Prepared by Julian García, based on material by
Arun Konagurthu

Workshop 19
Heuristics: Evolutionary Computation

FIT 3139 Computational Modelling and Simulation

MONASH University
Information Technology

Traveling Salesman

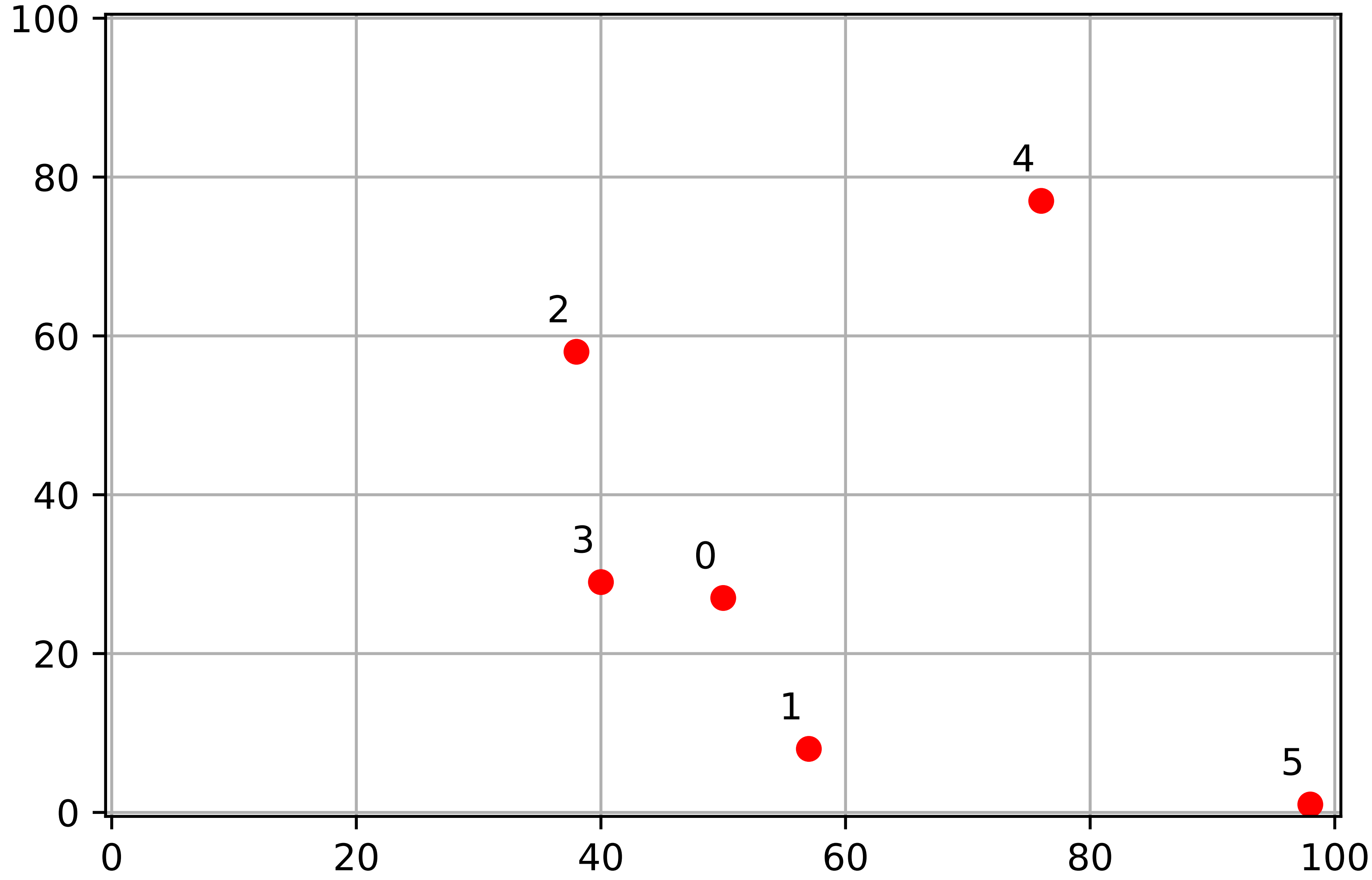
Suppose you are given the following driving distances in kms between the following capital cities.

distance

	Adelaide	Brisbane	Canberra	Darwin	Sydney
Adelaide		2053	1155	3017	1385
Brisbane	2053		1080	3415	939
Canberra	1155	1080		3940	285
Darwin	3017	3415	3940		3975
Sydney	1385	939	285	3975	

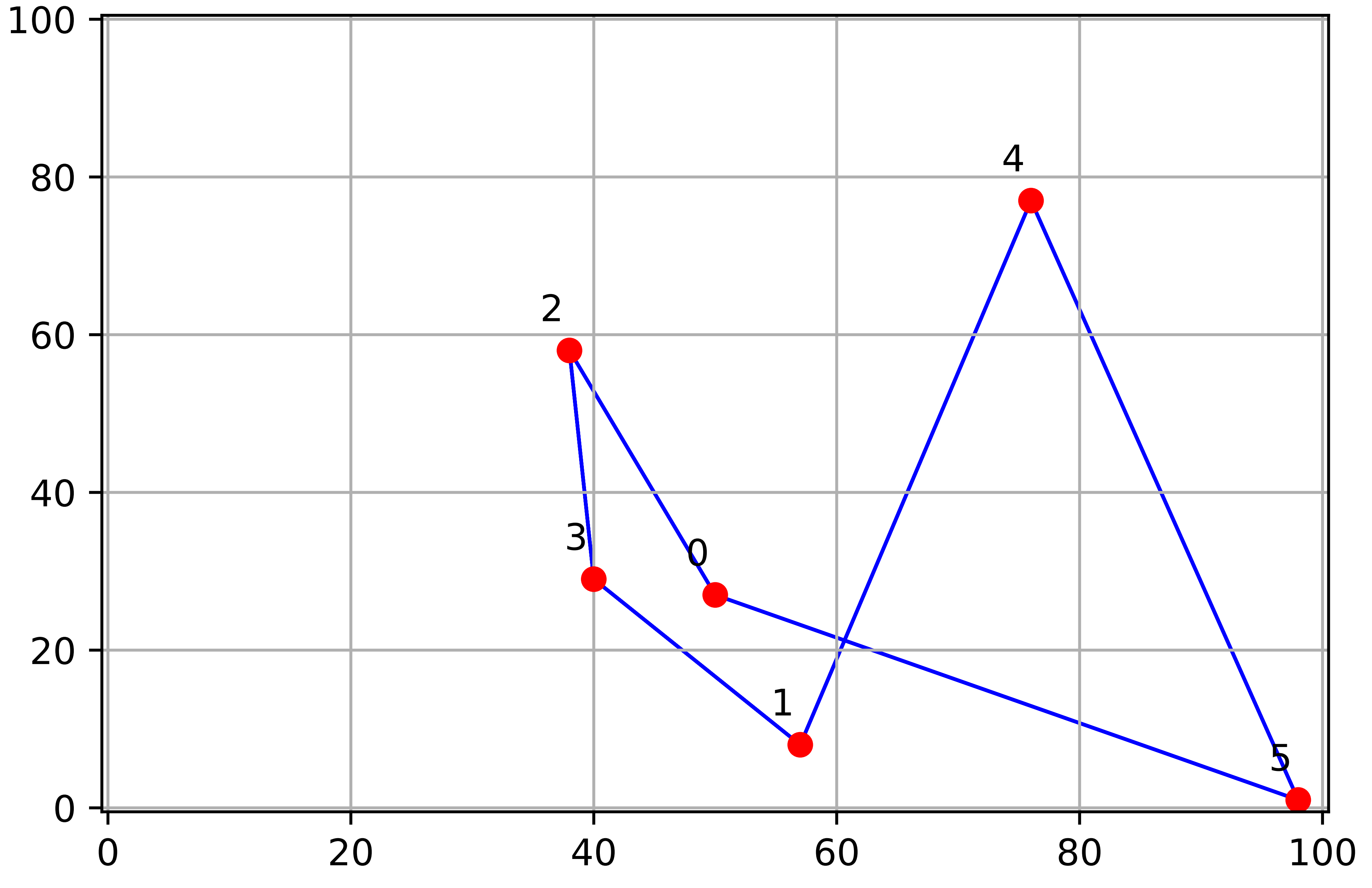
Find the shortest route that enables a salesman to <u>start at Canberra</u>, visit all the other cities, before <u>returning to Canberra.</u>

Euclidian TSP



Instance: collection of n points in $\mathbb{R}^2$

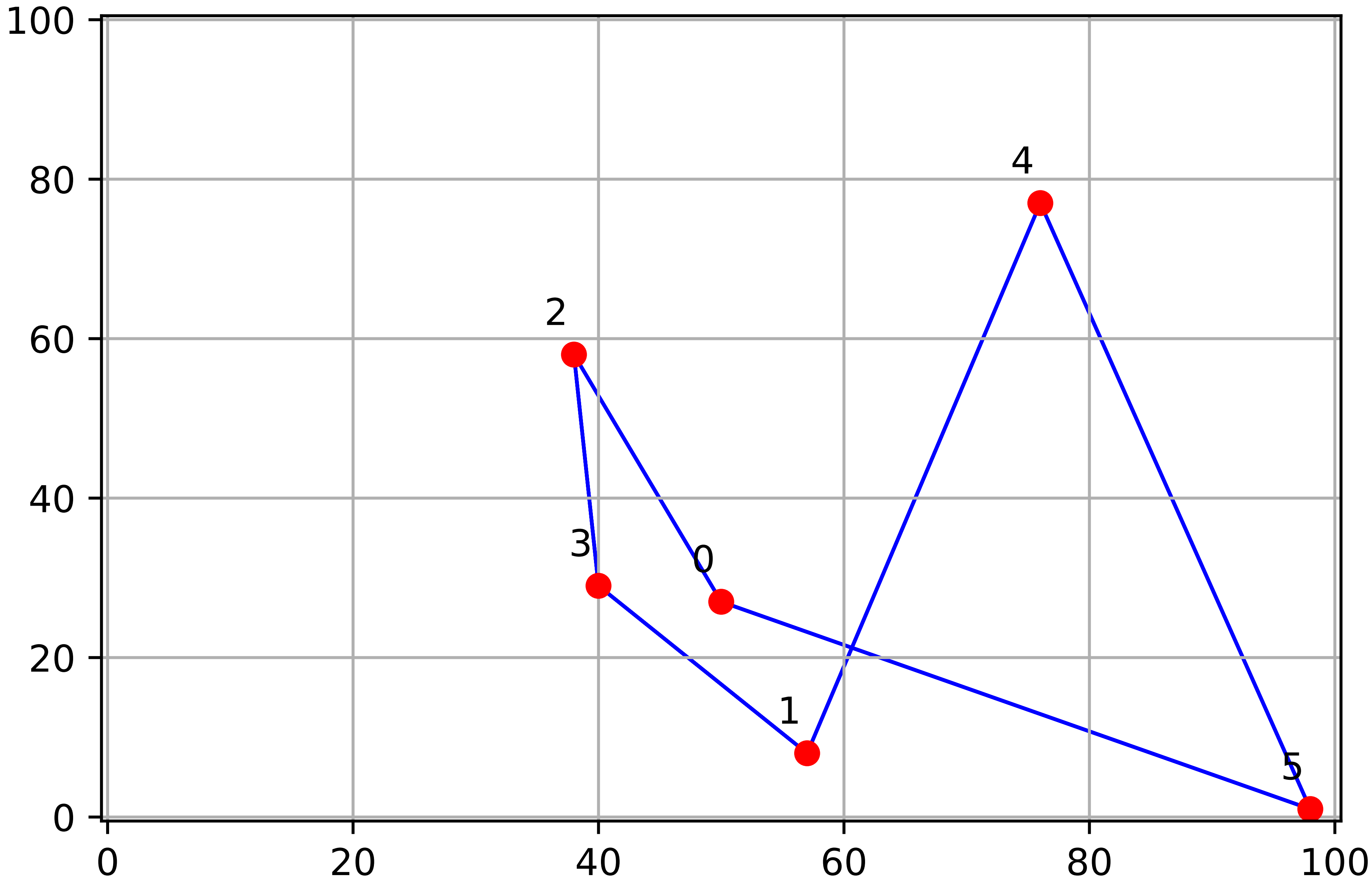
Euclidian TSP



Solution: permutation of the points (start=finish)

$$D = \sum_{i=1}^{n-1} \sqrt{|x_{i+1} - x_i|^2 + |y_{i+1} - y_i|^2} + \sqrt{|x_n - x_1|^2 + |y_n - y_1|^2}$$

~ 294.6



Cost: Total euclidian distance following the permutation

Simulated annealing — Algorithm

For a fixed temperature T:

1. Let x_i be the current solution to the problem.
2. Generate a perturbed solution $\tilde{x}$
3. Decide if $\tilde{x}$ is accepted or rejected.
4. If accepted, update new solution $x_{i+1} = \tilde{x}$, otherwise $x_{i+1} = x_i$
5. Repeat the process for many perturbations.

Decrease T.

```python
def simulated_annealing(function, search_space, perturbations_per_annealing_step, t0, cooling_factor):
    assert t0 > 0
    assert 0 < cooling_factor < 1
    current_solution = np.random.choice(search_space) # start with a random solution
    t = t0
    while abs(t) > 0.001:
        for _ in range(perturbations_per_annealing_step):
            current_value = function(current_solution)

            perturbed_solution = np.random.choice(search_space) #this perturbation can take many forms
            perturbation_value = function(perturbed_solution)

            delta = perturbation_value - current_value

            if delta > 0: # perturbation is better, so take it
                current_solution = perturbed_solution
                current_value = perturbation_value
            elif np.random.rand() < np.exp(delta/t): # perturbation is worse, but I may take it depending on temp
                current_solution = perturbed_solution # go random for large temp, don't for small
                current_value = perturbation_value
        t = cooling_factor*t
    return  current_solution, function(current_solution)
```

TSP with SA What differs?

- Search space: “Tours” — permutations of n points.
- Neighbourhood function — necessary for exploration.
- Minimise cost (instead of maximising value).

- Neighbourhood function — necessary for exploration.

Example:

[3, 1, 4, 5, 0, 2] ⟶ [3, 2, 4, 5, 0, 1]

```
def perturb(tour):
    # choose two cities at random
    i, j = np.random.choice(len(tour), 2, replace=False)
    new_tour = np.copy(tour)
    # swap them
    new_tour[i], new_tour[j] =  new_tour[j], new_tour[i]
    return new_tour
```

Metropolis criterion for <u>maximising</u> $f(x)$

Accept with probability

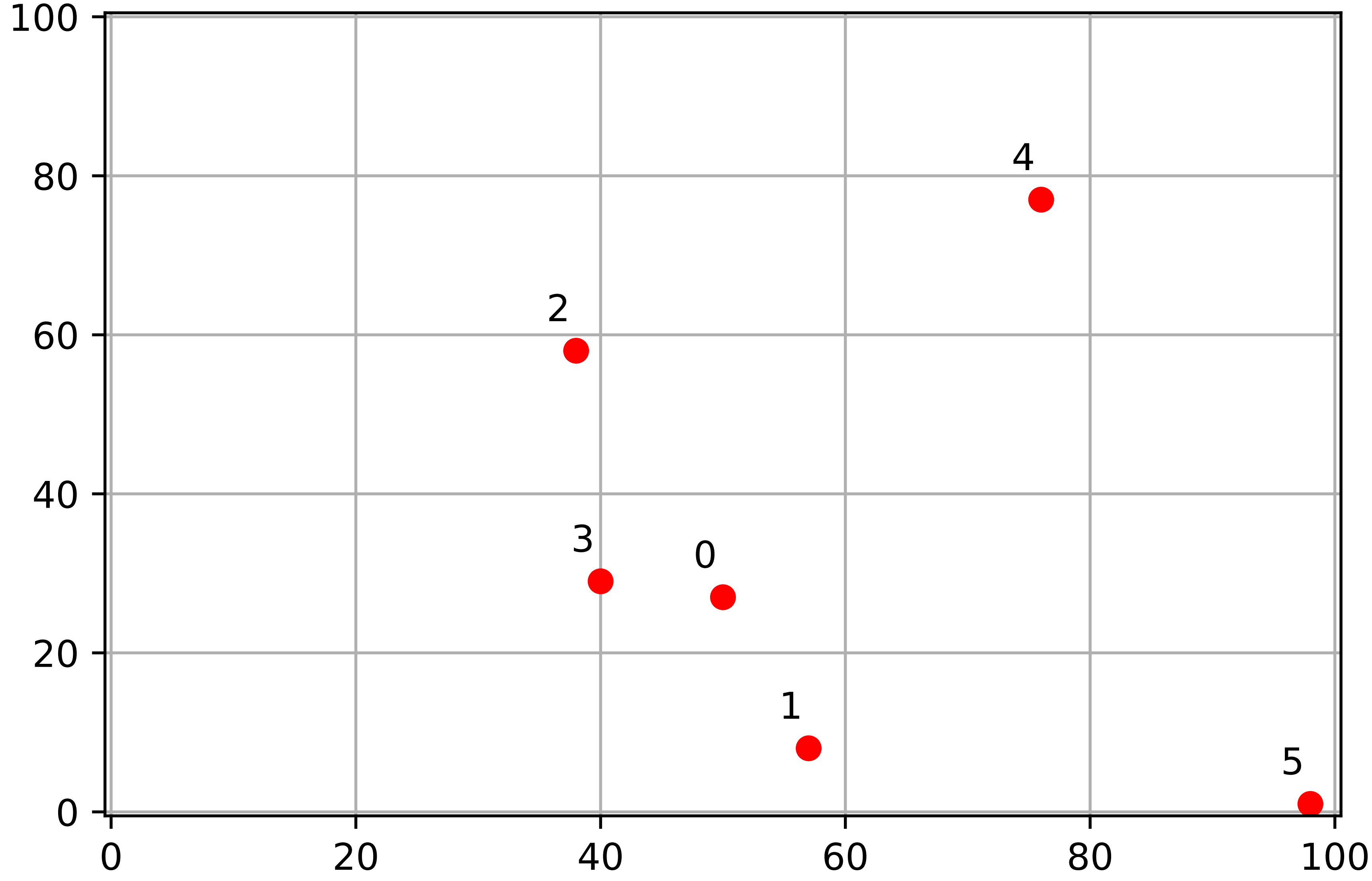
$$\begin{cases} 1 & \text{if} \quad f(\tilde{x}) > f(x_i) \\ \exp\left(\dfrac{f(\tilde{x}) - f(x_i)}{T}\right) & \text{otherwise} \end{cases}$$

Metropolis criterion for <u>minimising</u> $f(x)$

Accept with probability

$$\begin{cases} 1 & \text{if} \quad f(\tilde{x}) \boxed{<} f(x_i) \\ \exp\left(\boxed{-}\dfrac{f(\tilde{x}) - f(x_i)}{T}\right) & \text{otherwise} \end{cases}$$

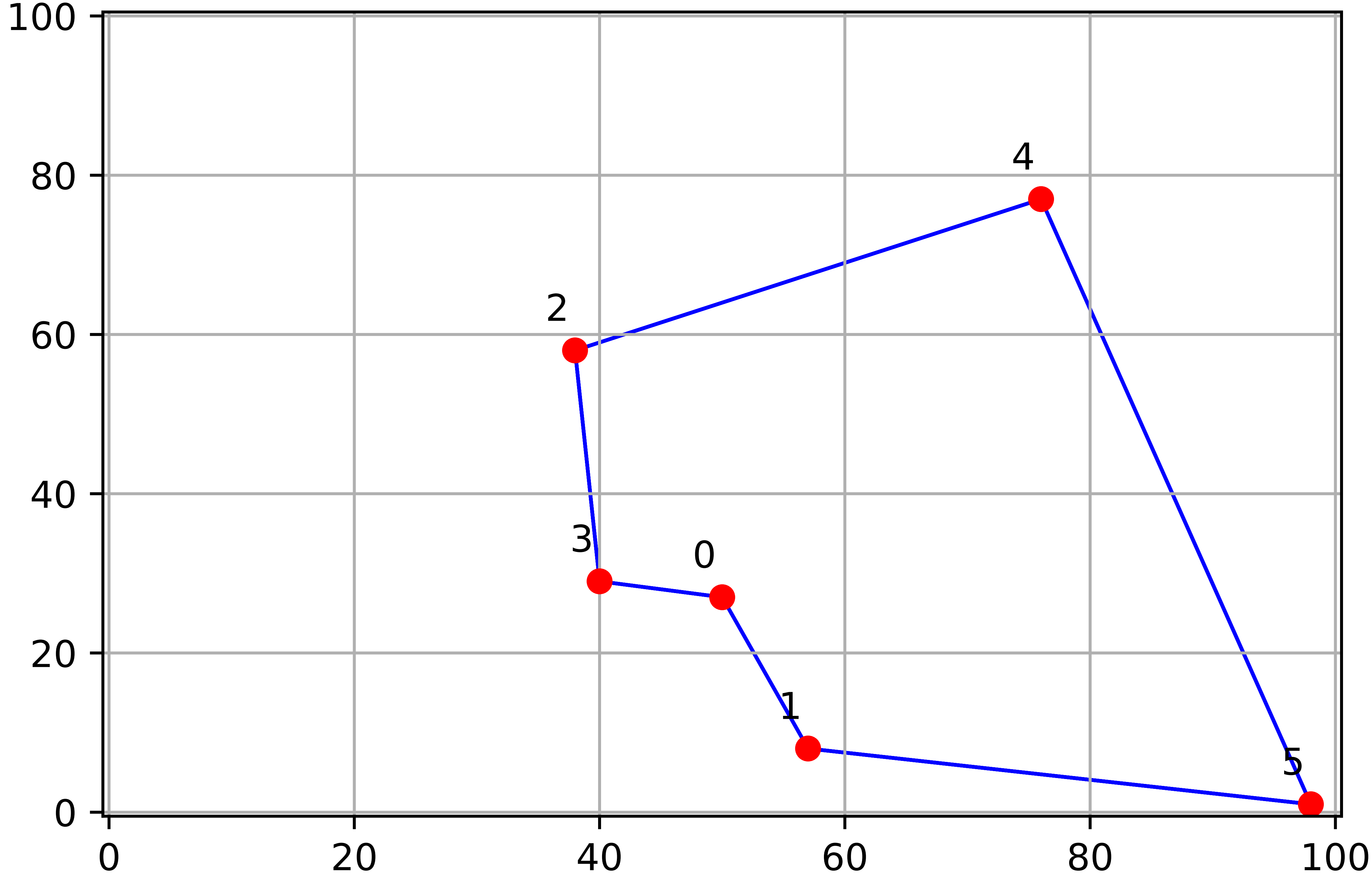
Instance: collection of n points in $\mathbb{R}^2$



```python
def SA_TSP(tsp_instance, perturbations_per_annealing_sep, t0,
           cooling_factor):
    number_of_cities = len(tsp_instance)
    current_solution = np.random.permutation(number_of_cities)
    t = t0
    while t > 0.001:
        for _ in range(perturbations_per_annealing_sep):
            current_value = cost(current_solution, tsp_instance)

            perturbation = perturb(current_solution)
            perturbation_value = cost(perturbation, tsp_instance)

            delta = perturbation_value - current_value
            if delta < 0:
                current_solution = perturbation
                current_value = perturbation_value
            elif np.random.rand() < np.exp(-delta/t):
                current_solution = perturbation
                current_value = perturbation_value
        t = cooling_factor*t
    return current_solution, cost(current_solution, tsp_instance)
```



perturbations_per_annealing_sep=100, t0=100, cooling_factor=0.95

Outline:

- Simulated Annealing (Metropolis)
- Evolutionary Computation (Genetic Algorithms)

Both techniques are *Heuristics*

“Natural computation”

inspired in natural processes.

Natural Computation

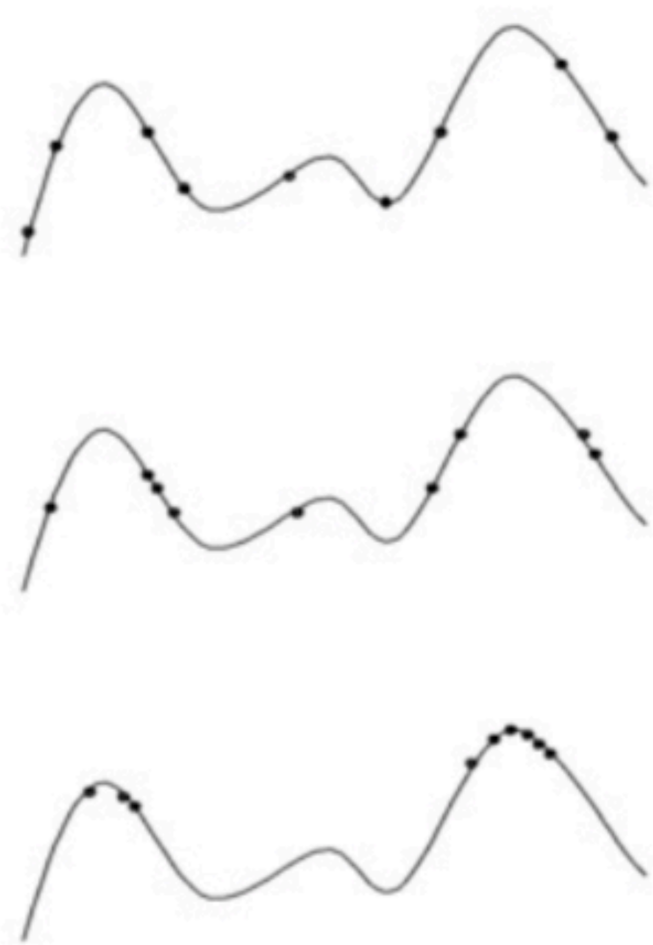
Simulated annealing $\longrightarrow$ Physics

Evolutionary Computation $\longrightarrow$ Evolutionary Biology

(simulated natural selection)

EC metaphor

- A population of individuals exists in an environment with limited resources
- **Competition** for those resources causes **selection** of those fitter individuals that are better adapted to the environment
- These individuals act as seeds for the generation of new individuals through **recombination and mutation**
- The new individuals have their **fitness** evaluated and compete (possibly also with parents) for **survival**.
- Over time Natural selection causes a rise in the fitness of the population



Early phase:
quasi-random population distribution

Mid-phase:
population arranged around/on hills

Late phase:
population concentrated on high hills

An Evolutionary Algorithm

```
begin
    t ← 0
    initialize P(t)
    evaluate P(t)
    while (not termination-condition) do
    begin
        t ← t + 1
        select P(t) from P(t − 1)
        alter P(t)
        evaluate P(t)
    end
end
```

Michalewicz and Fogel. Modern Heuristics. Chapter 6.

An Evolutionary Algorithm

```
begin
  t ← 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
  begin
    t ← t + 1
    select P(t) from P(t − 1)
    alter P(t)
    evaluate P(t)
  end
end
```

A *population* of solutions

A *fitness function*

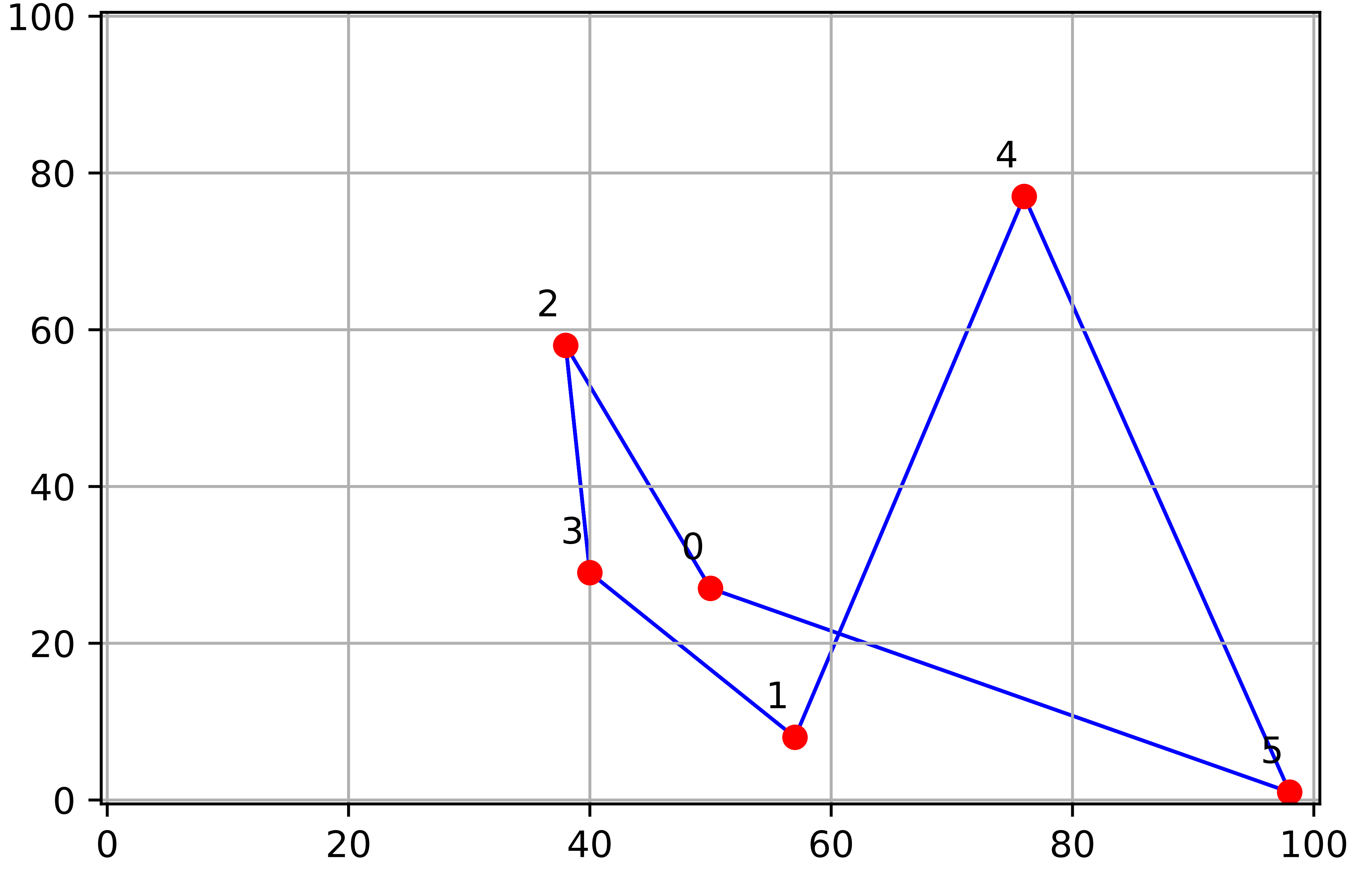
Individuals with large fitness
<u>more likely</u> to make it to the *next generation*

Variation operators:

- Mutation
- Crossover

Michalewicz and Fogel. Modern Heuristics. Chapter 6.

Euclidian TSP



Solution: permutation of the points (start=finish)

An Evolutionary Algorithm

```
begin
  t ← 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
  begin
    t ← t + 1
    select P(t) from P(t − 1)
    alter P(t)
    evaluate P(t)
  end
end
```

A *population* of solutions

A fitness function

Individuals with large fitness
more likely to make it to the *next generation*

Variation operators:

- Mutation
- Crossover

Michalewicz and Fogel. Modern Heuristics. Chapter 6.

```python
def initialise_population(number_of_cities, population_size):
    pop = []
    for _ in range(population_size):
        pop.append(np.random.permutation(number_of_cities))
    return pop
```

An Evolutionary Algorithm

```
begin
  t ← 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
  begin
    t ← t + 1
    select P(t) from P(t − 1)
    alter P(t)
    evaluate P(t)
  end
end
```

A *population* of solutions

A fitness function

Individuals with large fitness
<u>more likely</u> to make it to the *next generation*

Variation operators:

- Mutation
- Crossover

Michalewicz and Fogel. Modern Heuristics. Chapter 6.

```python
def distance(city1, city2):
    x_distance = abs(city1[0] - city2[0])
    y_distance = abs(city1[1] - city2[1])
    return np.sqrt(x_distance**2 + y_distance**2)
```

```python
def cost(tour, tsp_instance):
    cost = 0
    for i in range(0, len(tour)-1):
        cost += distance(tsp_instance[tour[i]], tsp_instance[tour[i+1]])
    cost += distance(tsp_instance[tour[-1]], tsp_instance[tour[0]])
    return cost
```

$$F_i = \frac{1}{c_i} \longrightarrow \text{Small cost, large fitness}$$

(one of many ways to do it)

An Evolutionary Algorithm

```
begin
  t ← 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
  begin
    t ← t + 1
    select P(t) from P(t − 1)
    alter P(t)
    evaluate P(t)
  end
end
```

A *population* of solutions

A *fitness function*

Individuals with large fitness
more likely to make it to the *next generation*

Variation operators:

- Mutation
- Crossover

Michalewicz and Fogel. Modern Heuristics. Chapter 6.

Roulette Wheel Selection

$$F_i = \frac{1}{c_i} \longrightarrow$$ Fitness of i

$$p_i = \frac{F_i}{\sum_{j=1}^{n} Fj} \longrightarrow$$ Probability that i will be part of the next generation

Individuals with large fitness
more likely to make it to the *next generation*

Alternative

Rank based selection: rank according to fitness, weight in the probability distribution depends on ranking, not fitness

An Evolutionary Algorithm

```
begin
  t ← 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
  begin
    t ← t + 1
    select P(t) from P(t – 1)
    alter P(t)
    evaluate P(t)
  end
end
```

A *population* of solutions

A *fitness function*

Individuals with large fitness
more likely to make it to the *next generation*

Variation operators:

- **Mutation**
- **Crossover**

Michalewicz and Fogel. Modern Heuristics. Chapter 6.

Mutation

[3, 1, 4, 5, 0, 2] → [3, 2, 4, 5, 0, 1]

```python
def perturb(tour):
    # choose two cities at random
    i, j = np.random.choice(len(tour), 2, replace=False)
    new_tour = np.copy(tour)
    # swap them
    new_tour[i], new_tour[j] =  new_tour[j], new_tour[i]
    return new_tour
```

Mutation = small (random variations) on candidate solutions

Crossover

random cutoff

Parent 1 [3, 1, 4, | 5, 0, 2]

Parent 2 [2, 3, 4, 0, 5, 1]

[3, 1, 4, 5, 0, 2]

+

[2, 3, 4, 0, 5, 1]

Offspring [3, 1, 4, 2, 0, 5]

```python
def crossover(tour_a, tour_b):
    cuttoff = np.random.randint(0, len(tour_a))
    ans = []
    for i in range(0, cuttoff):
        ans.append(tour_a[i])
    i = 0
    while len(ans) < len(tour_a):
        if tour_b[i] not in ans:
            ans.append(tour_b[i])
        i += 1
    return np.array(ans)
```

Crossover = combine “material” from multiple solutions

An Evolutionary Algorithm

begin
  $t \leftarrow 0$
  initialize $P(t)$
  evaluate $P(t)$
  while (**not** termination-condition) **do**
  begin
    $t \leftarrow t + 1$
    select $P(t)$ from $P(t-1)$
    alter $P(t)$
    evaluate $P(t)$
  end
end

A *population* of solutions

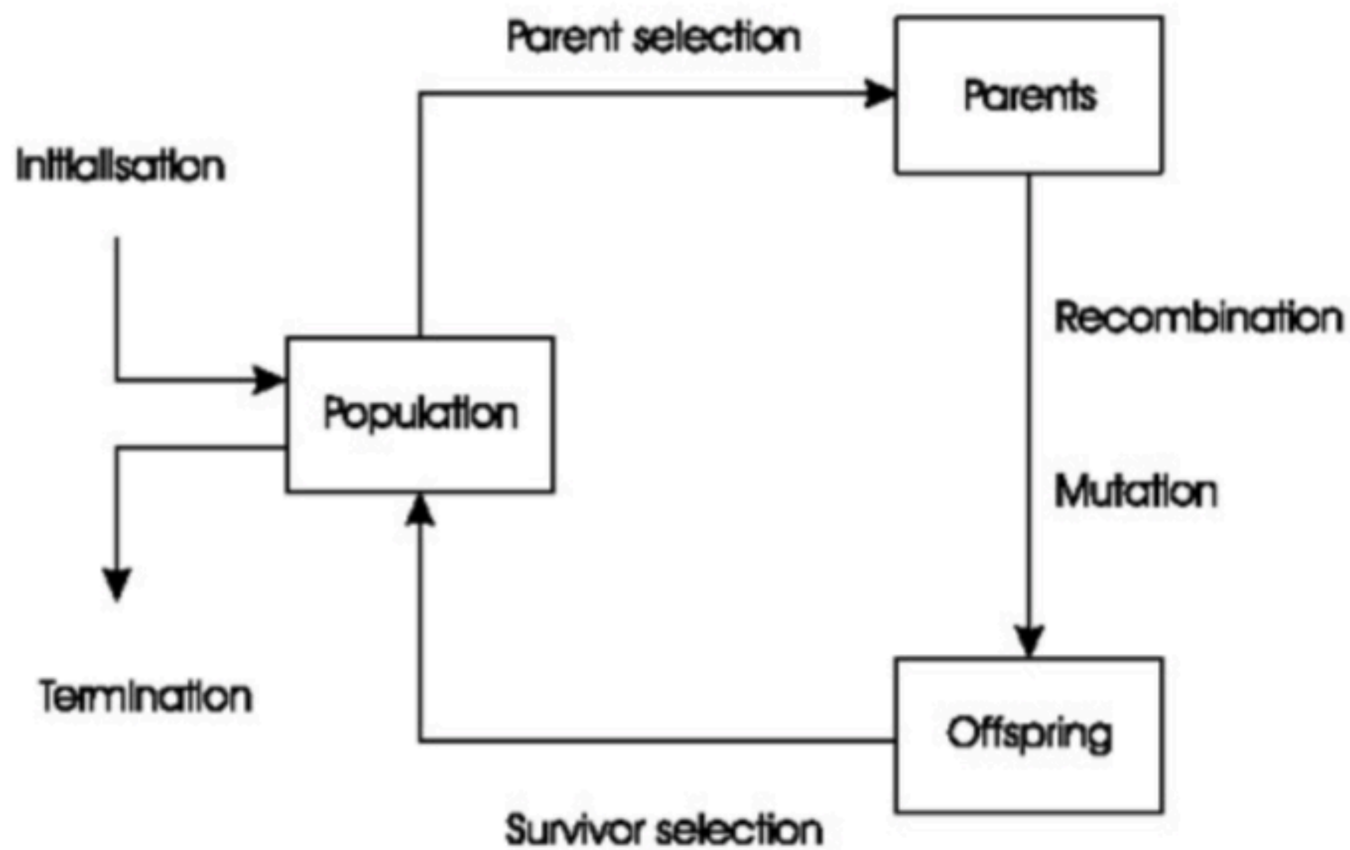
A *fitness function*

Individuals with large fitness
more likely to make it to the *next generation*

Variation operators:

- Mutation
- Crossover

Michalewicz and Fogel. Modern Heuristics. Chapter 6.


Parent selection
Parents
Initialisation
Recombination
Population
Mutation
Termination
Offspring
Survivor selection

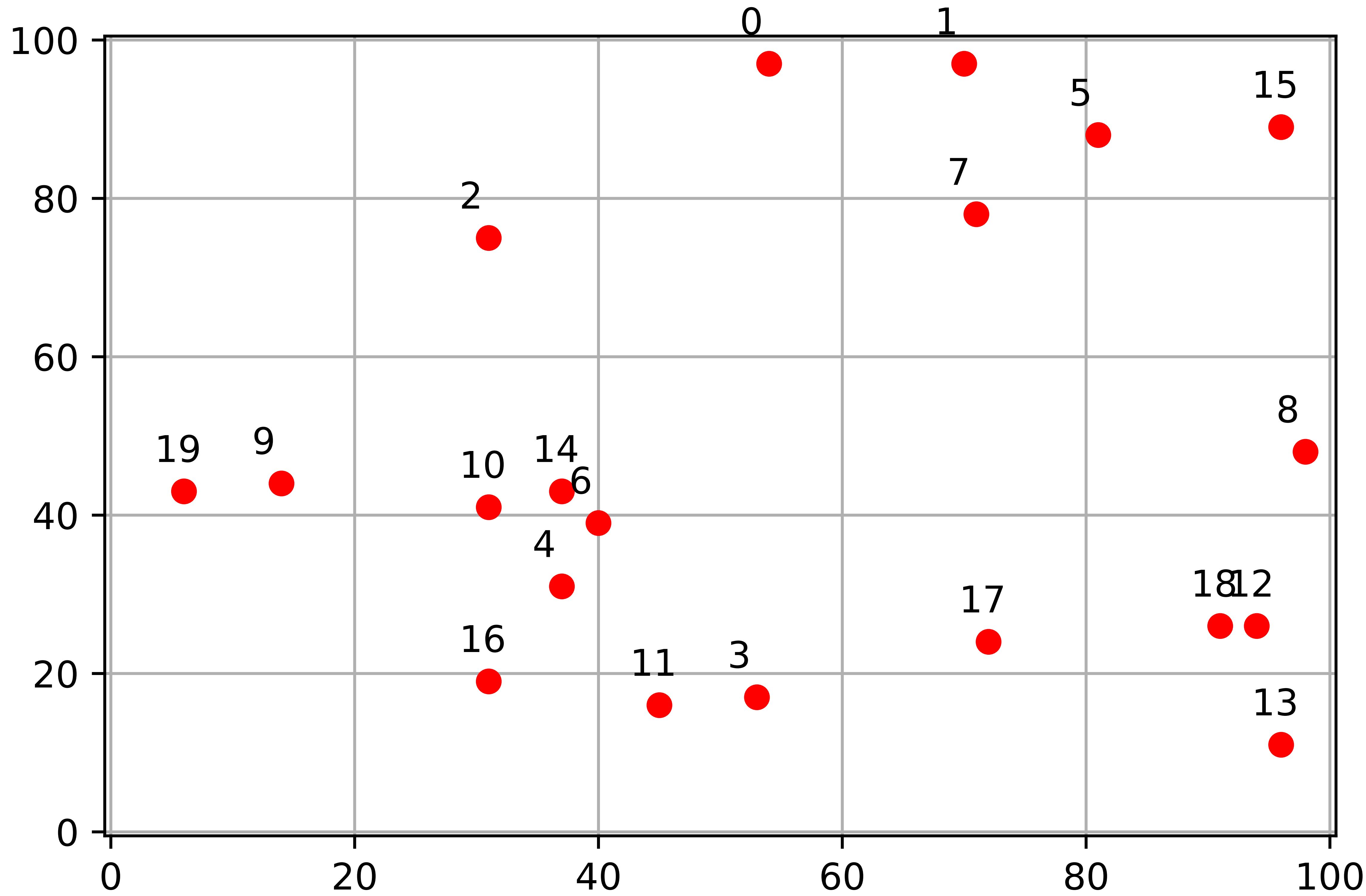
Selection + Variation

```
for _ in range(population_size-1):
    parent_index = np.random.choice(range(population_size), p = fitness_probability)
    parent = pop[parent_index]
    # sometimes mutate
    if rand() < mutation_prob:
        mutant = perturb(parent.copy())
        new_pop.append(mutant)
    # sometimes crossover
    elif rand() < crossover_prob:
        another_parent_index = np.random.choice(range(population_size), p = fitness_probability)
        another_parent = pop[another_parent_index]
        new_pop.append(crossover(parent, another_parent))
    # most times just copy the parent
    else:
        new_pop.append(parent.copy())
pop = new_pop
#compute fitness
cost_values = np.array([cost(x, tsp_instance) for x in pop])
fitness = 1.0/cost_values
fitness_probability = fitness/(np.sum(fitness))
```

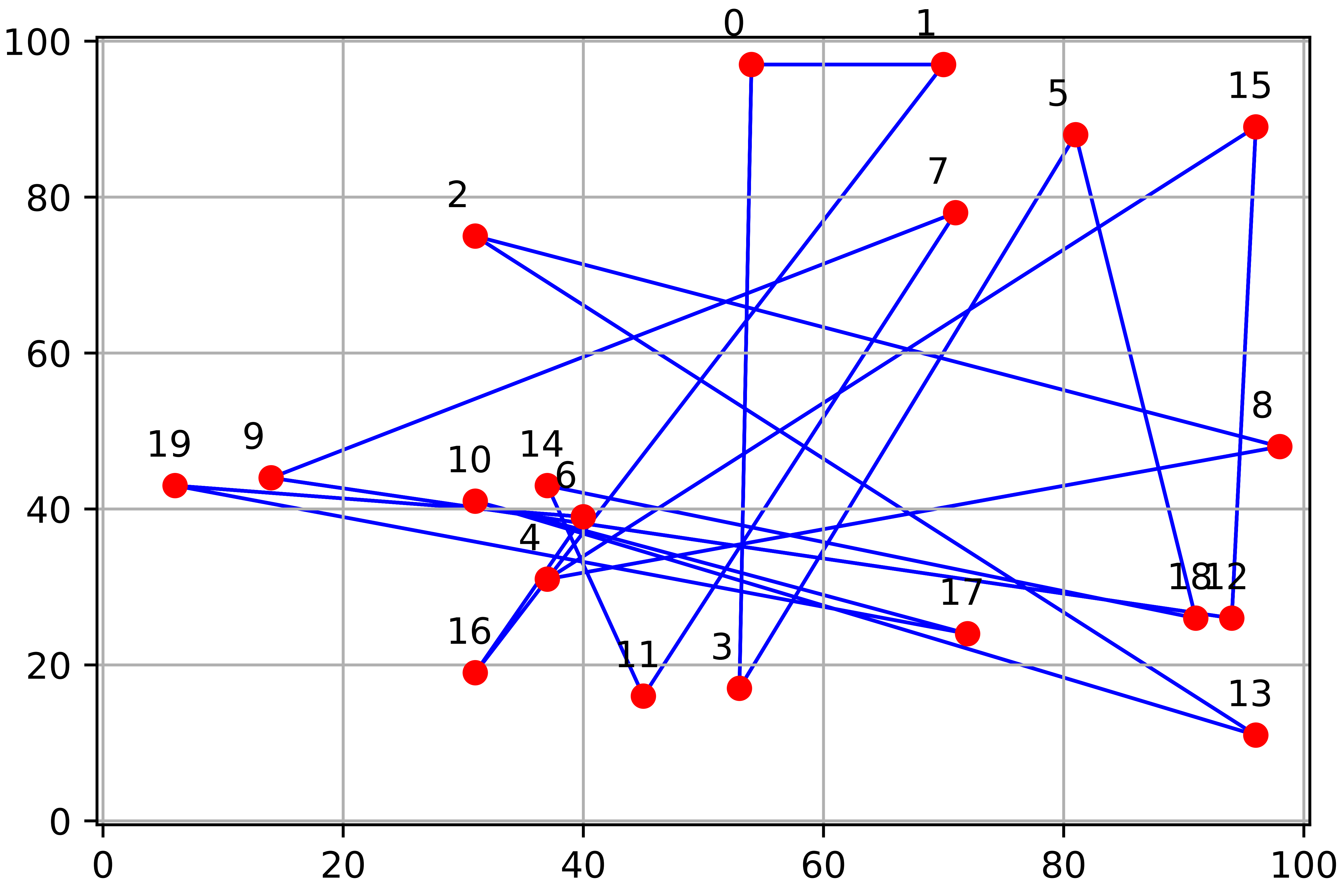
Always keep the fittest
"elitism"

```
for _ in range(number_of_generations):
    # create a new population
    new_pop = []
    fittest = pop[argmax(fitness)]
    new_pop.append(np.copy(fittest)) #always take fittes
    for _ in range(population_size-1):
        parent_index = np.random.choice(range(population_size), p = fitness_probability)
        parent = pop[parent_index]
        # sometimes mutate
        if rand() < mutation_prob:
            mutant = perturb(parent.copy())
            new_pop.append(mutant)
        # sometimes crossover
        elif rand() < crossover_prob:
            another_parent_index = np.random.choice(range(population_size), p = fitness_probability)
            another_parent = pop[another_parent_index]
            new_pop.append(crossover(parent, another_parent))
        # most times just copy the parent
        else:
            new_pop.append(parent.copy())
    pop = new_pop
    #compute fitness
    cost_values = np.array([cost(x, tsp_instance) for x in pop])
    fitness = 1.0/cost_values
    fitness_probability = fitness/(np.sum(fitness))
```

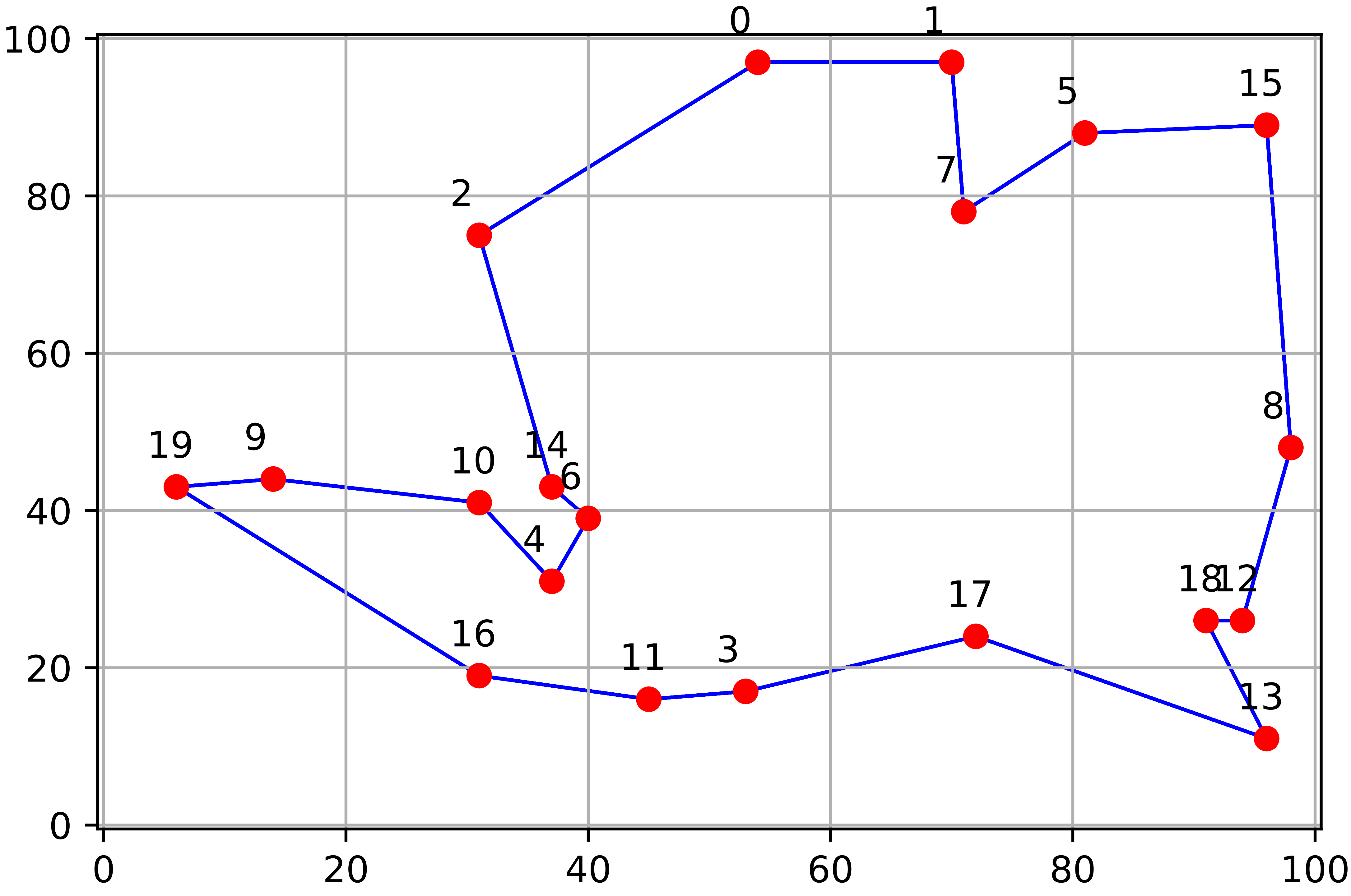
```python
def genetic_algorithm(tsp_instance, population_size,number_of_generations, mutation_prob, crossover_prob):
    number_of_cities = len(tsp_instance)
    pop = initialise_population(number_of_cities, population_size)
    #compute fitness
    cost_values = np.array([cost(x, tsp_instance) for x in pop])
    fitness = 1.0/cost_values
    fitness_probability = fitness/(np.sum(fitness))
    for _ in range(number_of_generations):
        # create a new population
        new_pop = []
        fittest = pop[argmax(fitness)]
        new_pop.append(np.copy(fittest)) #always take fittes
        for _ in range(population_size-1):
            parent_index = np.random.choice(range(population_size), p = fitness_probability)
            parent = pop[parent_index]
            # sometimes mutate
            if rand() < mutation_prob:
                mutant = perturb(parent.copy())
                new_pop.append(mutant)
            # sometimes crossover
            elif rand() < crossover_prob:
                another_parent_index = np.random.choice(range(population_size), p = fitness_probability)
                another_parent = pop[another_parent_index]
                new_pop.append(crossover(parent, another_parent))
            # most times just copy the parent
            else:
                new_pop.append(parent.copy())
        pop = new_pop
        #compute fitness
        cost_values = np.array([cost(x, tsp_instance) for x in pop])
        fitness = 1.0/cost_values
        fitness_probability = fitness/(np.sum(fitness))
    best = pop[argmin(cost_values)]
    best_cost = cost(best, tsp_instance)
    return best, best_cost
```



0
1
5
15
7
2
8
19
9
10
14
6
4
18
12
17
16
11
3
13
100
80
60
40
20
0
0
20
40
60
80
100



0
1
5
15
7
2
8
19
9
10
14
6
4
18
12
17
16
11
3
13
0
20
40
60
80
100
0
20
40
60
80
100



`population_size=50, number_of_generations=2000, mutation_prob=0.1, crossover_prob=0.1`

Permutation Encoding

Solutions are permutations: every *chromosome* is a string of numbers

Mutation

[3, 1, 4, 5, 0, 2] → [3, 2, 4, 5, 0, 1]

Pick two loci at random, swap them

Crossover

random cutoff

Parent 1 [3, 1, 4 | 5, 0, 2]

Parent 2 [2, 3, 4, 0, 5, 1]

→

[3, 1, 4, 5, 0, 2]

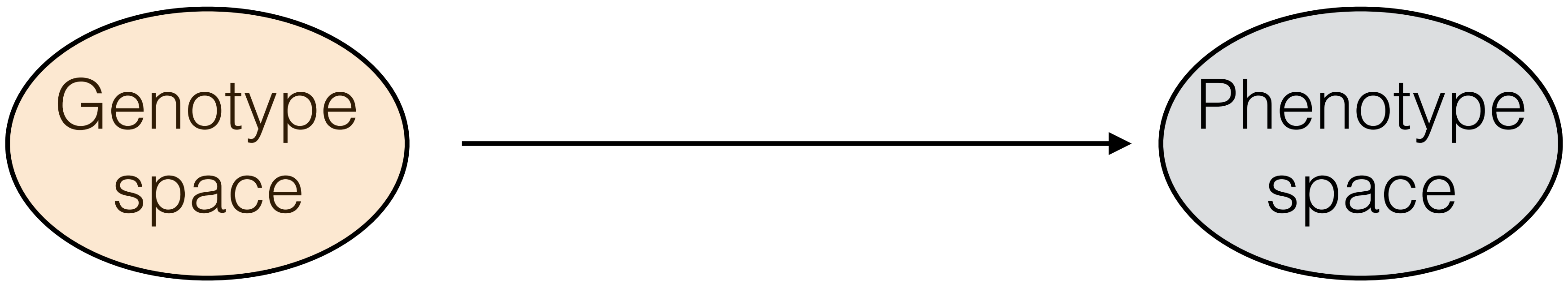
+

[2, 3, 4, 0, 5, 1]

Offspring [3, 1, 4, 2, 0, 5]

One crossover point is selected, till this point the permutation is copied from the first parent, then the second parent is scanned and if the number is not yet in the offspring it is added

Encoding



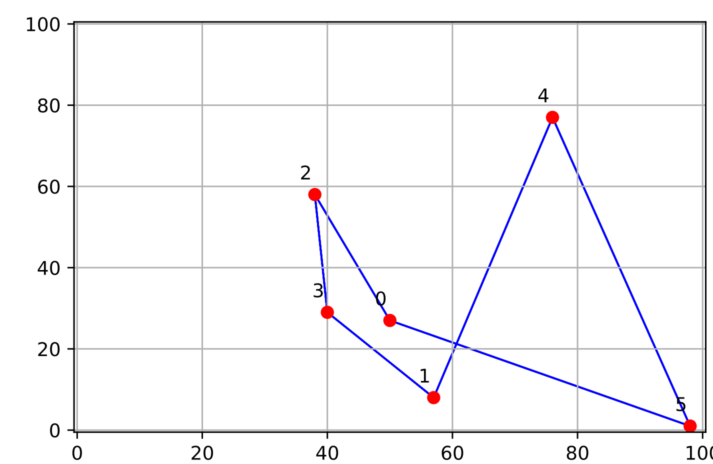
chromosomes

solutions

`[3, 2, 0, 5, 4, 1]`

`[5, 4, 1,3, 2, 0,]`

same solution, different genotypes



- To have a chance to find the global optimum, every feasible solution must be represented in genotype space

Different types of EA

Historically different flavours of EAs have been associated with different representations

Genetic algorithms: Bitstrings.

Real-valued vectors: Evolution Strategies

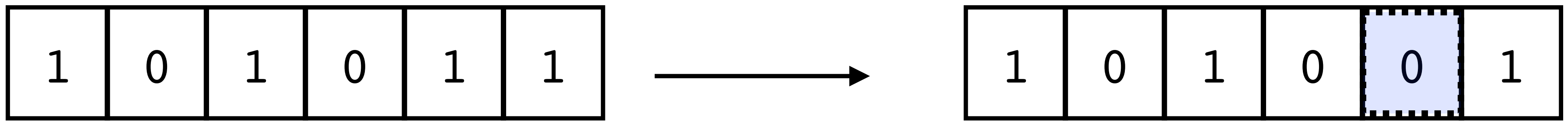
Finite state Machines: Evolutionary Programming

LISP or Expression Trees: Genetic Programming

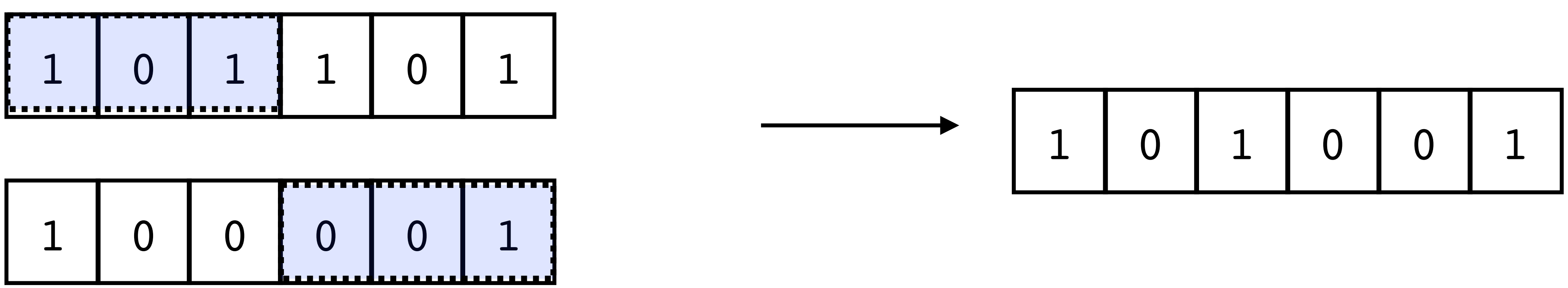
- Choose representation to suit problem.
    - Choose variation operators based on representation.
    - Selection operators are based on fitness, independent of representation

Genetic algorithms: Bitstrings.

Mutation

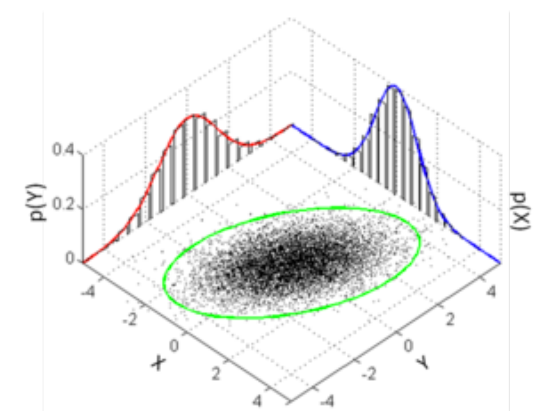


Crossover



Real-valued vectors: Evolution Strategies

- Solutions are real vectors.
- Variations are introduced by adding normally distributed random vectors — appropriately tuned according to the problem.
- Recent flavours of ES:



  - The population is represented by a Multivariate Normal Distribution, sampled to produce solutions. The co-variance matrix is adjusted by the search process.
    - CMA: Covariance Matrix Adaptation: Igel, Christian, Nikolaus Hansen, and Stefan Roth. "Covariance matrix adaptation for multi-objective optimization." Evolutionary computation 15.1 (2007): 1-28.
    - Natural-ES: Wierstra, Daan, et al. "Natural evolution strategies." 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence). IEEE, 2008.

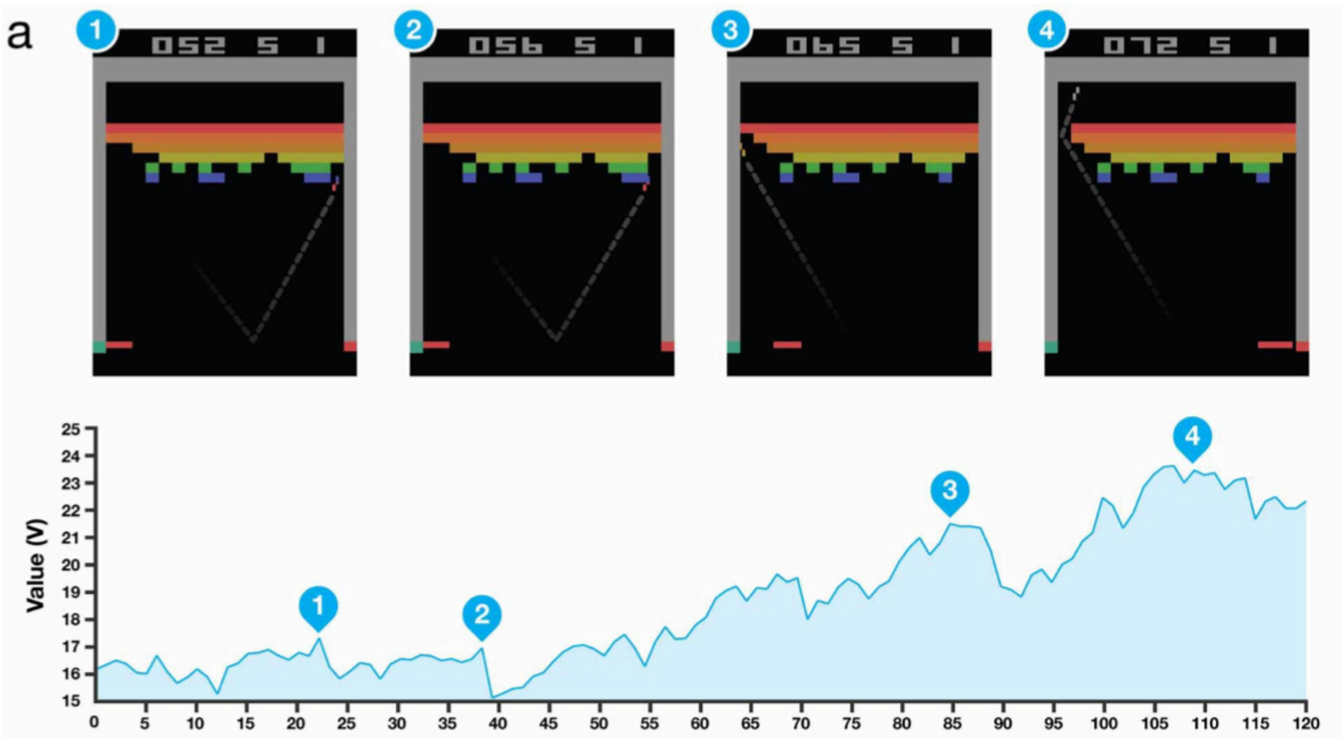
LETTER

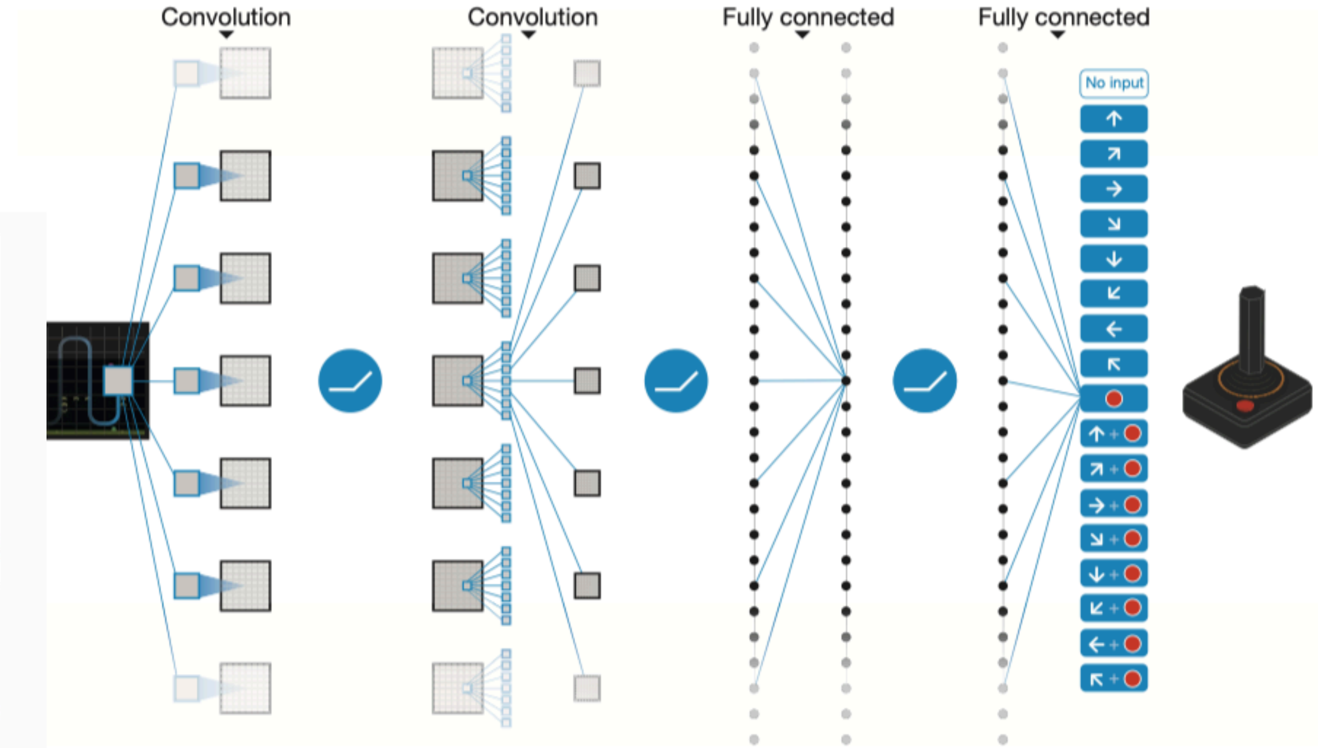
doi:10.1038/nature14236

2015

Human-level control through deep reinforcement learning

Volodymyr Mnih[1]*, Koray Kavukcuoglu[1]*, David Silver[1]*, Andrei A. Rusu[1], Joel Veness[1], Marc G. Bellemare[1], Alex Graves[1], Martin Riedmiller[1], Andreas K. Fidjeland[1], Georg Ostrovski[1], Stig Petersen[1], Charles Beattie[1], Amir Sadik[1], Ioannis Antonoglou[1], Helen King[1], Dharshan Kumaran[1], Daan Wierstra[1], Shane Legg[1] & Demis Hassabis[1]





2018

Example application

Similar performance using EC techniques.

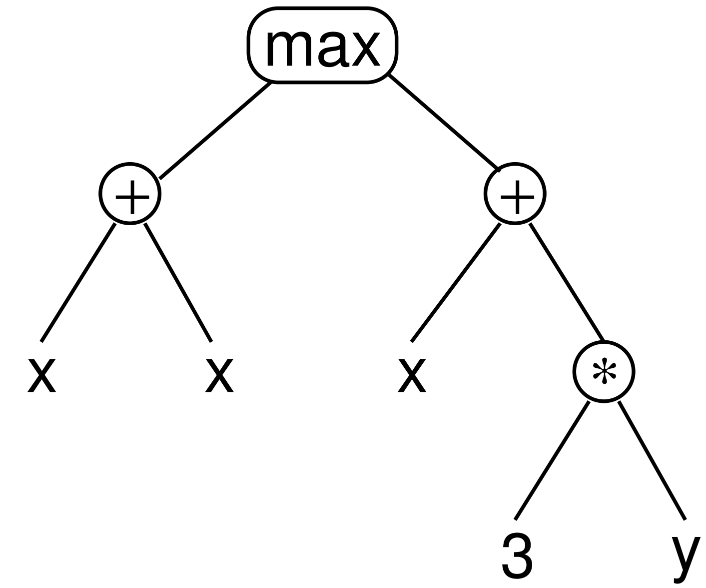
Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning

---

Felipe Petroski Such   Vashisht Madhavan   Edoardo Conti   Joel Lehman   Kenneth O. Stanley   Jeff Clune

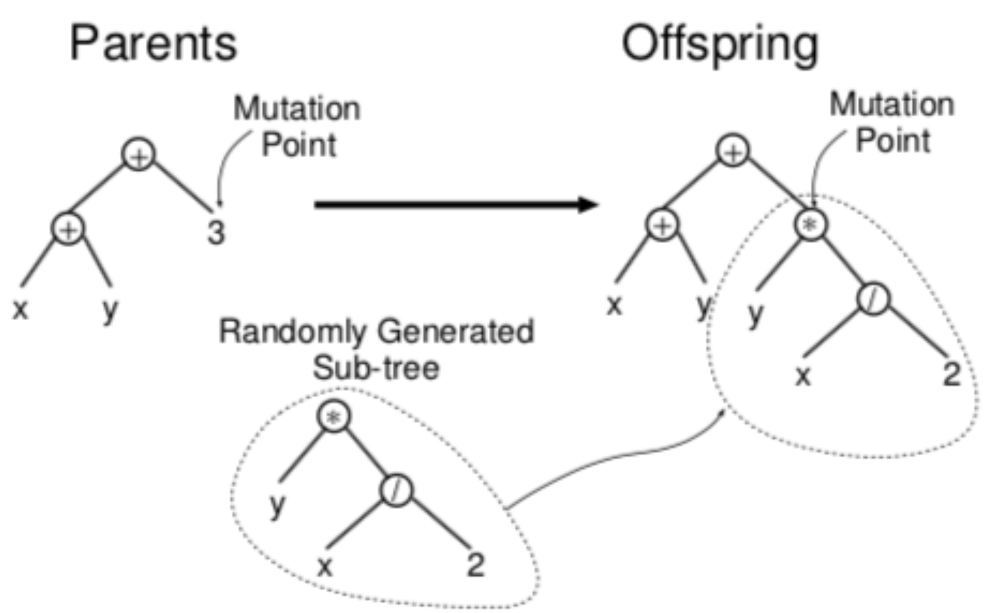
Uber AI Labs
{felipe.such, jeffclune}@uber.com

LISP or Expression Trees: Genetic Programming

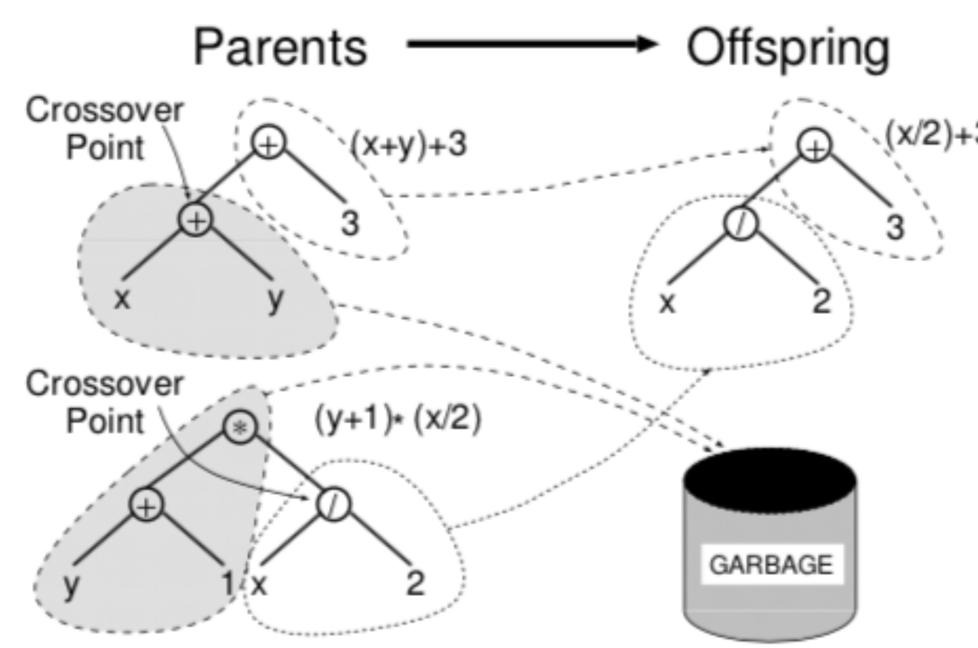


$$\max(x + x, 3y + 3x)$$

Mutation

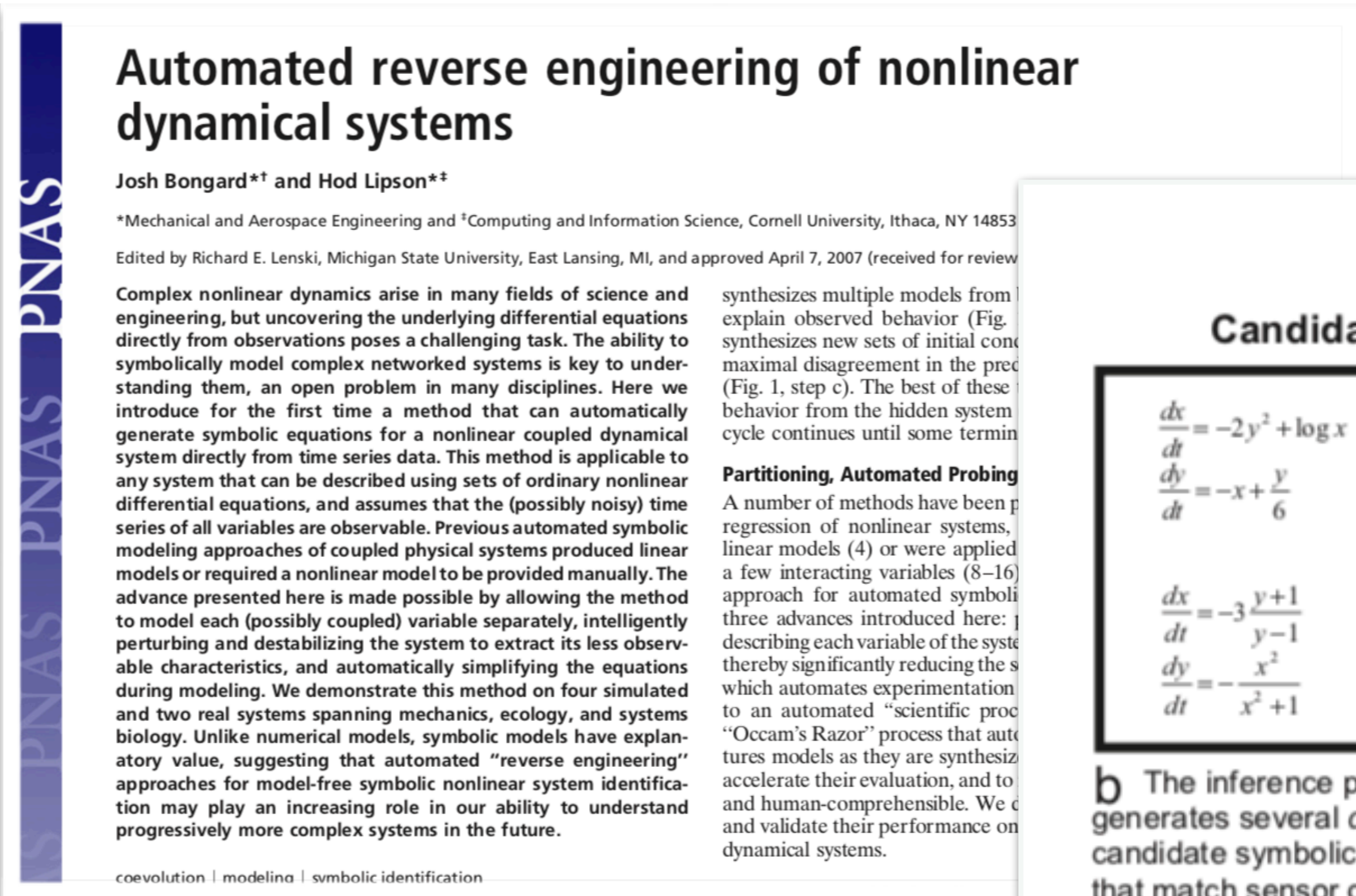


Crossover



Poli et al. A field guide to genetic programming. Chapter 2.

Example application



Automated reverse engineering of nonlinear dynamical systems

Josh Bongard*† and Hod Lipson*‡

*Mechanical and Aerospace Engineering and ‡Computing and Information Science, Cornell University, Ithaca, NY 14853

Edited by Richard E. Lenski, Michigan State University, East Lansing, MI, and approved April 7, 2007 (received for review

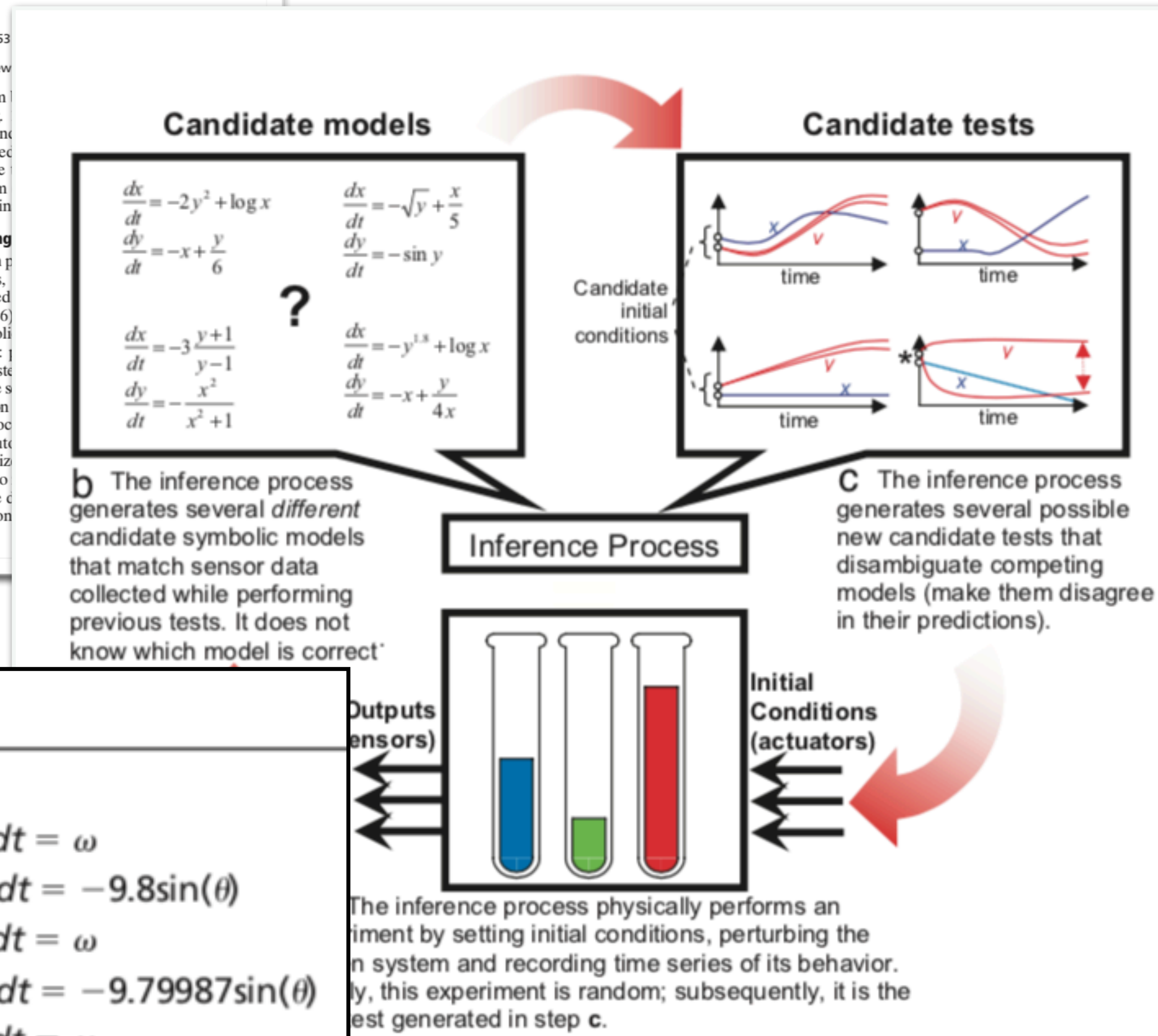
Complex nonlinear dynamics arise in many fields of science and engineering, but uncovering the underlying differential equations directly from observations poses a challenging task. The ability to symbolically model complex networked systems is key to understanding them, an open problem in many disciplines. Here we introduce for the first time a method that can automatically generate symbolic equations for a nonlinear coupled dynamical system directly from time series data. This method is applicable to any system that can be described using sets of ordinary nonlinear differential equations, and assumes that the (possibly noisy) time series of all variables are observable. Previous automated symbolic modeling approaches of coupled physical systems produced linear models or required a nonlinear model to be provided manually. The advance presented here is made possible by allowing the method to model each (possibly coupled) variable separately, intelligently perturbing and destabilizing the system to extract its less observable characteristics, and automatically simplifying the equations during modeling. We demonstrate this method on four simulated and two real systems spanning mechanics, ecology, and systems biology. Unlike numerical models, symbolic models have explanatory value, suggesting that automated "reverse engineering" approaches for model-free symbolic nonlinear system identification may play an increasing role in our ability to understand progressively more complex systems in the future.

coevolution | modeling | symbolic identification

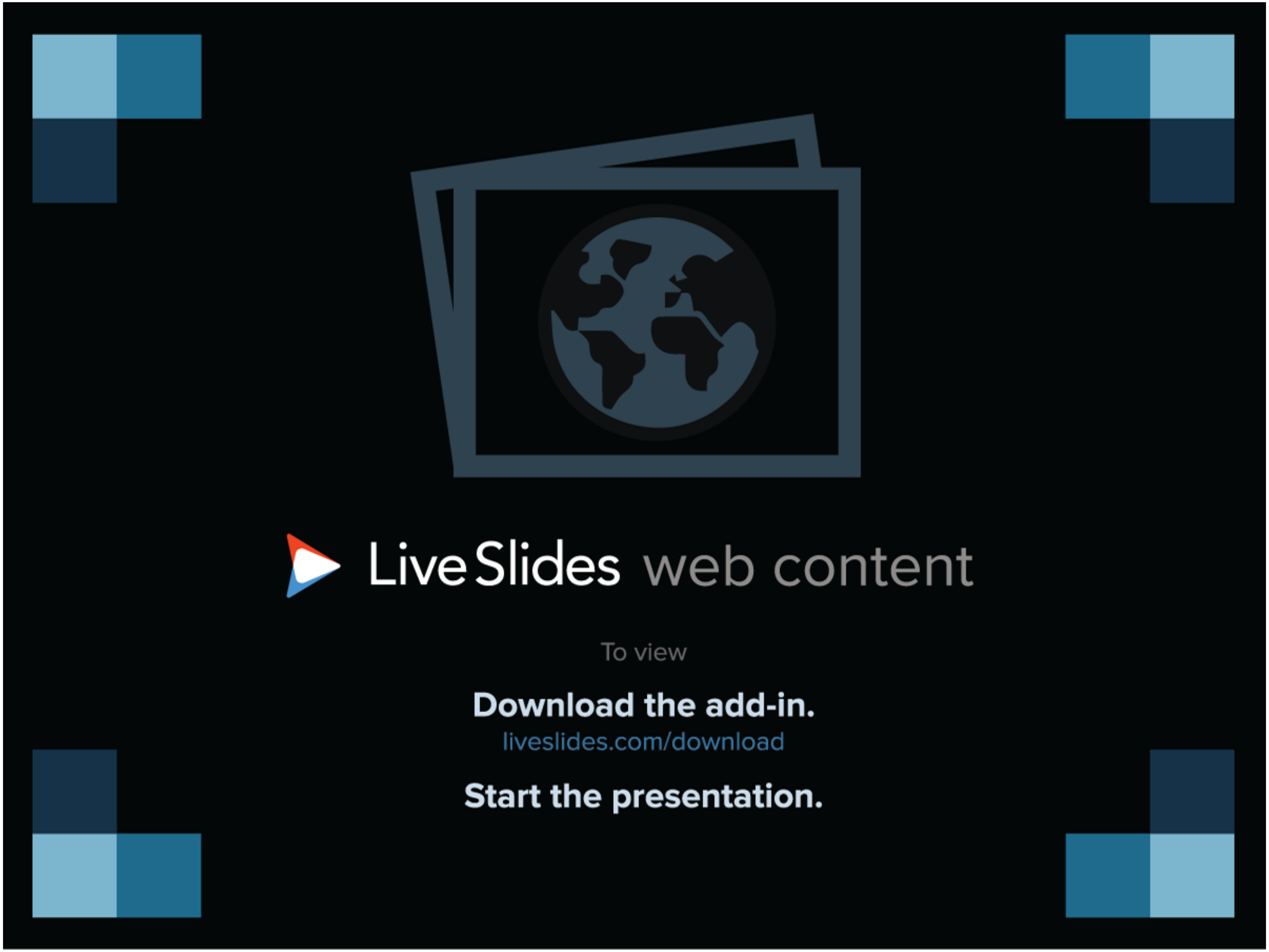
synthesizes multiple models from
explain observed behavior (Fig.
synthesizes new sets of initial con
maximal disagreement in the pred
(Fig. 1, step c). The best of these
behavior from the hidden system
cycle continues until some termin

Partitioning, Automated Probing

A number of methods have been p
regression of nonlinear systems,
linear models (4) or were applied
a few interacting variables (8–16)
approach for automated symboli
three advances introduced here:
describing each variable of the syste
thereby significantly reducing the s
which automates experimentation
to an automated "scientific proc
"Occam's Razor" process that aut
tures models as they are synthesiz
accelerate their evaluation, and to
and human-comprehensible. We
and validate their performance on
dynamical systems.



Synthetic system	
Single pendulum	
Target	$d\theta/dt = \omega$
	$d\omega/dt = -9.8\sin(\theta)$
Best model	$d\theta/dt = \omega$
	$d\omega/dt = -9.79987\sin(\theta)$
Median model	$d\theta/dt = \omega$
	$d\omega/dt = -9.8682\sin(\theta)$



EC in Robotics: https://www.youtube.com/watch?v=z9ptOeByLA4 almighty heuristics?

No Free Lunch Theorems for Optimization

David H. Wolpert and William G. Macready

***Abstract*—A framework is developed to explore the connection between effective optimization algorithms and the problems they are solving. A number of "no free lunch" (NFL) theorems are presented which establish that for any algorithm, any elevated performance over one class of problems is offset by performance over another class. These theorems result in a geometric interpretation of what it means for an algorithm to be well suited to an optimization problem. Applications of the NFL theorems to information-theoretic aspects of optimization and benchmark measures of performance are also presented. Other issues addressed include time-varying optimization problems and *a priori* "head-to-head" minimax distinctions between optimization algorithms, distinctions that result despite the NFL theorems' enforcing of a type of uniformity over all algorithms.**

***Index Terms*— Evolutionary algorithms, information theory, optimization.**

information theory and Bayesian analysis contribute to an understanding of these issues? How *a priori* generalizable are the performance results of a certain algorithm on a certain class of problems to its performance on other classes of problems? How should we even measure such generalization? How should we assess the performance of algorithms on problems so that we may programmatically compare those algorithms?

Broadly speaking, we take two approaches to these questions. First, we investigate what *a priori* restrictions there are on the performance of one or more algorithms as one runs over the set of all optimization problems. Our second approach is to instead focus on a particular problem and consider the effects of running over all algorithms. In the current paper we present results from both types of analyses but concentrate

assumption: *"all problems are equally likely"*

Conclusions

- Heuristics work in practice. They are part of the standard toolset in Computer Science / Computational Science.
- Generally lead to *many* evaluations of the objective/fitness function. *Tabu search* tries to mend this, by adding memory.
- There are plenty of *ad hoc* choices. “Too many degrees of freedom”.
- Theory results focus on convergence.
- Non-free lunch theorems: no one size fits all heuristic, all solutions must be problem specific.

Recommended Reading

Chapter 2, Introduction to Genetic Algorithms, S.N. Sivanandam and S.N. Deepa.
(Electronic version of the book available from Monash library)

For fundamental concepts of genetic algorithms:
http://www.obitko.com/tutorials/genetic-algorithms/index.php
(These slides are based on the tutorial at the link above.)