Week 1

Evolutionary Algorithms

Natural evolution achieves improvement by: (*) Small random changes (*) Natural selection (*) Replenishing population

> Nature-inspired algorithms are **stochastic** - randomness is added to escape local minima

Hill Climbing

Hill-climbing - maximization where a small change is applied to an individual $\mathbf{x} \in \mathbb{R}^D$ and the selection is based on some evaluation function, i.e., accept only if $F(\mathbf{x}') \geq F(\mathbf{x})$.

- **Operator formalism** - variation: instead of a single change, choose a class of operators to apply on $\mathbf{x}$, e.g., $O_d^{+\varepsilon}$ and $O_d^{-\varepsilon}$ add and subtract ε to element $\mathbf{x}_d$, respectively. **All-ones**: for $\mathbf{x} \in 0,1^D$ and $F(\mathbf{x}) = \mathrm{sum}(\mathbf{x})$, *hill-climbing* finds $\mathbf{x} = (1, 1, \dots)$.
- **Advantages**: Widely applicable, Usable as baseline or fine-tuning, **Disadvantages**: Tends to stuck in local optima, Does not acquire problem info

Week 2

Simulated Annealing

Annealing process in metals: at high temperatures atoms are random, at lower - crystalize. Crystals (lowest energy state) dominate after repeated heating and cooling. **Boltzmann distribution** - probability of a state x depends on energy F and temperature T (if $T \gg 0$, all x have nearly the same probability, if $T > \approx 0$, low energy x have the highest probability):

$$\rho(x) = \frac{\exp(-\frac{1}{T}F(x))}{\sum_{x'}\exp(-\frac{1}{T}F(x'))}$$

> If $T = \frac{T_0}{\log(1+t)}$, then optimal solution is found after long time, if temperature is reduced quicker, the algorithm may stuck. But it may be fine to reduce it faster if there are not many local minima after initial exploration.

	Simulated Annealing algorithm	**Genetic Algorithm**
1	Set: $T = 1$, $x = x_0$, $F(x) = \infty$	Generate initial candidate solutions (*bag of strings*)
2	Repeat for $t = 1, 2, \dots$	Repeat until termination condition:
3	Choose *solution*: $x_t \in O_k(x_{t-1}) \in X$ for random k	[*Intermediate generation*] Choose high-fitness solutions (based on objective)
4	Compute $F(x_t)$ and return *solution* if $F(x_t) < \Theta$	Calculate $\sum_d F_s(x_d)$ for each *solution* s
5	If *new cost* lower: $(\Delta F = F(x_t) - F(x_{t-1})) < 0$:	Rank solutions and select k best from N total
6	Continue	[*Next generation*] Modify some of them: breeding + mutation
7	Else:	Take a few solutions as parents, cross and re-glue
8	Accept *solution* with probability $\sim \exp(-\frac{1}{T}\Delta F)$ or set $x_t = x_{t-1}$	Generate copies of mixed solutions (e.g., 2 per mix) with few modifications
9	Update T in a suitable way	

> **Extra** Practical tips:

- Problem solved when *solution* is better (smaller) than Θ
- New solutions should be chosen in the neighbourhood, e.g., O_k flips k^{th} bit or a subset or is temperature-based
- Temperature should be changed such that fewer and fewer worse solutions are accepted

> **Simulated Annealing** works on populations with size $N = 1$, faster than random (though same complexity) but minimal-energy may not be reached even at low temperatures.

> **Note** Main idea of **SA**: bad solutions should not be abondoned immediately but more over time. It also cannot be proved that it is better than random search because of the deceptive functions.

Genetic Algorithms

Genetic Algorithms (GA) - cyclic nature-inspired algorithms that work on large populations, e.g., $N = 100$. There is current, intermediate and next generations. It is a type of **Evolutionary Algorithm (EA)** where an *individual* is a sequence of bytes $\mathbf{x} \in 0,1^N$ representing a *solution*.

Problem formulation: (*) Define *solution* representation in bits, e.g., $x = (\text{lengths of legs, age limits})$ (*) Define objective function to be computed for each x (*) Initial candidate solutions (*bag of strings*)

> Number of variables in *solution* should be a power of 2, otherwise, a lot of random solutions could be generated (due to bit-base)

> **Note** Some terms:

- **Reproduction** - copy *solution* into next generation
- **Crossover** - cut and glue, e.g., for single cut $ABDE \Leftrightarrow BADC$ we get $ABDC$ and $BADE$
- **Mutation** - change some values in *solution* to random new (~1% chance), e.g., one value change - $ABDC \Rightarrow ABCC$

> **Extra** **Crossover** can lead to large jumps in search space, keeps groups of good components together, brings innovation

Schema Theorem: if a problem has some level of compositionality (there are some substrings with wildcards, i.e., *schemata*, that everyone should have), then the algorithm finds this very quickly

Canonical GA

Roulette Wheel Selection - an *individual* (*genotype*, *chromosome*) is selected for *intermediate population* on each spin. Ratio of fitness to average fitness $f_i/\bar{f}$ corresponds to the number of offsprings. Two variants, both produce an unbiased sample of fitness in population:

- **Plain variant** (*stochastic sampling w.r.*) - selects i with $f_i/(n\bar{f})$ probability, e.g., if $f_i = \bar{f}$, then i "survives" with $1 - (1 - \frac{1}{n})^n$
- **Practical variant** (*remainder stochastic sampling*) - # offsprings is $\mathrm{int}(f_i/\bar{f})$ + one more with remainder probability $f_i/\bar{f} - \mathrm{int}(f_i/\bar{f})$

> Fitness values are often replaced by ranks making the selection process more natural

Crossover process: `1.` Shuffle population, ensure it is even, choose p_c (typically `0.7`) `2.` For every pair with probability p_c, choose index between 1 to $L - 1$ (*genotype* length), then cut and re-attach `3.` Move the obtained 2 children (or parent if no **crossover**) to the semi-new population

Mutation process: `1.` Finish **crossover**, choose p_m (typically `<0.01`, can be rank dependent) `2.` For all individuals loop through each *genotype* component and replace with a random one with probability p_m `3.` Move all the mutated/non-mutated individuals to a new population

> **Wary** Larger populations may improve exploration but may also contain many identical individuals (thus the need of mutations).

- **Advantages**: (*) Anytime behaviour - can stop and use current solution (*) Runs well on parallel machines (*) Easy to add new constraints: edit or penalise

- **Disadvantages**: (*) May be difficult to formulate (*solution* representation, fitness function) (*) Depends on initialization and selection pressure (*) Requires experimentation (form of **crossover**, **mutation**)

Week 3

Evolution Strategies

Evolution Strategies (ES) - one type of black-box optimization algorithms in the family of **Evolutionary Algorithms (EA)**. It is concerned with continuous representations (vectors of real numbers) of *solutions*. In contrast to **Simulated Annealing**, there is a population of *solutions*. **Simple Gaussian ES** - canonical version of **ES** where $p(\mathbf{x})$ is modelled as n-dimensional Gaussian distribution. From a populations of Λ *individuals*, λ offsprings are sampled and an elite subset of K best ones (based on fitness) is formed. It is used to estimate new $(\boldsymbol{\mu}, \boldsymbol{\sigma})$.

Offspring generation is a simple sampling from parent $\mathbf{m}$ with directional noise C (we can also compute mean of elite parents $\boldsymbol{\mu} \leftarrow \bar{\mathbf{m}}_{\text{elite}}$):

$$p(\mathbf{x}) \sim \mathbf{m} + \mathcal{N}(0, C)$$

Matrix C describes the preferred directions:

- $C = \text{diag}(\sigma, \dots, \sigma)$ - homogenous uncorrelated mutations (**Simple Gaussian**, isotropic: σI)
- $C = \text{diag}(\sigma_1, \dots, \sigma_D)$ - for gene-dependent mutation rates
- $C = (C_{ij})$ - full covariance matrix for correlated mutations (**CMA**)

Correlation Matrix Adaptation (CMA) ES - tracks pairwise dependencies between the samples in the distribution via covariance matrix C allowing to adjust exploration space more rapidly when the confidence level changes (in canonical **ES**, $\sigma^{(t+1)}$ was highly correlated with $\sigma^{(t)}$). Correlations among offsprings are computed as follows ($\mathbf{m}'$ - new elite state vector, e.g., $\bar{\mathbf{m}}$ or best $\mathbf{m}$):

$$C' = \frac{1}{K} \sum_{k=1}^{K} (\mathbf{x}_k - \mathbf{m}')(\mathbf{x}_k - \mathbf{m}')^\top$$

$$C \leftarrow (1 - \varepsilon) C + \varepsilon C'$$

> **Heuristic $1/5$ rule**: if less than $1/5$ of the children are better than their parents, then decrease the size of mutations: $C \leftarrow \rho C$, $\rho < 1$.

> **Wary** In **Evolution Strategies**, best *solution* is selected from noisy versions of initial *solutions* and the best strategy is problem-dependent; genetic encoding is not attempted unlike in **GA**.

Particle Swarm Optimization

Particle Swarm Optimization - a type of **EA** that emphasizes cooperation and communication which allows to form a *super-organism* (shows abilities not present in *individuals*) from the interaction of *individuals* (applications: movie effects, biological modelling, organic computing).

Reynolds' rules: (*) **Separation** - avoid collision with neighboring agents (*) **Alignment** - match the velocity of neighboring agents (*) **Cohesion** - stay near neighboring agents

> The direction of the update for the particle is unrelated (mostly) to other ones so **PSO** does not strictly follow **alignment** rule, like **ES**.

	PSO algorithm		**Differential Evolution algorithm**		**Genetic Programming algorithm**
1	Initialize:	1	Initialize similarly to **PSO**, also:	1	Choose a set of functions and terminals for the program to evolve
	$f : \mathbb{R}^D \to \mathbb{R}$: objective function		$F \in [0, 2]$: amplification factor of the differential variation		$/, *, +, -, <, >$, if: non-terminals
	N: number of particles		p: **crossover** probability		$x, y, -10, \dots, 10$: terminals
	$\mathbf{x}_n \in \mathbb{R}^D, \mathbf{v}_n \in \mathbb{R}^D, i = 1, \dots, N$: particle positions and velocities	2	**Mutation**: update velocities with 3 random ($\text{index} \neq n$) parents:		$N = 10000,\ p_{\text{cross}} = 0.9$: other
	$\hat{\mathbf{x}}_n, \hat{\mathbf{g}}$: best so-far (*simple nostalgia*) and global best (*group norm*)		$\mathbf{v}_n^{(t+1)} = \mathbf{x}_q^{(t)} + F(\mathbf{x}_r^{(t)} - \mathbf{x}_s^{(t)})$	2	Generate $P(0)$ of N trees of maximum depth D
	$\omega, \alpha_q, \alpha_2$: parameters	3	**Crossover**: choose random numbers $\rho_d \in [0, 1)$	3	Calculate fitness for each program and rank
2	Create random vectors $\mathbf{r}_1$ and $\mathbf{r}_2$ with components $U = [0, 1]$		$u_{nd}^{(t+1)} = \begin{cases} v_{nd}^{(t+1)} & \text{if } \rho_d < p,\ x_{nd}^{(t)} \quad \text{otherwise} \end{cases}$	4	Apply **selection**, **crossover**, **mutation**
3	Update velocities based on own experience and good example:	4	**Selection**: compare particle with computed $\mathbf{u}$:		
	$\mathbf{v_n}^{(t+1)} = \omega \mathbf{v_n}^{(t)} + \underbrace{\alpha_1 \mathbf{r}_1 \odot (\mathbf{l_n} - \mathbf{x_n}^{(t)})}_{\text{own experience}} + \underbrace{\alpha_2 \mathbf{r}_2 \odot (\mathbf{g} - \mathbf{x_n}^{(t)})}_{\text{good example}}$		$\mathbf{x}_n^{(t+1)} = \begin{cases} \mathbf{u}_n^{(t+1)} & \text{if } \mathbf{u}_n^{(t+1)} > \mathbf{x}_n^{(t)},\ \mathbf{x}_n^{(t)} \quad \text{otherwise} \end{cases}$		
4	Update positions and compute new f_n:				
	$\mathbf{x}_n^{(t+1)} \leftarrow \mathbf{x}_n^{(t)} + \mathbf{v}_n^{(t+1)}$				
5	Update local and global best:				
	$\mathbf{l}_n \leftarrow \mathbf{x}_n \quad \text{if } f(\mathbf{x}_n) < f(\mathbf{l}_n)$				
	$\mathbf{g} \leftarrow \mathbf{x}_n \quad \text{if } f(\mathbf{x}_n) < f(\mathbf{g})$				

> **PSO** shows exploratory (searches broad space region) and exploitative (approaches local optimum) behavior thus appropriate parameters must be chosen, though larger α_1, α_2 and ω lead to less stability (but better exploration).

> **Extra** *Local best* introduces a force towards a past state and both bests have a similar effect to gradient. Additional applications: evolving structure and weights of neural networks, robot path planning, localization, channel equalization in communication systems.

PSO Algorithm Details

Initialization tips:

- ```
  X = low + (up - low) * np.random.rand(N, D) # positions       f_g = -np.inf               # global (for maximization)
  V = np.zeros_like(X.shape)                  # velocities      f_l = np.repeat(f_g, N)     # local - for every particle
  ```

- ```
  omega = 0.7                 # range (0.1, 0.9)
  alpha_1 + alpha_2 = 4       # range (1, 5) - usually both equal
  N = 25                      # range (20, 200) - more for more complex problems
  max_vel = 0.15 * (up - low) # not larger than 10-20% range of x
  ```

> During the algorithm execution, ensure different realization for random R. Updates for $\mathbf{x}$ and $\mathbf{v}$ can be synchronous and asynchronous.

> **Note** Convergence is not ensured (the algorithm may oscillate around global optimum) but it is not necessary for a good solution, otherwise it is possible to add a **hill-climbing** stage to **PSO**.

Axis bias - particles often stick to the *solution* coordinate axes - when $\mathbf{x}_n^{(t)}$ coincides with $\mathbf{l}$ or $\mathbf{g}$ (even noise terms $\mathbf{r}_1$ and $\mathbf{r}_2$ are ineffective) - making it difficult to further explore those *solution* subspaces. *Bias* only exists if the algorithm is convergent, disappears for critical parameters.

Ways to control the *swarm* (may introduce additional parameters!):

- **Global best updates** - synchronous (after all particles moved), asynchronous (each tyme a particle finds a good solution)
- **Velocity control** - clamping, add a term $\kappa(\mathbf{v} - \mathbf{v_{target}})$ to reduce $\mathbf{v}_{\text{target}}$ over time
- **Search space limits** - reflect particles when they hit boundary
- **Premature convergence** - to avoid it, add noise, increase parameters, restart

PSO separation variant - a repulsive term is added when updating velocity, i.e., $\mathbf{v_n^{(t)}} = \ldots + \alpha_3 Z$ with $\alpha_3 > 0$. Z is small when particles are spread (small repulsion) and large when particles get close (big repulsion). For small ε, dynamics can become unstable.

$$Z = -\sum_{n' \neq n} \frac{\mathbf{r_3} \odot (\mathbf{x_{n'}} - \mathbf{x_n})}{\varepsilon + ||\mathbf{x_{n'}} - \mathbf{x_n}||^2}$$

Predator-Prey PSO - predators are attracted to prey (global-best particle) and prey is repulsed by predators - useful to control exploration-exploitation.

Differential Evolution

Differential Evolution - an algorithm similar to **PSO** but doesn't involve direct noisification $(\mathbf{r})$. Instead, population is kept diverse by mutating it based on differences between any vectors than differences between particle and better positions in the search space. If F is critical, the variance of *particle* vectors $\mathbf{x}$ remains constant between generations. If the variance is very small, F should be increased above this value:

$$F_{\text{crit}} = \sqrt{\frac{1}{N}\left(1 - \frac{p}{2}\right)}$$

> **DE** is often used as a component in hybrid metaheuristic algorithms. It also only has 3 parameters: F, N, p.

Week 4

Genetic Programming

Genetic Programming (GP) (type of **EA**) - a technique for evolving programs. A *genome* is a program (represented by a *tree*) that produces an *individual*, i.e., it is not an encoding of it. **GP** automatically creates a general solution in the form of parameterized topology.

> **Note** Evolution of programs is open-ended: the search space cannot be exhausted because **GPs** do not use a fixed-length encoding.

Given a population of randomly generated programs $P(0)$, each p_n is run on some input and evaluated. Fitter programs are bred. `(*)` **Closure** - ensures return by dealing with edge cases: redefine division by `0` to return `np.inf`, overload operators to handle variable number of arguments. Use grammar rules (e.g., with components `op`, `var`, `(`, `)`, `x`) to help achieve closure. `(*)` **Sufficiency** - terminals need to be defined, set of non-terminals must be sufficiently large.

LISP - a functional program that can be expressed as a tree (non-terminals, operators are nodes and numbers/variables are leaves), e.g., $f(x) = x^2 + 3$ is expressed as `f(x)=(+(*xx)3)`. Programs can be constructed from a random bag of terminals and non-terminals.

Fitness is rated by using example labeled data and *cross-validation*. We compute **MSE** (mean of $||GP(x_d) - y_d||^2$), adjust and normalize:

$$F_{\text{adj}} = \frac{1}{1 + \text{MSE}}; \qquad F_{\text{norm}}(n) = \frac{F_{\text{adj}}(n)}{\sum_{n'} F_{\text{adj}}(n')}$$

Most fit individuals have fitness ≤ 1, least fit have ≥ 0. Once selection is done, further tasks are:

- **Crossover** - same as in **GA**, except, here subtrees are swapped
- **Mutation** - vary numbers, change symbols, *shrink* (subtree is replaced by one of its terminals), *hoist* (make subtree an *individual*)

Genetic Programming Examples

There are a few types of problems:

1. **Symbolic Regression**: find a program that produces $\hat{y}$ given $\mathbf{x}$.
   - `(*)` Terminals: $\mathbf{x}$ and constants `(*)` Operators: $+, -, *$ `(*)` Fitness: sum of absolute errors over test set
2. **Learning to Plan**: find a program to transform any initial state to goal state, e.g., rearrange characters into some word.
   - Terminals: `CS` (current stack's head), `TB` (highest correct char in stack), `NN` (next needed char)
   - Operators: `MS(x)` (move `x` from table to stack), `MT` (move `x` to table), `DU(exp1, exp2)` (do `exp1` until `exp2` becomes `TRUE`)
   - Fitness: captures correctness, efficiency, small tree; example generation: `(DU (MS NN) (NOT NN))`

> **Extra** **GP** is very useful if quite a bit about the problem is known and not much engineering is needed.

> **Wary** **GP** can be troublesome: ever increasing depth/size, stalled fitness, very long runs. Sometimes it's good to have unused code in *individuals* as it helps to conserve information that may be needed later.

Application areas:

- *Good approximate solution is satisfactory*: control, pattern recognition (can simplify DNNs), data mining
- *Difficult for humans to write programs*: cellular automata, multi-agent strategies, distributed AI
- *Black art problems*: synthesis of topology, quantum computing circuits
- *Practical*: security checks in transport, smart homes, financial trading rules

Week 5

Ant Colony Optimization

Ant Colony Optimization (ACO) - population-based search technique for the solution of combinatorial optimization problem. Idea: ants lay and follow *pheromone* trails (*solutions*) which decays over time to reach food. Shorter paths have bigger build up so ants follow it.

ACO for **TSP**: ants follow a probability rule and after a tour, *pheromone* levels are updated on each edge visited. $\tau(i,j)$ - favourability (*global*) to go $i \to j$, $\eta(i,j)$ - visibility (*local*), e.g., greedy $\frac{1}{d(i,j)}$. $\alpha = 1$, $\beta = 2$, $\sum_{k \in \text{Allowed}}$ normalizes over non-visited towns.

$$p(i,j) = \frac{\tau(i,j)^\alpha \eta(i,j)^\beta}{\sum_{k \in \text{Allowed}} \tau(i,k)^\alpha \eta(i,k)^\beta}$$

Pheromone evaporates on each iteration with constant ρ, density depends on M ants. Q is constant, L_m is length of $m's$ tour.

$$\tau_{ij} = \rho\tau_{ij} + \Delta\tau_{ij} \qquad \Delta\tau_{ij} = \sum_{m=1}^{M} \Delta_m \tau_{ij} \qquad \Delta_m \tau_{ij} = \begin{cases} Q/L_m & \text{if } m \text{ used edge}(i,j), \\ 0 & \text{otherwise} \end{cases}$$

> To initialize **ACO**, choose small *pheromone* strength, initialize their intensities on edges, position ants on different towns.

We have variants:

- **Ant System (AS)**: $\Delta\tau_{ij}$ is a contribution from all M ants, also sets $Q = 1$
- **Ant Colony System (ACS)**: *pheromone* updated globally by the best ant m^*: $\tau_{ij} \leftarrow \rho\tau_{ij} + \Delta_{m^*}\tau_{ij}$, where $\Delta_{m^*}\tau_{ij} = 1/L_{m^*}$
- **Max-Min Ant System (MMAS)**: like **ACS** but $\tau_{\min} \leq \Delta_{m^*}\tau_{ij} \leq \tau_{\max}$ and initializes $\tau_{ij} = \tau_{\max}$. In theory, $\tau_{\max} = \frac{1}{L^*(1-\rho)}$ where L^* - shortest tour length (optimistic initialization). $\tau_{\min}$ guarantees continuous exploration, $\tau_{\max}$ makes non-visited regions equally attractive

Bin Packing Problem: minimizing slack when putting N weighted items to bins of capacity C. Trials of item placements are evaluated. *Pheromone* indicates either item i is placed in bin j or that items ij are in the same bin (encourages grouping). Preference of largest item is used for *local* heuristics and $\Delta_{m^*}\tau_{ij}$ is switched between best-ever and interaction-best. Fitness that promotes full bins (minimize # bins Λ):

$$f(s) = \frac{\sum_{\lambda=1}^{\Lambda} \sum_{i \in \text{Bin}_\lambda} \left(\frac{w_i}{C}\right)^2}{\Lambda}$$

> **Local search** in **Bin Packing Problem**: from n_h bins larger items are tried to be taken and replaced with smaller ones in the rest of the bins. Then, if smaller items become available, they are greedily inserted into fuller bins.

ACO parameters: (*) N: 10 to 20, (*) ρ (evaporation constant): 0.9, closer to 1 for longer runs, (*) α, β: 1 and non-tunable because it should depend on local heuristics, (*) τ_0: non-tunable and not critical because of normalization but can be used to insert prior knowledge.

Whether **ACO** is useful, depends on a problem (its strength and weakness the merging of solutions). Applications include decision making but many of them are typically better solved with **RL**: (*) Data stream optimization (*) Protein folding (*) Bus routes, garbage collection, delivery routes (*) Autonomous driving (*) Project scheduling

The more complex the problem is, the less *pheromone* should be evaporated (ρ should be bigger) so that more information about the problem is accumulated over time.

No-Free-Lunch Theorem

No Free Lunch Theorem (NFLT): no shortcuts for solutions, all strategies perform similarly on average, best one depends on prior knowledge.

Optimization as **search**: we sample $d_M = (x_1, f_1), \ldots, (x_M, f_M)$ points and their fitnesses in discrete search space and the algorithm A tells where to look next $A(d_M) = d_{M+1}$. To measuere performance, algorithm is not needed: $\Psi_M(f, A) = \min_{m \leq M} f_m$. **NFLT** says:

$$\forall A, A' \in \mathcal{A} : \sum_{f \in \mathcal{F}} \Psi_M(f, A) = \sum_{f \in \mathcal{F}} \Psi_M(f, A')$$

NFLT is not applicable:

- If a probability distribution is applied over problem function, e.g., some have less noise, because some algorithms will be better than the others and they will be aligned with the probability distribution.
- If we can draw conclusions from the first N observations about the later ones (happens in practice), then algorithms can be adapted.
- If the algorithm is resampling points, because it could converge quickly, e.g., to a local optimum.

> For non-discrete search problems, **NFLT** is not applicable. We should specify the problem set to understand the algorithms that are being used, otherwise they're all equal, regardless of min/max

Week 6

Metaheuristic Optimization

Metaheuristic Optimization (MHO) algorithms - procedures designed to find aad generate a *heuristic* that provides a sufficiently good solution, often dependent on a set of random variables. Some examples: **GA**, **PSO**, **Cuckoo Search**.

> **MHO** achieves a balance between 2 opposed effects, e.g., *exploration* vs *exploitation*, *cooperation* vs *competition*, *global* vs *local* search etc. It is successful if it can provide a balance between *exploitation* of accumulated experience and *exploration* of the search space.

> **Wary** Not all **MHO** algorithms are **EAs** but all **EAs** can be considered a type of **MHO**.

Random Walk - undirected local search: $\mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}$ with population size 1 - at each timestep t, with $p_d = 0.5$ we do step ± 1. For $D = 1$, the ensemble of walks approaches *Gaussian* with $\sigma^2 \sim T$. Used to introduce diversity in **MHO** and find problem characteristics.

- For $D \leq 2$, we get arbitrarily close to global optimum $\mathbf{x}^*$ with probability 1. For $D > 2$, we need extra information.
- **Levy Flights** - a **Random Walk** with a step that follows **Levy** distribution: $P_{\text{Levy}}(\mathbf{x}) \sim a\mathbf{x}^{-\gamma}$ for $\mathbf{x} > \mathbf{0}$ and $\gamma \in (1, 3]$. Mean and variance are infinite. I.e., computed mean does not tell anything about future. Used for **MHO** algorithms for *scale-free* search.
- **Informed Search** - biased **Random Walks** that uses an information from the optimization algorithm to decide on candidate *solutions*; can change statistics based on fitness (**Adaptive Walks**) Examples: *best-first*, *A**.
- **Bayesian Framework** - uses a probability distribution that covers all the search space. Posterior is updated based on fitness: $P^{(t)}(\mathbf{x}|\mathbf{x}_n, F_n) = \frac{P(\mathbf{x}|\mathbf{x}_n, F_n)P^{(t)}(\mathbf{x})}{P(\mathbf{x}_n, F_n)}$ where $P^{(t)}(\mathbf{x}) = P^{(t-1)}(\mathbf{x}|\mathbf{x}_n, F_n)$.

Biologically and Physically Inspired Algorithms

Most biologically and physically inspired algorithms are like **PSO** (attraction to better results, though no dynamics $\alpha = 1, \omega = 0$) or **ACO** (probability rule to explore near optimum; preference of fitter solutions counteracted by randomness):

- **Bat Algorithm**: like **PSO** but involves noise events that occur with probability τ_n for every *individual* (all τ_n approach 1):
  - With $p = \tau_n$, sample $\rho \sim U[\rho_{\min}, \rho_{\max}]$. Then, $\mathbf{v}_n^{(t+1)} = \mathbf{v}_n^{(t)} + \rho(\mathbf{g} - \mathbf{x}_n^{(t)})$ as well as $\mathbf{x}_n^{(t+1)} = \mathbf{x}_n^{(t)} + \mathbf{v}_n^{(t+1)}$
  - With $p = 1 - \tau_n$, sample $\rho_L \sim U[-1, 1]^D$. Then, $\mathbf{x}_n = \mathbf{x}_{n'} + \rho_L \text{avg}(\mathbf{L})$. Note: L_n - noise strength reduced on each t
- **Artificial Bee Colonies**: like **PSO** (multiplies difference with random α), **DE** (adds differences) and **ACO** (prefers higher fitness *solutions*):
  - Given a set of bests $\mathbf{s}_n$ mutate each by mixing with a random *solution*: $\mathbf{s}_n^{(t+1)} = \mathbf{s}_n^{(t)} + \rho_B(\mathbf{s}_n^{(t)} - \mathbf{s}_{n'}^{(t)})$ with $\rho_B \sim U[-1, 1]^D$
  - Compute probability of selection $p_n = \frac{F(\mathbf{s}_n)}{\sum_{n'} F(\mathbf{s}_{n'})}$, accept $\mathbf{s}_n^{(t+1)}$ only if better
  - The fact that **ABC** has no adjustable parameters to control the dynamics of the solutions, is sometimes presented as an advantage
- **Cuckoo Search**: moves particle in two directions and swaps a fraction of bad solutions with random ones:
  - Move particle *locally* (with step width α and random factor $\rho \in 0, 1$): $\mathbf{s}_n^{(t+1)} = \mathbf{s}_n^{(t)} + \alpha\rho \odot (\mathbf{s}_{n'}^{(t)} - \mathbf{s}_{n''}^{(t)})$
  - Move particle *globally* (with $|\xi_d| \sim P_{\text{Levy}}$): $\mathbf{s}_n^{(t+1)} = \mathbf{s}_n^{(t)} + \alpha\boldsymbol{\xi}$
  - Beneficial properties of Levy flights that have more small random steps as well as relatively more large steps, so that they provide an interesting combination of exploitation and exploration
- **Spider Monkey**: hierarchical version of **ABC**, e.g., 40 *individuals* forming 5 subgroups. Groups are combined and split with probability:
  - *Individuals* are updated based on attraction to group/global best and attraction/repulsion from other *individual* or group best
  - *Global* and *local* bests are updated
- **Glow-Worm**: compares own fitness to *individuals* with neighborhood:
  - Calculate probability of *glow-worms* moving to neighbors and choose a neighbor to move towards them
  - Update neighborhood range to maintain a group-based competition
- **Harmony Search**: a special case of **ES** at zero temperature + random search:
  - Choose random best and check whether a change leads to improvement $s'_{nd} = s_{nd} + U[-\rho, \rho]$
  - Accept when better, with $p = \tau$, add new random best from some rang. τ and ρ decrease over time
  - As the noise range, governed by ρ, decreases over time, the current set of solutions contracts quickly to the region near best solution. This is counteracted by inserting now random solutions point with a small probability τ.
- **Physics- and Chemistry-based**: based on *principle of least action* (non-local optimization in space and time). Forces in physics are often given as gradients of potentials.
  - Particle starting at point $\mathbf{x}_1$ at time t_1 and reaching $\mathbf{x}_2$ at t_2 follows a trajectory that is an extremum of action integral
  - Physics systems are well understood but computational problems may not have physics counterpart or they may be local
- **Electromagnetism-like optimization (EMO)**: uses *Coulomb's law* (force between 2 particles like gravitation)

- Determine charge based on fitness, compute force $F_{n,n'}$ based on *EM* law, then $\mathbf{x_n} \leftarrow \mathbf{x_n} + \sum_{n'=n} \frac{\rho F_{n,n'}}{|F_{n,n'}|}$
- Direction of force is a combination of towards the particle with higher fitness and from the one with worse
- If the global optimum is known to be likely near the centre of the search space, then the search is more efficient.

- **Spiral-Based Search**: a particle $\mathbf{x}$ spirals towards global optimum $\mathbf{x}^*$. More controllable but has poor coverage when more dims:
  - Rotate a vector pointing from $\mathbf{x}^*$ to $\mathbf{x}$ by ϕ and reduce distance by $g(\cdot)$: $\mathbf{x}^{(t+1)} = \mathbf{x}^* + g(|\mathbf{x}^{(t)} - \mathbf{x}^*|, t) R_{\phi(t)} (\mathbf{x}^{(t)} - \mathbf{x}^*)$
  - For $D > 2$, we can choose 2 dimensions randomly and apply $\begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}$ (or construct full rotation matrix)
  - Still the approach is largely focused on local search and would need some additional method for choosing "bests" in order to produce competitive results.

Continuous Metaheuristic Optimization algorithm

1	Initialize parameters
	$\hat{\mathbf{x}}_{n'}, \hat{\mathbf{x}}_{n''}$: *global* best, *group* best or other more/less fit *individuals*
	$\alpha, \omega, \boldsymbol{\rho}$: often 1, 0, random from $[\rho_{\min}, \rho_{\max}]$, respectively
	$\boldsymbol{\xi}$: a random solution to replace one *individual* when $\kappa = 0$
2	For a population of vectors, update each:
	$\mathbf{v}_n^{(t+1)} = \omega \mathbf{v}_n^{(t)} + \alpha \sum_{n',n''} \boldsymbol{\rho} \odot \left(\hat{\mathbf{x}}_{n'}^{(t)} - \hat{\mathbf{x}}_{n''}^{(t)} \right)$
	$\mathbf{x}_n^{t+1} = \begin{cases} \mathbf{x}_n^{(t)} + \mathbf{v}_n^{(t+1)} & \text{with prob } 1-p \\ \kappa \mathbf{x}_n^{(t)} + \boldsymbol{\xi} & \text{with prob } p \end{cases}$

Extra Both biology- and physics-based algorithms are designed based on low dimensional intuition. Biological optimization is usually w.r.t. a niche thus physics-based algorithms generalize better than biological ones (except neural networks).

Week 7

Multi-Objective Optimization

Multi-Objective Optimization - optimization over several (weighted) objective functions, which can be conflicting. Population-based algorithms are particularly useful due to simultaneous search.

Each point in search space $\mathbf{x} \in \mathbb{R}^D$ corresponds to a point in objective space $\mathbf{f} \in \mathbb{R}^K$ for K objectives. We are concerned with a set X of optimal points. For instance:

- Given $\mathbf{x} \in \mathbb{R}^{D=1}$ and objectives $f_1(\mathbf{x}) = 2^{x_0}$ and $f_2(\mathbf{x}) = -x_0^2$, and equal-weight scalarization $f_{1/2} = \frac{1}{2} f_1 + \frac{1}{2} f_2$, we can solve:
- Global maximum is at $\text{argmax}_{\mathbf{x}} f_{1/2}(\mathbf{x}) \approx 0.5$ for $x_d \in [0, 1]$ (if space is too large, we can ignore some objective functions)

Improvement is selected w.r.t. either objective but *diversification* is maintained - the need to find solutions that correspond to different combinations of the fitness

Pareto front - the set of *Pareto efficient* solutions, i.e., there's no other solution that would make all objectives better off - improving one would deteriorate at least one other. $\mathbf{x}^*$ is *Pareto optimal* if there is no $\mathbf{x}$ such that $f_i(\mathbf{x}) \geq f_i(\mathbf{x}^*)$ for all i.

Extra **GAs** are useful when searching for **Pareto front** - collective search in sampling, non-connected sets and fitness constraints are possible, however, it is not as efficient in high-dimensional problems and it's difficult to select fit *individuals*.

MOO Algorithms

Hill Climbing and **Simulated Annealing** for **MOO** - start with empty set of *solutions*, repeatedly make step from one state to another and accept based on algorithm. If new solution is *Pareto efficient* w.r.t. S, then $S \leftarrow S \cup \mathbf{x}$. Apply anti-crowding; for **HC** restart.

Nondominated Sorting GA (NSGA-II) - sorts a population into different *fronts* based on **domination** and for each *solution* in each *front* computes a *crowding distance* (**Manhattan**). *Solutions* are selected based on their *rank* and *crowding distance*.

- A *solution* **dominates** another if it is better in at least one objective and not worse in any other.

There are 3 helper functions (in `crowd_distance_assignment` objectives are normalized):

- ```python
  def find_non_dominated_front(population):
      front = []

      for p in population:
        # Check if any solution dominates p, if not - append
        is_dominated = [dominates(q, p) for q in population]
        front += [] if any(is_dominated) else [p]

      return front
  ```

  ```python
  def fast_non_dominated_sort(population):
      fronts = []

      while population != []:
          # Keep finding non-dominated fronts until population is empty
          fronts.append(find_non_dominated_front(population))
          population = [p for p in population if p not in fronts[-1]]

      return fronts
  ```

- ```python
  def crowd_distance_assignment(I):
      # Init distance for each individual in front
      for x in I: x.distance = 0

      for k in range(len(I[0].objectives)):
          # Sort individuals based on kth objective
          I = sorted(I, key=lambda x: x.objectives[k])
          I[0].distance = I[-1].distance = float("inf")

          for i in range(1, len(I) - 1):
              # CD is a measure of how close an individual is to its neighbors
              I[i].distance += (I[i+1].objectives[k] - I[i-1].objectives[k]) ** 2

      return I
  ```

NSGA-II algorithm

1	$R^{(t)} = P^{(t)} \cup Q^{(t)}$: combine parents and children
2	$\mathcal{F} =$ `fast_non_dominated_sort` $(R^{(t)})$
3	Include *fronts* until the parent population is filled $\lvert P^{(t+1)} \rvert + \lvert \mathcal{F}_i \rvert \leq N$:
	$\mathcal{F}_i =$ `crowd_distance_assignment` $(\mathcal{F}_i)$
	$P^{(t+1)} = P^{(t+1)} \cup \mathcal{F}_i$
	$i = i + 1$

Hybrid Metaheuristics algorithm

1	Initialize pool of solutions P
2	While not terminated:
	$S \leftarrow \text{OutputFunction}(P)$:
	if $\lvert S \rvert > 1$
	$S' \leftarrow \text{SolutionCombinationMethod}(S)$
	else:

	NSGA-II algorithm		**Hybrid Metaheuristics algorithm**
4	Add the remainder i^{th} *front* to finalize parent $P^{(t+1)}$:		$S' \leftarrow S$
	$\mathcal{F}_i = \text{sort}(\mathcal{F}_i)$		$S'' \leftarrow \text{ImprovementMethod}(S')$
	$P^{(t+1)} = P^{(t+1)} \cup \mathcal{F}_i[0 : (N - \lvert P^{(t+1)} \rvert)]$		$P \leftarrow \text{InputFunction}(S'')$
5	Use **selection**, **crossover**, **mutation** to create new population (standard **GA**):	3	Apply post-optimization procedure to P
	$Q^{(t+1)} = \text{make-new-pop}(P^{(t+1)})$		
6	$t = t + 1$		

Some alternative approaches (other than **dominance-based**): (*) **Priority-based**: order results lexicographically based on goal priority (*) **Stepping-stone-based** - optimize first what is most easily optimizable (*) **Indicator-based** - use performance indicator (e.g., distance from **nadir** point - worst point for each objective) to drive search

> **Extra** **MOO** provide an approach to analyse practical problems where various objectives have been identified by the customer.

Hybrid Metaheuristics

Metaheuristics - algorithmic frameworks for developing *heuristic* optimization algorithms which find near-optimal solutions with incomplete information or limited computation capacity. Given a set of potential solutions and a fitness function, they try to find a sampling procedure $G\left(s_{t=1}^{T-1}\right) \rightarrow f(s_T)$ such that $f(s_T)$ is near-optimal for $T = T_{\text{termination}}$. Examples: **SA**, **ACO**, **memetic algorithms**.

From a set of previous samples $s_{i\,t=1}^{T-1}$ we can infer: (*) Search direction (*) Step width (*) Prediction of new samples (*) Search dimensions

Memetic algorithms - a type of **metaheuristic** approach - a population-based algorithm with separate *individual* (local) learning. It is a cultural algorithm - based on social evolution. Can be **GA** with **ES**/**HC** for local search. There are several approaches: (*) **Lamarckian**: individual learning changes original representation (*) **Haeckelian**: social protection for some individuals (*) **Fisherian**: the rate of increase in teh mean fitness of any organism is equal to its genetic variance

Hybrid metaheuristic - an algorithm that combines **metaheuristic** with other optimization approaches. **Hybridization** can be done by combining with greedy heuristics, local search, complementary **metaheuristics**, mathematical programming, ML and data mining etc.

Classification **Hybrid Metaheuristics**:

- **Level**: A function of one **MH** is replaced by another **MH** (*low*) or two complete algorithms are cooperating (*high*)
- **Mode**: Two or more **MHs** are applied sequentially (*relay*) or is there a direct cooperation (*teamwork*)
- **Type**: Always the same **metaheuristics** is used (*homogeneous*) or a choice among several **MHs** is made (*heterogeneous*)
- **Domain**: All algorithms work on same search space (*global*) or on different problem aspects, e.g., single objectives in **MOO** (*partial*)
- **Function**: All algorithms work on the full problem (*generalist*) or just on aspects such as diversification (*specialist*)
- **Interaction**: The combination is fixed (*static*) or depends on the runtime properties of the algorithm (*adaptive*)

> Classic **memetic algorithms** can be considered as *low*-**level** *teamwork* hybrids

> **Extra** **Metaheuristics** can be combined with ML - good starting values for gradient-based algorithms can be provided by diversifying **MH**, parameters can be adapted by *greedy search*, representation of **Pareto front** can be improved using path-finding.

Hyperheuristics

Hyperheuristics - a *heuristic* search method that automates the process of selecting, combining, generating simpler *heuristics* to solve search problems. Unlike **metaheuristics**, they operate on a search space of *heuristics* rather than the search space of problem solutions. Unlike **Hyper metaheuristics**, they automatically find **hybridisation**. Two ways to use fitness values:

- **Online learning HH** - request new fitness from the problem (costly but flexible and accurate)
- **Offline learning HH** - learn from a set of training examples (requires large data and model, falis at non-stationarity)

Hyperheuristic GA - given, e.g., **PSO** with many terms (repulsion, alignment etc), add flags g_i to switch them on or off, e.g., $v_n \leftarrow g_0\omega v_n + g_1\alpha_1 R_1 \text{Expression } n_1 + g_2\alpha_2 \ldots$. We can use **GA** with various crossover and mutations to find best $\mathbf{g}$.

> **Wary** **Hyperheuristics** are computationally very demanding thus restricted to low-cost fitness functions.

> **Extra** Good **EA** algorithms (multi-objective) that have won the competitions in the recent past include **ES** + restart, **GA** + multiple parent crossover, **CMA-ES** (covariance matrix adaptation). Algorithms are compared based on 3 principles:
>
> 1. Problems for assessment are not used in development
> 2. Designer is equipped with domain-specific knowledge and similar studies
> 3. Algorithms are compared on equal computational effort

Week 8

Applications of Metaheuristic Algorithms

Application areas:

- **Combinatorial optimization**: determining optimal way to deliver packages, working out best allocation of jobs to people
- **Design problems in engineering**: behavior finding, developing airline network, designing water distribution network

Things to consider in example applications:

- **Flight scheduling**: cost, flight duration, total travel time, CO2
- **Load balancing in telecommunication**: performance, persistance, dynamics, fault tolerance
- **Wind farm layout**: cost, nominal power, maintenance, environmental impact

> An application using **metaheuristic** algorithms is successful if it outperforms other/traditional methods, enables success in new domains, can solve idealized problems, is actually used in industry/society

Information about the problem can help in designing the application: (1) fitness function, (2) search space representation, (3) algorithm:

- **Problem prior knowledge** - discrete/continuous, size of search space, fitness function(s), time scales
- **Application-related knowledge** - previous applications of other/same **MHOAs** in the domain of interest (or similar)
- **Experiences with the algorithm in domain** - properties of fitness landscape, rate of failure, parameter values, **hybridizations**

> **Extra** Successful applications include: recommendation systems, symbolic regression, feature selection, data mining, robot design

Actual Examples

- **Design of wall-floor building systems**: best layout for free and irregular wall arrangements (structural performance, easy fabrication)
- **Diagnosis of chronic disorders**: diagnosis of diabetes and cancer (need to consider noisy data, many subcases)
- **Optimization of flight connections**: finding optimal routes (need to consider time constraints, operators are swap, change airport etc)