# Local and Global Search

Parts adapted from:

- Chapter 4 of AI by David Poole and Alan Macworth;
- AI a modern approach by Stuart Russel and Peter Norvig

# Optimisation problems

*Optimisation problem*: given

- a set of variables and their domains; and
- an objective function (aka cost function) that takes a complete assignment of the variables as parameters and returns a real number;

find a complete assignment so that the value of the objective function is optimised (maximised or minimised).

**Example**

Variables: $a$, $b$   $dom(a) = dom(b) = \{-1,0,1\}$

Objective function: $f(a, b) = ab - a$

Maximise $f$.

# Solving an optimisation instance

- A straightforward algorithm is the exhaustive search (generate and evaluate).
  - Good: guarantees to find an optimal solution
  - Bad: for many problems it is intractable
- In this lecture we look at two families of algorithms: local search and global search.
  - Good: can find optimal or near-optimal solutions in reasonable time
  - Bad: do not guarantee optimality (in most cases)

# Local search for optimisation

- A ***local search*** algorithm is an iterative algorithm that keeps a single current state (complete assignment) and in each iteration tries to improve it by moving to one of its neighbouring states.

- Two key aspects to define:
  - Neighbourhood: which states are the neighbours of a given state
  - Movement: which neighbouring state should the algorithm go to

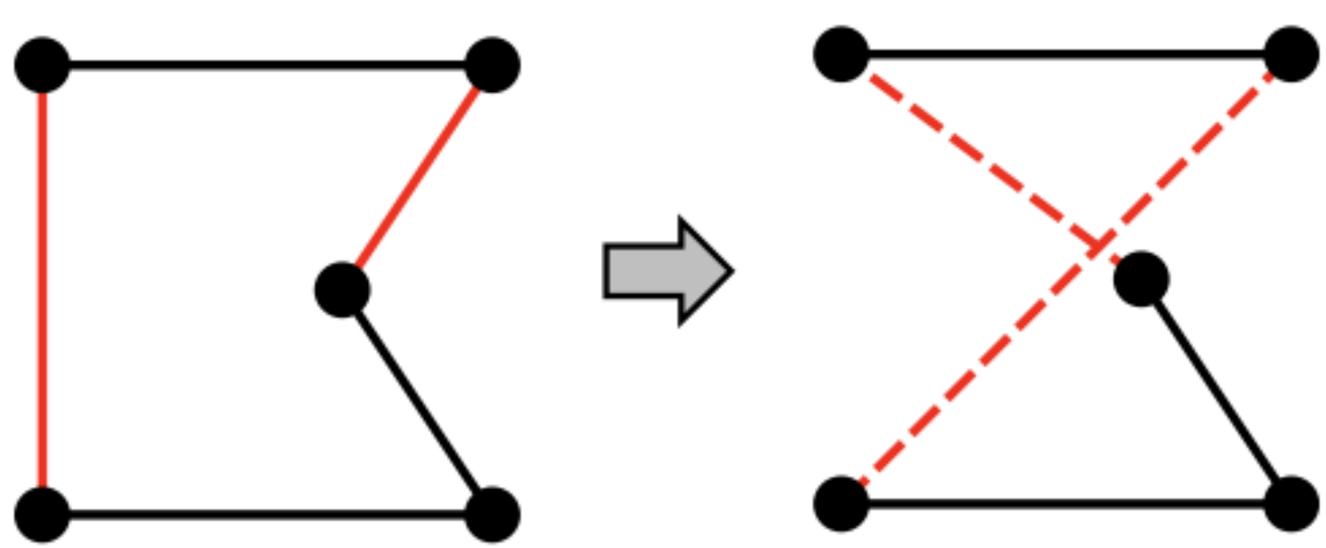
- A search algorithm is considered to be greedy if it always moves to the best neighbour. Two variants:
  - *hill climbing* (or greedy ascent) for maximisation
  - *greedy descent* for minimisation.

# Example: Local Search for TSP

- Traveling Salesperson Problem (TSP): Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

- Start with any complete tour, in each iteration perform pairwise exchanges if it improves the total cost.

- Variants of this approach can get close to optimal solution quickly (even with a large number of cities).



# Local search for CSPs

- A constrained satisfaction problem (CSP) can be reduced to an optimisation problem.
- Given an assignment, a **conflict** is an unsatisfied constraint.
- The goal is to find an assignment that does not produce any conflict (i.e. all the constraints are satisfied).
- Heuristic (or cost or objective) function: the number of conflicts produced by an assignment.
- Optimisation problem: find an assignment that minimises this heuristic function.

# Local Search for CSP: Neighbourhood

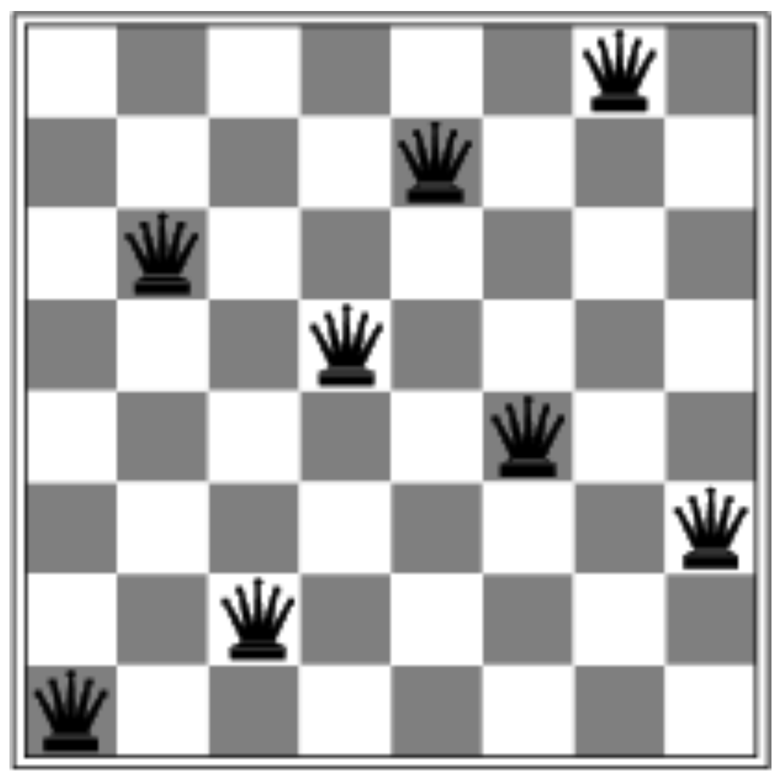
Neighbours of a given state (assignment) can be defined in many ways. Examples:

- All possible assignments except the current one (poor design - why?)
- Select a variable. Neighbours are assignments in which that variable takes a different value from its domain.
- Select a variable that appears in any conflict. Neighbours are assignments in which that variable takes a different value from its domain.
- Select a variable in the current assignment that participates in the most number of conflicts. Neighbours are assignments in which that variable takes a different value from its domain.

# Example: local search for *n*-queens problem

- Aim: Put $n$ queens on an $n \times n$ board with no two queens attacking each other.

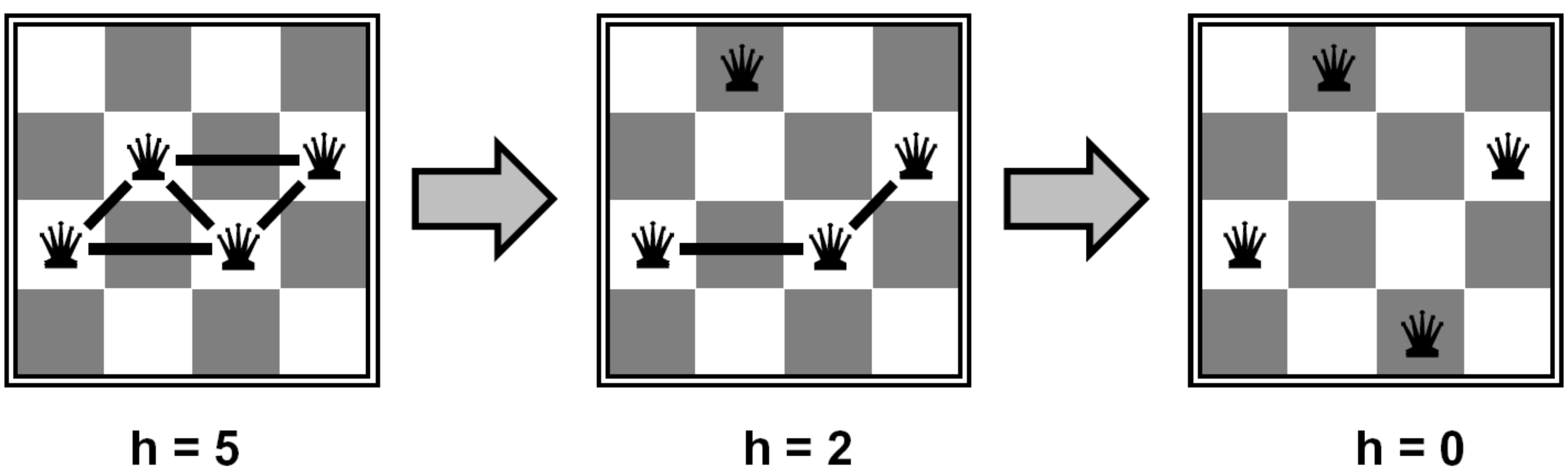
- The objective (heuristic) function to minimise: number of conflicts.



$h = 1$

# Example: 4-Queens

- States: 4 queens in 4 columns ($4^4 = 256$ states)
- Obtaining neighbours: move each queen in its own column
- Objective function to minimise: h(board) = number of pairs of queens that are attacking each other (number of conflicts)



# Example: neighbours

- Objective function (conflict count): number of pairs of queens that are attacking each other.

- Number of conflicts in the current state: 17

# Local Search Issues

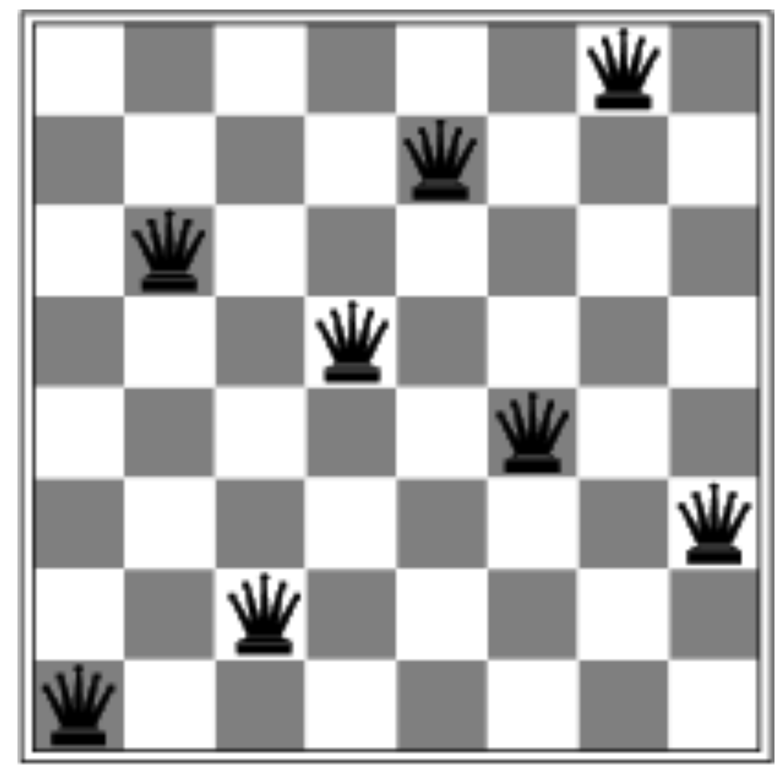
- Local (greedy) search can get stuck in local optima or flat ares of the landscape of the objective function.

- Randomised greedy descent can help sometimes:

  - **random step**: move to a random neighbour.

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

  - these make the search so called ***global***.



**a local minimum with a conflict count of 1.**

# Parallel search

- A total assignment is called an **individual**.
- Idea: maintain a population of *k* individuals instead of one.
- At every stage, update each individual in the population. Whenever an individual is a solution, it can be reported.
- Like *k* restarts, but uses *k* times the minimum number of steps.
- A basic form of global search.

# Simulated Annealing

- Pick a variable at random and a new value at random.
- If it is an improvement, adopt it.
- If it isn't an improvement, adopt it probabilistically depending on a temperature parameter, $T$.
  - With current assignment $n$ and proposed assignment $n'$ we move to $n'$ with probability $e^{(h(n)-h(n'))/T}$
- Temperature can be reduced.

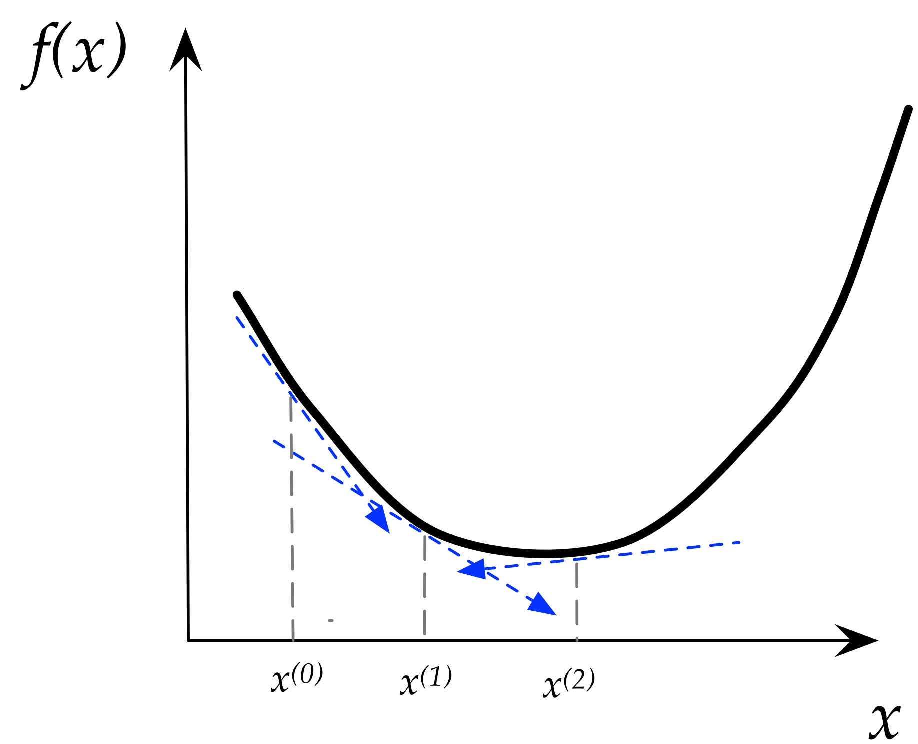
Probability of accepting a change:

| Temperature | 1-worse | 2-worse | 3-worse |
|---|---|---|---|
| 10 | 0.91 | 0.81 | 0.74 |
| 1 | 0.37 | 0.14 | 0.05 |
| 0.25 | 0.02 | 0.0003 | 0.000005 |
| 0.1 | 0.00005 | 0 | 0 |

# Gradient Descent

- A widely-used local search algorithm in numeric optimisation (e.g. in machine learning)
- Used when the variables are numeric and continuous.
- The objective function must be differentiable (mostly).

Guess $\mathbf{x}^{(0)}$, set $k \leftarrow 0$
**while** $||\nabla f(\mathbf{x}^{(k)})|| \geq \epsilon$ **do**
$\quad \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - t_k \nabla f(\mathbf{x}^{(k)})$
$\quad k \leftarrow k + 1$
**end while**
**return** $\mathbf{x}^{(k)}$



# Evolutionary Algorithms

References:

A.E. Eiben and J.E. Smith, Introduction to Evolutionary Computing, Springer

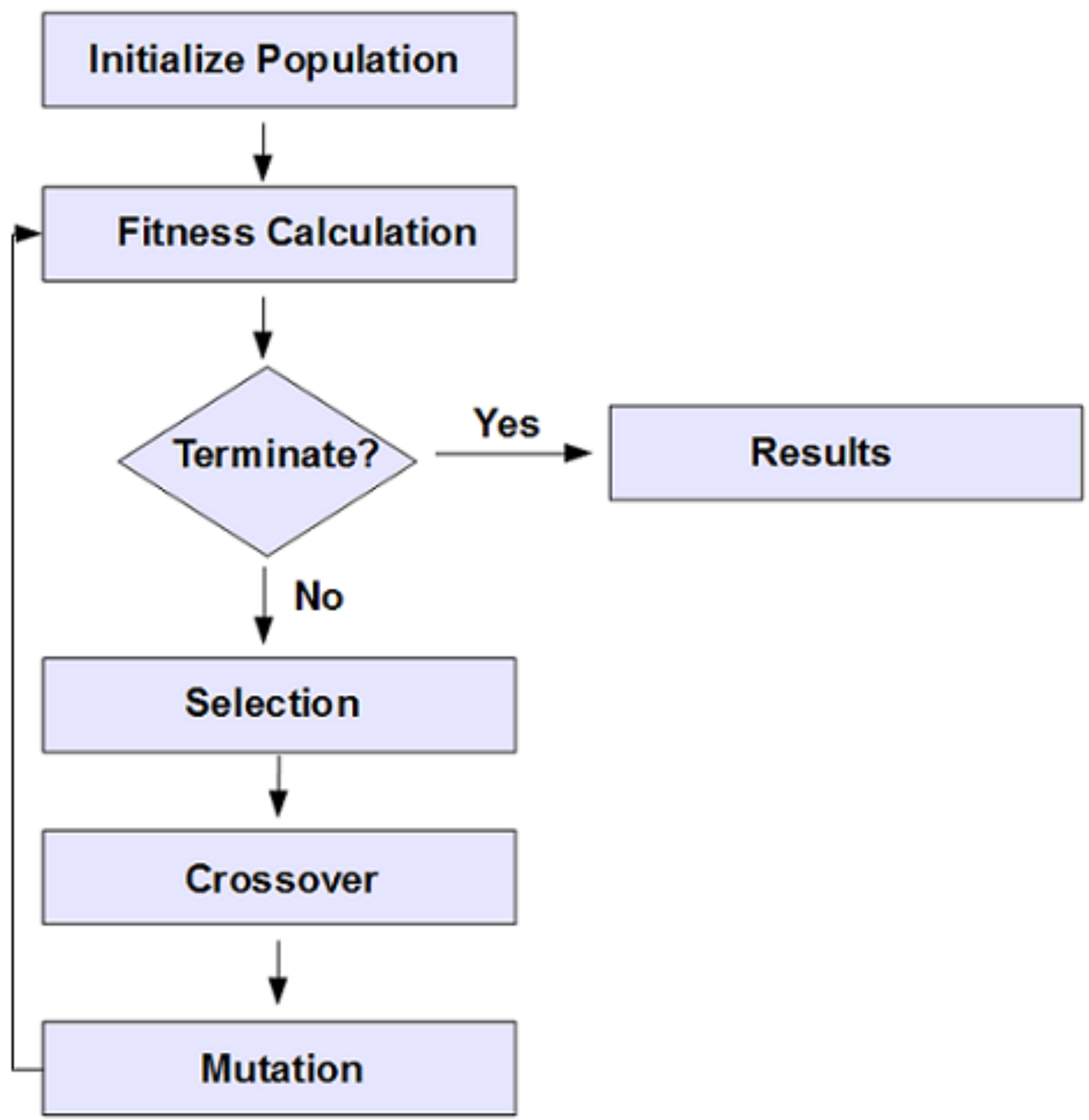
K. A. De Jong, Evolutionary Computation, MIT Press

J. C. Spall
*Introduction to Stochastic Search and Optimization,* John Wiley and Sons

# Genetic (Evolutionary) Algorithms

- Genetic Algorithms (GAs) and the whole family of Evolutionary Algorithms (EAs) are inspired by natural selection.
- They are in the global search family.
- The algorithm maintains a population of *individuals* (aka chromosomes) which evolves over time.
- We can make an individual to represent anything we want (e.g. for CSP it would be an assignment or list of values)
- A *fitness function* is needed. The function takes an individual as input and returns a numeric value indicating how good/bad the individual is.
- A mechanism is needed to create an initial population (usually randomly)
- A mechanism is needed to “evolve” the current generation (population) to the next one. This involves the following mechanisms (also called operators or functions):
  - selection (decide which individuals survive or can reproduce/breed)
  - crossover (given a number of parent individuals, create a number of children)
  - mutation (make some random changes to individuals)

# GA: Flowchart



# Evaluation: Fitness Function

Purpose:

- Parent selection
- Measure for convergence
- For Steady state: Selection of individuals to die
- Should reflect the value of the chromosome (individual)
- It is a critical part of any EA / GA

# Selection

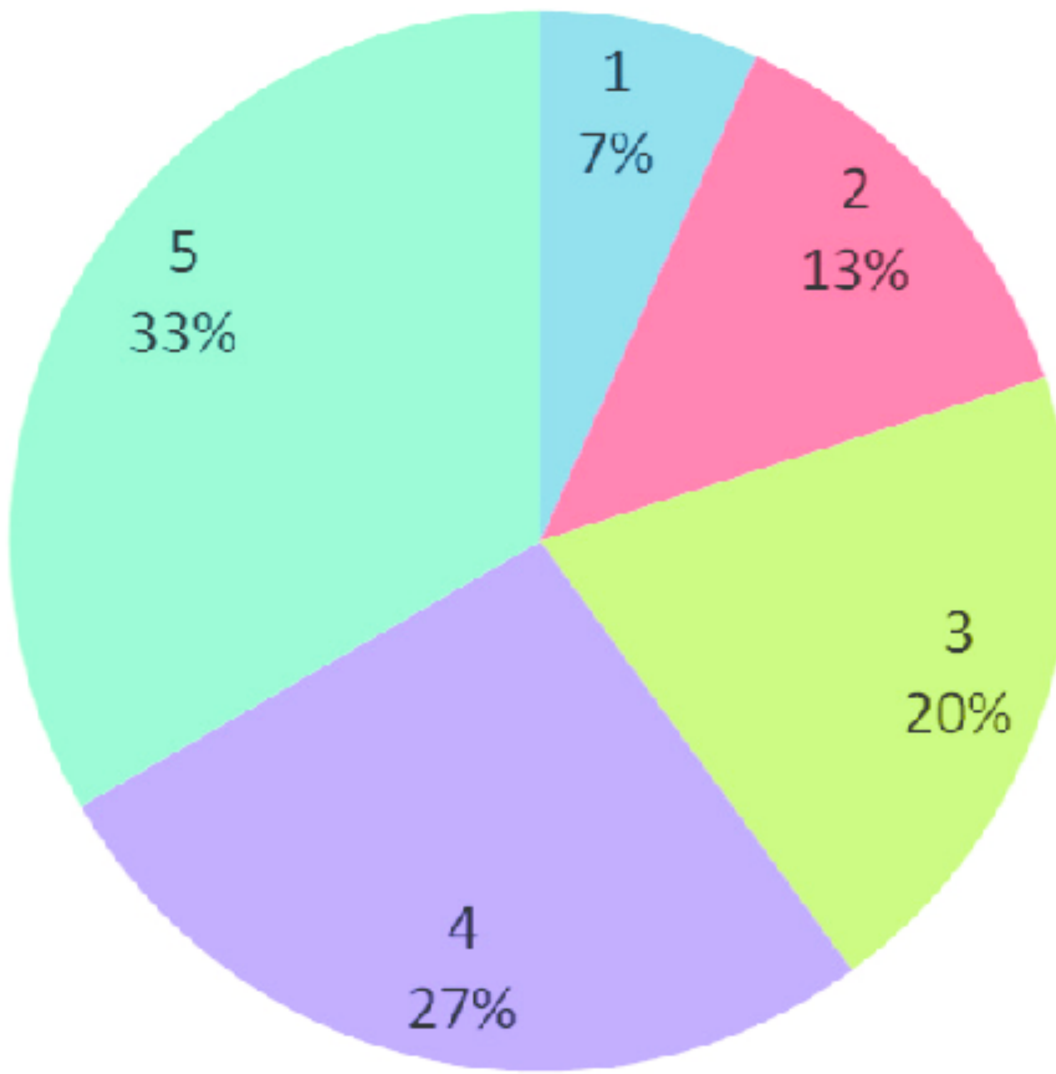
Main idea: better individuals should have higher chance of surviving and breeding.

Types:

- Roulette wheel selection
- Tournament selection
- … any mechanism that somehow overall achieves the main idea.

# Roulette Wheel Selection

- Chances proportional to fitness
- Assign to each individual a part of the roulette wheel
- Spin the wheel *n* times to select *n* individuals




| Individual | Fitness |
|---|---|
| 1 | 1.0 |
| 2 | 2.0 |
| 3 | 3.0 |
| 4 | 4.0 |
| 5 | 5.0 |

# Roulette Wheel Selection: Example

- Sum the fitness of all individuals, call it T
- Generate a random number N between 1 and T
- Return individual whose fitness added to the running total is equal to or larger than N
- Chance to be selected is exactly proportional to fitness
- Individual: 1, 2, 3, 4, 5, 6
- Fitness: 8, 2, 17, 7, 4, 11
- Running total: 8, 10, 27, 34, 38, 49
- N: 23
- Selected: 3

# Selection: Tournaments

- *n* individuals are randomly chosen; the fittest one is selected as a parent.
- *n* is the "size" of the tournament.
- By changing the size, selection pressure can be adjusted.

# Elitism

- Always keep at least one copy of the fittest individual so far
- Results in non-decreasing fitness (of the best individual in the population) over generations
- Widely used in population models

# Reproduction Operators

Selected individuals will be, with different proportions:

- copied to the next generation (unchanged);

- combined with each other (with crossover) to generate child individuals (offsprings) for the next generation; or

- mutated for the next generation.

**Crossover**

- Some portion of the next generation is created by combining selected parents and creating offsprings.
- Usually two parents produce two offspring.
- Typically the probability of crossover (proportion of the next population created by crossover) is between 0.6 and 1.0

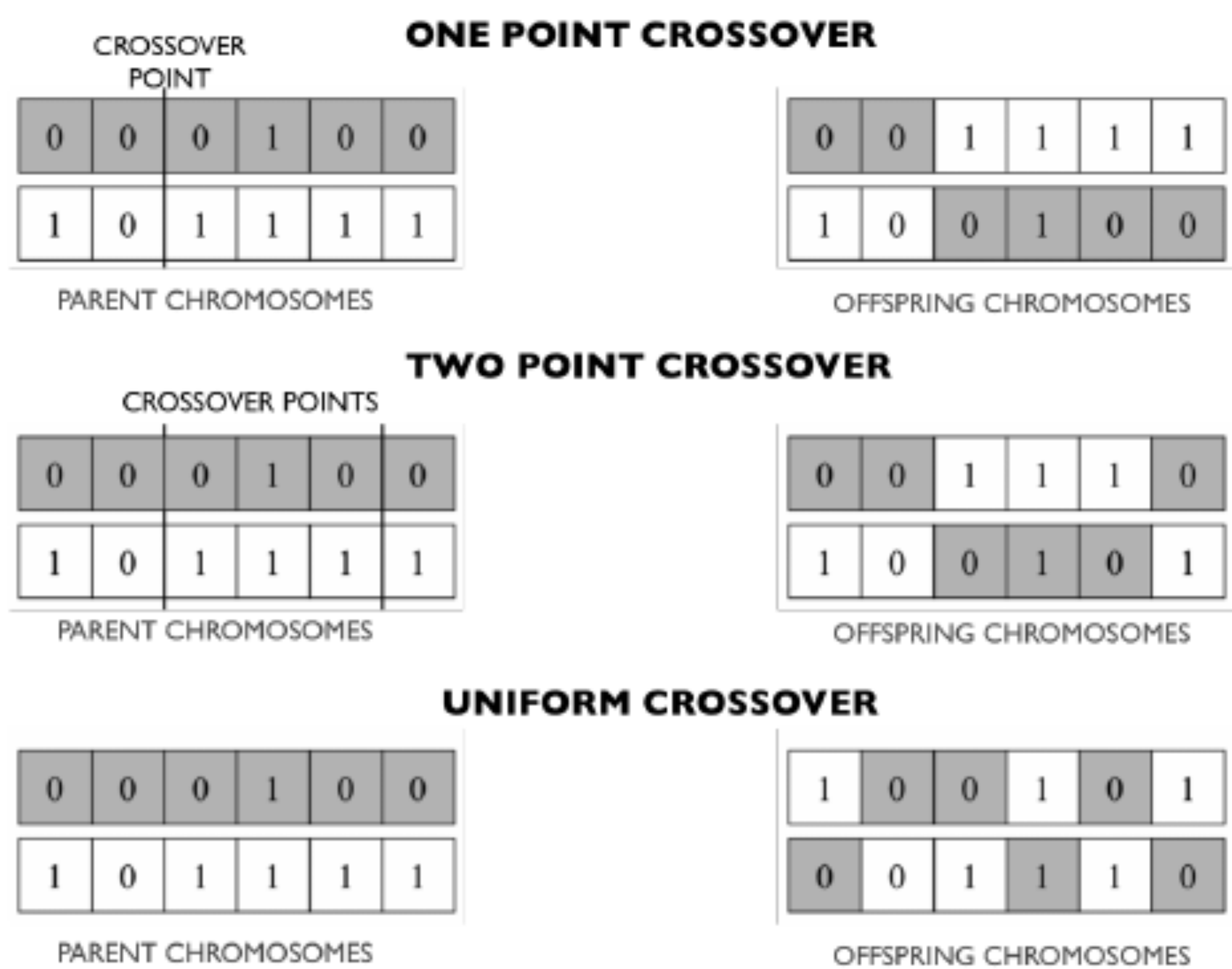
**Mutation**

- Some portion of the next generation is created by mutation.
- In mutation a few “genes“ of an individual is changed randomly (e.g. for CSP the value of a variable changes to another random value in its domain)
- Usually the probability of mutation is low - typically less than 5%

# Crossover

Often individuals are represented as a sequence (tuple) of values (e.g. zeros and ones). With this representation, cross over can be performed very easily:

- Generate 1, 2, or a number of random *crossover points*.
- Split the parents at these points.
- Create offsprings by exchanging alternate segments.



# Mutation

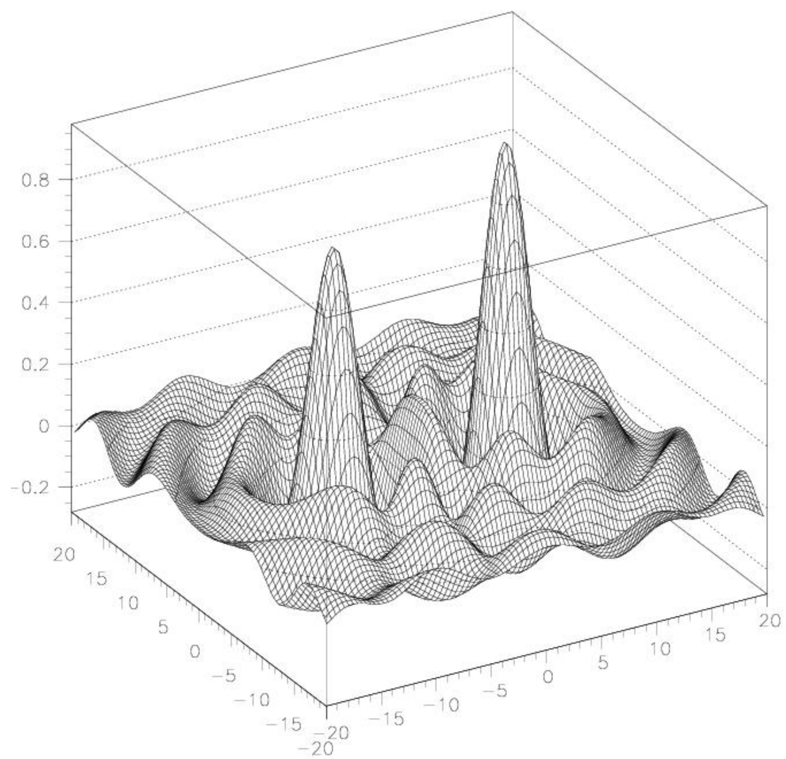
With sequential representation (e.g. tuples), mutation is performed by selecting 1 or more random locations (indices) and changing the values at those locations to some random value (from the domain).

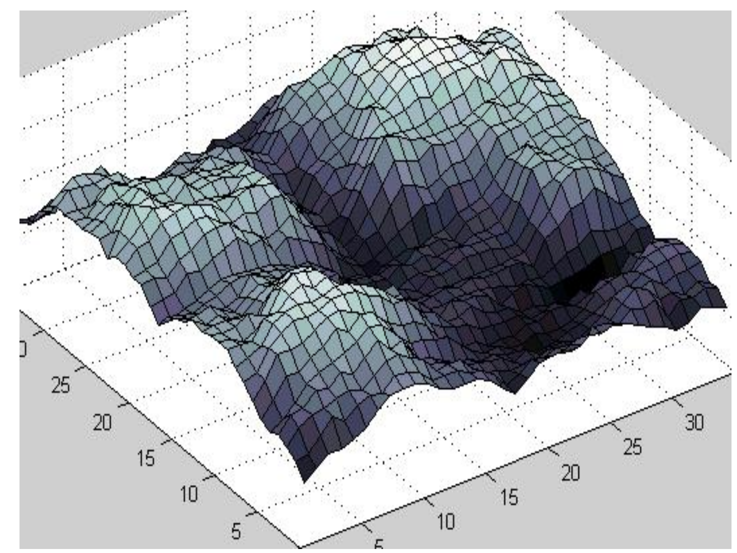
**Mutation vs Crossover**

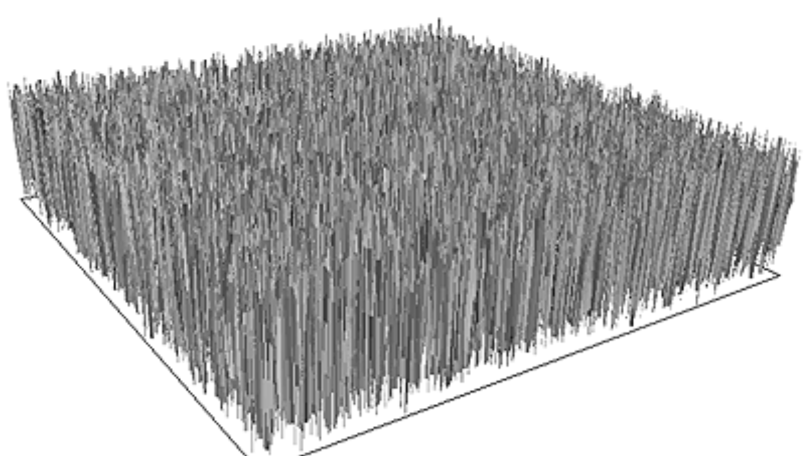
- Purpose of crossover: combining somewhat good candidates in the hope of producing better children
- Purpose of mutation: bring diversity (new “ideas”)
- It is good to have both.
- Mutation-only-EA is possible, crossover-only-EA would not work.

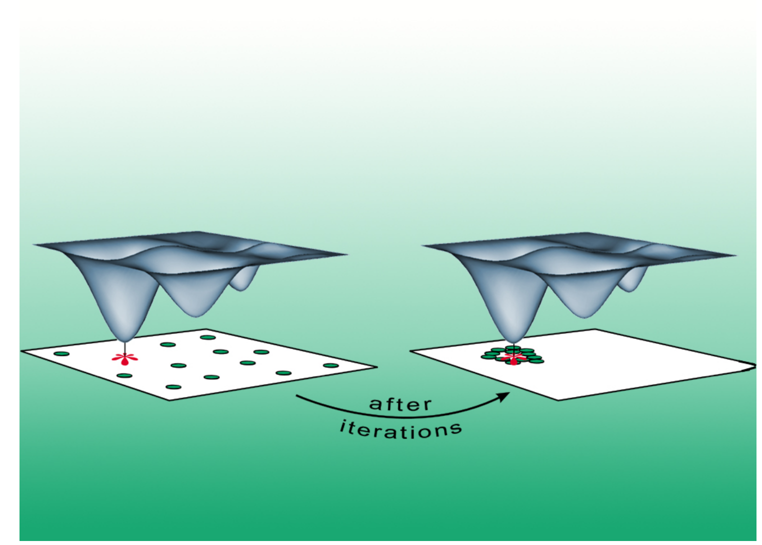
# Fitness landscapes

- EAs are known to be able to handle relatively challenging fitness landscapes.
- Example fitness landscapes where the search space has two continuous variables are shown below.
    - The vertical axis is the value of the objective function.
    - Neighbours of a point are its surrounding points on the 2D plane.



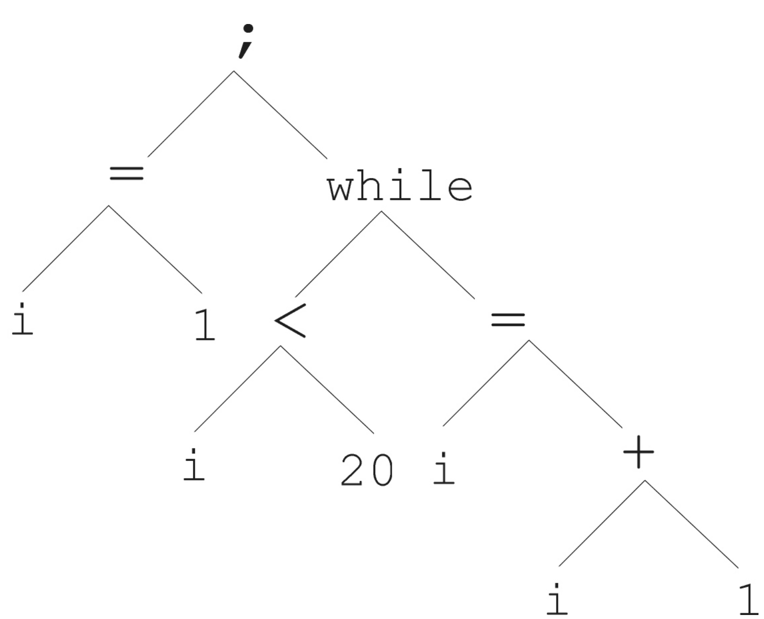



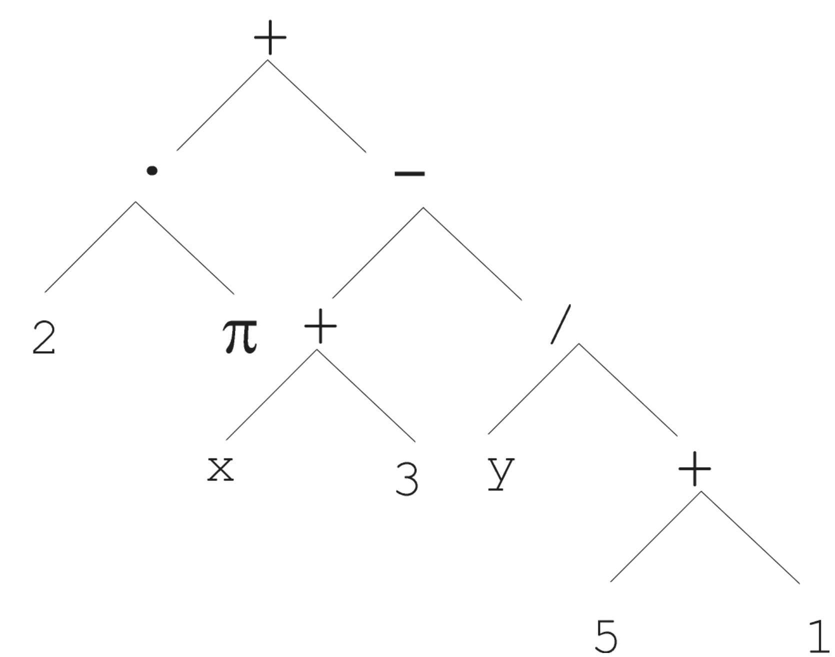





# Tree representation

- Individuals can have more sophisticated structures
- For example when we need to do optimisation (search) in the space of computer programs or expressions, a tree representation can be used. Trees can be represented as nested lists.
- The following shows two example trees representing statements and expressions.



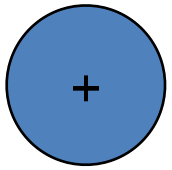




```
i =1;
while (i < 20)
{
        i = i +1
}
```

$$2 \cdot \pi + \left( (x+3) - \frac{y}{5+1} \right)$$

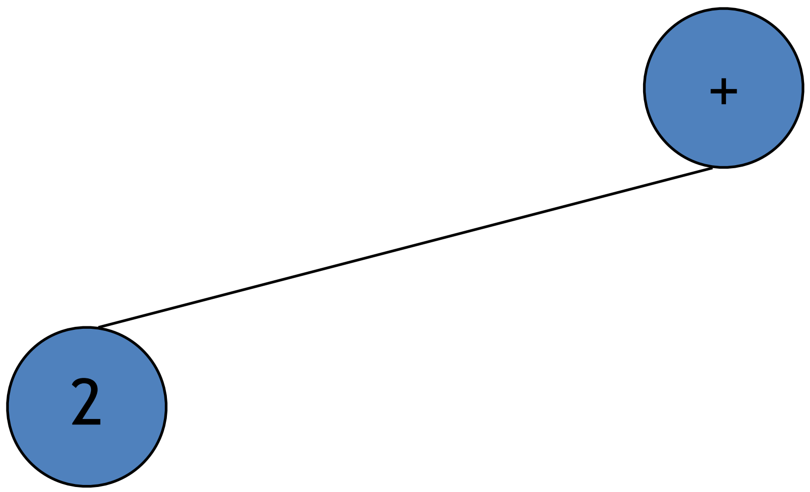
# Randomly Generating Programs

(+ ...)



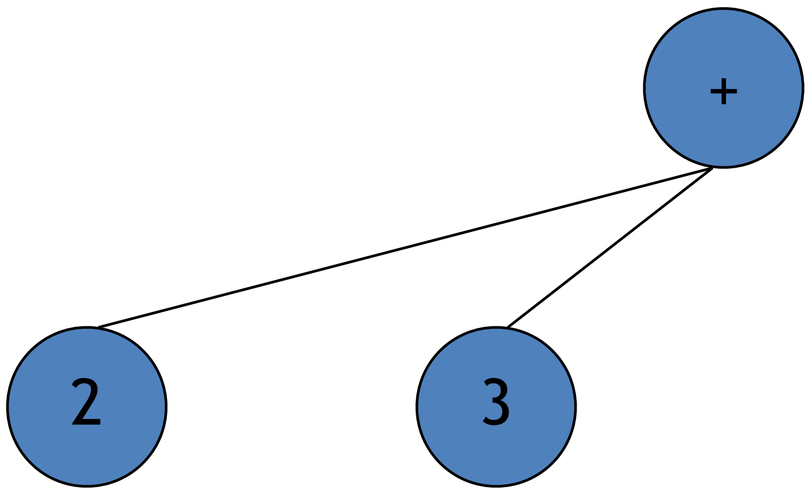
# Randomly Generating Programs

(+ 2 …)



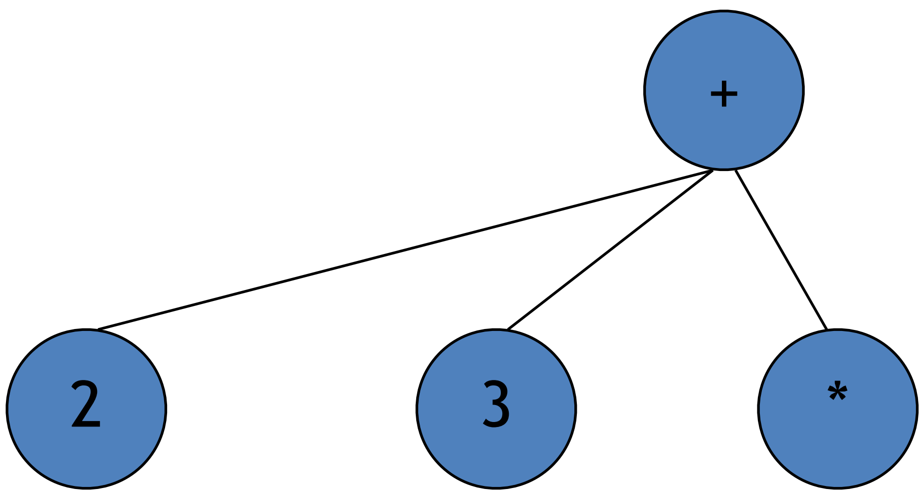
# Randomly Generating Programs

(+ 2 3 ...)



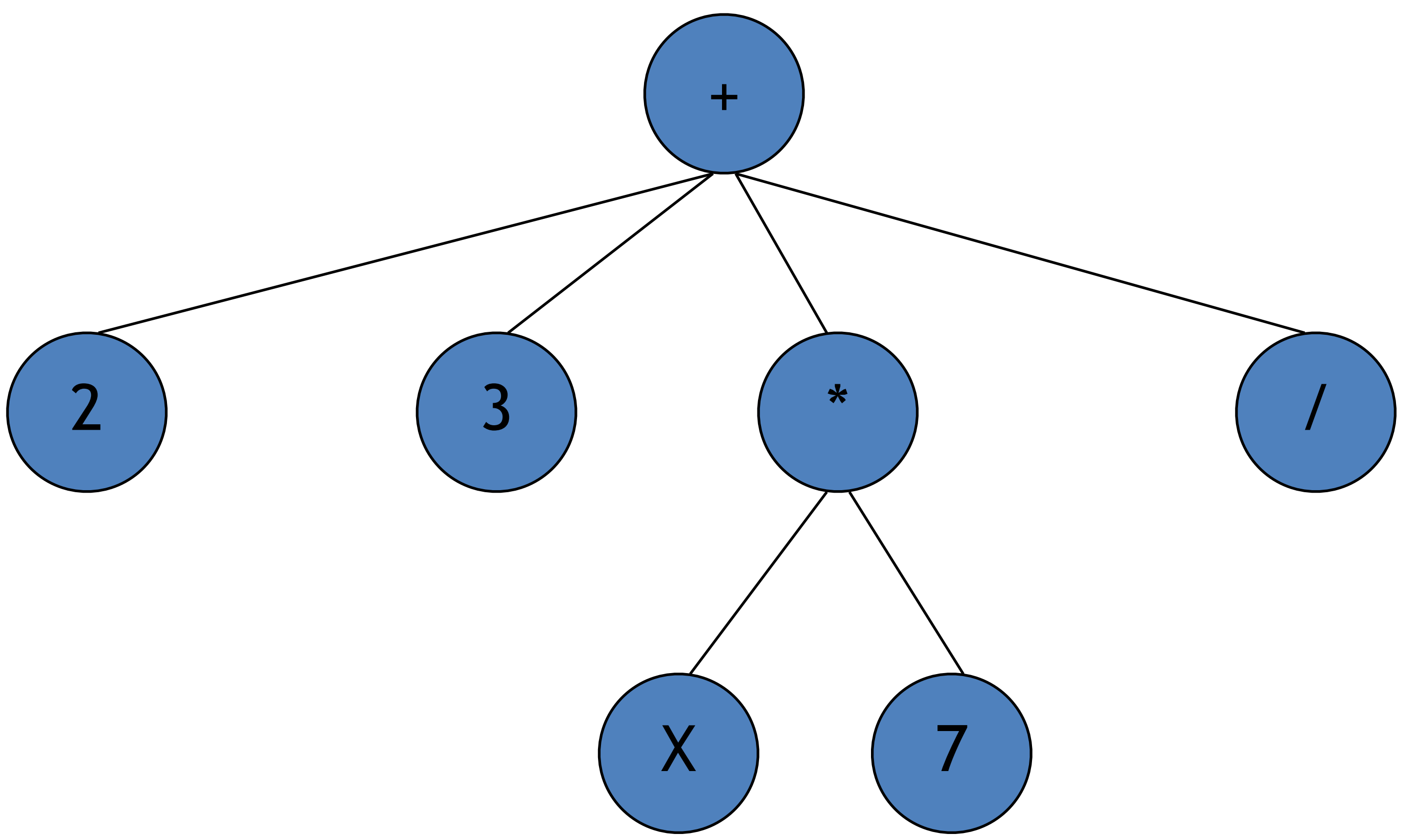
# Randomly Generating Programs

(+ 2 3 (* ...) ...)



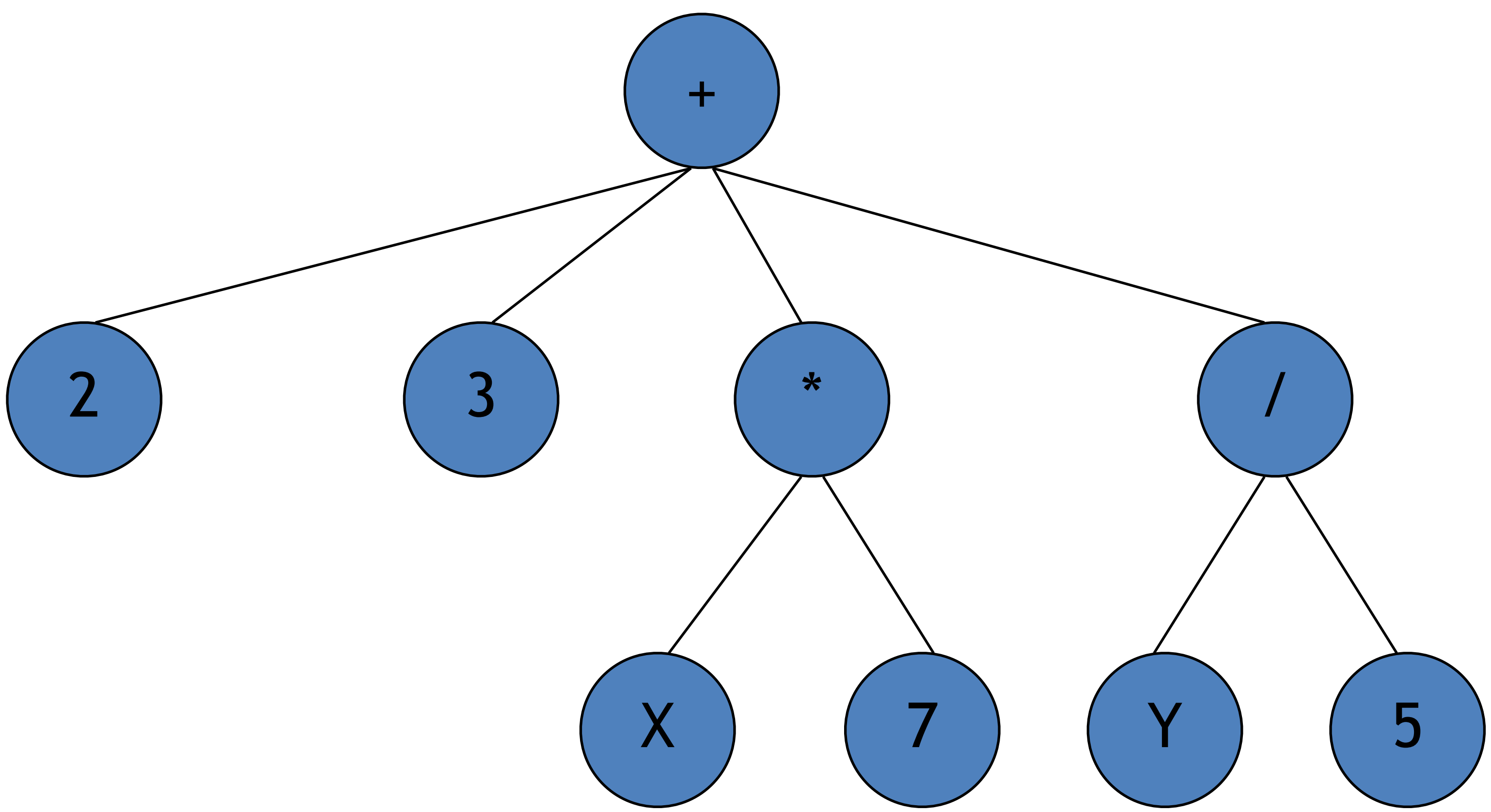
# Randomly Generating Programs

(+ 2 3 (* X 7) (/ …))



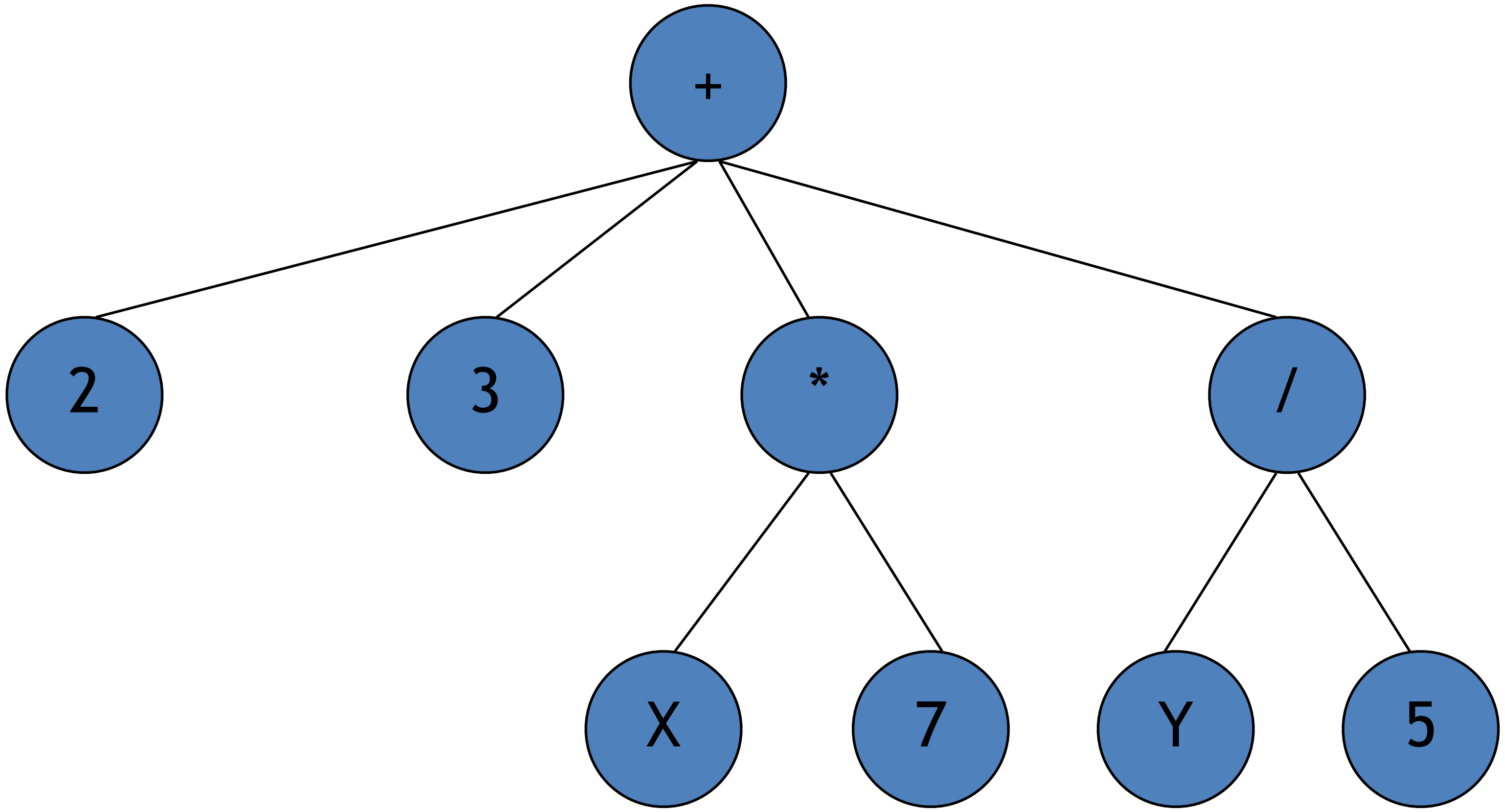
# Randomly Generating Programs

(+ 2 3 (* X 7) (/ Y 5))



# Mutation

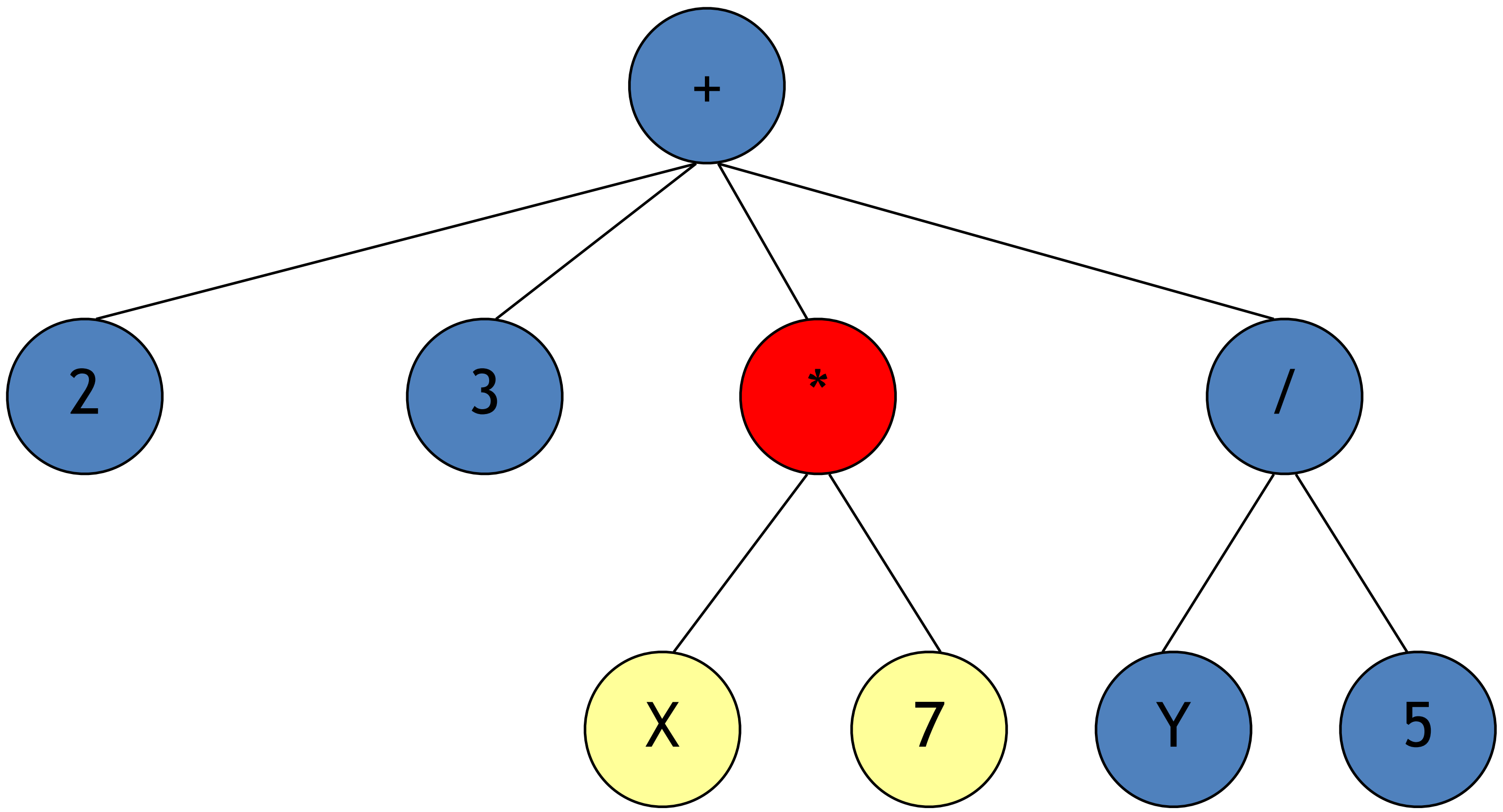
(+ 2 3 (* X 7) (/ Y 5))



# Mutation

First pick a random point (node)

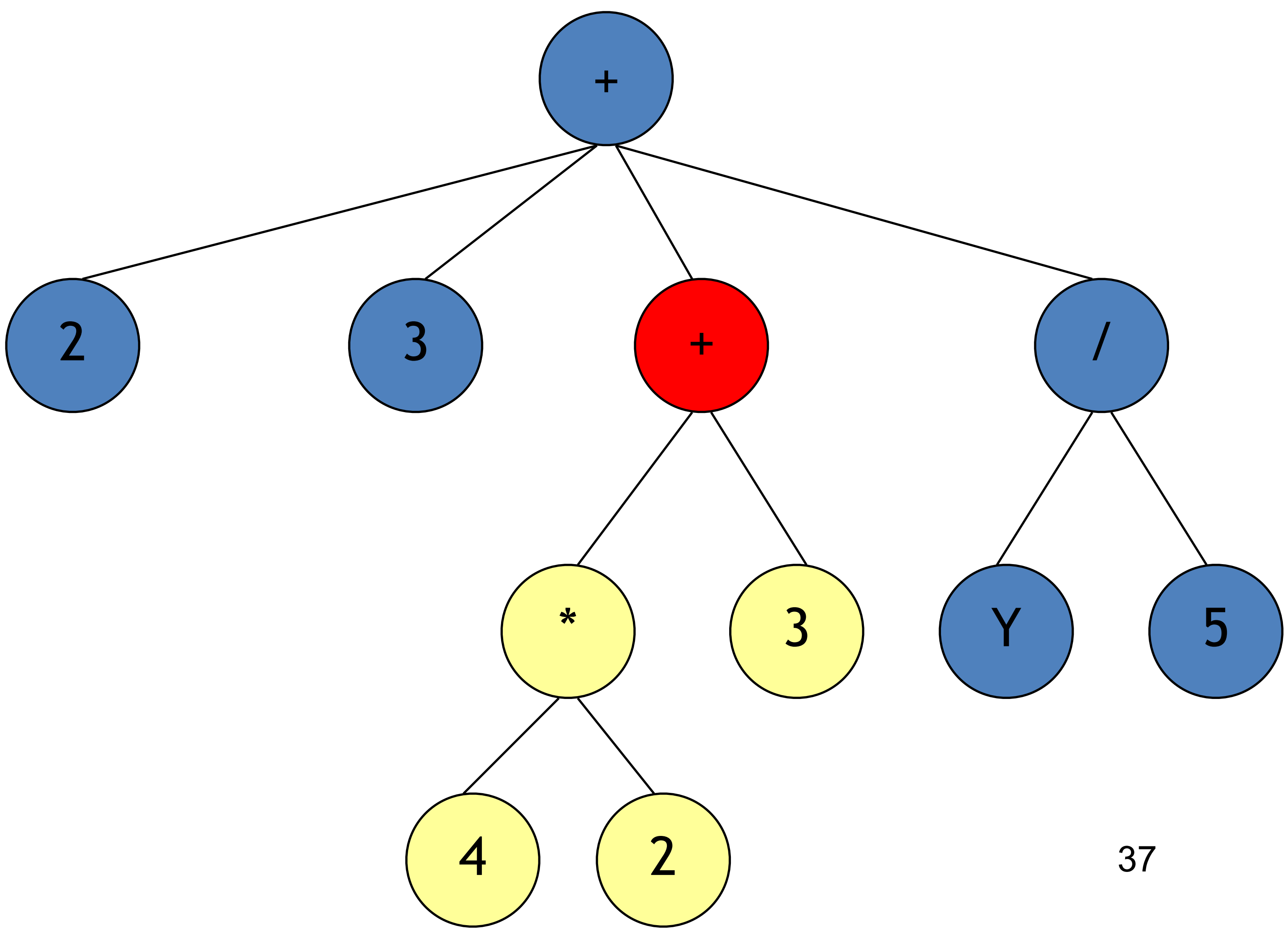
(+ 2 3 (* X 7) (/ Y 5))



# Mutation

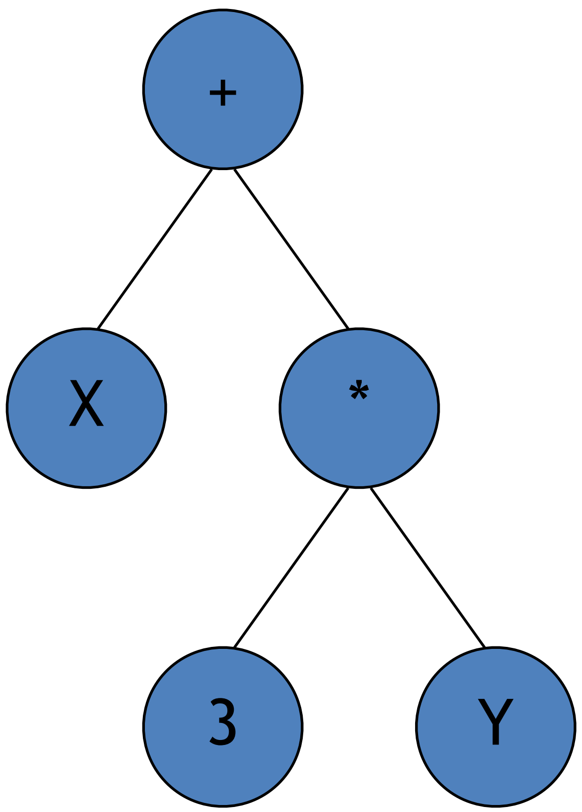
(+ 2 3 (+ (* 4 2) 3) (/ Y 5))

Delete the node and its children, and replace with a randomly generated tree

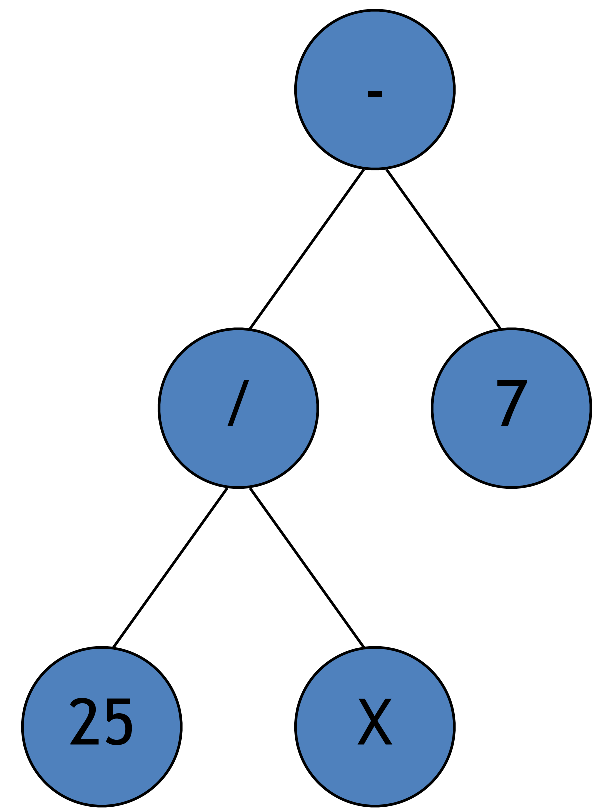


# Crossover

(+ X (* 3 Y))

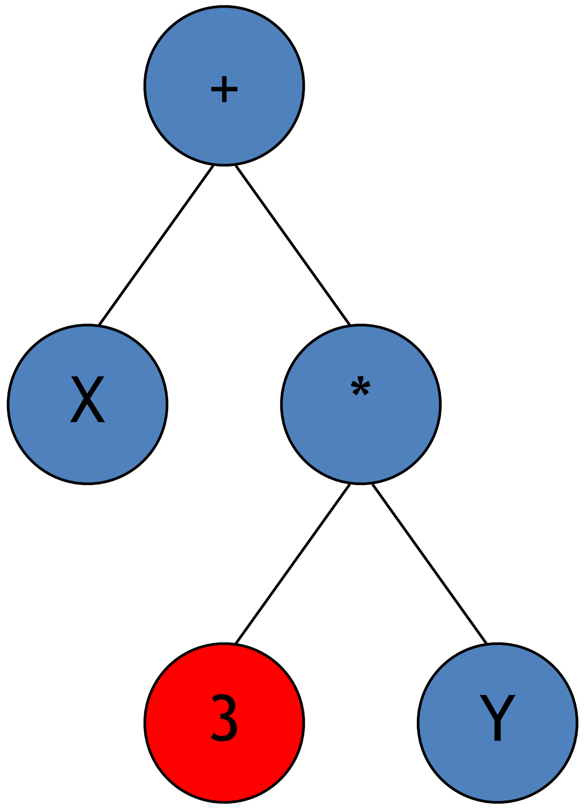



(- (/ 25 X) 7)



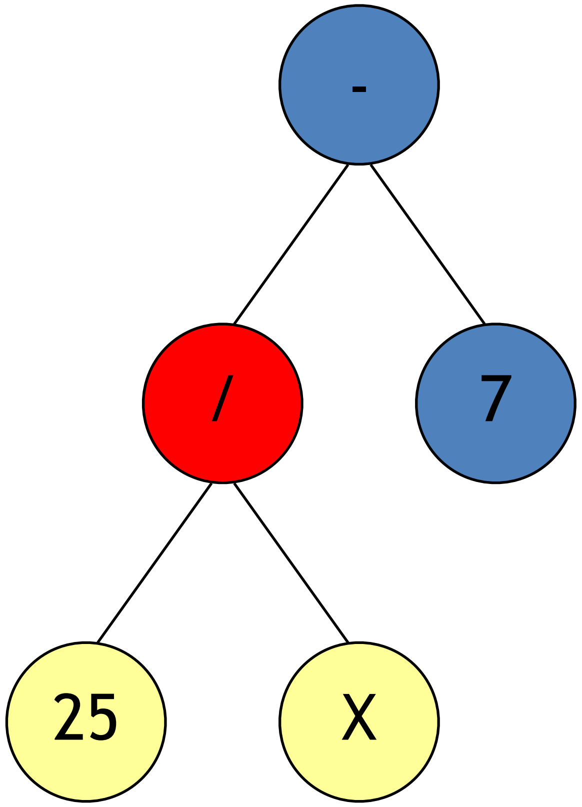
# Crossover

(+ X (* 3 Y))




Pick crossover points (a random node in each tree)

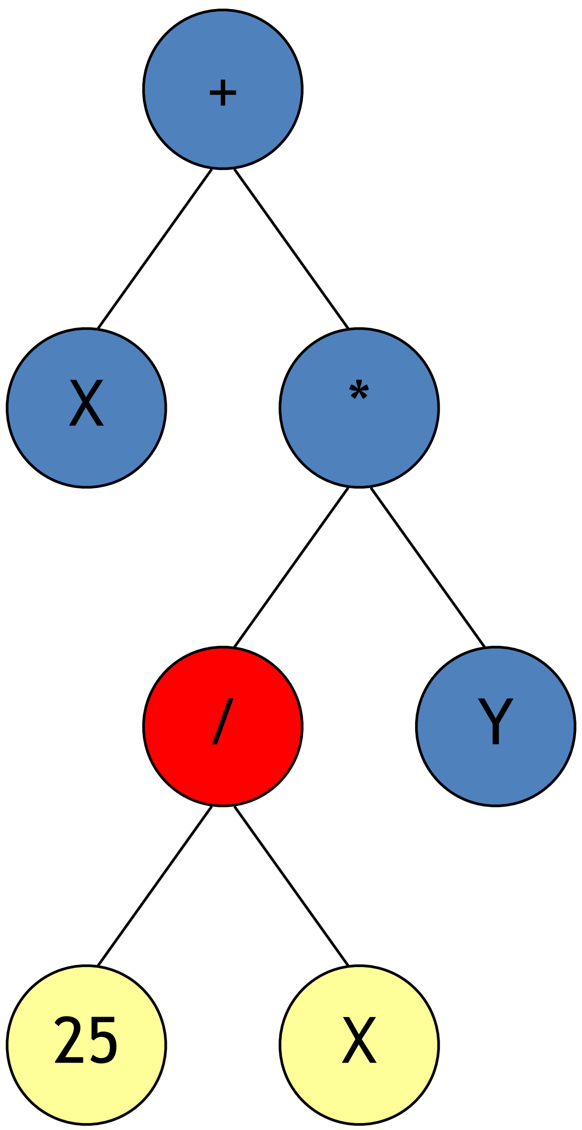
(- (/ 25 X) 7)



# Crossover

(+ X (* (/ 25 X) Y))

(- 3 7)




Swap the two nodes