Lecture 5 – Search Based Software Testing (SBST)

AAA705: Software Testing and Quality Assurance

Jihyeok Park

PLRG

2024 Spring

Recall – White-Box (Structural) Testing

Sometimes called **structural testing** because it uses the **internal structure** of the program to derive test cases.

- **Coverage Criteria**
  - The adequacy of a test suite is measured in terms of the **coverage** of the program's internal structure.
- **Search Based Software Testing (SBST)**
  - A technique that uses **meta-heuristic search** algorithms to maximize/minimize a certain **fitness function**.
- **Dynamic Symbolic Execution (DSE)**
  - A technique that systematically explores the input space using **symbolic execution** with **dynamic analysis**.

Recall – White-Box (Structural) Testing

Sometimes called **structural testing** because it uses the **internal structure** of the program to derive test cases.

- **Coverage Criteria**
  - The adequacy of a test suite is measured in terms of the **coverage** of the program's internal structure.
- **Search Based Software Testing (SBST)**
  - A technique that uses **meta-heuristic search** algorithms to maximize/minimize a certain **fitness function**.
- **Dynamic Symbolic Execution (DSE)**
  - A technique that systematically explores the input space using **symbolic execution** with **dynamic analysis**.

Let's focus on the **SBST** in this lecture, and start from **search-based software engineering (SBSE)**!

Contents

Contents

Search Based Software Engineering (SBSE)

- The **search-based software engineering** (SBSE) is a **large movement** that seeks to apply various **optimization** techniques to software engineering problems.

Search Based Software Engineering (SBSE)

- The **search-based software engineering** (SBSE) is a **large movement** that seeks to apply various **optimization** techniques to software engineering problems.

- **Meta-heuristic** and **computational intelligence** techniques are found increasingly in SE research.

Search Based Software Engineering (SBSE)

- The **search-based software engineering** (SBSE) is a **large movement** that seeks to apply various **optimization** techniques to software engineering problems.

- **Meta-heuristic** and **computational intelligence** techniques are found increasingly in SE research.

- Two major conferences (ICSE and ESEC/FSE) now tend to have whole sessions dedicated to SBSE.

Search Based Software Engineering (SBSE)

- The **search-based software engineering** (SBSE) is a **large movement** that seeks to apply various **optimization** techniques to software engineering problems.
- **Meta-heuristic** and **computational intelligence** techniques are found increasingly in SE research.
- Two major conferences (ICSE and ESEC/FSE) now tend to have whole sessions dedicated to SBSE.
- Dedicated international conference (e.g., SSBSE) and many other workshops.

Meta-heuristic

- Strategies that **guide** the search process to find **acceptable solutions**

Meta-heuristic

- Strategies that **guide** the search process to find **acceptable solutions**

- **Approximate** and usually non-deterministic

Meta-heuristic

- Strategies that **guide** the search process to find **acceptable solutions**

- **Approximate** and usually non-deterministic

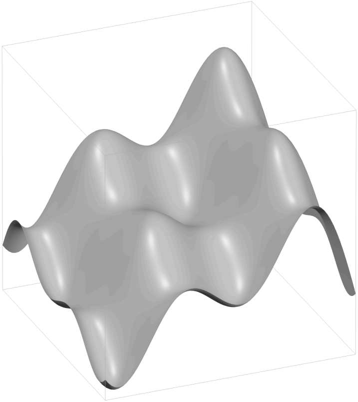
- **General** and not problem-specific

Meta-heuristic

- Strategies that **guide** the search process to find **acceptable solutions**
- **Approximate** and usually non-deterministic
- **General** and not problem-specific
- **Iterative** improvement by **exploring** the search space

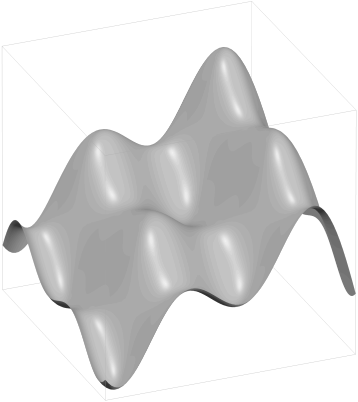
Search Space

How to find the **best** or at least an **acceptable** solution?



Search Space

How to find the **best** or at least an **acceptable** solution?



Try and automatically **learn** from the **experience** for the next **trial**.

Key Ingredients

- **Representation – What** are we going to try this time?

Key Ingredients

- **Representation – What** are we going to try this time?

- **Operators – How to** change the representation for search?

Key Ingredients

- **Representation – What** are we going to try this time?

- **Operators – How to** change the representation for search?

- **Fitness Function – How well** are we doing?

Key Ingredients

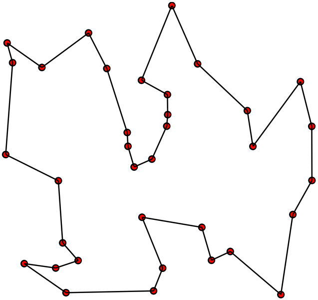
- **Representation** – **What** are we going to try this time?

- **Operators** – **How to** change the representation for search?

- **Fitness Function** – **How well** are we doing?

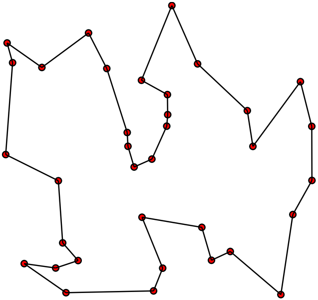
- Constraints, etc.

Example: Travelling Salesman Problem (TSP)



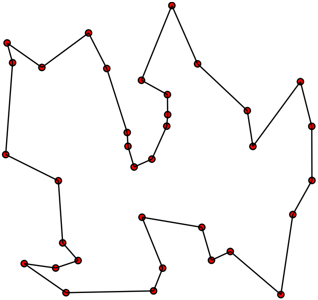
- Assume that you are a salesman.

Example: Travelling Salesman Problem (TSP)



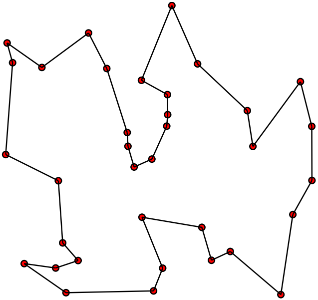
- Assume that you are a salesman.
- You want to **visit all** the cities and **return** to the starting city with the **minimum cost** (e.g., distance, time, etc.).

Example: Travelling Salesman Problem (TSP)



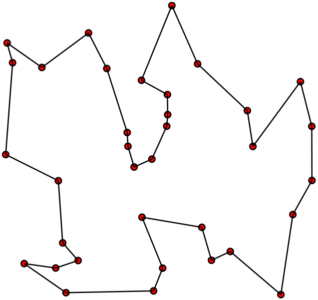
- Assume that you are a salesman.
- You want to **visit all** the cities and **return** to the starting city with the **minimum cost** (e.g., distance, time, etc.).
- Unfortunately, the TSP is a **NP-hard** problem. It means that there is **no known algorithm** that can solve it in **polynomial time**.

Example: Travelling Salesman Problem (TSP)



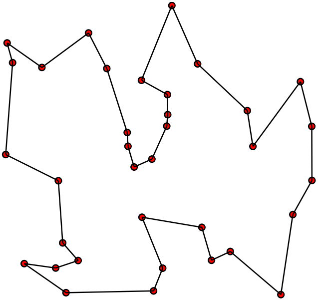
- **Representation**: A sequence of cities

Example: Travelling Salesman Problem (TSP)



- **Representation**: A sequence of cities
- **Operators**: Swap two cities

Example: Travelling Salesman Problem (TSP)



- **Representation**: A sequence of cities
- **Operators**: Swap two cities
- **Fitness Function**: Total distance

Exploitation vs. Exploration

- **Exploitation**: If we have found a good solution, we should try to search around it or do something similar.

Exploitation vs. Exploration

- **Exploitation**: If we have found a good solution, we should try to search around it or do something similar.

- **Exploration**: Unexplored search space may contain **much better** solutions.

Exploitation vs. Exploration

- **Exploitation**: If we have found a good solution, we should try to search around it or do something similar.

- **Exploration**: Unexplored search space may contain **much better** solutions.

- How to **balance** these two is a **key** to the success of SBSE.

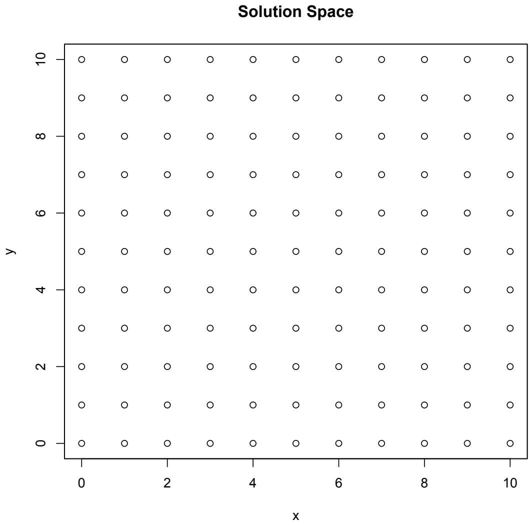
Key Topics

- **Fitness Landscape**
- **Local Search**
- **Genetic Algorithms**
- **Bio-inspired Algorithms**

Contents

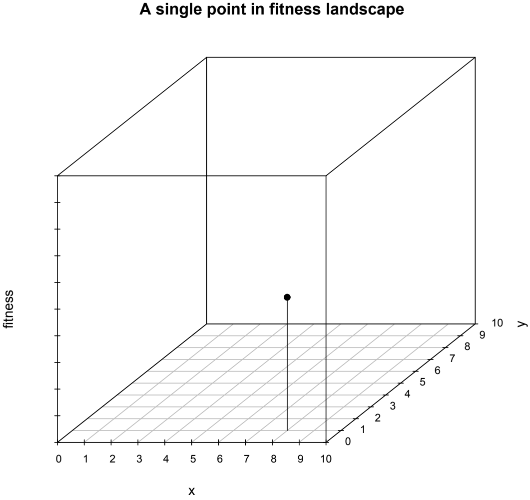
Fitness Landscape

Let's consider a fake problem: Find the pair (x, y) such that $x + y = 10$ for $0 \leq x \leq 10$ and $0 \leq y \leq 10$.



Fitness Landscape

Let's consider a fake problem: Find the pair (x, y) such that $x + y = 10$ for $0 \leq x \leq 10$ and $0 \leq y \leq 10$.



Fitness Landscape

- For each **representation** (x, y), how to know **how good** it is?

Fitness Landscape

- For each **representation** (x, y), how to know **how good** it is?

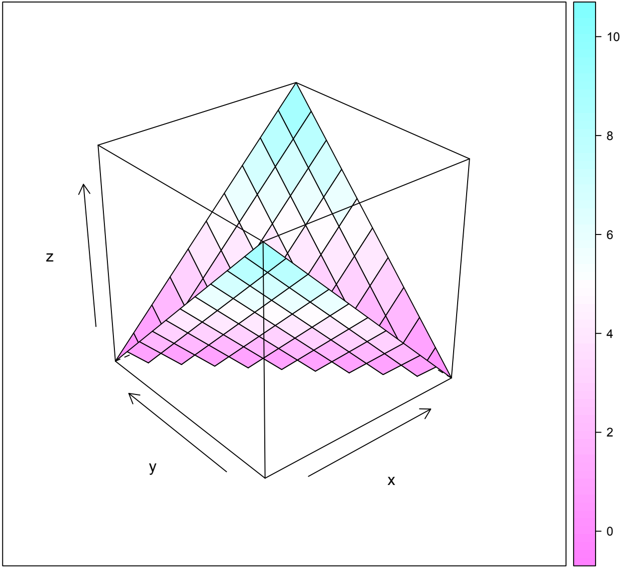
- We need to solve the problem $x + y = 10$.

Fitness Landscape

- For each **representation** (x, y), how to know **how good** it is?
- We need to solve the problem $x + y = 10$.
- We can **change** the problem into a **minimization** problem:

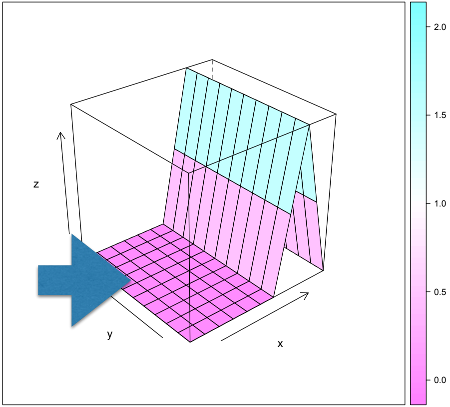
$$f(x, y) = |10 - (x + y)|$$

Fitness Landscape



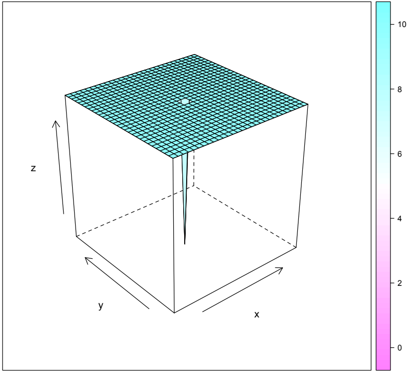
Fitness Landscape – Plateau

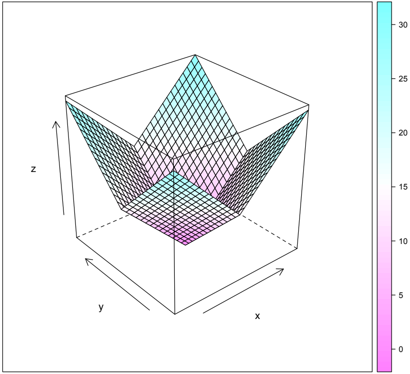
It is difficult to escape from the large and flat region (i.e., **plateau**) in the fitness landscape



Fitness Landscape – Needle in a Haystack

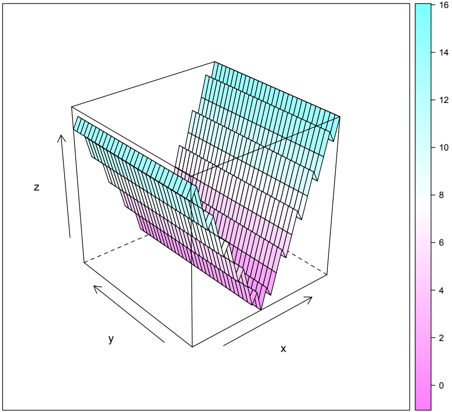
If the fitness landscape has a small region of high fitness surrounded by a large region of low fitness, it is called a **needle in a haystack**, and it is the worst case for search algorithms. We need to find a way to change the landscape into a more favorable one.

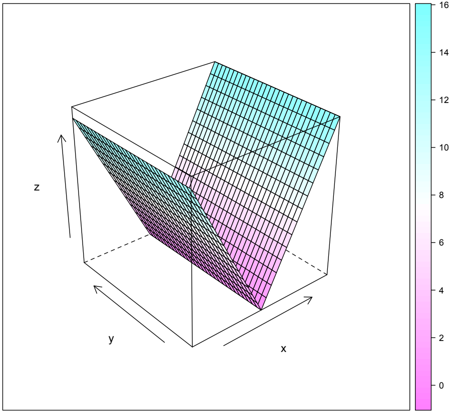




Fitness Landscape – Ruggedness

If the fitness landscape has many local optima, it is called a **rugged** landscape. In this case, the search algorithm may get stuck in one of the many local optima and fail to find the global optimum.





Contents

Local Search

- **Local search** is one of the simplest and most widely used meta-heuristic algorithms.

Local Search

- **Local search** is one of the simplest and most widely used meta-heuristic algorithms.

- It **starts** from a **random solution**.

Local Search

- **Local search** is one of the simplest and most widely used meta-heuristic algorithms.
- It **starts** from a **random solution**.
- Consider multiple **neighboring** solutions.

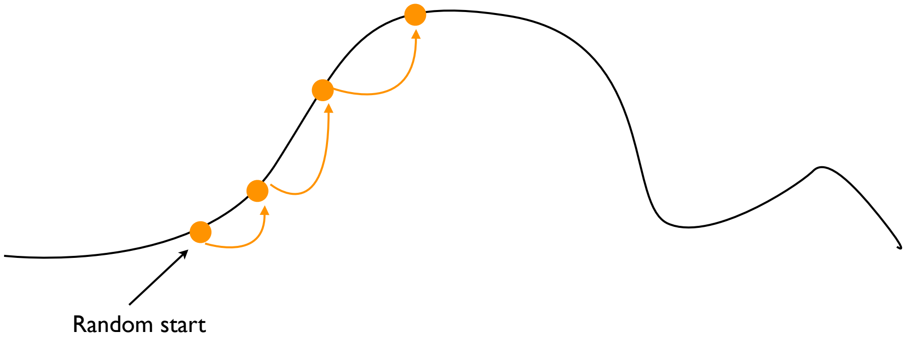
Local Search

- **Local search** is one of the simplest and most widely used meta-heuristic algorithms.
- It **starts** from a **random solution**.
- Consider multiple **neighboring** solutions.
- **Move** to one of **better** solutions according to the fitness function.

Local Search

- **Local search** is one of the simplest and most widely used meta-heuristic algorithms.
- It **starts** from a **random solution**.
- Consider multiple **neighboring** solutions.
- **Move** to one of **better** solutions according to the fitness function.
- **Repeat** the process until **no better solution** is found.

Local Search



Local Search – Hill Climbing (Steepest Ascent)

The most popular local search algorithm is the **hill climbing** algorithm with the **steepest ascent** strategy.

```
HillClimbing()
    climb ← True
    s ← GetRandom()
    while climb
        N ← GetNeighbours(s)
        climb ← False
        foreach n ∈ N
            if Fitness(n) > Fitness(s)
                climb ← True
                s ← n
            return s
```

Local Search – Hill Climbing (First Ascent)

One of variations of the hill climbing algorithm is the **first ascent** strategy by selecting the first better solution.

```
HillClimbing()
    climb ← True
    s ← GetRandom()
    while climb
        N ← GetNeighbours(s)
        climb ← False
        foreach n ∈ N
            if Fitness(n) > Fitness(s)
                climb ← True
                s ← n
                break
            return s
```

Local Search – Hill Climbing (Random)

Or, we can **randomly** select a solution among the better neighboring solutions in the hill climbing algorithm.

$\textsc{HillClimbing}()$

    $s \leftarrow \textsc{GetRandom}()$

    while True

        $N \leftarrow \textsc{GetNeighbours}(s)$

        $N' \leftarrow \{n \in N | \textsc{Fitness}(n) > \textsc{Fitness}(s)\}$

        if $|N'| > 0$

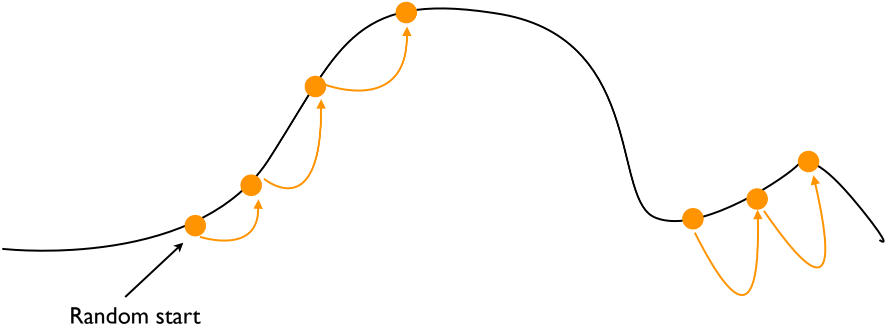
            $s \leftarrow \textsc{RandomPick}(N')$

        else

            break

    return s

Local Search – Stuck in Local Optima



Local Search – Stuck in Local Optima

- The local search algorithm may get **stuck in a local optima**.

Local Search – Stuck in Local Optima

- The local search algorithm may get **stuck in a local optima**.
- Then, how to **escape** from the local optima?

Local Search – Stuck in Local Optima

- The local search algorithm may get **stuck in a local optima**.

- Then, how to **escape** from the local optima?

- There are many strategies to **escape** from the local optima.

Local Search – Simulated Annealing

- Let's mimic the process of **annealing** in metallurgy.

Local Search – Simulated Annealing

- Let's mimic the process of **annealing** in metallurgy.
- We introduce a **temperature** parameter that controls the **probability** of accepting a **worse solution** for **exploration** purposes.

Local Search – Simulated Annealing

- Let's mimic the process of **annealing** in metallurgy.
- We introduce a **temperature** parameter that controls the **probability** of accepting a **worse solution** for **exploration** purposes.
- The temperature is **gradually decreased** to **reduce** the probability of accepting a worse solution.

Local Search – Simulated Annealing

SimulatedAnnealing()

(1) $s = s_0$
(2) $T \leftarrow T_0$
(3) **for** $k = 0$ **to** n
(4) $s_{new} \leftarrow \textsc{GetRandomNeighbour}(s)$
(5) **if** $P(\mathrm{F}(s), \mathrm{F}(s_{new}), T) \geq random(0,1)$ **then** $s \leftarrow s_{new}$
(6) $T \leftarrow \textsc{Cool}(T)$
(7) **return** s

$\mathrm{P}(F(s)$,$F(s_{new})$, $T)$

(1) **if** $F(s_{new}) > F(s)$ **then return** 1.0
(2) **else return** $e^{\frac{F(s_{new})-F(s)}{T}}$

Local Search – Simulated Annealing

There are several strategies to **decrease** the temperature (**cooling**):

- Linear cooling

$$T(t) = T_0 - \alpha t$$

Local Search – Simulated Annealing

There are several strategies to **decrease** the temperature (**cooling**):

- Linear cooling

$$T(t) = T_0 - \alpha t$$

- Exponential cooling

$$T(t) = T_0 \cdot \alpha^t (0 < \alpha < 1)$$

Local Search – Simulated Annealing

There are several strategies to **decrease** the temperature (**cooling**):

- Linear cooling

$$T(t) = T_0 - \alpha t$$

- Exponential cooling

$$T(t) = T_0 \cdot \alpha^t (0 < \alpha < 1)$$

- Logarithmic cooling

$$T(t) = \frac{c}{\log(t + d)}$$

  - With large c, slow cooling
  - Surprisingly, there exists a proof that says that the logarithmic cooling will find the global optimum in infinite time.
  - Theoretically interesting, but not practical.

Local Search – Tabu Search

- **Tabu search** is another approach to **escape** from the local optima.

Local Search – Tabu Search

- **Tabu search** is another approach to **escape** from the local optima.
- Two main ideas:
  - **Memory**: Keep track of **recently visited** solutions and **avoid** them.
  - **Diversification**: Introduce randomness to **explore** the search space.

Local Search – Tabu Search

```
TabuSearch()
    s ← s_0
    s_best ← s
    T ← [] // tabu list
    while not stoppingCondition()
        c_best ← null
        foreach c ∈ GetNeighbours(s)
            if (c ∉ T) ∧ (F(c) > F(c_best)) then c_best ← c
        s ← c_best
        if F(c_best) > F(s_best) then s_best ← c_best
        append(T, c_best)
        if |T| > maxTabuSize then removeAt(T, 0)
    return sBest
```

Tabu list stores the **recently visited** solutions using a FIFO queue, and we can control the **size** of the tabu list.

Local Search – Random Restart

- In common situations, we have a **search budget** (e.g., time, # of fitness evaluations, etc.) for the local search algorithm.

Local Search – Random Restart

- In common situations, we have a **search budget** (e.g., time, # of fitness evaluations, etc.) for the local search algorithm.

- What if the local search algorithm **stops** but the **budget still remains**?

Local Search – Random Restart

- In common situations, we have a **search budget** (e.g., time, # of fitness evaluations, etc.) for the local search algorithm.

- What if the local search algorithm **stops** but the **budget still remains**?

- We can **restart** the local search algorithm from a **new random solution** to keep searching for the global optimum.

Local Search – Search Radius

- The **effectiveness** of the local search algorithm depends on the **search radius** rather than the size of the search space.

Local Search – Search Radius

- The **effectiveness** of the local search algorithm depends on the **search radius** rather than the size of the search space.

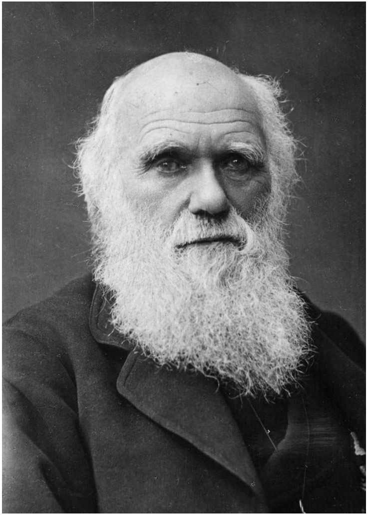
- Search radius is the **maximum number of moves** required to go **across** the search space.

Local Search – Search Radius

- The **effectiveness** of the local search algorithm depends on the **search radius** rather than the size of the search space.
- Search radius is the **maximum number of moves** required to go **across** the search space.
- For example, consider the **TSP problem** with **20 cities**.
  - **Search Space**: $N! = 20! \approx 2.4 \times 10^{18}$
  - **Search Radius**: $\frac{N(N-1)}{2} = \frac{20\times 19}{2} = 190$
  - It means that the local search algorithm can find the global optimum within 190 moves in a good situation.

Contents

Genetic Algorithms



Genetic Algorithms

- Let's **mimic** the process of **natural selection** in biology.

Genetic Algorithms

- Let's **mimic** the process of **natural selection** in biology.
- We will keep multiple solutions as a **population**.

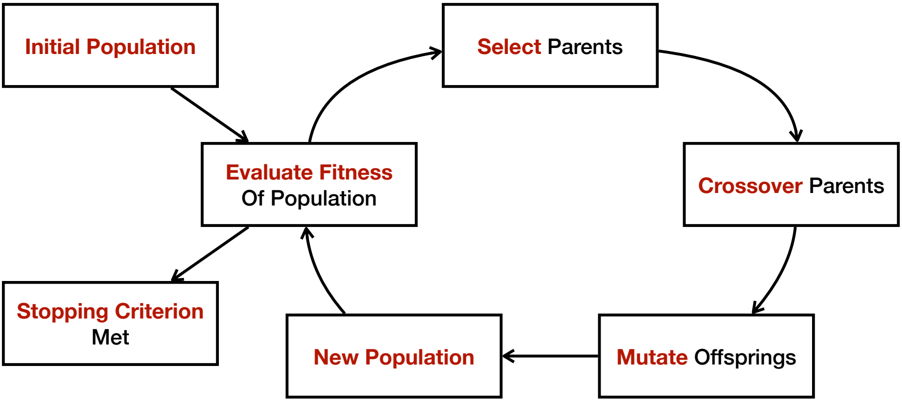
Genetic Algorithms

- Let's **mimic** the process of **natural selection** in biology.
- We will keep multiple solutions as a **population**.
- In **each generation**, we apply **selection pressure** to **evolve** the population of solutions towards better fitness values.

Genetic Algorithms

- Let's **mimic** the process of **natural selection** in biology.
- We will keep multiple solutions as a **population**.
- In **each generation**, we apply **selection pressure** to **evolve** the population of solutions towards better fitness values.
- Remember: **exploration** and **exploitation**
  - If **too much pressure**, the search converges to a local optimum.
  - If **too little pressure**, the search goes nowhere.

Genetic Algorithms



Genetic Algorithms – Selection Strategies

- We need to **select** two parent individuals to produce a new offspring.

Genetic Algorithms – Selection Strategies

- We need to **select** two parent individuals to produce a new offspring.

- This is one of two places where we apply the **selection pressure**.

Genetic Algorithms – Selection Strategies

- We need to **select** two parent individuals to produce a new offspring.
- This is one of two places where we apply the **selection pressure**.
- The **better** individuals selected as parents, the **more selection pressure** is applied.

Genetic Algorithms – Selection Strategies

Fitness Proportional Selection (FPS): The probability of selecting an individual is proportional to its fitness value.

$$P_{\text{FPS}}(i) = \frac{f(i)}{\sum_{j=1}^{\mu} f(j)}$$

where i is an **individual**, $f(i)$ its **fitness value**, and μ the **population size**.

Fitness Proportional Selection (FPS): The probability of selecting an individual is proportional to its fitness value.

$$P_{\text{FPS}}(i) = \frac{f(i)}{\sum_{j=1}^{\mu} f(j)}$$

where i is an **individual**, $f(i)$ its **fitness value**, and μ the **population size**.

If there is an **outstanding individual**, it will quickly dominate the population (**premature convergence**).

Genetic Algorithms – Selection Strategies

Fitness Proportional Selection (FPS): The probability of selecting an individual is proportional to its fitness value.

$$P_{\mathrm{FPS}}(i) = \frac{f(i)}{\sum_{j=1}^{\mu} f(j)}$$

where i is an **individual**, $f(i)$ its **fitness value**, and μ the **population size**.

If there is an **outstanding individual**, it will quickly dominate the population (**premature convergence**). To avoid this, we can do:

- **Windowing** – At each generation, fitness is transformed by subtracting the minimum fitness of the current population: $\beta(t) = \min_{i \in P} f(i)$

Fitness Proportional Selection (FPS): The probability of selecting an individual is proportional to its fitness value.

$$P_{\mathrm{FPS}}(i) = \frac{f(i)}{\sum_{j=1}^{\mu} f(j)}$$

where i is an **individual**, $f(i)$ its **fitness value**, and μ the **population size**.

If there is an **outstanding individual**, it will quickly dominate the population (**premature convergence**). To avoid this, we can do:

- **Windowing** – At each generation, fitness is transformed by subtracting the minimum fitness of the current population: $\beta(t) = \min_{i \in P} f(i)$
- **Sigma scaling** – The fitness is transformed by subtracting the mean fitness and dividing by the standard deviation of the fitness values.

$$f'(i) = \max(1 + \frac{f(i) - \bar{f}}{2\sigma}, 0.1)$$

Genetic Algorithms – Selection Strategies

Ranking Selection – Individuals are ranked by their fitness values and selected according to their ranks (best $= \mu - 1$, worst $= 0$).

Genetic Algorithms – Selection Strategies

Ranking Selection – Individuals are ranked by their fitness values and selected according to their ranks (best $= \mu - 1$, worst $= 0$).

There are different ways to utilize ranks to select individuals:

- **Linear ranking** – parameterizes by $1 \leq s \leq 2$

$$P_{\text{linear}}(i) = \frac{2-s}{\mu} + \frac{i(s-1)}{\sum_{j=1}^{\mu} j}$$

Genetic Algorithms – Selection Strategies

Ranking Selection – Individuals are ranked by their fitness values and selected according to their ranks (best $= \mu - 1$, worst $= 0$).

There are different ways to utilize ranks to select individuals:

- **Linear ranking** – parameterizes by $1 \leq s \leq 2$

$$P_{\text{linear}}(i) = \frac{2-s}{\mu} + \frac{i(s-1)}{\sum_{j=1}^{\mu} j}$$

- **Exponential ranking** – more selection pressure than linear ranking

$$P_{\exp}(i) = \frac{1-e^{-i}}{\sum_{j=1}^{\mu}(1-e^{-j})}$$

Genetic Algorithms – Selection Strategies

Ranking Selection – Individuals are ranked by their fitness values and selected according to their ranks (best $= \mu - 1$, worst $= 0$).

There are different ways to utilize ranks to select individuals:

- **Linear ranking** – parameterizes by $1 \leq s \leq 2$

$$P_{\text{linear}}(i) = \frac{2 - s}{\mu} + \frac{i(s-1)}{\sum_{j=1}^{\mu} j}$$

- **Exponential ranking** – more selection pressure than linear ranking

$$P_{\exp}(i) = \frac{1 - e^{-i}}{\sum_{j=1}^{\mu} (1 - e^{-j})}$$

Individual	Fitness	Rank	P_{FPS}	$P_{\text{linear}}(s = 1.5)$	$P_{\text{linear}}(s = 2)$	$P_{\exp}$
A	1	0	0.10	0.17	0.00	0.00
B	4	1	0.40	0.33	0.33	0.42
C	5	2	0.50	0.50	0.67	0.58

Genetic Algorithms – Selection Strategies

There are many other selection strategies:

- **Roulette Wheel Selection**
- **Stochastic Universal Sampling (SUS)**
- **Tournament Selection**
- **Over-Selection**
- etc.

Genetic Algorithms – Crossover Operators

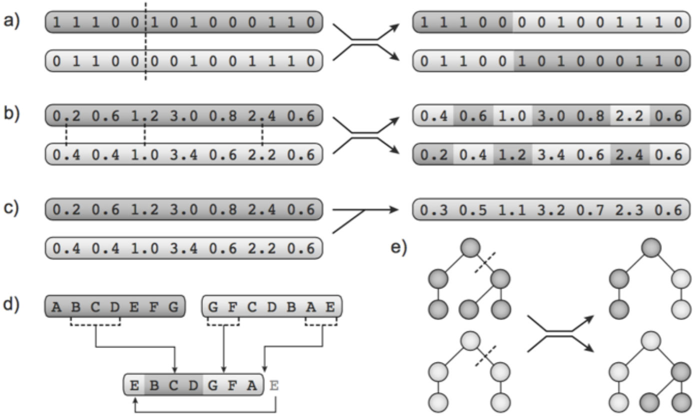


Figure 1.11 Examples of crossover operators. *a*) one-point; *b*) uniform; *c*) arithmetic; *d*) for sequences; *e*) for trees.

(from "Bio-inspired Artificial Intelligence: Theories, Methods, and Technologies" by Dario Floreano and Claudio Mattiussi)

Genetic Algorithms – Mutation Operators

- The **mutation** operator makes small changes to the representation of an individual.

Genetic Algorithms – Mutation Operators

- The **mutation** operator makes small changes to the representation of an individual.

- This is, usually, the **only** way **new genetic material** is introduced into the population.

Genetic Algorithms – Mutation Operators

- The **mutation** operator makes small changes to the representation of an individual.

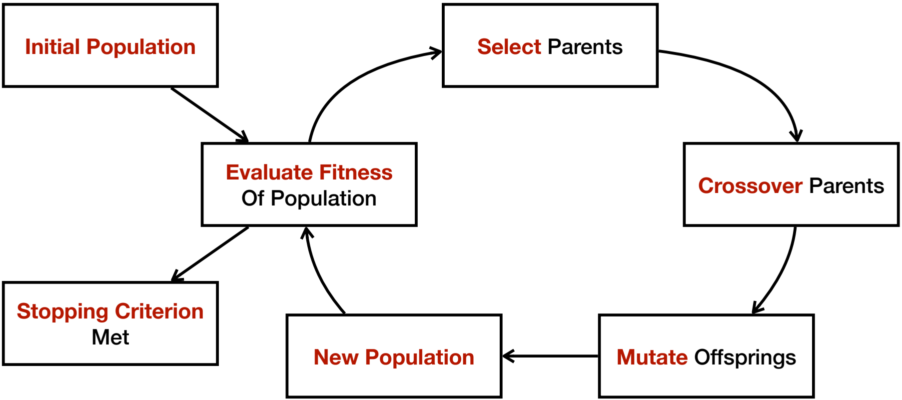
- This is, usually, the **only** way **new genetic material** is introduced into the population.

- **Without mutation**, all we can do is **recombine** the genetic material that is already present in the **initial population**.

Genetic Algorithms – Mutation Operators

- The **mutation** operator makes small changes to the representation of an individual.
- This is, usually, the **only** way **new genetic material** is introduced into the population.
- **Without mutation**, all we can do is **recombine** the genetic material that is already present in the **initial population**.
- The effective way to define the mutation operator is highly **dependent on the problem domain**.

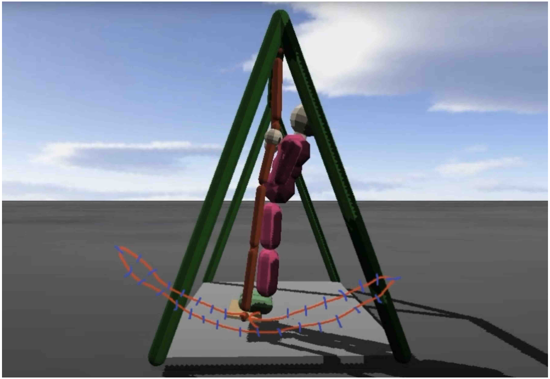
Genetic Algorithms



Genetic Algorithms – Example

One interesting example of GA is to **learn** how to **ride a swing**.

https://www.youtube.com/watch?v=Yr_nRnqeDp0



Let's split one cycle of the swing into 32 time steps and define 32-bit representation for the solution (**1** for **standing** and **0** for **sitting**).

Genetic Algorithms – Example

- **Knapsack Problem** – NP-hard problem
- **Travelling Salesman Problem (TSP)** – NP-hard problem
- **Program Synthesis** – Automatically generate programs
- **Program Repair** – Automatically repair buggy programs
- **Automotive Design** – Optimize the design of a car
- **Robotics** – Optimize the motion of a robot
- **Molecular structure optimization**
- **Protein folding prediction**
- etc.

Contents

Biomimicry

Imitation of the models, systems, and elements of **nature** for the purpose of solving **complex human problems**.

Biomimicry

Imitation of the models, systems, and elements of **nature** for the purpose of solving **complex human problems**.

- **Morpho Butterfly**
  - **Structural coloration** for the blue color
  - Mirasol display technology from Qualcomm is based on this

Biomimicry

Imitation of the models, systems, and elements of **nature** for the purpose of solving **complex human problems**.

- **Morpho Butterfly**
  - **Structural coloration** for the blue color
  - Mirasol display technology from Qualcomm is based on this
- **Burrs**
  - Swiss electrical engineer, George de Mestral, Had to remove **burdock burrs** (seeds) from his cloths and his dog's furs whenever he returned from walks in Alps.
  - Eventually, he invented **Velcro hooks** in 1951.

Let's apply the same idea to solve **software engineering problems**.

Bio-inspired – Particle Swarm Optimization (PSO)

- Let's mimic the behavior of a **flock of birds**!

Bio-inspired – Particle Swarm Optimization (PSO)

- Let's mimic the behavior of a **flock of birds**!

- Each bird is a **particle** in the search space.

Bio-inspired – Particle Swarm Optimization (PSO)

- Let's mimic the behavior of a **flock of birds**!
- Each bird is a **particle** in the search space.
- The goal is to find the **best position** (maximum food source) in the search space by **communicating** with other birds.

Bio-inspired – Particle Swarm Optimization (PSO)

- Let's mimic the behavior of a **flock of birds**!
- Each bird is a **particle** in the search space.
- The goal is to find the **best position** (maximum food source) in the search space by **communicating** with other birds.
  1. Each bird has an inertia to keep flying in the **same direction**.

Bio-inspired – Particle Swarm Optimization (PSO)

- Let's mimic the behavior of a **flock of birds**!
- Each bird is a **particle** in the search space.
- The goal is to find the **best position** (maximum food source) in the search space by **communicating** with other birds.
  1. Each bird has an inertia to keep flying in the **same direction**.
  2. Each bird remembers and has a tendency to return to the **local best position** it has ever **visited by itself**.

Bio-inspired – Particle Swarm Optimization (PSO)

- Let's mimic the behavior of a **flock of birds**!
- Each bird is a **particle** in the search space.
- The goal is to find the **best position** (maximum food source) in the search space by **communicating** with other birds.
  1. Each bird has an inertia to keep flying in the **same direction**.
  2. Each bird remembers and has a tendency to return to the **local best position** it has ever **visited by itself**.
  3. Each bird has a tendency to **follow** the known **global best position** in the flock by **communicating** with other birds.

Bio-inspired – Particle Swarm Optimization (PSO)

- Let's mimic the behavior of a **flock of birds**!
- Each bird is a **particle** in the search space.
- The goal is to find the **best position** (maximum food source) in the search space by **communicating** with other birds.
    1. Each bird has an inertia to keep flying in the **same direction**.
    2. Each bird remembers and has a tendency to return to the **local best position** it has ever **visited by itself**.
    3. Each bird has a tendency to **follow** the known **global best position** in the flock by **communicating** with other birds.
- **GA** is **competitive** vs. **PSO** is **cooperative**.

$$x_i^{t+1} = x_i^t + v_i^t$$

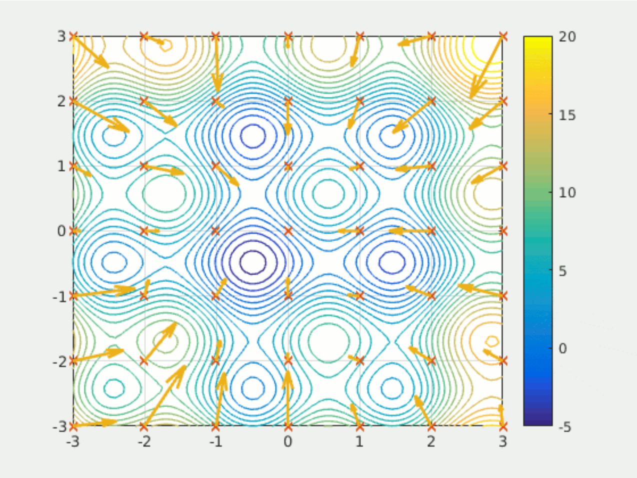
$$v_i^{t+1} = ① \, wv_i^t + ② \, c_1(p_i - x_i^t) + ③ \, c_2(g - x_i^t)$$

- x_i^t – position of the i-th particle at time t
- v_i^t – velocity of the i-th particle at time t
- p_i – best position of the i-th particle (local best)
- g – best position of the entire flock (global best)

It follows the three rules of the flock of birds.

1. Each bird has an inertia to keep flying in the **same direction**.
2. Each bird remembers and has a tendency to return to the **best position** it has ever **visited by itself** (**local best**).
3. Each bird has a tendency to **follow** the known **global best position** in the flock by **communicating** with other birds. (**global best)**

Bio-inspired – Particle Swarm Optimization (PSO)



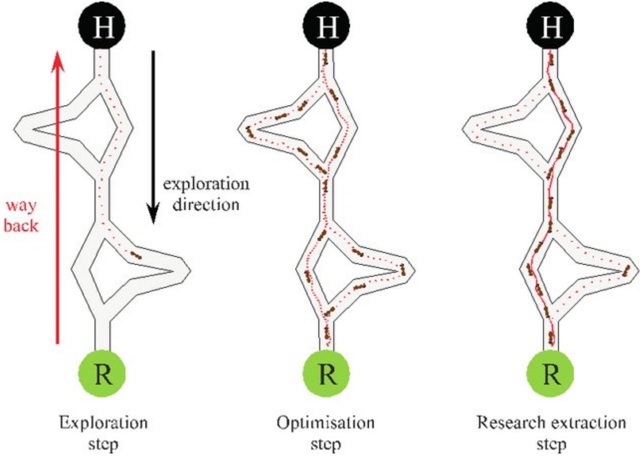
Link

Bio-inspired – Ant Colony Optimization (ACO)

Can we **mimic** the behavior of an **ant colony**?

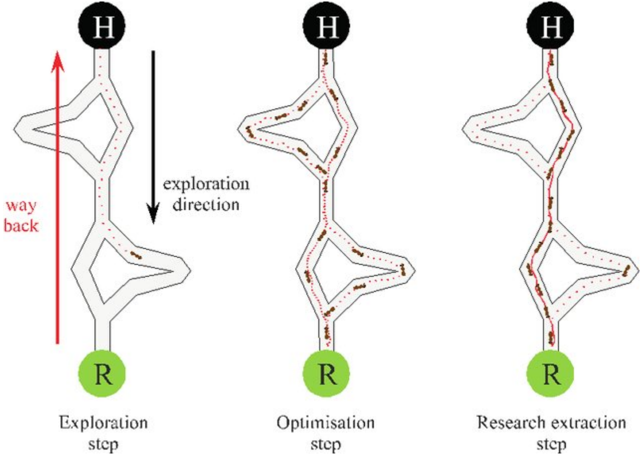
Bio-inspired – Ant Colony Optimization (ACO)

Ant colony utilizes a **pheromone** to **communicate** with other ants to find the **shortest path** to the food source.



Bio-inspired – Ant Colony Optimization (ACO)

Ant colony utilizes a **pheromone** to **communicate** with other ants to find the **shortest path** to the food source.



Bio-inspired – Ant Colony Optimization (ACO)

The **ant colony optimization (ACO)** algorithm is a meta-heuristic algorithm that is inspired by the foraging behavior of ants.

Bio-inspired – Ant Colony Optimization (ACO)

The **ant colony optimization (ACO)** algorithm is a meta-heuristic algorithm that is inspired by the foraging behavior of ants.

Let's consider the **TSP problem**.

Bio-inspired – Ant Colony Optimization (ACO)

The **ant colony optimization (ACO)** algorithm is a meta-heuristic algorithm that is inspired by the foraging behavior of ants.

Let's consider the **TSP problem**.

1. For **initialization**, we drop ants on **random nodes** on the graph, and deposit small amount of **pheromone** on all edges **uniformly**.

The **ant colony optimization (ACO)** algorithm is a meta-heuristic algorithm that is inspired by the foraging behavior of ants.

Let's consider the **TSP problem**.

1. For **initialization**, we drop ants on **random nodes** on the graph, and deposit small amount of **pheromone** on all edges **uniformly**.
2. Ants choose which edge to cross by considering the 1) **amount of pheromone** and 2) the **length of the edge**.

The **ant colony optimization (ACO)** algorithm is a meta-heuristic algorithm that is inspired by the foraging behavior of ants.

Let's consider the **TSP problem**.

1. For **initialization**, we drop ants on **random nodes** on the graph, and deposit small amount of **pheromone** on all edges **uniformly**.
2. Ants choose which edge to cross by considering the 1) **amount of pheromone** and 2) the **length of the edge**.
3. When ants finish a tour, the amount of pheromone on each edge is **updated** inversely proportional to the length of the tour.

Bio-inspired – Ant Colony Optimization (ACO)

The **ant colony optimization (ACO)** algorithm is a meta-heuristic algorithm that is inspired by the foraging behavior of ants.

Let's consider the **TSP problem**.

1. For **initialization**, we drop ants on **random nodes** on the graph, and deposit small amount of **pheromone** on all edges **uniformly**.
2. Ants choose which edge to cross by considering the 1) **amount of pheromone** and 2) the **length of the edge**.
3. When ants finish a tour, the amount of pheromone on each edge is **updated** inversely proportional to the length of the tour.
4. The amount of pheromone is slightly **evaporated** at each iteration.

Bio-inspired – Ant Colony Optimization (ACO)

The **ant colony optimization (ACO)** algorithm is a meta-heuristic algorithm that is inspired by the foraging behavior of ants.

Let's consider the **TSP problem**.

1. For **initialization**, we drop ants on **random nodes** on the graph, and deposit small amount of **pheromone** on all edges **uniformly**.
2. Ants choose which edge to cross by considering the 1) **amount of pheromone** and 2) the **length of the edge**.
3. When ants finish a tour, the amount of pheromone on each edge is **updated** inversely proportional to the length of the tour.
4. The amount of pheromone is slightly **evaporated** at each iteration.
5. By repeating the process, ants converge to the **shortest path**.

- **Probability of ant k choosing edge (i,j):**

$$p_{i,j}^k = \frac{(\tau_{i,j})^\alpha \cdot (\eta_{i,j})^\beta}{\sum_{h \in J^k} (\tau_{i,h})^\alpha \cdot (\eta_{i,h})^\beta}$$

where $\tau_{i,j}$ is the **amount of pheromone** on edge (i,j), $\eta_{i,j} = \frac{1}{d_{i,j}}$ is the **inverse of the length** of edge (i,j), and α and β are the parameters to control the **importance of pheromone and the length** of the edge. J^k is the set of nodes **not yet visited** by ant $1 \leq k \leq m$.

- **Probability of ant k choosing edge (i, j)**:

$$p_{i,j}^{k} = \frac{(\tau_{i,j})^{\alpha} \cdot (\eta_{i,j})^{\beta}}{\sum_{h \in J^k} (\tau_{i,h})^{\alpha} \cdot (\eta_{i,h})^{\beta}}$$

  where $\tau_{i,j}$ is the **amount of pheromone** on edge (i, j), $\eta_{i,j} = \frac{1}{d_{i,j}}$ is the **inverse of the length** of edge (i, j), and α and β are the parameters to control the **importance of pheromone and the length** of the edge. J^k is the set of nodes **not yet visited** by ant $1 \leq k \leq m$.

- **Pheromone update**: $\Delta\tau_{i,j} = \frac{Q}{L_k}$, where Q is the constant, and L_k is the length of the tour of ant k.

- **Probability of ant k choosing edge** (i, j):

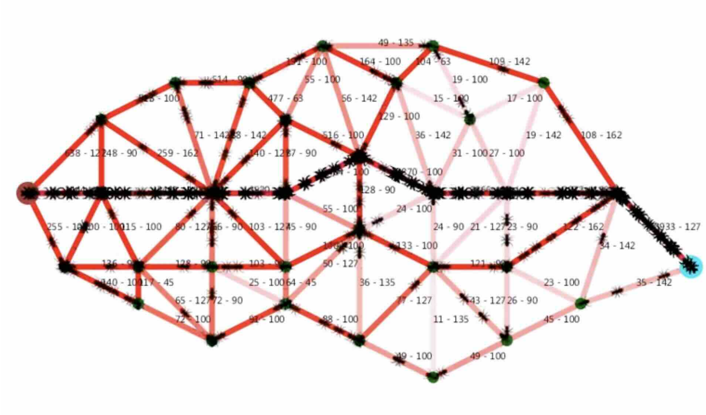
$$p_{i,j}^{k} = \frac{(\tau_{i,j})^{\alpha} \cdot (\eta_{i,j})^{\beta}}{\sum_{h \in J^k} (\tau_{i,h})^{\alpha} \cdot (\eta_{i,h})^{\beta}}$$

  where $\tau_{i,j}$ is the **amount of pheromone** on edge (i, j), $\eta_{i,j} = \frac{1}{d_{i,j}}$ is the **inverse of the length** of edge (i, j), and α and β are the parameters to control the **importance of pheromone and the length** of the edge. J^k is the set of nodes **not yet visited** by ant $1 \leq k \leq m$.

- **Pheromone update**: $\Delta\tau_{i,j} = \frac{Q}{L_k}$, where Q is the constant, and L_k is the length of the tour of ant k.

- **Evaluation**: $\tau_{i,j} = (1 - \rho)\tau_{i,j} + \sum_{k=1}^{m} \Delta\tau_{i,j}^{k}$, where ρ is the **evaporation rate**.

Bio-inspired – Ant Colony Optimization (ACO)



Link

Bio-inspired – Ant Colony Optimization (ACO)

- When the **graph changes**, the ACO algorithm can **adapt** with the **second-best** solution by **reusing** the pheromone.

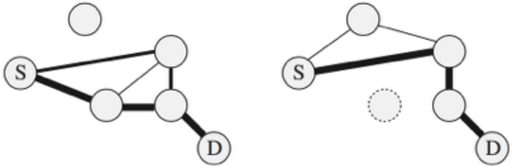


Figure 7.8 *Left*: Virtual ants maintain multiple paths between source and destination nodes. Shorter paths are traversed by more ants (thicker line). *Right*: If a node (or edge) fails, ants immediately use and reinforce the second shortest path available.

Dario Floreano and Claudio Mattiussi, Bio-inspired Artificial Intelligence, MIT Press

Bio-inspired

There are many other bio-inspired algorithms:

Bio-inspired

There are many other bio-inspired algorithms:

- **Artificial Immune System (AIS)** – Inspired by the human immune system to detect and eliminate **vulnerabilities** in computer systems.

Bio-inspired

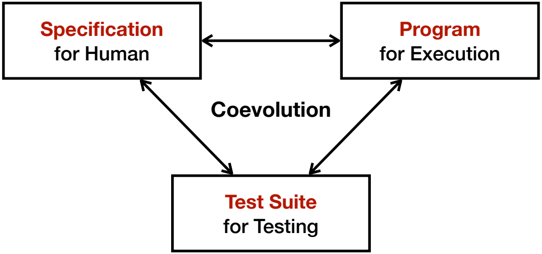
There are many other bio-inspired algorithms:

- **Artificial Immune System (AIS)** – Inspired by the human immune system to detect and eliminate **vulnerabilities** in computer systems.
- **Artificial Neural Network (ANN)** – Inspired by the human brain to solve complex problems.

Bio-inspired

There are many other bio-inspired algorithms:

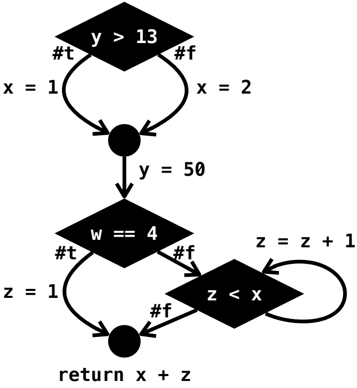
- **Artificial Immune System (AIS)** – Inspired by the human immune system to detect and eliminate **vulnerabilities** in computer systems.
- **Artificial Neural Network (ANN)** – Inspired by the human brain to solve complex problems.
- **Co-evolutionary Algorithms** – Inspired by the **co-evolution** of species in nature.



Contents

Search Based Software Testing (SBST)

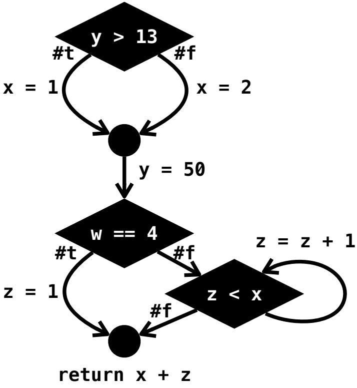
```
int foo(int x, int y, int w) {
  int z = 0;
  if (y > 13) { x = 1; }
  else { x = 2; }
  y = 50;
  if (w == 4) z = 1;
  else {
    while (z < x) { z = z + 1; }
  }
  return x + z;
}
return x + z;
```



- Our goal is to **automatically generate test cases** to **maximize** the **coverage** of the software under test.

Search Based Software Testing (SBST)

```
int foo(int x, int y, int w) {
  int z = 0;
  if (y > 13) { x = 1; }
  else { x = 2; }
  y = 50;
  if (w == 4) z = 1;
  else {
    while (z < x) { z = z + 1; }
  }
  return x + z;
}
return x + z;
```



- Our goal is to **automatically generate test cases** to **maximize** the **coverage** of the software under test.
- Let's apply the **search-based** approach to **software testing**!

Search Based Software Testing (SBST)

- Convert path conditions into a mathematical **fitness function**.

Search Based Software Testing (SBST)

- Convert path conditions into a mathematical **fitness function**.
- Use meta-heuristic algorithms to **maximize/minimize** fitness function.

Search Based Software Testing (SBST)

- Convert path conditions into a mathematical **fitness function**.
- Use meta-heuristic algorithms to **maximize/minimize** fitness function.
- When the goal is met, you have your **test case**.

Search Based Software Testing (SBST)

- Convert path conditions into a mathematical **fitness function**.
- Use meta-heuristic algorithms to **maximize/minimize** fitness function.
- When the goal is met, you have your **test case**.
- For example, we can define a **fitness function** for branch coverage as:

$$\textbf{[Approach Level]} + \text{normalize}(\textbf{[Branch Distance]})$$

  - **Approach Level –** The number of un-penetrated **nesting levels** surrounding the target branch.
  - **Branch Distance –** How close the input came to satisfying the condition of the target branch. For example, if the condition is $x + y == 10$, the branch distance is $|10 - (x + y)|$.

Example – Alternating Variable Method (AVM)

- The **alternating variable method (AVM)** is meta-heuristic algorithm to search for **test input vectors** that maximize/minimum a given fitness function.

Example – Alternating Variable Method (AVM)

- The **alternating variable method (AVM)** is meta-heuristic algorithm to search for **test input vectors** that maximize/minimum a given fitness function.

- Based on the known empirical results, AVM is one of the most effective algorithm for achieving C/C++ structural coverage.

Example – Alternating Variable Method (AVM)

- The **alternating variable method (AVM)** is meta-heuristic algorithm to search for **test input vectors** that maximize/minimum a given fitness function.

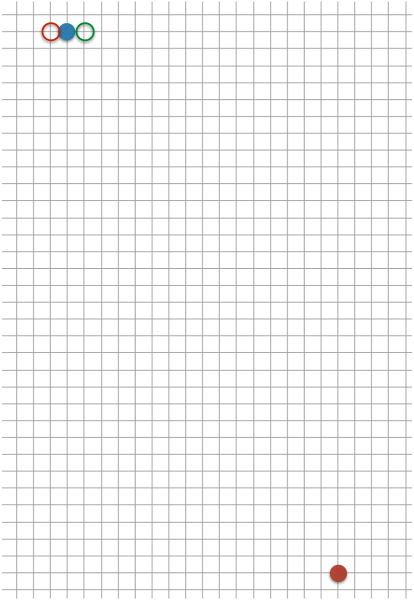
- Based on the known empirical results, AVM is one of the most effective algorithm for achieving C/C++ structural coverage.

- It has two operation modes:

  1. **Exploratory Move** – Decide **which direction** results in fitter solutions by exploring neighboring solutions.

  2. **Pattern Move** – **Accelerate** the search in the selected direction.

Example – Alternating Variable Method (AVM)

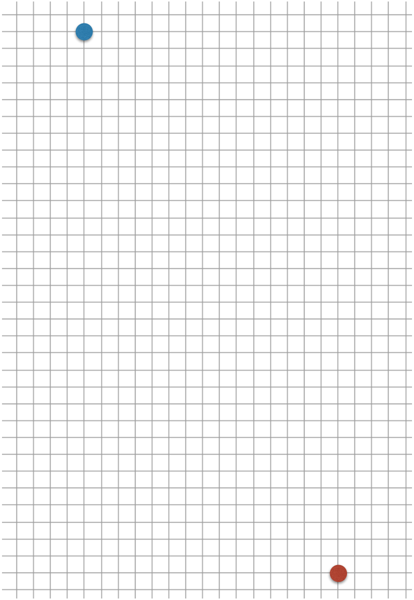


Our goal is to **minimize** the fitness function.

(x, y)	Δ	$f(x, y)$	▲/▼
$(5, 2)$	$(-1, 0)$	36.23	▲
$\mathbf{(7, 2)}$	$\mathbf{(1, 0)}$	**35.34**	▼

Exploratory Move – $x \uparrow$

Example – Alternating Variable Method (AVM)

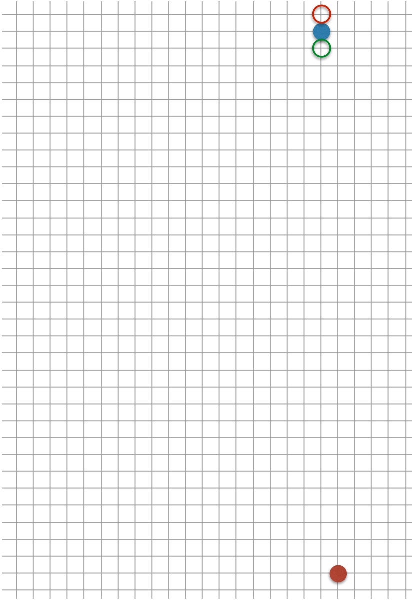


Our goal is to **minimize** the fitness function.

(x, y)	Δ	$f(x, y)$	▲/▼
$(7, 2)$	$(1, 0)$	35.34	▼
$(9, 2)$	$(2, 0)$	34.53	▼
$(13, 2)$	$(4, 0)$	33.24	▼
$\mathbf{(21, 2)}$	$\mathbf{(8, 0)}$	**32.01**	▼
$(37, 2)$	$(16, 0)$	35.34	▲

Pattern Move – $x \uparrow$

Example – Alternating Variable Method (AVM)

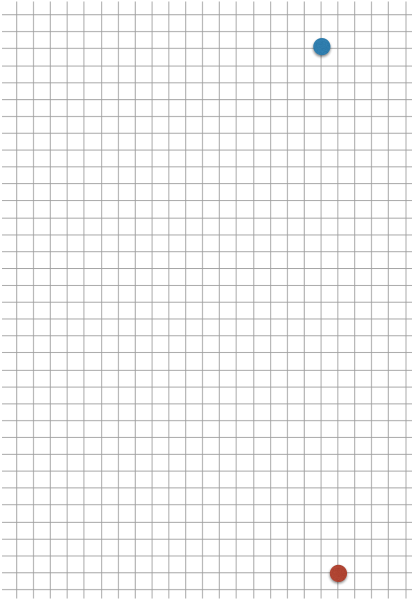


Our goal is to **minimize** the fitness function.

(x, y)	Δ	$f(x, y)$	▲/▼
$(21, 1)$	$(0, -1)$	33.01	▲
$\mathbf{(21, 3)}$	$\mathbf{(0, 1)}$	**31.01**	▼

Exploratory Move – $y \uparrow$

Example – Alternating Variable Method (AVM)

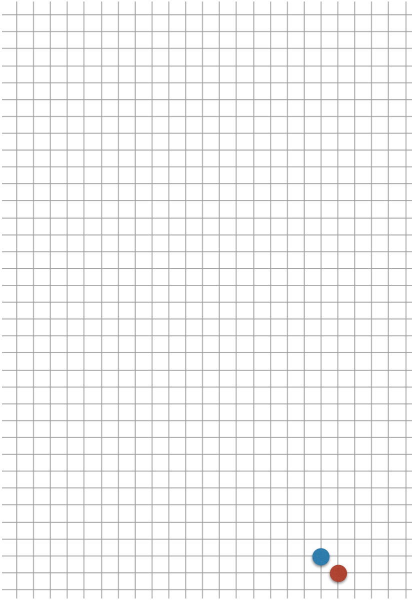


Our goal is to **minimize** the fitness function.

(x, y)	Δ	$f(x, y)$	▲/▼
$(21, 5)$	$(0, 2)$	29.01	▼
$(21, 9)$	$(0, 4)$	25.01	▼
$(21, 17)$	$(0, 8)$	17.02	▼
$\mathbf{(21, 33)}$	$\mathbf{(0, 16)}$	**1.41**	▼
$(21, 65)$	$(0, 32)$	26.03	▲

Pattern Move – $y \uparrow$

Example – Alternating Variable Method (AVM)



Our goal is to **minimize** the fitness function.

After one or two more iterations, we can find the **optimal solution**.

$$(x, y) = (22, 34)$$

Summary

1. Search Based Software Engineering (SBSE)
2. Fitness Landscape
3. Local Search
    - Hill Climbing
    - Simulated Annealing
    - Tabu Search
4. Genetic Algorithms
    - Selection Strategies
    - Crossover Operators
    - Mutation Operators
5. Bio-inspired Algorithms
    - Particle Swarm Optimization (PSO)
    - Ant Colony Optimization (ACO)
6. Search Based Software Testing (SBST)
    - Alternating Variable Method (AVM)

Next Lecture

- Dynamic Symbolic Execution (DSE)

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr