# CPSC 322: Introduction to Artificial Intelligence

## CSPs: Stochastic Local Search

Textbook reference: [4.7.2,4.7.4]

Instructor: Varada Kolhatkar
University of British Columbia

Credit: These slides are adapted from the slides of the previous offerings of the course. Thanks to all instructors for creating and improving the teaching material and making it available!

# Announcements

- Assignment 2 has been released and is due on **21 Oct 11:59pm**.
- Midterm **practice questions** are available on Piazza.
- Midterm time and location
  **Time:** Friday, Oct 25th, from 6pm to 7pm
  **Location:** Woodward 2
  (Instructional Resources Centre-IRC) (WOOD) - 2
- My office hours: Fridays from 11am to noon in ICCS 185. Will also hold extra office hour for midterm next Wednesday. (Details will be posted on Piazza.)

# Lecture outline

- Recap local search (~5 mins) 👈
- Stochastic local search (~25 mins)
- Class activity (~15 mins)
- Evaluating random algorithms (~10 mins)
- SLS pros and cons (~5 mins)
- Summary and wrap up (~5 mins)

# Local search problem

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.

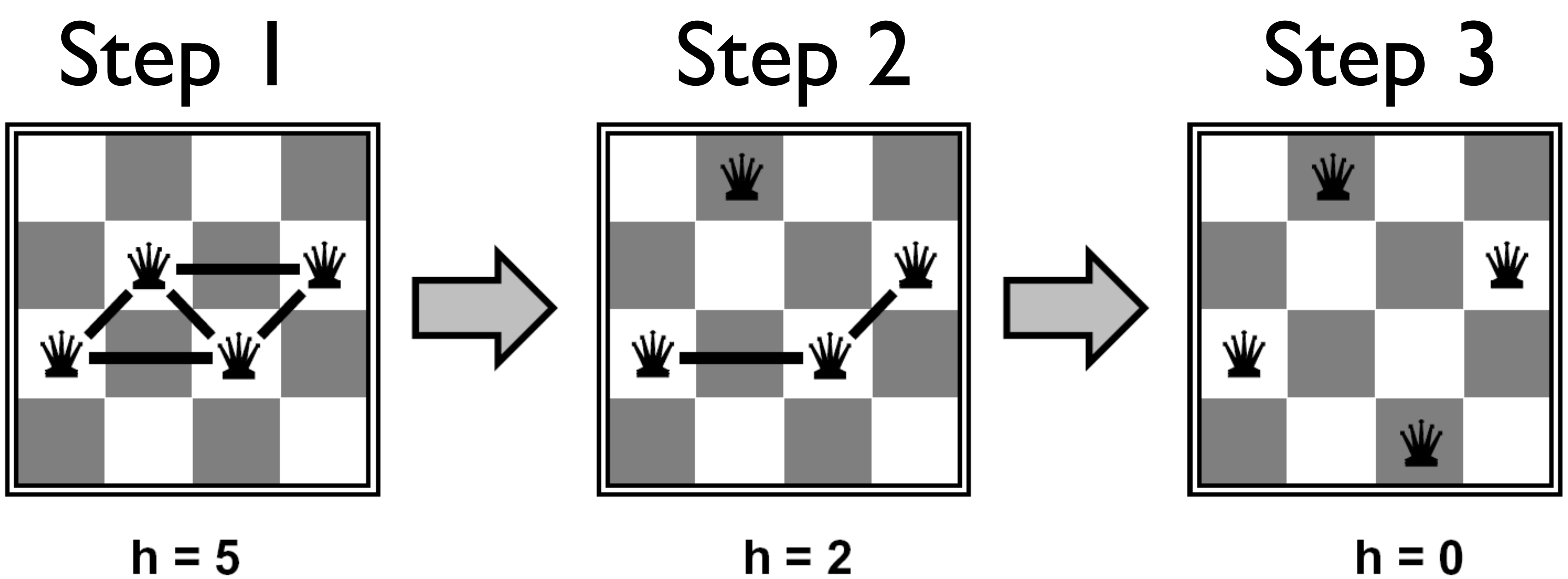
**Neighbour relation**: an edge in the search space will exist when the neighbour relation holds between a pair of nodes.

**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: 4-Queen problem

**CSP**: 4-Queen CSP
One variable per column;
Domains {1,2,3,4}: row where the queen in the $i^{th}$ column sits;
Constraints: no two queens in the same row, column or diagonal
**Neighbour relation:** value of a single column differs
**Scoring function:** number of attacks

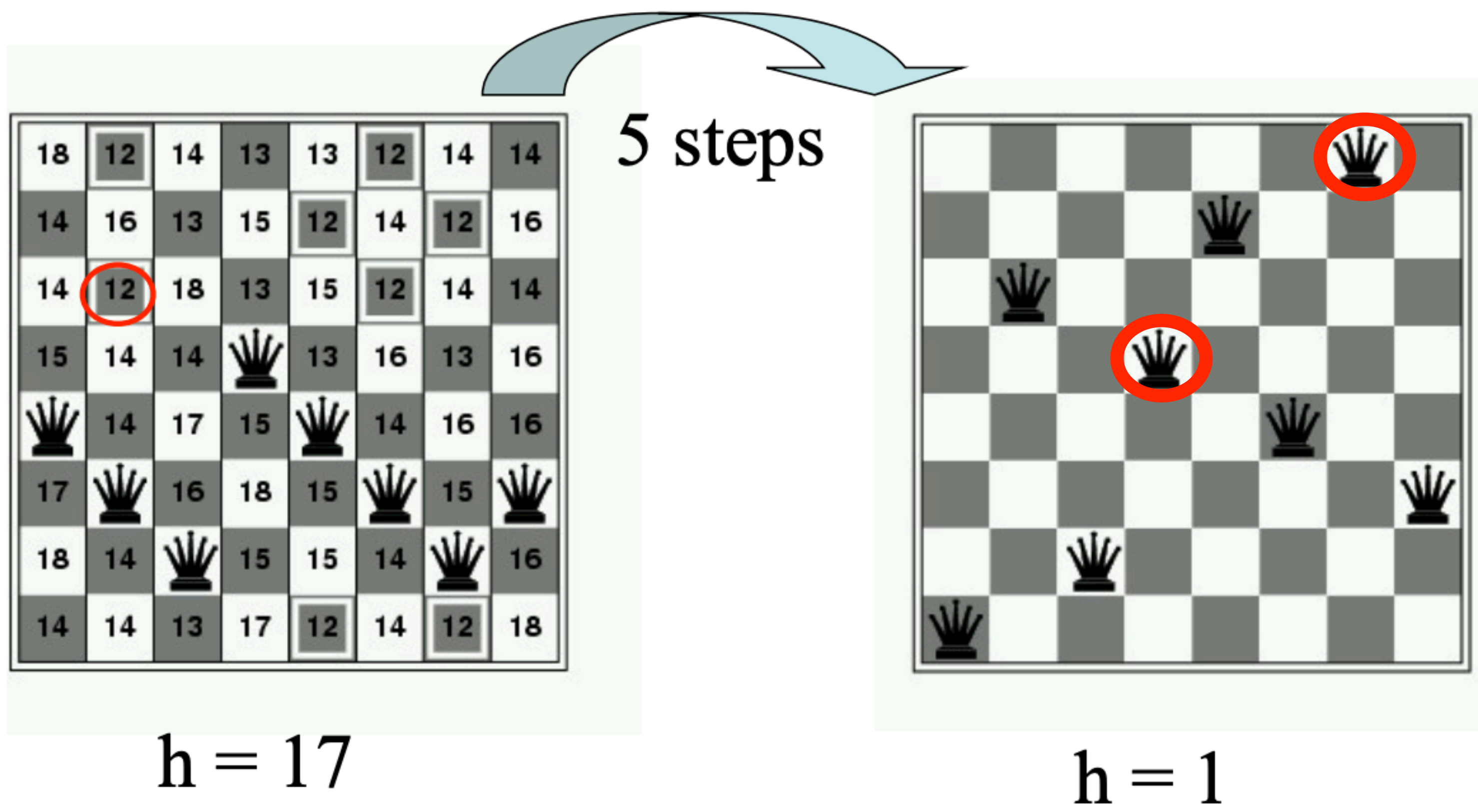


# Determining the "best" neighbour

- **Iterative best improvement**: Select the neighbour that optimizes some scoring/evaluation function $h(n)$.
  - **Greedy descent**: Evaluate $h(n)$ for each neighbour, pick the neighbour $n$ with **minimal** $h(n)$
  - **Hill climbing**: Evaluate $h(n)$ for each neighbour, pick the neighbour $n$ with **maximum** $h(n)$
  - Note that Minimizing $h(n)$ is identical to maximizing $-h(n)$

# Local minima

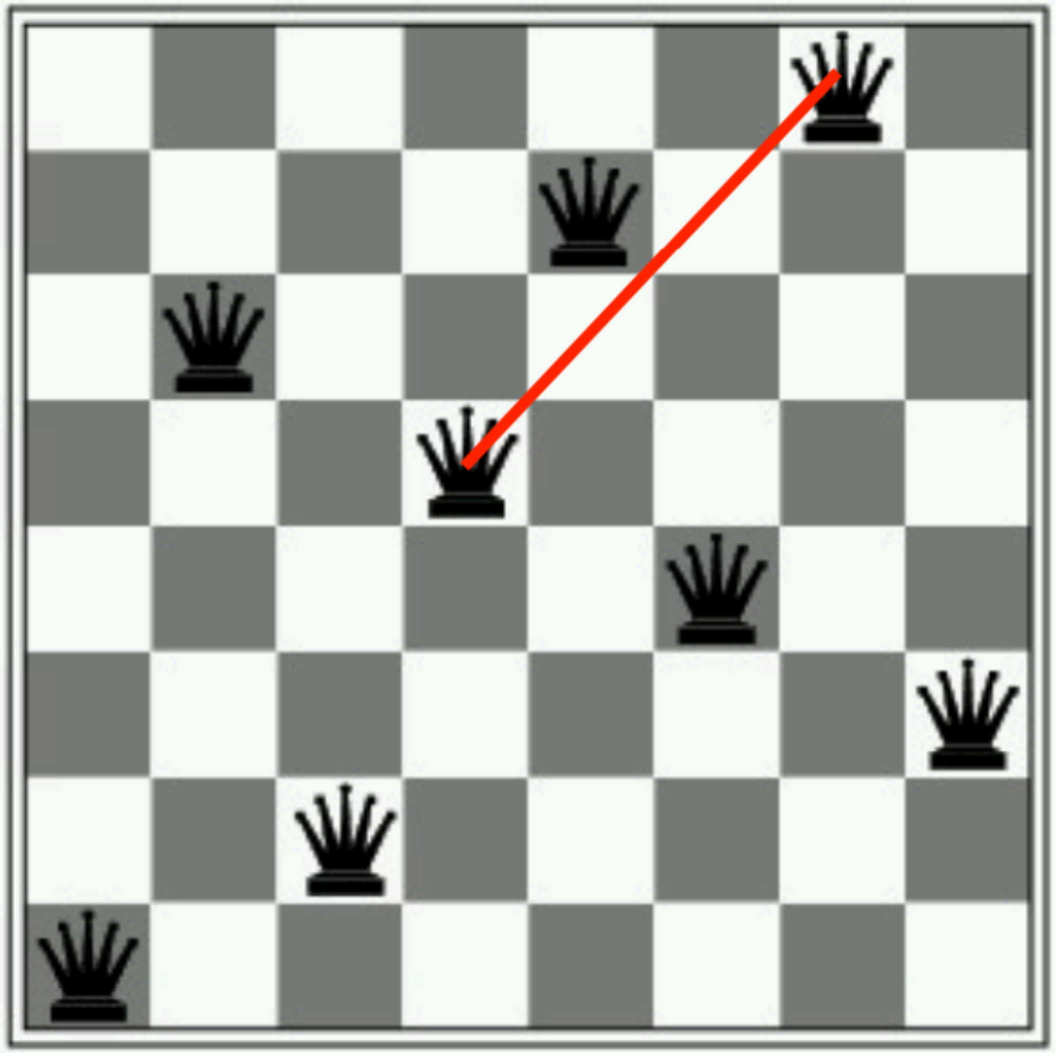
Iterative best improvement picks one of the best neighbours (successor) of the current assignment, but it can get stuck in local minima that are not global minima.




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

# Local minima

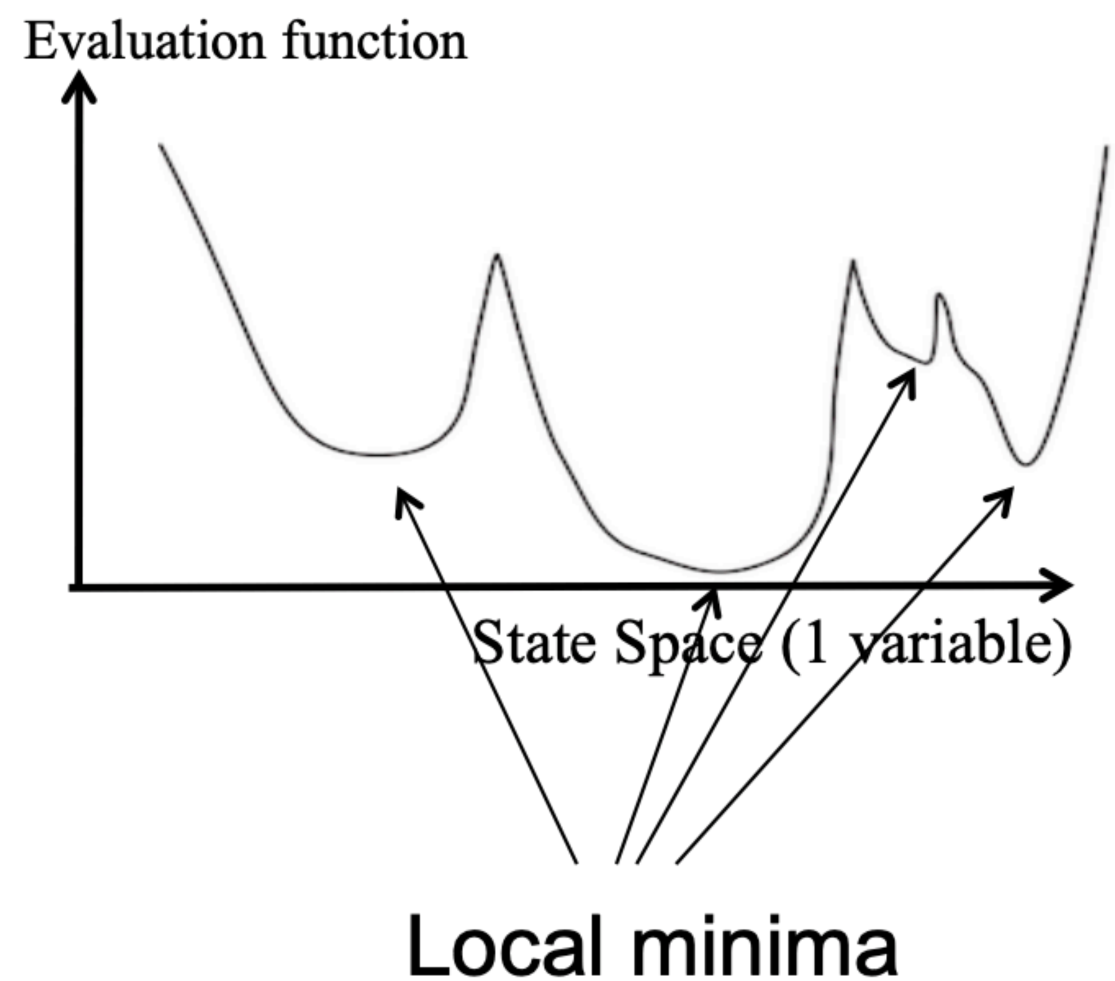
- Which move should we pick in this situation?
- Current cost: h=1
- No single move can improve on this
- In fact, every single move only makes things worse ($h \geq 2$)
- Locally optimal solution. Since we are minimizing: local minimum



# 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.



# Stochastic local search (SLS)

Iterative best improvement picks one of the best neighbours (successor) of the current assignment, but it can get stuck in local minima that are not global minima.

A mix of iterative best improvement with random moves is an instance of a class of algorithms known as **stochastic local search**.

# Today: Learning outcomes

From this lecture, students are expected to be able to:

- Implement SLS with
  - Random steps (1-step, 2-step versions)
  - Random restart
- Compare SLS algorithms with runtime distributions
- Explain pros and cons of SLS

# Lecture outline

- Recap local search (~5 mins)
- Stochastic local search (~25 mins)
- Class activity (~15 mins)
- Evaluating random algorithms (~10 mins)
- SLS pros and cons (~5 mins)
- Summary and wrap up (~5 mins)

# SLS: Successful application

Scheduling of Hubble Space Telescope:

- reducing time to schedule 3 weeks of observations:
    - from **one week** to around **10 sec**.

# Stochastic local search

GOAL: We want our local search

- To be guided by the scoring function
- Not to get stuck in local maxima/minima, plateaus etc.

Add 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 initialization

Local search step

# General local search for greedy descent

```
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

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

# General local search for random sampling

```
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

Do not go in the while loop. Always start fresh.

# Tracing SLS algorithms in Aispace

Let’s look at these algorithms in AIspace:

- Greedy Descent
- Random Sampling

Simple scheduling problem 2 in AIspace:

# 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:

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

- **Random steps**: move to a random neighbour

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

# Stochastic local search

We can alternate

A. Greedy descent steps

B. Random steps: Move to a random neighbour

C. Random restart: reassign random values to all variables

# Stochastic local search

```
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 or restart

Sometimes select the “best” neighbour

Sometimes select a neighbour at random

# General local search for random walk

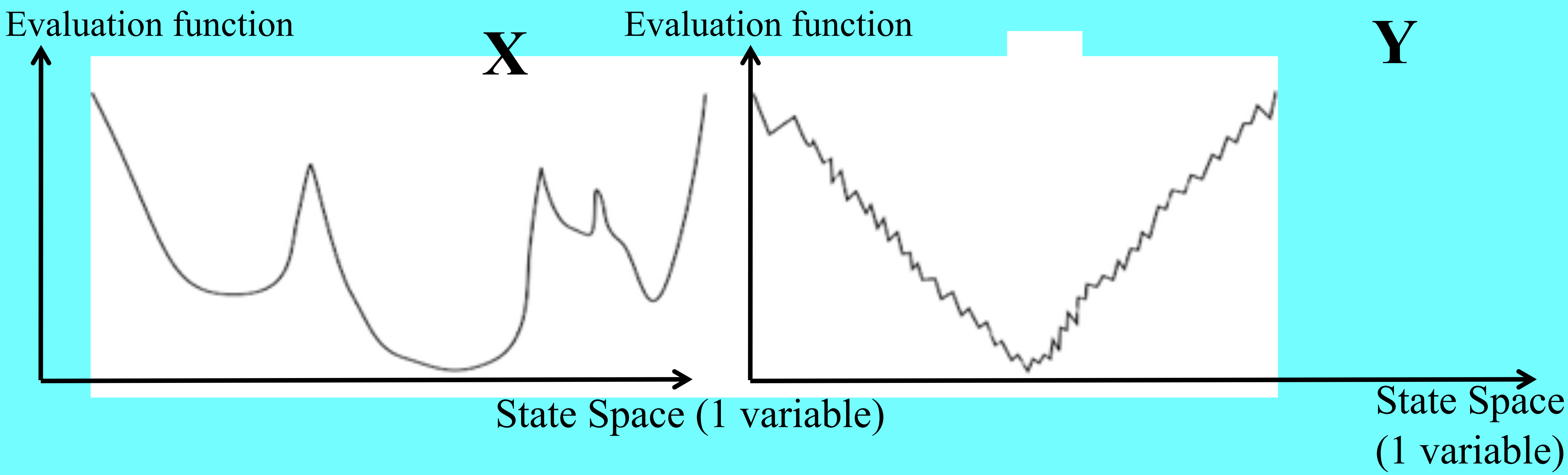
```
Procedure Local-Search(V,dom,C)
        Inputs
                V: a set of variables
                dom: a function such that dom(X) is the domain of variable X
                C: set of constraints to be satisfied
        Output
                complete assignment that satisfies the constraints
        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

Keep choosing a neighbour randomly instead of “best” neighbour.

# Random steps vs. Random restart

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

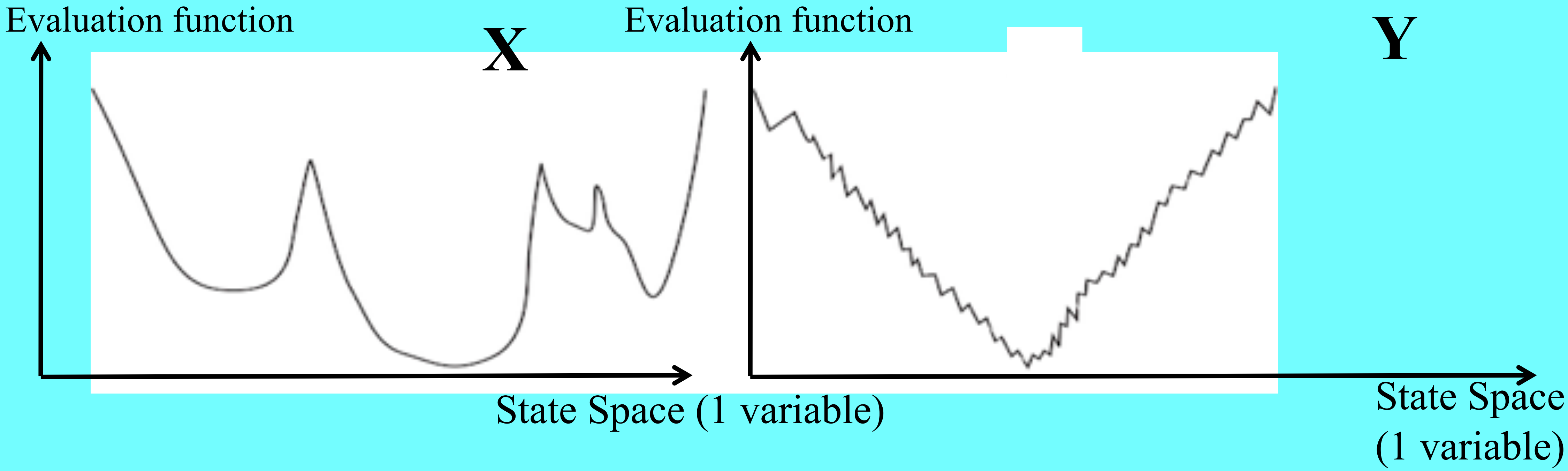



A. Greedy descent with random steps best on X
   Greedy descent with random restart best on Y

B. Greedy descent with random steps best on Y
   Greedy descent with random restart best on X

# Random steps vs. Random restart

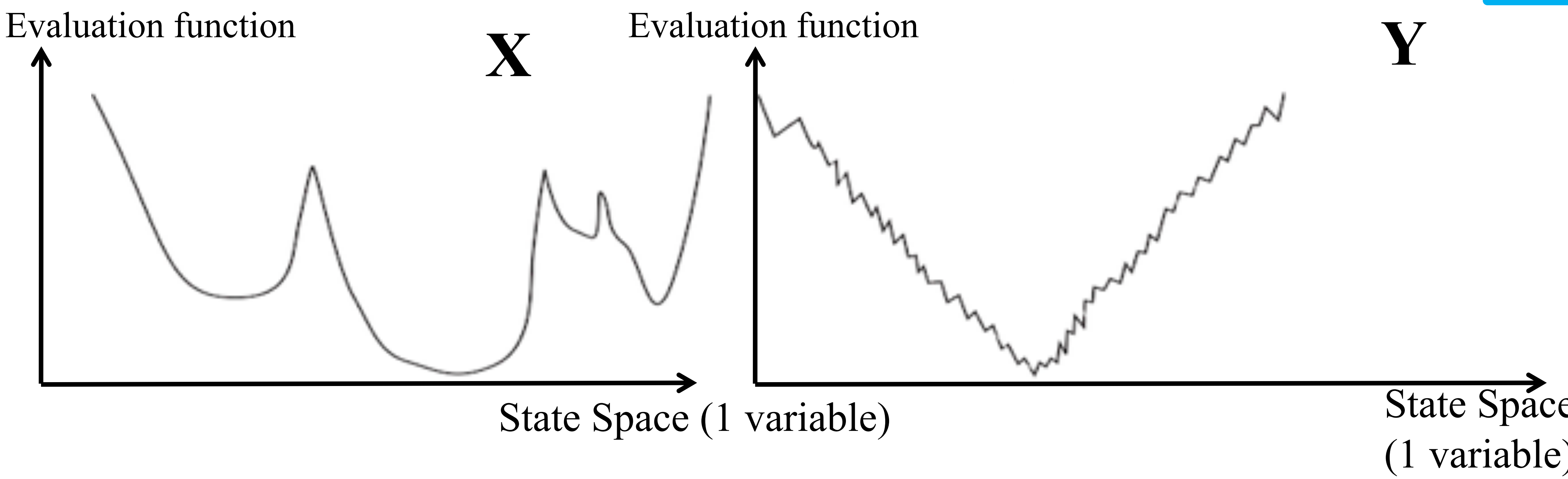
Which randomized method would work best in each of these two search spaces?




A. Greedy descent with random steps best on X
Greedy descent with random restart best on Y

B. Greedy descent with random steps best on Y
Greedy descent with random restart best on X ✅

# Random steps vs. Random restart




- These examples are simplified extreme cases for illustration
  - In practice you don’t know how the search space looks like
- Usually integrating both kinds of randomization works best

# ASIDE: Random walk view of PageRank

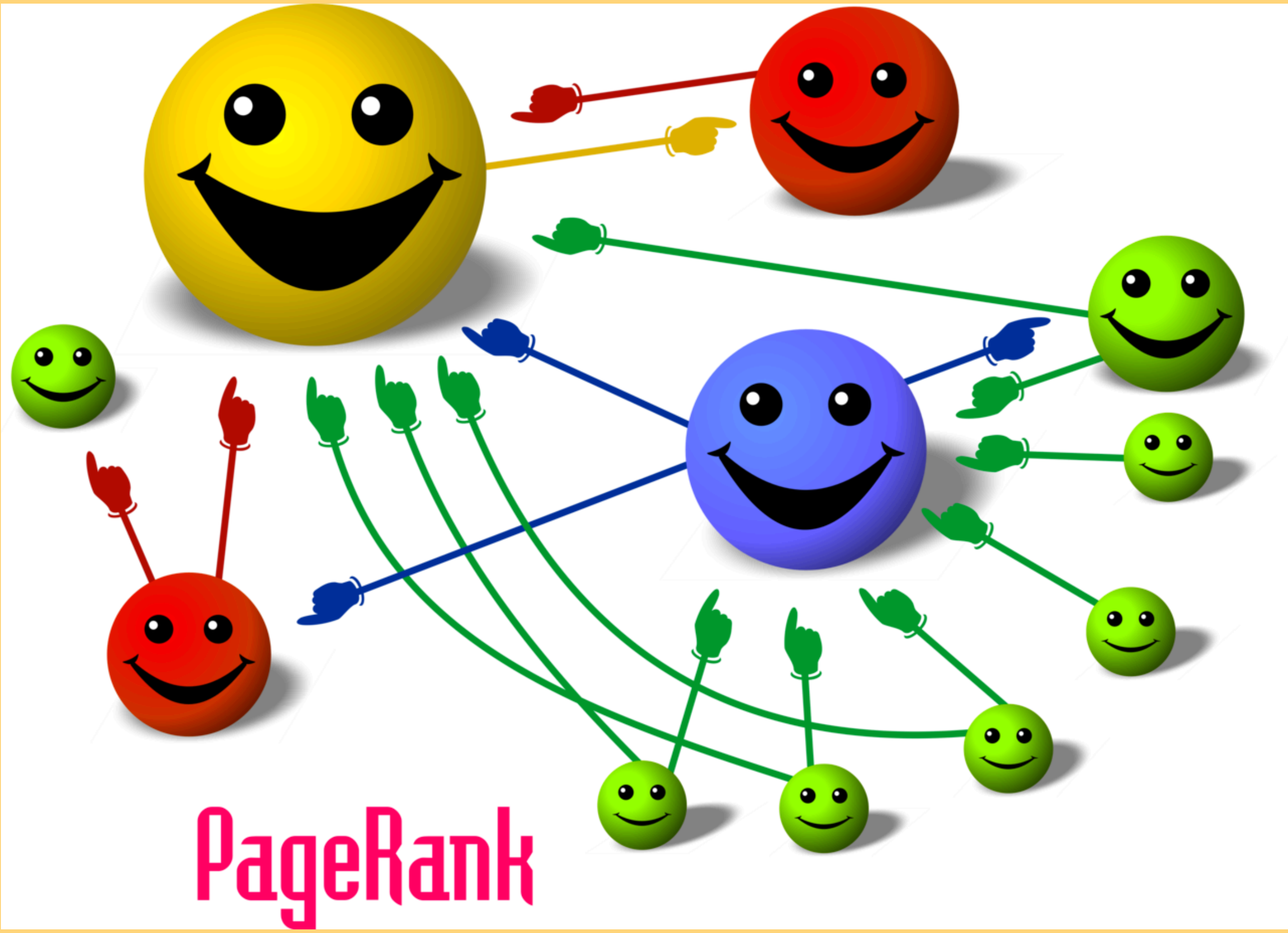
PageRank algorithms can be interpreted as a random walk

At $t = 0$ start at a random webpage

At $t = 1$ follow a random link on the current webpage

At $t = 2$ follow a random link on the current webpage

Probability of landing at page as $t \to \infty$ is the pagerank.




Wikipedia's cartoon illustration of PageRank. Large face = Higher rank

# Stochastic local search for CSPs

**Start node**: random assignment
**Goal**: assignment with zero unsatisfied constraints
**Heuristic function** $h$: number of unsatisfied constraints

**Stochastic local search** is a mix of:

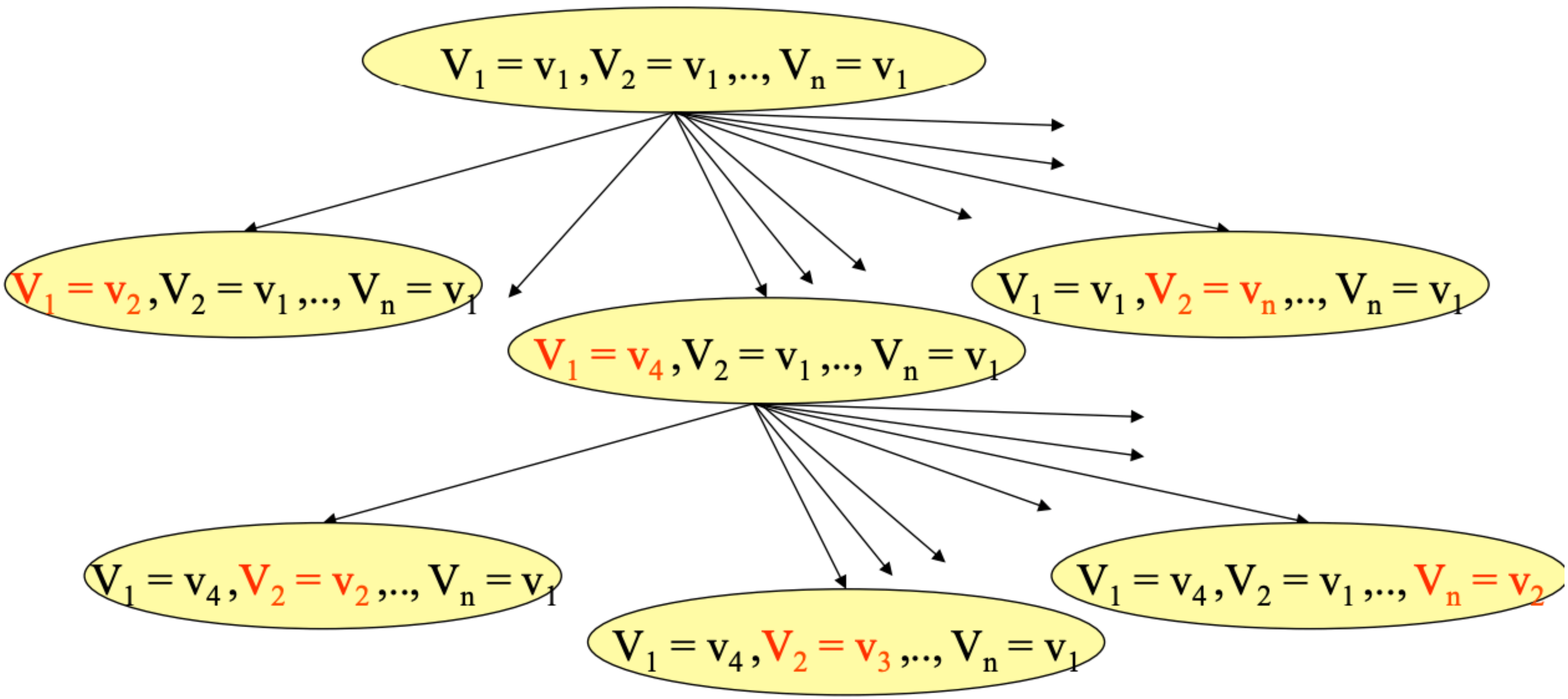
- **Greedy descent**: move to neighbour with lowest $h$
- **Random walk**: take some random steps
- **Random restart**: reassigning values to all variables

# Stochastic local search for CSPs

More examples of ways to add randomness to local search for a CSP

- **One-stage** selection: Choose a random variable-value pair
- **Two-stage** selection: First select variable V, then new value for V

# Variable selection



# One stage selection

One stage selection: all assignments that **differ in exactly one variable.** How many of those are there for N variables and domain size d?

A. $O(Nd)$

B. $O(d^N)$

C. $O(N^d)$

D. $O(N+d)$

# One stage selection

One stage selection: all assignments that **differ in exactly one variable.** How many of those are there for N variables and domain size d?

A. $O(Nd)$ ✅

B. $O(d^N)$

C. $O(N^d)$

D. $O(N + d)$

# 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.

# Two-stage selection: selecting variables

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**

# Two-stage selection: selecting values

First select variable V, then new value for V

- 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

# Greedy descent with min-conflict heuristic

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

# When do we stop?

- When you know the solution found is optimal (e.g., no constraint violations)
- Or when you're out of time: you have to act **NOW**

```
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
```

# Lecture outline

- Recap local search (~5 mins)
- Stochastic local search (~25 mins)
- Class activity (~15 mins) 👈
- Evaluating random algorithms (~10 mins)
- SLS pros and cons (~5 mins)
- Summary and wrap up (~5 mins)

# Class activity (~10 mins)

| Local search algorithm | Stopping criteria condition (line 5) | Variable and value selection (line 6) |
|---|---|---|
| Random sampling | | |
| Random walk | | |
| Greedy descent | | |
| Greedy descent with random walk | | |
| Greedy descent with random restart | | |

# Lecture outline

- Recap local search (~5 mins)
- Stochastic local search (~25 mins)
- Class activity (~15 mins)
- Evaluating random algorithms (~10 mins) 👈
- SLS pros and cons (~5 mins)
- Summary and wrap up (~5 mins)

# 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 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

# Evaluating SLS algorithms

- 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
  - Still, summary statistics, such as mean run time or median run time, don't tell the whole story
    - E.g., it would penalize an algorithm that often finds  a solution quickly but sometime stagnates

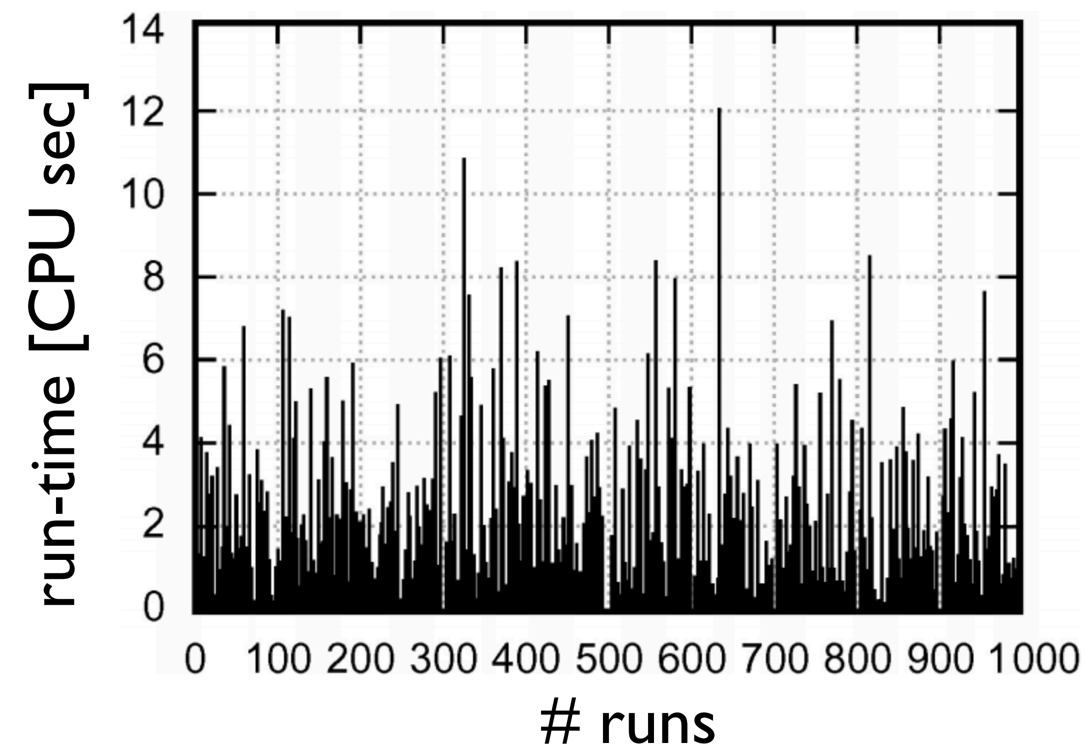
# Comparing SLS algorithms

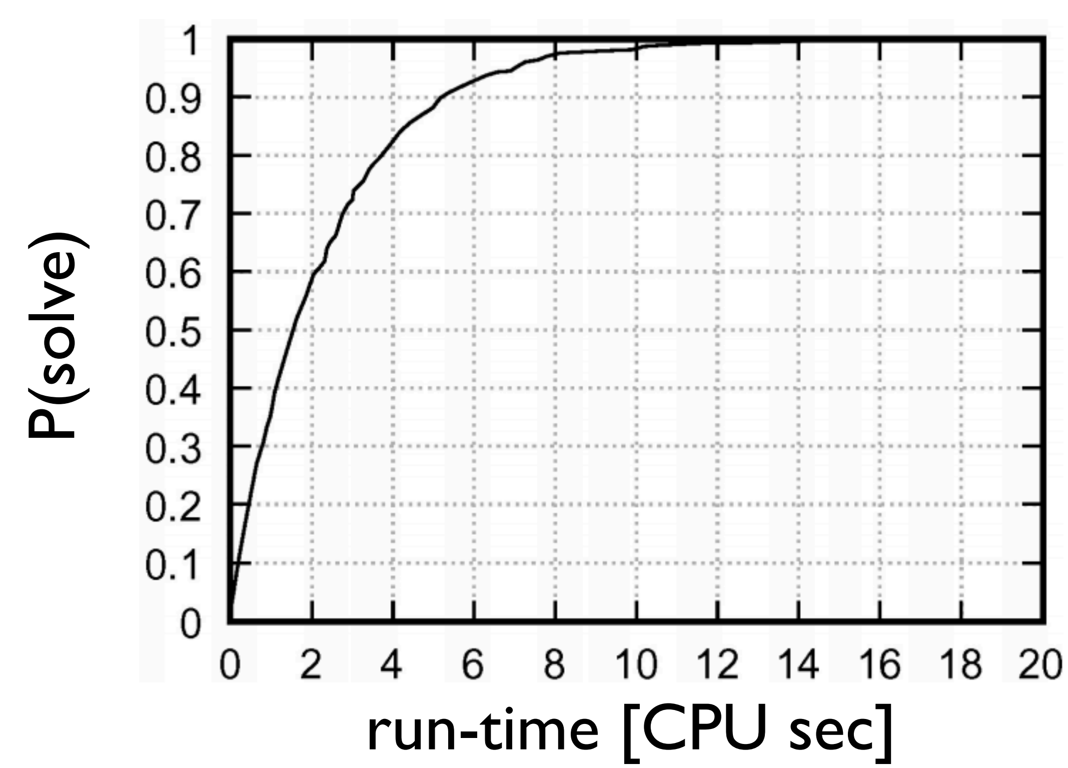
- A better way to evaluate empirical performance is **Runtime distributions**.

- Perform many runs (e.g., 1000 runs)

- Consider the empirical distribution of the runtimes

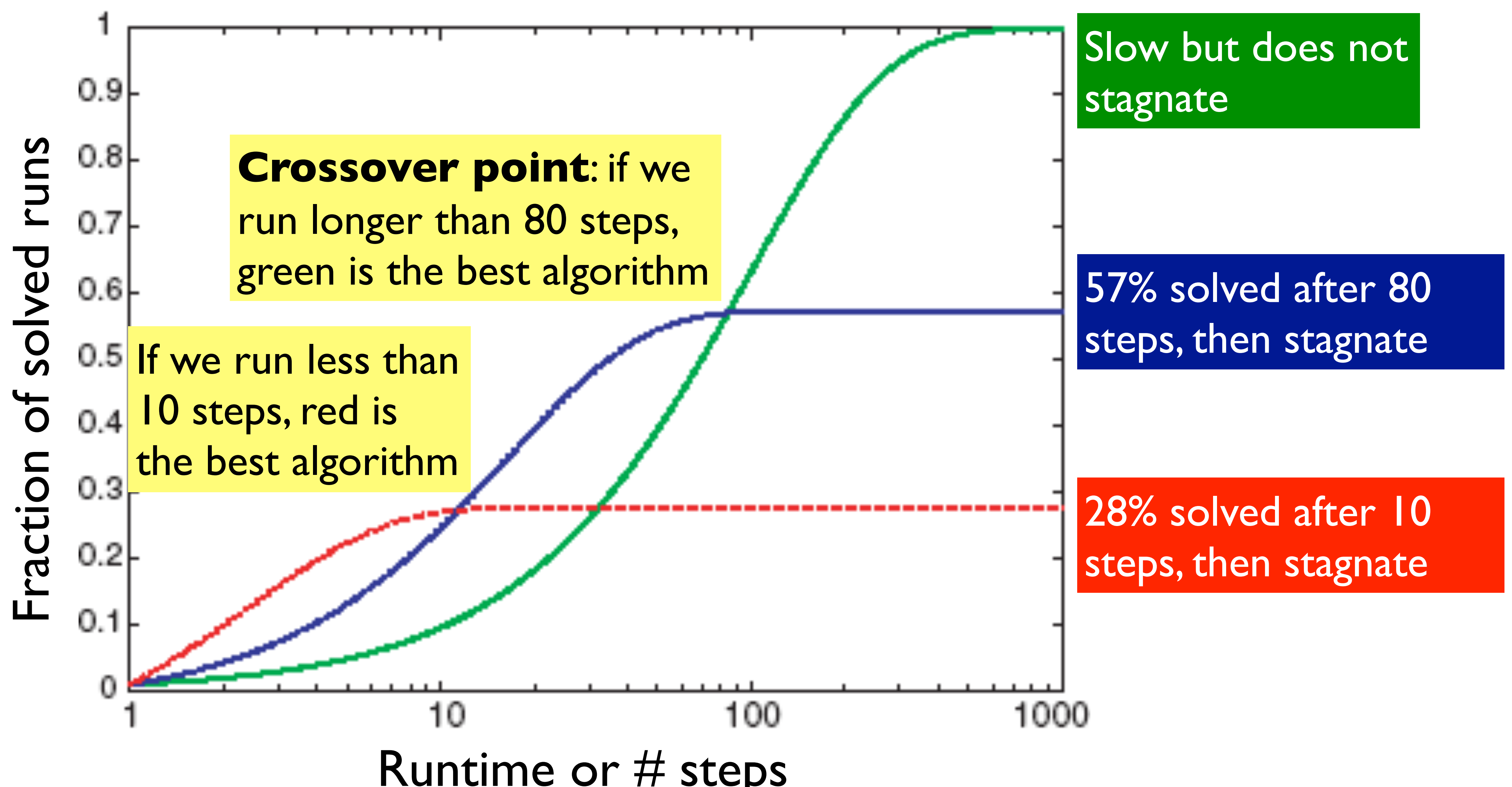
- Sort the empirical runtimes (decreasing)



# Comparing runtime distributions

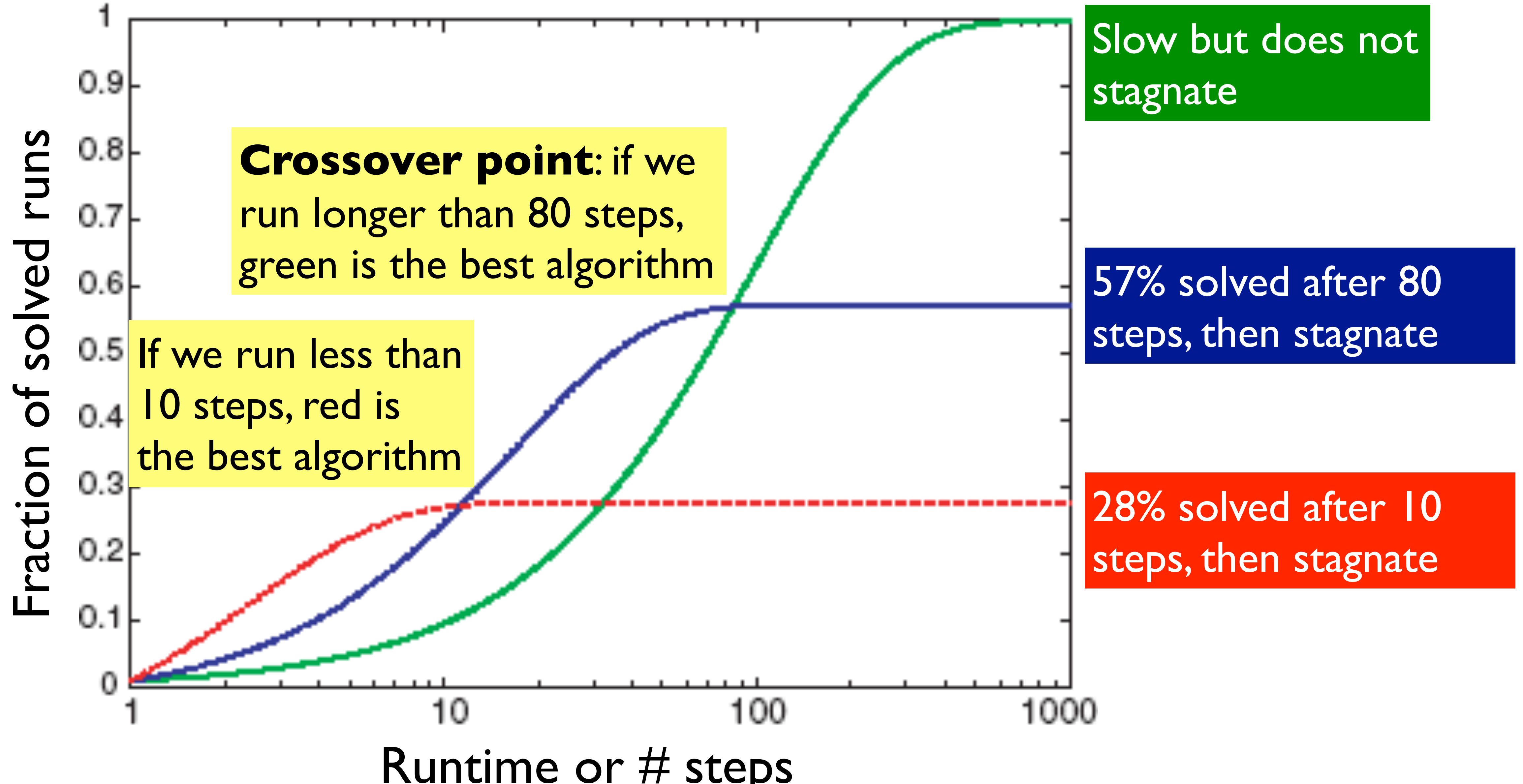
Plots runtime (or number of steps) and the proportion (or number) of the runs that are solved within that runtime.




log scale on the x axis is commonly used

# Comparing runtime distributions

Which algorithm takes the fewest number of steps to be successful in 70% of the cases?




log scale on the x axis is commonly used

# Runtime distributions in Aispace

Let’s look at some algorithms and their runtime distributions:

1. Greedy Descent
2. Random Sampling
3. Random Walk
4. Greedy Descent with random walk

Simple scheduling problem 2 in AIspace

# Lecture outline

- Recap local search (~5 mins)
- Stochastic local search (~25 mins)
- Class activity (~15 mins)
- Evaluating random algorithms (~10 mins)
- SLS pros and cons (~5 mins) 👈
- Summary and wrap up (~5 mins)

# Pros and cons of SLS

**Typically no guarantee to find a solution even if one exists**

- Most SLS algorithms can sometimes stagnate and usually it's not clear whether problem is infeasible or the algorithm stagnates
- Very hard to analyze theoretically

**Generality**

- Fast and do not require much memory
- can optimize arbitrary functions with *n* inputs Example: constraint optimization
- Work well for dynamically changing problems

# Random restart

- Randomized algorithm that succeeds some of the time can be extended to an algorithm that succeeds more often by running it multiple times

- Guaranteed to find global minimum as time $\to \infty$

  - In particular, random sampling and random walk: strictly positive probability of making *N* lucky choices in a row

# SLS: dynamically changing problems

- When the problem can change (particularly important in scheduling). E.g., schedule for airline: thousands of flights and thousands of personnel assignment
  - Storm can render the schedule infeasible
- Goal: Repair with minimum number of changes
- This can be easily done with a local search starting form the current schedule
- Other techniques usually require more time and might find solution requiring many more changes

# SLS: anytime algorithms

- Maintain the node with best $h$ found so far (the "incumbent")

- Given more time, can improve its incumbent

# Constrained optimization problems

**Constraint satisfaction problems**

- Hard constraints: Need to satisfy all of them
- All models are equally good

**Constrained optimization problems**

- Hard constraints: Need to satisfy all of them
- Soft constraints: need to satisfy them as well as possible
- Can have weighted constraints

# Constrained optimization problems

Possible 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

All local search methods we will discuss work just as well for constrained optimization. All they need is an evaluation function $h$.

# Exam scheduling: constrained optimization problem

- Example hard constraints
  - Cannot have an exam in too small a room
  - Cannot have multiple exams in the same room in the same time slot …
- Example soft constraints
  - Student should not have to write multiple exams on the same day
  - It would be nice if students had their exams spread out

# Lecture outline

- Recap local search (~5 mins)
- Stochastic local search (~25 mins)
- Class activity (~15 mins)
- Evaluating random algorithms (~10 mins)
- SLS pros and cons (~5 mins)
- Summary and wrap up (~5 mins) 👈🏻

# Stochastic local search: Summary

- Key idea: combine greedily improving moves with **randomization**

- Along with improving steps we can allow a "small probability" of:

    - **Random steps**: move to a random neighbour.

    - **Random restart**: reassign random values to all variables.

# Stochastic local search: Summary

- Always keep **best solution** found so far
- Stop when
  - Solution is found (in vanilla CSP, all constraints satisfied)
  - Run out of time (return best solution so far)

# Coming up

Readings for next class

- 4.7.3 Local Search Variants
- 4.8 Population-Based Methods