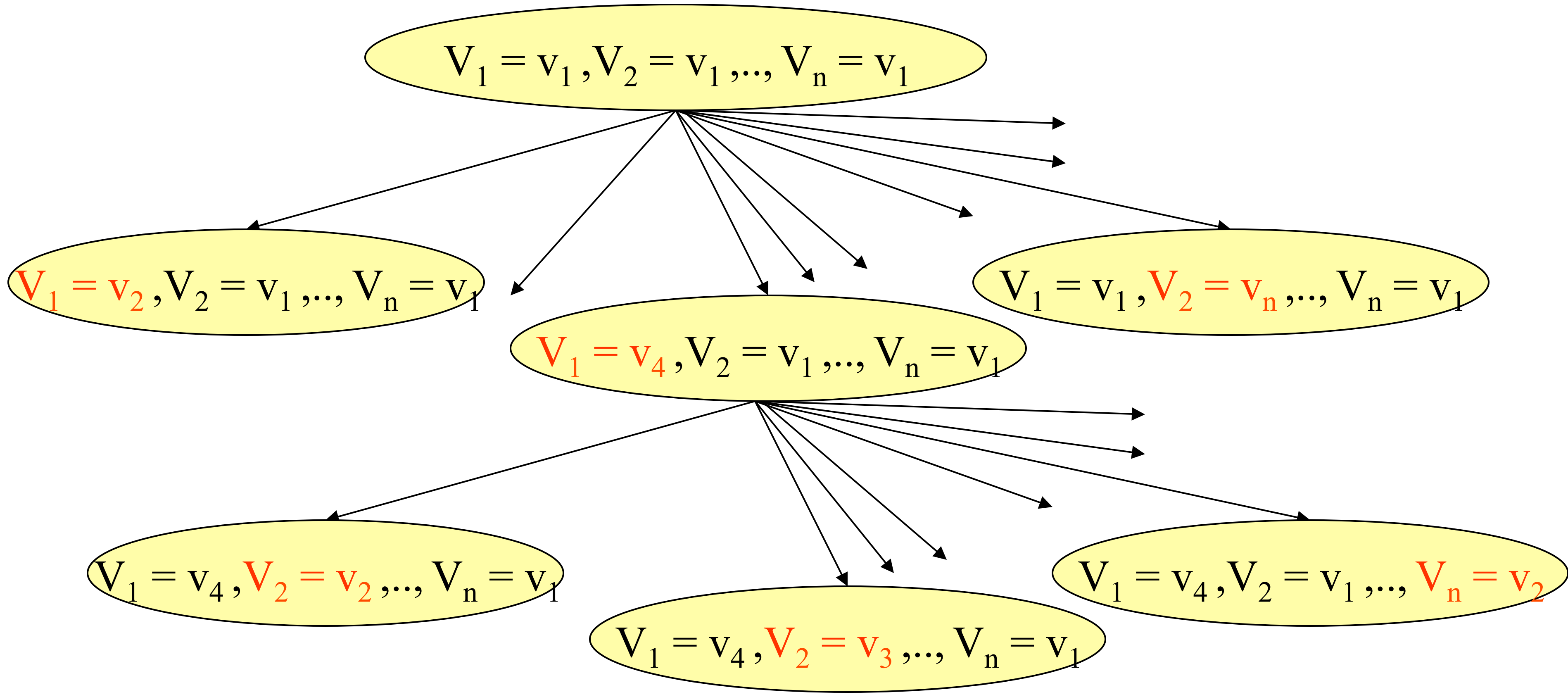
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 8 | 1 | 4 | 8 | 3 | 4 | 3 | 5 |
| 7 | 9 | 3 | 6 | 2 | 8 | 1 | 4 | 7 |
| 4 | 6 | 5 | 7 | 1 | 2 | 8 | 5 | 6 |
| 3 | 3 | 7 | 3 | 1 | 4 | 1 | 9 | 3 |
| 8 | 5 | 7 | 8 | 2 | 2 | 9 | 7 | 8 |
| 5 | 4 | 4 | 3 | 7 | 8 | 7 | 6 | 2 |
| 4 | 8 | 7 | 1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

$\rightarrow$



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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


$V_1 = v_1, V_2 = v_1, ..., V_n = v_1$
$V_1 = v_2, V_2 = v_1, ..., V_n = v_1$
$V_1 = v_4, V_2 = v_1, ..., V_n = v_1$
$V_1 = v_1, V_2 = v_n, ..., V_n = v_1$
$V_1 = v_4, V_2 = v_2, ..., V_n = v_1$
$V_1 = v_4, V_2 = v_3, ..., V_n = v_1$
$V_1 = v_4, V_2 = v_1, ..., V_n = v_2$

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!

# 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)

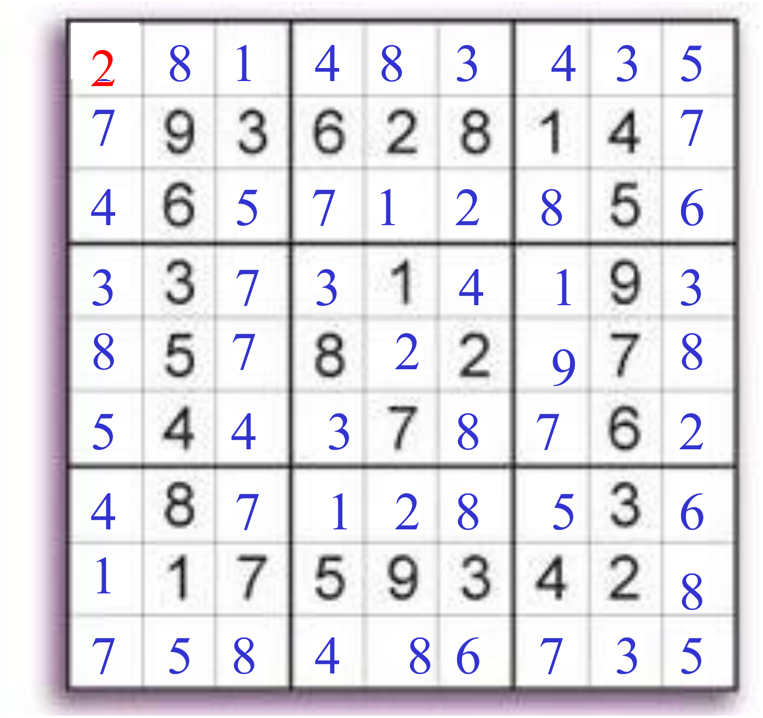
# Example: Greedy descent for Sudoku

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



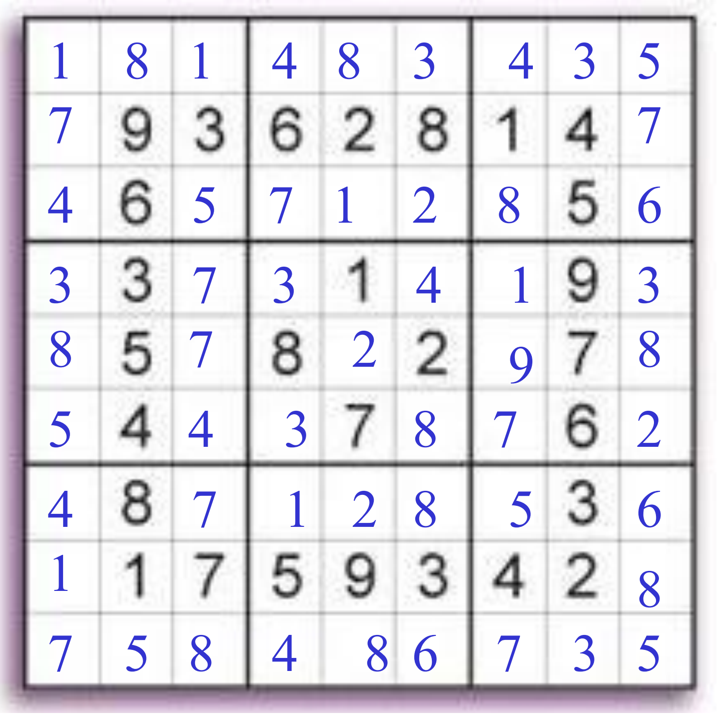
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 8 | 1 | 4 | 8 | 3 | 4 | 3 | 5 |
| 7 | 9 | 3 | 6 | 2 | 8 | 1 | 4 | 7 |
| 4 | 6 | 5 | 7 | 1 | 2 | 8 | 5 | 6 |
| 3 | 3 | 7 | 3 | 1 | 4 | 1 | 9 | 3 |
| 8 | 5 | 7 | 8 | 2 | 2 | 9 | 7 | 8 |
| 5 | 4 | 4 | 3 | 7 | 8 | 7 | 6 | 2 |
| 4 | 8 | 7 | 1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

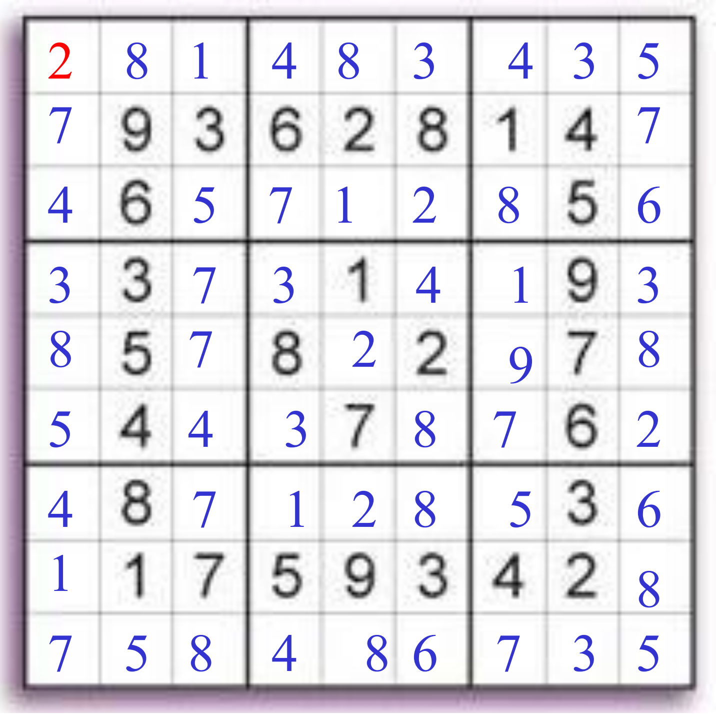
Example for one local search step:

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 |



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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
```

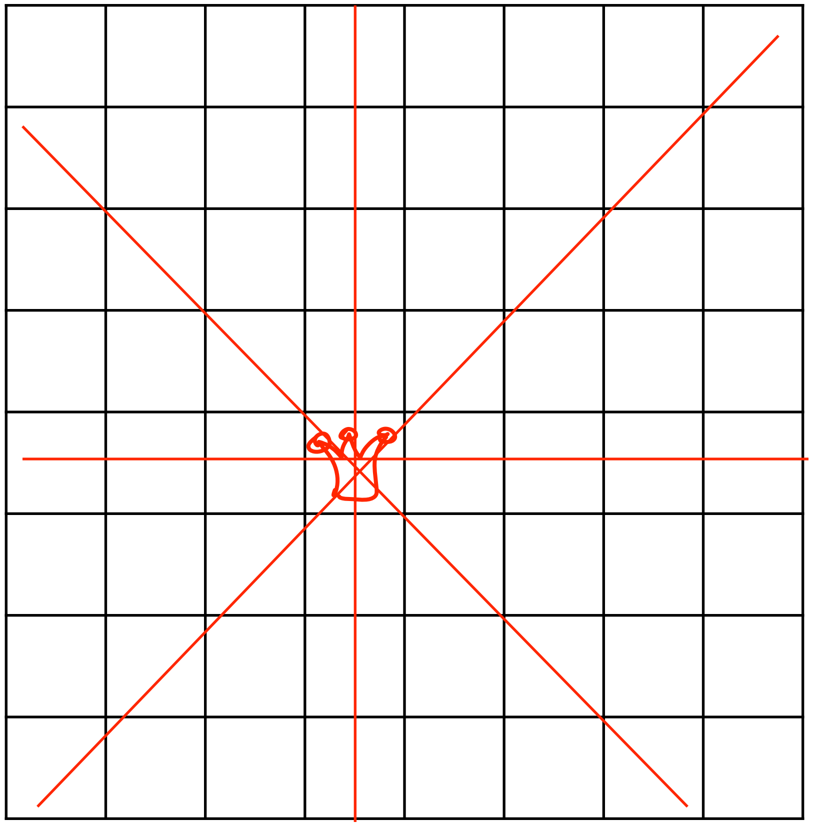
Based on local information.
E.g., for each neighbour evaluate how many constraints are unsatisfied.

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

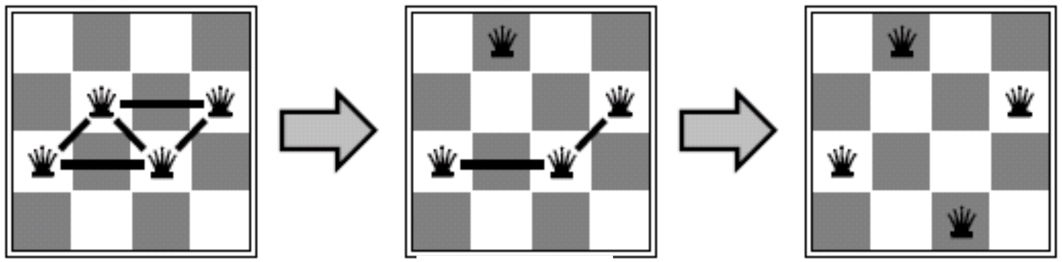
## Example: 4-Queens

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

Operators: move queen in column

Goal test: no attacks

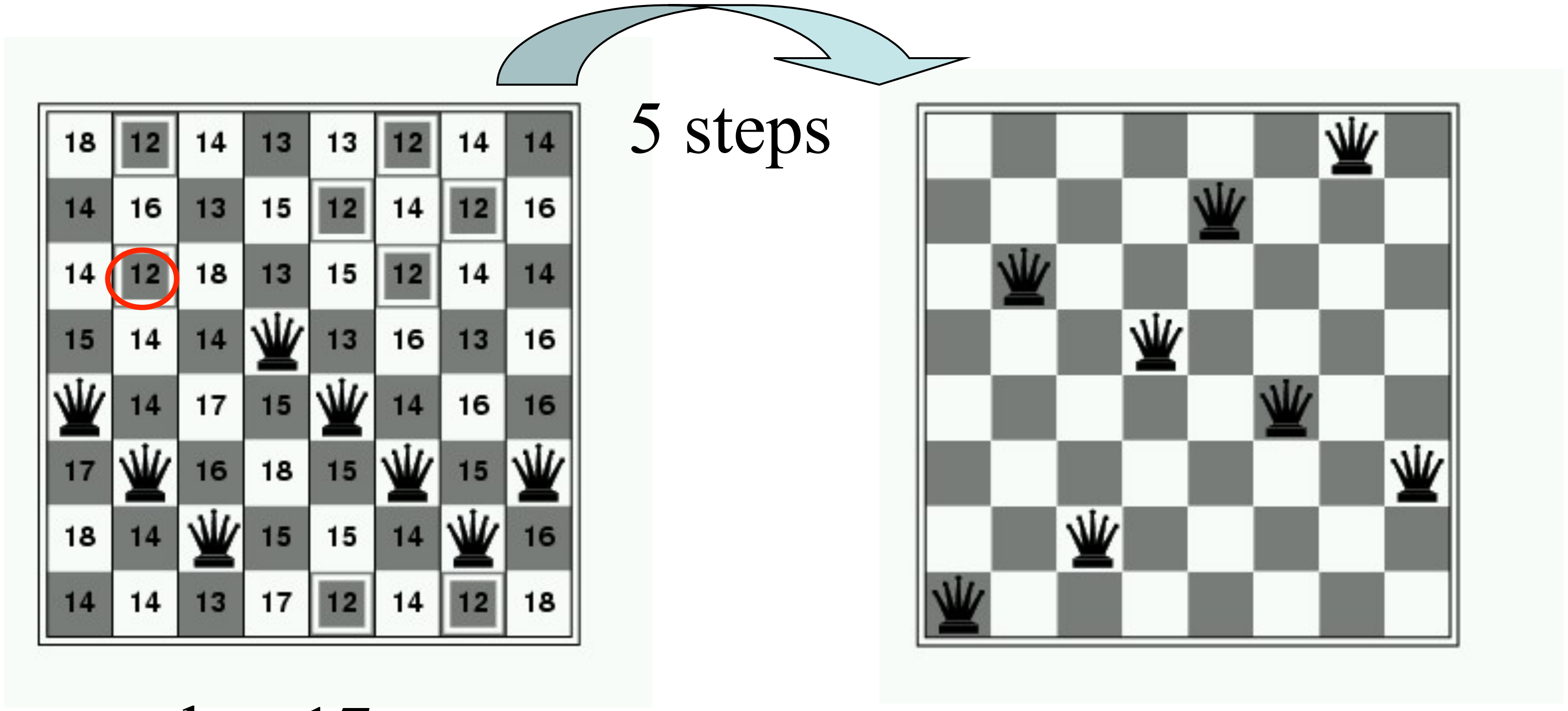
Evaluation: $h(n)$ = number of attacks



h = 5 h = ? h = ?

1 0 2 3

# Example: N-Queens



h = 17

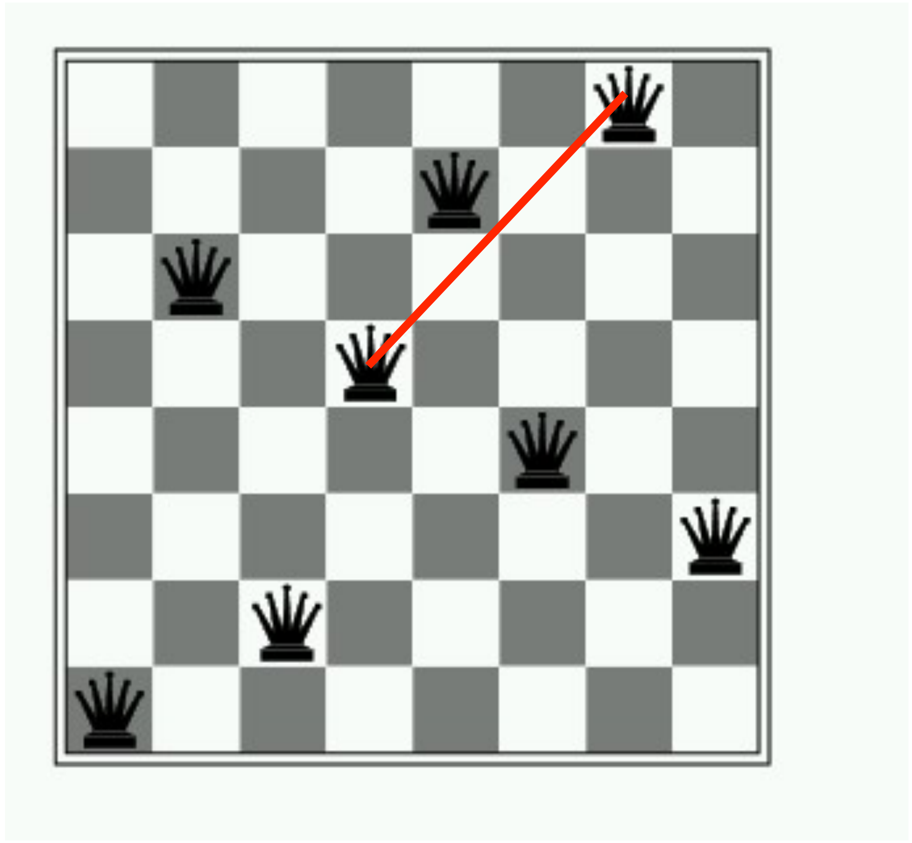
h = 1

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

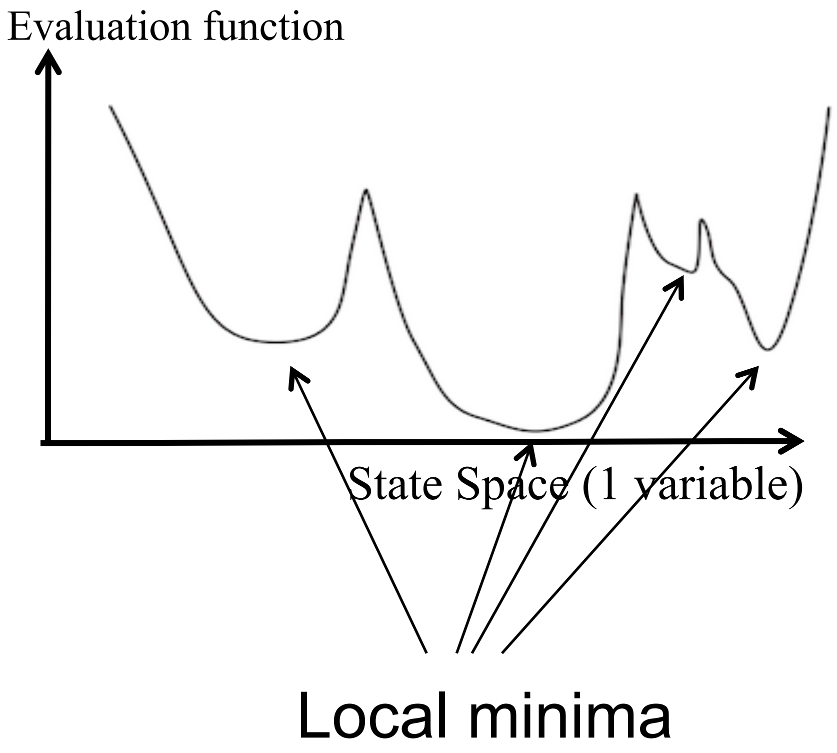
# The problem of local minima

- 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

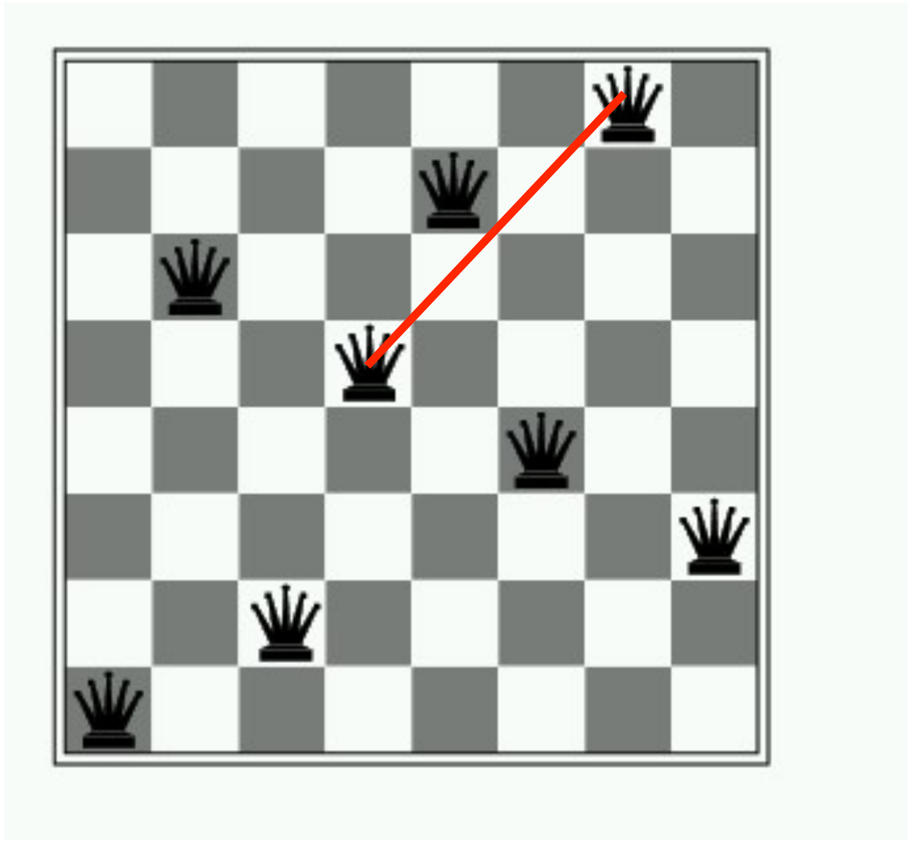
# Different neighbourhoods

- 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

# Different neighbourhoods

- How about an 8-exchange neighbourhood?
  - All minima with respect to the 8-exchange neighbourhood are global minima
    - Why?
  - How expensive is the 8-exchange neighbourhood?
    - $O(N^8 d^8)$
- In general, N-exchange neighbourhood includes all solutions
  - Where N is the number of variables
  - But is exponentially large



# 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

```
1: Procedure Local-Search(V,dom,C)
2:      Inputs
3:           V: a set of variables
4:           dom: a function such that dom(X) is the domain of variable X
5:           C: set of constraints to be satisfied
6:      Output    complete assignment that satisfies the constraints
7:      Local
8:           A[V] an array of values indexed by V
9:      repeat
10:          for each variable X do
11:               A[X] ←a random value in dom(X);
12:
13:          while (stopping criterion not met & A is not a satisfying assignment)
14:               Select a variable Y and a value V ∈dom(Y)
15:               Set A[Y] ←V
16:
17:          if (A is a satisfying assignment) then
18:               return A
19:
20:     until termination
```

Extreme case 2: greedy descent
Only restart in local minima:
Stopping criterion is "no more improvement in eval. function h"
Select variable/value greedily.

## Greedy descent vs. Random sampling

- Greedy descent is
  - good for finding local minima
  - bad for exploring new parts of the search space
- Random sampling is
  - good for exploring new parts of the search space
  - bad for finding local minima
- A mix of the two can work very well

# Greedy Descent + Randomness

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

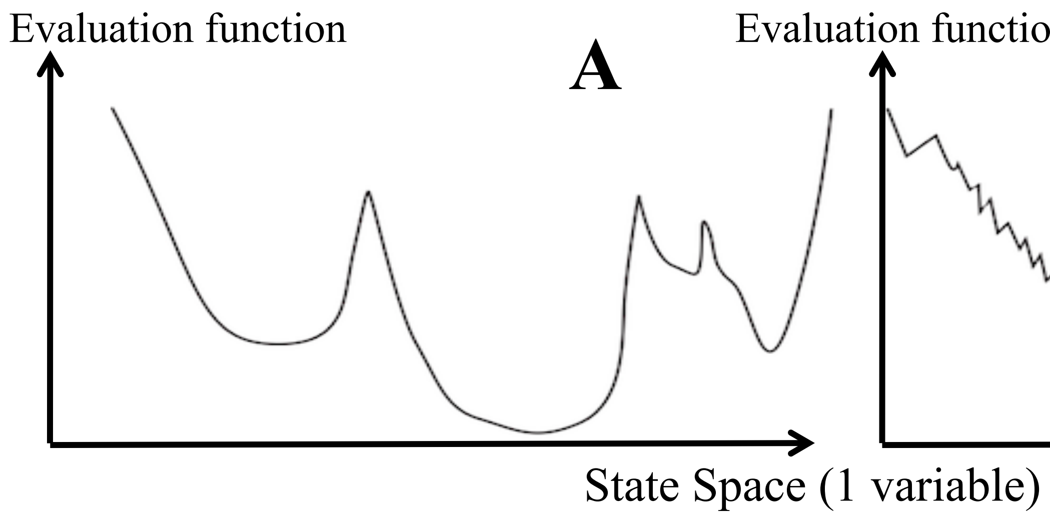
- Next to greedy steps, we can allow for:

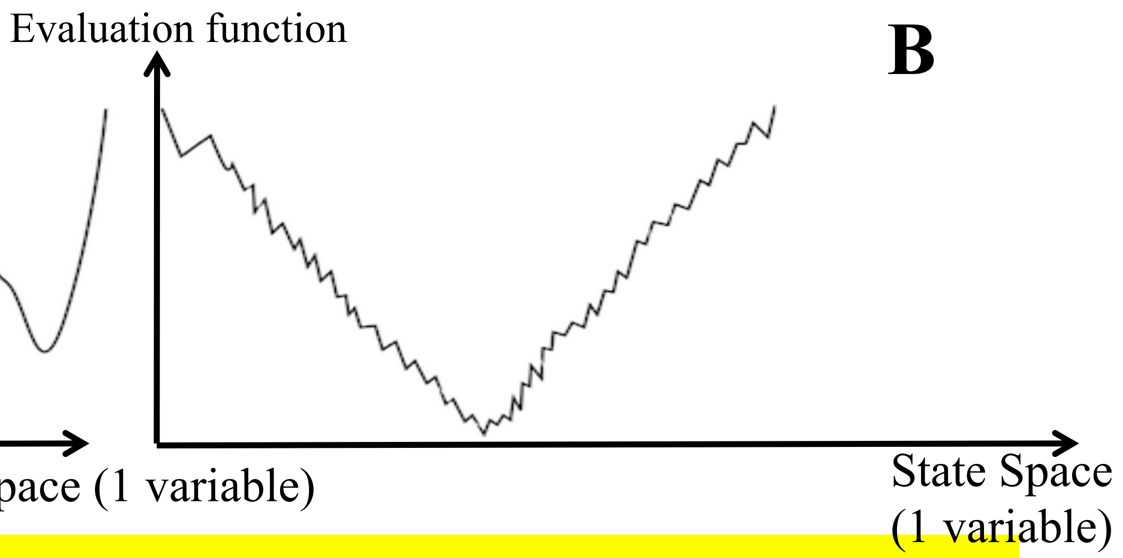
  1. Random restart:
     reassign random values to all variables (i.e. start fresh)

  2. Random steps:
     move to a random neighbour

- Only doing random steps (no greedy steps at all) is called "random walk"

## Which randomized method would work best in each of the these two search spaces?



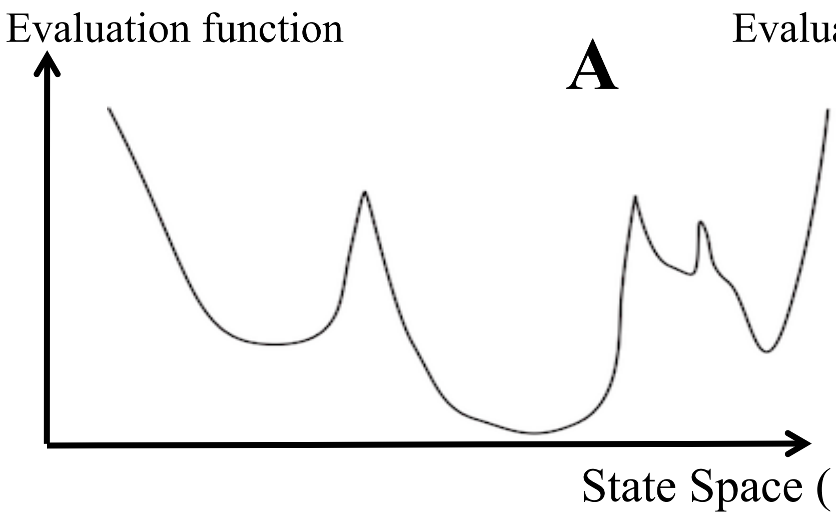


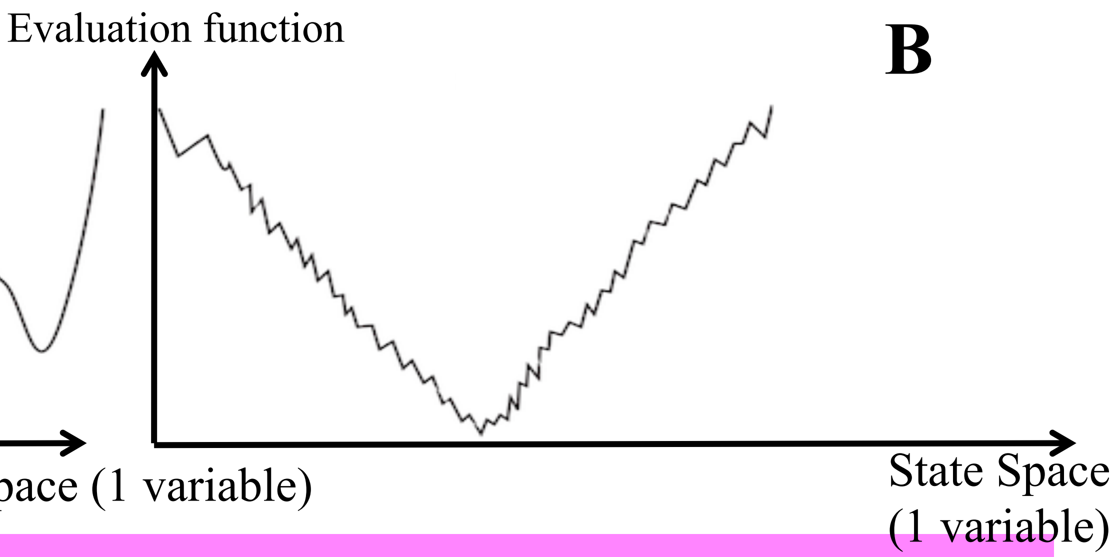
Greedy descent with random steps best on A
Greedy descent with random restart best on B

Greedy descent with random steps best on B
Greedy descent with random restart best on A

equivalent

## Which randomized method would work best in each of the these two search spaces?





Greedy descent with random steps best on B

Greedy descent with random restart best on A

- But these examples are simplified extreme cases for illustration
  - in practice, you don't know what your search space looks like
- Usually integrating both kinds of randomization works best

## 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

- More 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: the one yielding minimal h(n)
    - Sometimes choose a random value for the chosen variable

## Greedy Descent with Min-Conflict Heuristic

- One of the best SLS techniques for CSP solving:
  - At random, select one of the variables v that participates in a violated constraint
  - Set v to one of the values that minimizes the number of unsatisfied constraints

- Can be implemented efficiently:
  - Data structure 1 stores currently violated constraints
  - Data structure 2 stores variables that are involved in violated constraints

  - Each step only yields incremental changes to these data structures

- Most SLS algorithms can be implemented similarly efficiently → very small complexity per search step

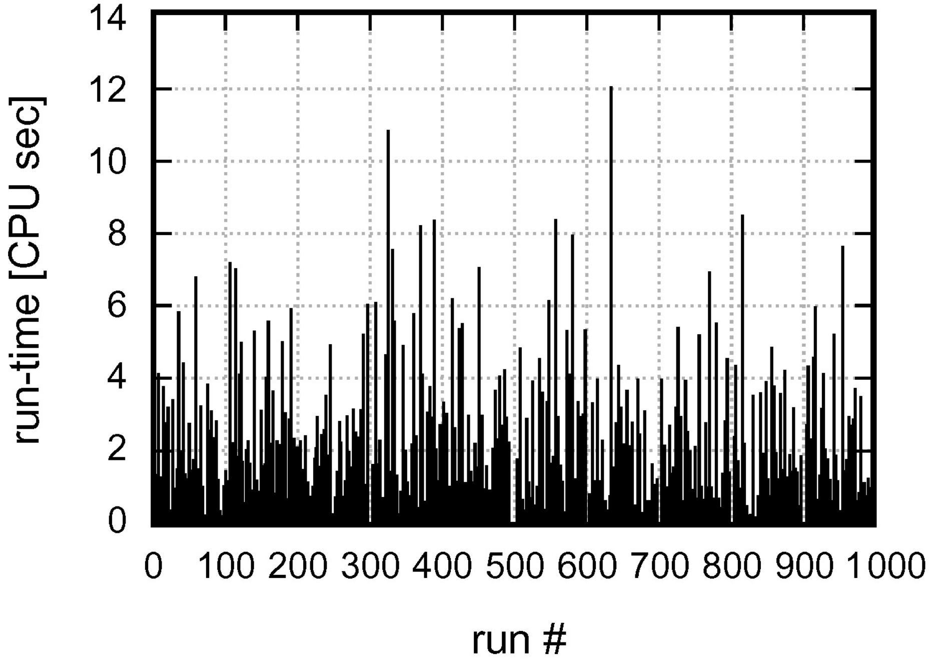
# Evaluating SLS algorithms

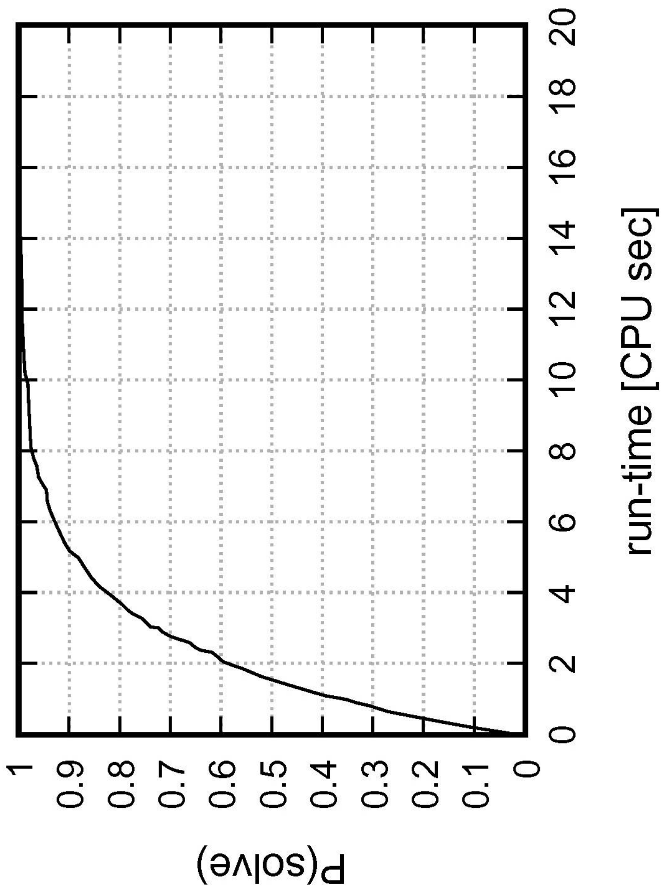
- SLS algorithms are randomized
  - The time taken until they solve a problem is a random variable
  - It is entirely normal to have runtime variations of 2 orders of magnitude in repeated runs!
    - E.g. 0.1 seconds in one run, 10 seconds in the next one
    - On the same problem instance (only difference: random seed)
    - Sometimes SLS algorithm doesn't even terminate at all: stagnation

- If an SLS algorithm sometimes stagnates, what is its mean runtime (across many runs)?
  - Infinity!
  - In practice, one often counts timeouts as some fixed large value X
    - But results depend on which X is chosen

# Comparing SLS algorithms

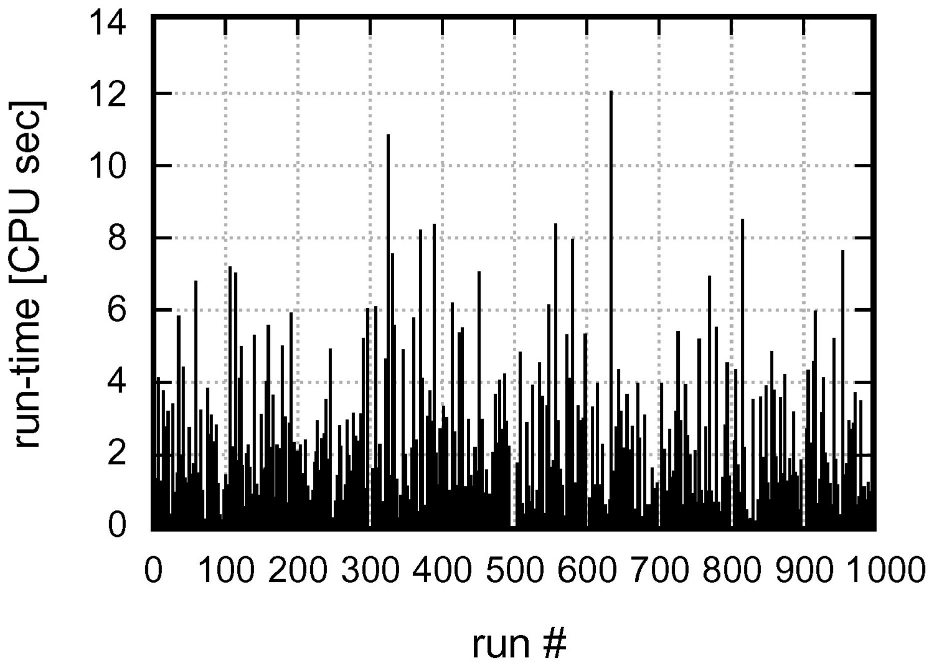
- A better way to evaluate empirical performance
  - Runtime distributions
    - Perform many runs (e.g. below: 1000 runs)
    - Consider the empirical distribution of the runtimes
      - Sort the empirical runtimes (decreasing)

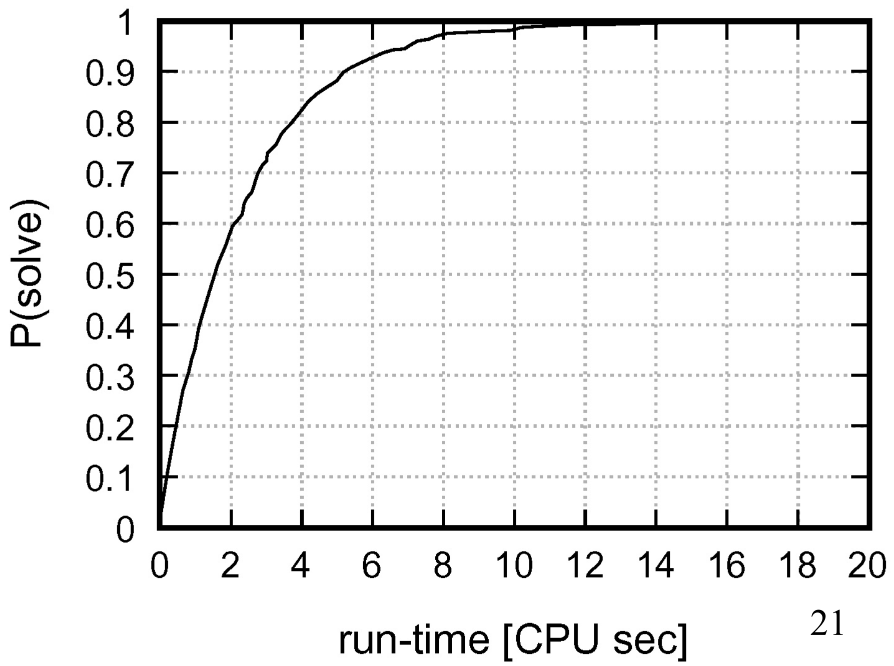




# Comparing SLS algorithms

- A better way to evaluate empirical performance
  - Runtime distributions
    - Perform many runs (e.g. below: 1000 runs)
    - Consider the empirical distribution of the runtimes
      - Sort the empirical runtimes (decreasing)
      - Rotate graph 90 degrees. E.g. below: longest run took 12 seconds





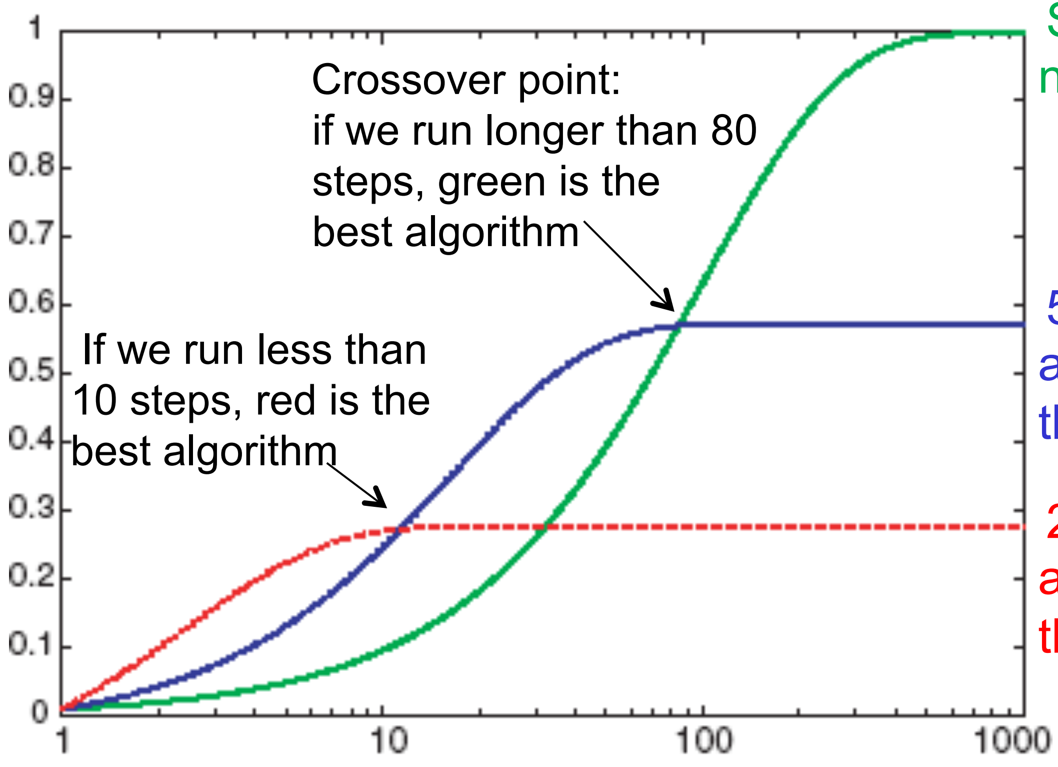
# Comparing runtime distributions

x axis: runtime (or number of steps)
y axis: proportion (or number) of runs solved in that runtime

– Typically use a log scale on the x axis

Fraction of solved runs, i.e.

P(solved by this time)



Slow, but does not stagnate

57% solved after 80 steps, then stagnate

28% solved after 10 steps, then stagnate

# of steps

Which algorithm is most likely to solve the problem within 30 steps?

blue

red

green

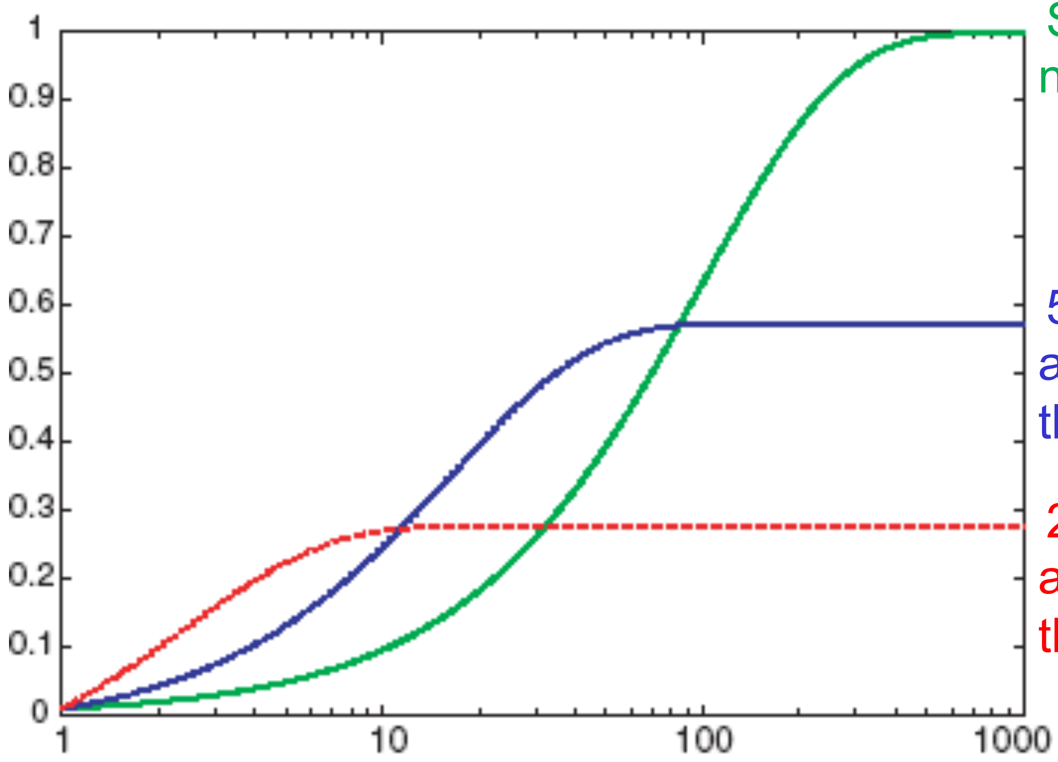
# Comparing runtime distributions

- Which algorithm has the best median performance?
  - I.e., which algorithm takes the fewest number of steps to be successful in 50% of the cases?

blue red green

Fraction of solved runs, i.e.

P(solved by this time)



Slow, but does not stagnate

57% solved after 80 steps, then stagnate

28% solved after 10 steps, then stagnate

# of steps

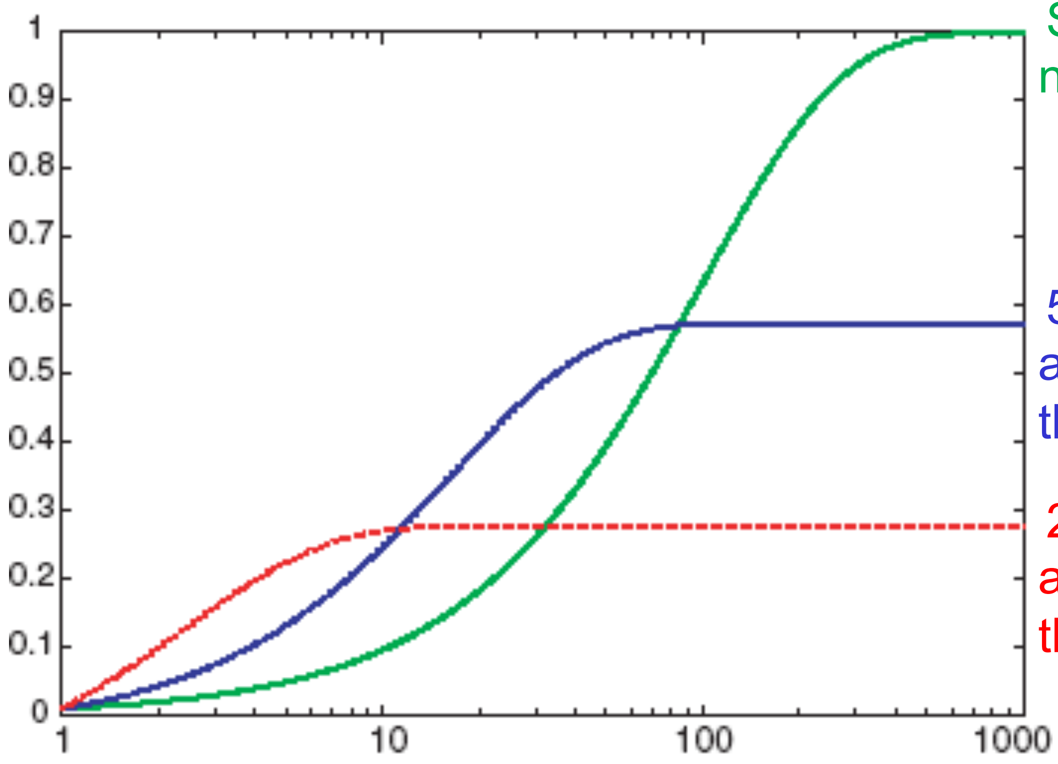
# Comparing runtime distributions

- Which algorithm has the best 70% quantile performance?
  - I.e., which algorithm takes the fewest number of steps to be successful in 70% of the cases?

blue red green

Fraction of solved runs, i.e.

P(solved by this time)



Slow, but does not stagnate

57% solved after 80 steps, then stagnate

28% solved after 10 steps, then stagnate

# of steps

# Pro’s and Con’s of SLS

- Typically no guarantee to find a solution even if one exists
  - Most SLS algorithms can sometimes stagnate
    - Not clear whether problem is infeasible or the algorithm stagnates
    - Very hard to analyze theoretically
  - Some exceptions: guaranteed to find global minimum as time $\rightarrow \infty$
    - In particular random sampling and random walk: strictly positive probability of making N lucky choices in a row
- Anytime algorithms
  - maintain the node with best h found so far (the “incumbent”)
  - given more time, can improve their incumbent
- Generality: can optimize arbitrary functions with n inputs
  - Example: constraint optimization
  - Example: RNA secondary structure design
- Generality: dynamically changing problems

## SLS generality: Constraint Optimization Problems

- Constraint Satisfaction Problems
  - Hard constraints: need to satisfy all of them
  - All models are equally good

- Constraint Optimization Problems
  - Hard constraints: need to satisfy all of them
  - Soft constraints: need to satisfy them as well as possible
  - Can have weighted constraints
    - Minimize h(n) = sum of weights of constraints unsatisfied in n
    - Hard constraints have a very large weight
    - Some soft constraints can be more important than other soft constraints → larger weight
  - All local search methods we will discuss work just as well for constraint optimization
    - all they need is an evaluation function h

## Example for constraint optimization problem

Exam scheduling

- Hard constraints:
  - Cannot have an exam in too small a room
  - Cannot have multiple exams in the same room in the same time slot
  - …
- Soft constraints
  - Student should not have to write two exams at the same time (important)
  - Students should not have multiple exams on the same day
  - It would be nice if students had their exams spread out
  - …

## SLS generality: optimization of arbitrary functions

- SLS is even more general
  - SLS's generality doesn't stop at constraint optimization
  - We can optimize arbitrary functions $f(x_1, \ldots, x_n)$ that we can evaluate for any complete assignment of their n inputs
  - The function's inputs correspond to our possible worlds, i.e. to the SLS search states

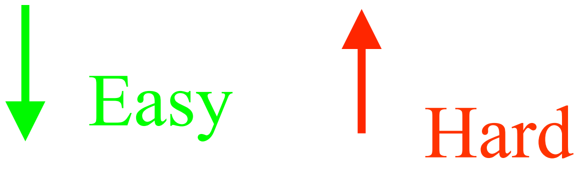
- Example: RNA secondary structure design

## Example: SLS for RNA secondary structure design

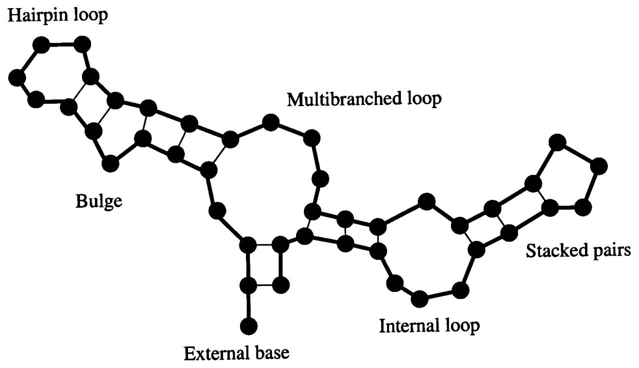
- RNA strand made up of four bases: cytosine (C), guanine (G), adenine (A), and uracil (U)
- 2D/3D structure RNA strand folds into is important for its function
- Predicting structure for a strand is "easy": $O(n^3)$
- But what if we want a strand that folds into a certain structure?
  - Local search over strands
    - Search for one that folds into the right structure
  - Evaluation function for a strand
    - Run $O(n^3)$ prediction algorithm
    - Evaluate how different the result is from our target structure
    - Only defined implicitly, but can be evaluated by running the prediction algorithm

RNA strand

GUCCCAUAGGAUGUCCCAUAGGA



Secondary structure



Best algorithm to date: Local search algorithm RNA-SSD developed at UBC [Andronescu, Fejes, Hutter, Condon, and Hoos, Journal of Molecular Biology, 2004]

## SLS generality: dynamically changing problems

- The problem may change over time
  - Particularly important in scheduling
  - E.g., schedule for airline:
    - Thousands of flights and thousands of personnel assignments
    - A storm can render the schedule infeasible

- Goal: Repair the schedule with minimum number of changes
  - Often easy for SLS starting from the current schedule
  - Other techniques usually:
    - Require more time
    - Might find solution requiring many more changes

## Many different types of local search

- There are many different SLS algorithms
  - Each could easily be a lecture by itself
  - We will only touch on each of them very briefly
  - Only need to know them on a high level
- For more details, see
  - UBC CS grad course “Empirical Algorithmics” by Holger Hoos
  - Book “Stochastic Local Search: Foundations and Applications” by Holger Hoos & Thomas Stützle, 2004 (in reading room)

# Simulated Annealing

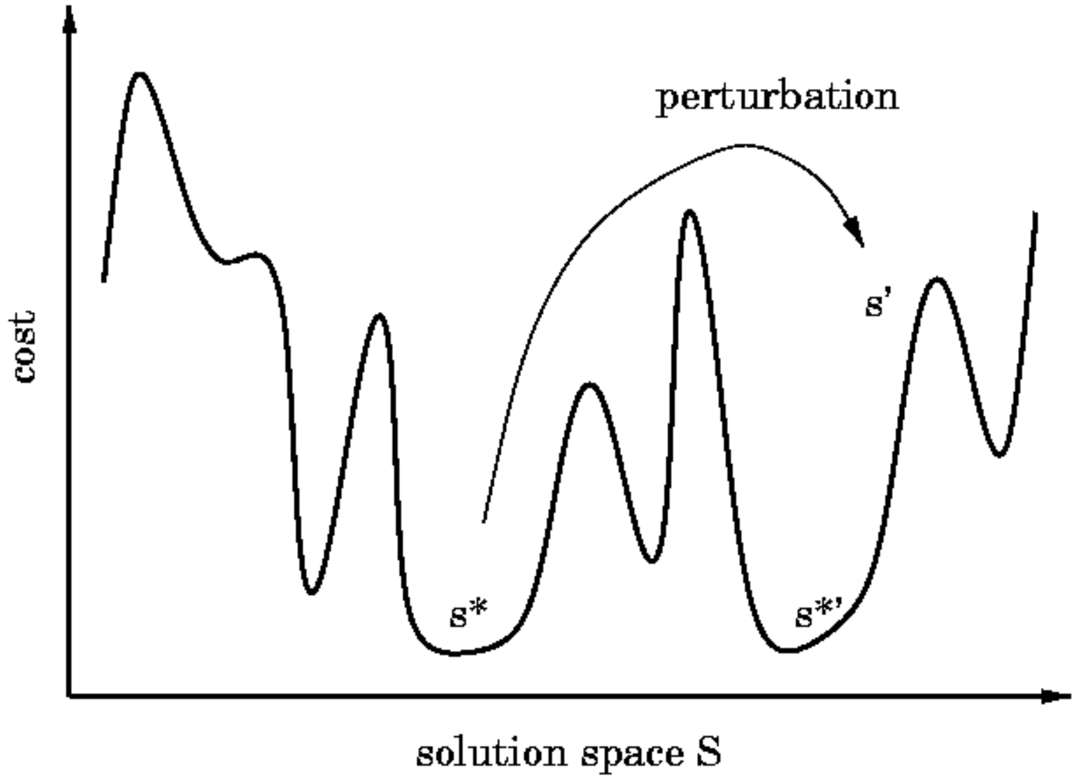
- Annealing: a metallurgical process where metals are hardened by being slowly cooled so settle into lowest energy state
- Analogy:
  - start with a high ‘temperature’: great tendency to take random steps
  - Over time, cool down: only take random steps that are not too bad
- Details:
  - At node n, select a random neighbour n’
  - If h(n’) < h(n), move to n’ (i.e. accept all improving steps)
  - Otherwise, adopt it with a probability depending on
    - How much worse n’ is than n
    - the current temperature T: high T tends to accept even very bad moves
    - Probability of accepting worsening move: $\exp( (h(n) – h(n’) )/ T )$
  - Temperature reduces over time, according to an annealing schedule
    - “Finding a good annealing schedule is an art”
    - E.g. geometric cooling: every step multiply T by some constant < 1

# Tabu Search

- Mark partial assignments as tabu (‘taboo’= forbidden)
  - Prevents repeatedly visiting the same (or similar) local minima
  - Maintain a queue of k variable=value assignments that are tabu
  - E.g., when changing $V_7$’s value from 2 to 4, we cannot change $V_7$ back to 2 for the next k steps
  - k is a parameter that needs to be optimized empirically

# Iterated Local Search

- Perform iterative best improvement to get to local minimum
- Perform perturbation step to get to different parts of the search space
  - E.g. a series of random steps (random walk)
  - Or a short tabu search



# Beam Search

- Keep not just 1 assignment, but k assignments at once
  - A 'beam' with k different assignments (k is the 'beam width')
- The neighbourhood is the union of the k neighbourhoods
  - At each step, keep only the k best neighbours
  - Never backtrack
- When k=1, this is identical to:

Greedy descent | Breadth first search | Best first search

  - Single node, always move to best neighbour: greedy descent
- When k=∞, this is basically:

Greedy descent | Breadth first search | Best first search

  - At step m, the beam contains all nodes m steps away from the start node
  - Like breadth first search,
    but expanding a whole level of the search tree at once
- The value of k lets us limit space and parallelism

## Stochastic Beam Search

- Like beam search, but you probabilistically choose the k nodes at the next step (‘generation’)

- The probability that neighbour n is chosen depends on h(n)
  - Neighbours with low h(n) are chosen more frequently
  - E.g. rank-based: node n with lowest h(n) has highest probability
    - probability only depends on the order, not the exact differences in h
  - This maintains diversity amongst the nodes

- Biological metaphor:
  - like asexual reproduction:
    each node gives its mutations and the fittest ones survive

## Genetic Algorithms

- Like stochastic beam search, but pairs of nodes are combined to create the offspring

- For each generation:
  - Choose pairs of nodes $n_1$ and $n_2$ (‘parents’),
    where nodes with low h(n) are more likely to be chosen from the population
  - For each pair ($n_1$, $n_2$), perform a cross-over:
    create offspring combining parts of their parents
  - Mutate some values for each offspring
  - Select from previous population and all offspring which nodes to keep in the population

# Example for Crossover Operator

- Given two nodes:

  $X_1 = a_1,\ \ X_2 = a_2,\ \ \ldots,\ \ X_m = a_m$

  $X_1 = b_1;\ \ X_2 = b_2,\ \ \ldots,\ \ X_m = b_m$

- Select i at random, form two offspring:

  $X_1 = a_1,\ \ X_2 = a_2,\ \ \ldots, X_i = a_i,\ \ X_{i+1} = b_{i+1},\ \ \ldots,\ \ X_m = b_m$

  $X_1 = b_1,\ \ X_2 = b_2,\ \ \ldots, X_i = b_i,\ \ X_{i+1} = a_{i+1},\ \ \ldots,\ \ X_m = a_m$

- Many different crossover operators are possible

- Genetic algorithms is a large research field
  - Appealing biological metaphor
  - Several conferences are devoted to the topic

## Parameters in stochastic local search

- Simple SLS
  - Neighbourhoods, variable and value selection heuristics, percentages of random steps, restart probability
- Tabu Search
  - Tabu length (or interval for randomized tabu length)
- Iterated Local Search
  - Perturbation types, acceptance criteria
- Genetic algorithms
  - Population size, mating scheme, cross-over operator, mutation rate
- Hybridizations of algorithms: many more parameters

# The Algorithm Configuration Problem

## Definition

- Given:
  - Runnable algorithm A, its parameters and their domains
  - Benchmark set of instances B
  - Performance metric m
- Find:
  - Parameter setting ('configuration') of A optimizing m on B

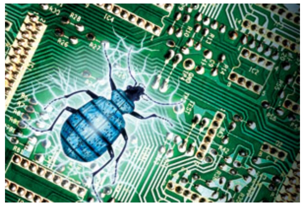
UBC Ph.D. thesis (Hutter, 2009): "Automated configuration of algorithms for solving hard computational problems"

## Motivation for automated algorithm configuration

Customize versatile algorithms for different application domains

- Fully automated
  - Saves valuable human time
  - Can improve performance dramatically

Solver config 1


Solver config 2

## Generality of Algorithm Configuration

Arbitrary problems, e.g.

- SAT, MIP, Timetabling, Probabilistic Reasoning, Protein Folding, AI Planning, ….

Arbitrary parameterized algorithms, e.g.

- Local search
  - Neighbourhoods, restarts, perturbation types, tabu length, etc
- Genetic algorithms & evolutionary strategies
  - Population size, mating scheme, crossover operators, mutation rate, hybridizations, etc
- Systematic tree search
  (advanced versions of arc consistency + domain splitting)
  - Branching heuristics, no-good learning, restart strategy, pre-processing, etc

## Simple Manual Approach for Configuration

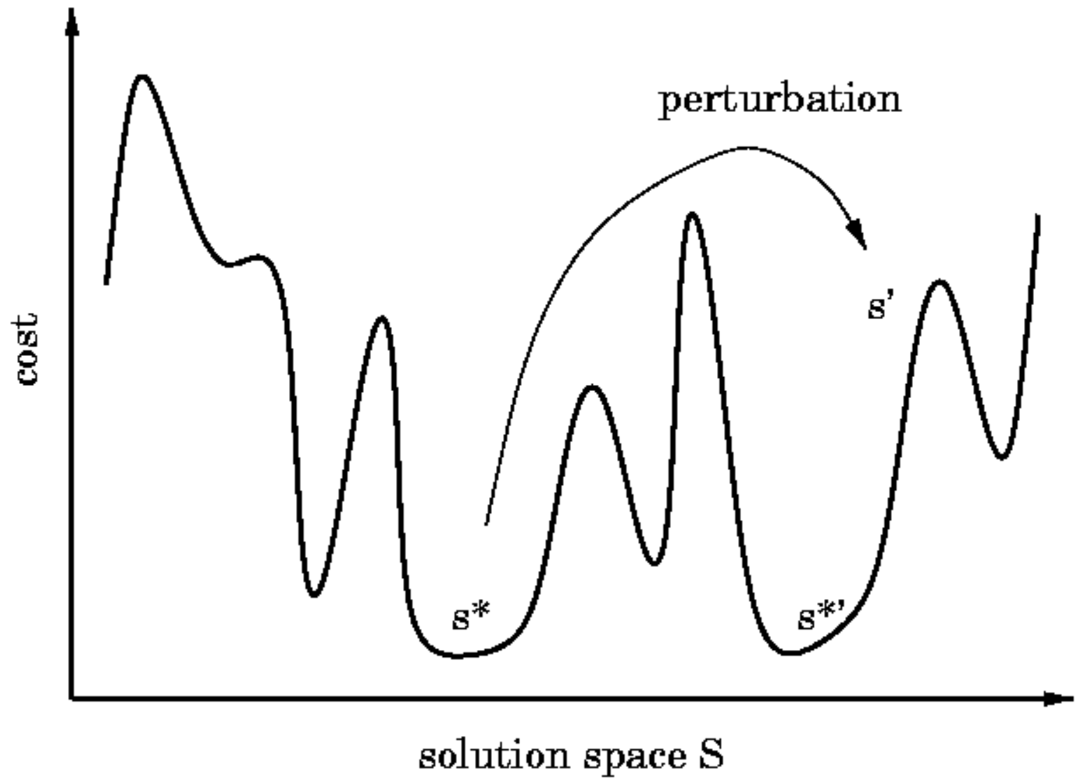
```
Start with some configuration
repeat
    Modify a single parameter
    if results on benchmark set improve then
        keep new configuration
until no more improvement possible (or “good enough")
```

→ *Manually executed local search*

# The ParamILS Framework

[Hutter, Hoos & Stützle; AAAI '07 & Hutter, Hoos, Leyton-Brown & Stützle; JAIR'09]

Iterated Local Search in parameter configuration space:



# Example application for ParamILS: solver for mixed integer programming (MIP)

IP: NP-hard constraint optimization problem

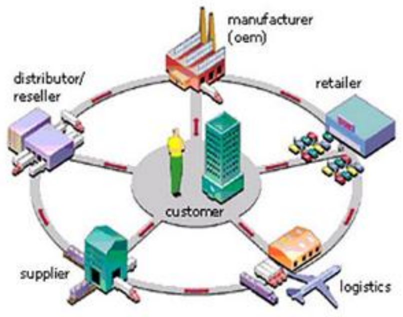
$$\begin{aligned} \min \quad & c^{\mathsf{T}} x \\ \text{s. t.} \quad & Ax \le b \\ & x_i \in \mathbb{Z} \text{ for i} \in \text{I} \end{aligned}$$

MIP = IP with only some integer variables

Commercial state-of-the-art MIP solver IBM ILOG CPLEX:

– licensed by > 1000 universities and 1300 corporations, including ⅓ of the Global 500

Transportation/Logistics: SNCF, United Airlines, UPS, United States Postal Service, …



Supply chain management software: Oracle, SAP, …

Production planning and optimization: Airbus, Dell, Porsche, Thyssen Krupp, Toyota, Nissan, …

Up to 50-fold speedups just by optimizing the parameters!