AIML426 Assignment 2

Salvadalvi—300614650

October 5, 2025

1 Evolutionary Programming and Evolution Strategy

1.1 Method

1.1.1 Evolutionary Programming (EP)

The EP individual used in this algorithm is outlined in Algorithm 0. Each EP individual consists of an array x representing the solution vector for the optimisation functions $f1$ and $f2$. Associated with x is a corresponding mutation level array mut_lev, which determines the magnitude of change applied to each element of x during evolution. Two learning rates, τ and τ', are computed based on the problem dimension to control the rate of self-adaptation in mutation.

During each update, the mutation levels are adjusted using normally distributed random variables to introduce stochastic variation. The updated mutation levels are then used to generate new candidate solutions by adding scaled perturbations, ensuring that each updated value of x remains within its defined bounds.

This self-adaptive mechanism enables each individual to dynamically control its mutation strength throughout the evolutionary process, improving convergence and diversity. The class also includes a `clone()` method that allows deep copying of individuals for reproduction without altering the original.

Algorithm 1 Individual_EP Class

Class Individual_EP
function INITIALIZE($l\_bound, h\_bound, min\_mut, max\_mut, dimension, rng$)
    $self.l\_bound \leftarrow l\_bound$
    $self.h\_bound \leftarrow h\_bound$
    $self.rng \leftarrow rng$
    $self.x \leftarrow$ random uniform values between l_bound and h_bound for each dimension using rng
    $self.mut\_lev \leftarrow$ random uniform values between min_mut and max_mut for each dimension using rng
    $self.fitness \leftarrow \infty$
    $self.\tau \leftarrow \frac{1}{\sqrt{2\sqrt{dimension}}}$
    $self.\tau' \leftarrow \frac{1}{\sqrt{2 \times dimension}}$
end function
function UPDATE_MUT_LEVEL
    $a \leftarrow rng.normal(0, 1)$
    for $i \leftarrow 1$ **to** length of mut_lev **do**
        $b \leftarrow rng.normal(0, 1)$
        $old\_mut \leftarrow mut\_lev[i]$
        $mut\_lev[i] \leftarrow old\_mut \times e^{(\tau' \times a + \tau \times b)}$
    end for
end function
function UPDATE_X($dist$)
    Call UPDATE_MUT_LEVEL()
    for $i \leftarrow 1$ **to** length of x **do**
        $old\_x \leftarrow x[i]$
        $mutated\_x \leftarrow old\_x + mut\_lev[i] \times dist[i]$
        $x[i] \leftarrow$ clamp($mutated\_x$, l_bound, h_bound)
    end for
end function
function CLONE
    return a deep copy of the current individual
end function

The EP algorithm as suggested above in the individual, integrates elements from Fast EP and improved EP. To enhance mutation, a self-adaptive strategy is employed, while both normal and cauchy distributions are used to provide a more effective mutation scheme, allowing the algorithm to explore diverse regions of the search space. The overall algorithm is illustrated in 2.

The algorithm begins by initialising a population of pop_size individuals. In each generation, every individual produces two mutated offspring, one using a normal distribution and the other using a Cauchy distribution. This dual mutation strategy is employed because certain mutations are more effectively explored with one distribution over the other. The mutated offspring with the higher fitness is selected as the representative child and becomes part of the next generation.

At the end of each generation, the parent population and offspring are combined and sorted by fitness. The top pop_size individuals are retained for the subsequent

generation, ensuring high selection pressure, and the population continually evolves toward higher fitness solutions.

Algorithm 2 Evolutionary Programming (EP) Algorithm

dimension, population size pop_size, $fitness\_function$, l_bound, h_bound, mutation range $[min\_mut, max\_mut]$, max generations max_iter, seed final population, best individual per generation
rng = random.setSeed(seed)
pop = [EP_Ind for _ in range(pop_size)]
$fitness\_function(pop)$

for generation $\leftarrow 1$ max_iter **do** offspring = []
ind in population
ind_1, ind_2 = mutate_ind(ind, dimension, rng)
best_child = select_best_ind(ind_1, ind_2, fitness)
offspring.append(best_child)
combine = pop + offspring
combine.sort(by_fitness)
pop = combine(return pop_size)
return final population, best individuals per generation

The hyper parameter used for EP can be displayed in table 1.

Parameter	Value	Justification
Population size	100	100 population size enabled for a wide area of the search space to be covered, effectively finding the optimal solution.
Generation	100	From testing, 100 was perfect for the algorithm to find a optimum.
min_{mut}	0.1	0.1 min mutation allowed to maintain small adaptive changes and avoid premature convergence.
max_{mut}	1	max mutation allow broader exploration during the early stages of evolution.

Figure 1: Hyper parameters for EP algorithm

1.1.2 Evolutionary Strategy (ES)

The algorithm employs a self-adaptive ES, allowing mutation parameters to evolve alongside the individuals. This adaptive mutation mechanism facilitates faster convergence and reduces the need for manual parameter tuning. Additionally, the algorithm uses an $ES(\mu + \lambda)$ selection scheme, which maintains strong selection pressure by ensuring that the best individuals even from the parent population are retained in the next generation. This approach is particularly important for f_1 and f_2, given their unbounded search spaces.

It is important to note that each individual consists of x_n, representing the solution values, and σ_n, representing the mutation step sizes associated with each x_n. During mutation (illustrated in Figure 2), a scalar p_0 and a vector p_n are sampled from a stan-

dard normal distribution $N(0, 1)$. These values are then used to update both x_n and σ_n, generating the new mutated individual.

$$ind = \begin{bmatrix} x_0 & x_1 & \dots & x_n \\ \sigma_0 & \sigma_1 & \dots & \sigma_n \end{bmatrix}$$

Figure 2: Individual representation for ES

The Evolution Strategy (ES) algorithm is presented in 3. The process begins by initialising a population of size μ, which is then evaluated for fitness. For each of the max_iter generations, an offspring array of λ individuals is created. Each offspring is generated by selecting two parents and performing crossover. The offspring are then mutated, evaluated for fitness, and added to the offspring array. Once the offspring array is complete, it is combined with the parent population, and the next generation is formed by selecting the top μ individuals. This process is repeated until the maximum number of generations, max_iter, is reached. This strategy, as explained before, ensures that selection pressure is kept, enforcing better individuals.

Algorithm 3 Evolution Strategy (ES) Algorithm

population size μ, offspring size λ, crossover probability $cxpb$, mutation probability $mutpb$, max generations max_iter final population, best solution
pop = init_pop(μ)
evalute_fitness(pop)

for generation $\leftarrow 1$ max_iter **do** offspring = []

while number of offsprings $< \lambda$ **do**
p1, p2 = select_parents(pop, 2) children = mate(p1,p2)
child in children mutate(child) evaluate_fitness(child)
offsprings.add(children)
combine = pop + offspring
pop = selectBest(combine, μ)
return pop

Parameter	Value	Justification
Generation	100	From testing, 100 generation provided more stable fit results.
(offspring size) λ	80	Chosen offspring size allowed for a greater exploration in the search space
μ	30	Chosen value allowed only a select best individuals to be mutated for the next generation, increasing selection pressure.
p_{min}	0.001	Set to maintain a steady baseline mutation rate throughout evolution.
p_{max}	0.1	Selected to cap mutation intensity, preventing instability in offspring generation.

Figure 3: Hyper parameters for ES algorithm

1.2 Results & discussion

–	D=20	D=50
f_1		
Mean	3.682754e+05	3.090507e+06
STD	3.914734e+05	1.524339e+06
f_2		
Mean	1.091286	1.211502
STD	0.033131	0.061194

Figure 4: Results for EP

–	D=20	D=50
f_1		
Mean	3.324667e+05	4.564804e+06
STD	4.481414e+05	2.121105e+06
f_2		
Mean	1.031442	1.225517
STD	0.020649	0.052774

Figure 5: Results for ES

1.2.1 EP: $f1$ & $f2$

For the Rosenbrock function ($f1$), increasing the dimension from D = 20 to D = 50 significantly worsens the performance of the EP algorithm. The mean fitness increases from $3.68 * 10^5$ to $3.09 * 10^6$, while the standard deviation also rises sharply, indicating more variability across runs. This behaviour is expected because Rosenbrock's function is highly non-linear, and higher dimensions create a more complex landscape that is harder for EP to navigate, leading to poorer and less consistent solutions. In contrast, for the Griewank function ($f2$), the increase in dimension results in only a modest increase in mean fitness (from 1.09 to 1.21) and a slight rise in standard deviation. Griewank's function has a more regular structure that allows EP to maintain relatively consistent performance even as the number of variables grows. Overall, these results demonstrate that EP's performance is sensitive to the number of variables, with higher dimensions generally increasing difficulty and variability, particularly for complex landscapes, whereas smoother or more regular functions are less affected.

1.2.2 EP: $D = 20$ & $D = 50$

For D = 20, the EP algorithm performs much better on $f2$ than on the $f1$. Specifically, the mean fitness for $f2$ is 1.09 with a low standard deviation of 0.033, whereas for $f1$, the mean fitness is $3.68 * 10^5$ with a much higher standard deviation of $3.91 * 10^5$. A similar trend is observed for D = 50, where $f2$ maintains a relatively low mean fitness of 1.21 and a moderate standard deviation of 0.061, whereas $f1$'s mean fitness rises dramatically to $3.09 * 10^6$ with a high standard deviation of $1.52 * 10^5$. These results highlight that the behaviour and performance of EP strongly depend on the characteristics of the objective function. Functions with smoother, more regular like Griewank allow EP to explore the search space more effectively and converge reliably, while functions with narrow valleys and steep gradients like Rosenbrock pose greater challenges, resulting in poorer convergence and higher variability across runs.

1.2.3 ES: $f1$ & $f2$

For the Rosenbrock function $(f1)$, increasing the dimension from $D = 20$ to $D = 50$ significantly impacts the performance of the ES algorithm. The mean fitness increases from $3.32 * 10^5$ to $4.56 * 10^6$, and the standard deviation rises from $4.48 * 10^5$ to $2.12 * 10^6$, indicating both worse performance and higher variability as the problem dimensionality increases. This is expected because Rosenbrock's function has a narrow, curved

valley that becomes increasingly difficult to navigate in higher dimensions, making it challenging for ES to converge to optimal solutions. For the Griewank function, the mean fitness increases modestly from 1.03 to 1.23, with a slight increase in standard deviation from 0.021 to 0.053. Although higher dimensions introduce more variables to optimise, ES is still able to explore the smoother landscape of Griewank effectively, resulting in relatively consistent performance. Overall, these results show that the behaviour and performance of ES are highly sensitive to problem dimensionality, with higher dimensions generally leading to worse fitness outcomes and more variability, particularly for functions with steep gradients and complex landscapes, whereas smoother and more regular functions are less affected.

1.2.4 ES: $D = 20$ & $D = 50$

For $D = 20$, the ES algorithm performs better on the Griewank function than on the Rosenbrock function. The mean fitness for $f2$ is 1.03 with a low standard deviation of 0.021, whereas for $f1$, the mean fitness is $3.32 * 10^5$ with a higher standard deviation of $4.48 * 10^5$. This indicates that ES can reliably find good solutions for $f2$, while its performance on $f1$ is both worse and more variable. Similarly, for $D = 50$, $f2$ maintains a relatively low mean fitness of 1.23 and a moderate standard deviation of 0.053, whereas $f1$'s mean fitness rises dramatically to $4.56 * 10^6$ with a high standard deviation of $2.12 * 10^6$. These results demonstrate that the behavior and performance of ES strongly depend on the characteristics of the objective function. Functions with smoother, more regular, or multi-modal landscapes like Griewank allow ES to explore the search space effectively and converge reliably, whereas functions with narrow valleys and steep gradients like Rosenbrock pose greater challenges, resulting in poorer convergence and higher variability across runs.

2 Differential Evolution and Particle Swarm Optimization Algorithms

2.1 Method

2.1.1 DE

DE algorithm, as shown in Algorithm 4, begins by initialising a population of pop_size individuals, each representing a candidate solution in the search space. The fitness of each individual is evaluated initially.

For each generation, the algorithm iterates through all individuals in the population. For each individual (called the agent), three distinct individuals are randomly selected from the population. A trial vector is created by combining these three individuals using the mutation formula, scaled by the factor F, and controlled by the crossover probability CR. A randomly selected index j_rand ensures that at least one component is mutated.

The trial vector is then evaluated, and a selection step determines whether the agent is replaced by the trial vector. Specifically, if the trial vector has a better (lower) fitness than the current agent, it replaces the agent in the population. This process continues for all individuals, and statistics are recorded at the end of each generation.

Through repeated mutation, crossover, and selection, the population gradually evolves toward high-quality solutions. By the final generation, the population is expected to have converged around optimal or near-optimal solutions.

Algorithm 4 Differential Evolution (DE)

procedure DE(dimension, fitness, pop_size, F, CR, max_iter)
    Initialize population pop with pop_size individuals
    for each individual ind in pop **do**
        Evaluate fitness: $ind.fitness \leftarrow fitness(ind)$
    end for
    Record initial statistics
    for generation $g = 1$ to max_iter **do**
        for each individual $agent$ in pop **do**
            Select three distinct individuals a, b, c from pop
            Create a copy: $y \leftarrow agent$
            Randomly select an index $j\_rand \in [0, dimension)$
            for $i = 0$ to $dimension - 1$ **do**
                if $i = j\_rand$ **or** $random() < CR$ **then**
                    $y[i] \leftarrow a[i] + F \cdot (b[i] - c[i])$
                end if
            end for
            Evaluate trial vector: $y.fitness \leftarrow fitness(y)$
            if $y.fitness < agent.fitness$ **then**
                Replace: $agent \leftarrow y$
            end if
        end for
        Record statistics for generation g
    end for
end procedure=0

2.1.2 PSO

The particles in the PSO algorithm can be represented as shown in 5. Each particle has a position, which represents a potential solution to either f_1 or f_2, and a velocity, which determines how its position changes in the next generation. Additionally, each particle keeps track of its personal best position and fitness, as well as the global best position and fitness found by the swarm. This information is used to update the particle's velocity, ensuring that the particle moves toward optimal regions in the search space, thereby improving its fitness over successive generations.

Algorithm 5 Particle Class for PSO

Particledimension, l_bound, h_bound

Initialize position: $position[i] \leftarrow$ random value in $[l\_bound, h\_bound]$ for $i = 1$ to $dimension$
$v_{max} \leftarrow 0.1 \cdot (h\_bound - l\_bound)$
Initialize velocity: $velocity[i] \leftarrow$ random value in $[-v_{max}, v_{max}]$ for $i = 1$ to $dimension$
$p\_best \leftarrow position$
$g\_best \leftarrow$ None
$fitness \leftarrow +\infty$
$b\_fitness \leftarrow +\infty$
procedure CALC_NEW_VELOCITY(r1, r2, a_co1, a_co2, inertia)
    $cog\_component[i] \leftarrow r1 \cdot a\_co1 \cdot (p\_best[i] - position[i])$
    $soc\_component[i] \leftarrow r2 \cdot a\_co2 \cdot (g\_best[i] - position[i])$
    $vel\_int[i] \leftarrow inertia \cdot velocity[i]$
    Update velocity: $velocity[i] \leftarrow vel\_int[i] + cog\_component[i] + soc\_component[i]$
end procedure
procedure CALC_NEW_POSITION(r1, r2, a_co1, a_co2, inertia)
    CALC_NEW_VELOCITY(r1, r2, a_co1, a_co2, inertia)
    Update position: $position[i] \leftarrow position[i] + velocity[i]$
end procedure

PSO algorithm, as shown in Algorithm 6, begins by initializing a swarm of $swarm\_size$ particles. Each particle is assigned a random position within the given bounds and an initial velocity. The algorithm also maintains records of each particle's personal best position and fitness, as well as the global best position and fitness found by the entire swarm.

During each iteration, the fitness of every particle is evaluated. If a particle achieves a fitness better than its personal best, its personal best position and fitness are updated. Similarly, if the particle's fitness is better than the global best, the global best position and fitness are updated. After evaluating all particles, the algorithm updates their velocities and positions based on their personal best and the global best, guided by acceleration coefficients and an inertia factor.

This iterative process continues for a specified number of generations. By the end of the optimisation, the particles are expected to converge around high-quality regions of the search space.

Algorithm 6 Particle Swarm Optimization (PSO)

procedure PSO(dimension, fitness, swarm_size, l_bound, h_bound, a_co1, a_co2, inertia, max_iter=100, seed=100)
    Initialize swarm: $swarm \leftarrow$ [Particle(dimension, l_bound, h_bound) for each particle in swarm_size]
    $best\_fitnesses \leftarrow []$
    $best\_position \leftarrow$ None
    $best\_fitness \leftarrow +\infty$
    for $iter = 1$ to max_iter **do**
        for each $particle$ in $swarm$ **do**
            Evaluate fitness: $particle.fitness \leftarrow fitness(particle.position)$
            if $particle.p\_best$ is None **or** $particle.fitness < particle.b\_fitness$ **then**
                $particle.p\_best \leftarrow particle.position.copy()$
                $particle.b\_fitness \leftarrow particle.fitness$
            end if
            if $best\_position$ is None **or** $particle.fitness < best\_fitness$ **then**
                $best\_position \leftarrow particle.position.copy()$
                $best\_fitness \leftarrow particle.fitness$
            end if
        end for
        ▷ Update velocities and positions
        for each $particle$ in $swarm$ **do**
            $particle.g\_best \leftarrow best\_position.copy()$
            $r1 \leftarrow rnd.random()$
            $r2 \leftarrow rnd.random()$
            $particle.calc\_new\_position(r1, r2, a\_co1, a\_co2, inertia)$
        end for
        Append $best\_fitness$ to $best\_fitnesses$
    end for
    return $(best\_position, best\_fitness, best\_fitnesses)$
end procedure

2.2 Results

–	D=20	D=50
$f1$		
Mean	4364.547269	2.615994e+06
STD	3092.123828	1.274122e+06
$f2$		
Mean	0.901779	1.095919
STD	0.091875	0.024335

Figure 6: Results for DE

–	D=20	D=50
$f1$		
Mean	12406.209840	6.213937e+05
STD	9805.772120	2.609382e+05
$f2$		
Mean	0.638000	1.094653
STD	0.173278	0.026917

Figure 7: Results for PSO

2.3 Discussion

2.3.1 DE: $f1$ & $f2$

For $f1$, the DE algorithm shows a significant increase in mean fitness as the number of variables increases from $D = 20$ to $D = 50$. Specifically, the mean rises from 4364.55 with a standard deviation of 3092.12 at $D = 20$ to $2.6159 * 10^6$ with a standard deviation of $1.2741 * 10^6$ at $D = 50$. This indicates that as the problem dimensionality increases, the search space becomes substantially larger, making it more difficult for DE to find optimal solutions consistently. The higher variability at $D = 50$ reflects less stable convergence and greater sensitivity to the initial population and mutation strategies. For $f2$, the mean fitness increases only moderately from 0.9018 (STD 0.0919) at $D = 20$ to 1.0959 (STD 0.0243) at $D = 50$. The standard deviation decreases slightly, suggesting that DE maintains stable performance even as dimensionality grows. This indicates that $f2$ is less sensitive to the increase in variables, and the algorithm can explore the search space effectively without large fluctuations in results. Overall, these results demonstrate that the number of variables strongly affects DE performance. For complex or highly variable functions like $f1$, higher dimensionality leads to increased mean fitness values and greater variability, indicating more challenging optimization. In contrast, for smoother or less complex functions like $f2$, DE remains relatively stable, highlighting that the algorithm's performance scaling depends on the landscape of the objective function.

2.3.2 DE: $D = 20$ & $D = 50$

For $D = 20$, the DE algorithm performs much better on $f2$ than on $f1$. The mean fitness for $f2$ is 0.9018 with a low standard deviation of 0.0919, indicating that DE consistently finds near-optimal solutions. In contrast, $f1$ has a mean fitness of 4364.55 with a high standard deviation of 3092.12, showing that DE struggles to converge and produces more variable results on this function. For $D = 50$, a similar trend is observed. DE achieves a mean fitness of 1.0959 with a standard deviation of 0.0243 on $f2$, demonstrating stable and reliable performance even in higher dimensions. Meanwhile, $f1$'s mean fitness rises sharply to $2.6159 * 10^6$ with a high standard deviation of $1.2741 * 10^6$, reflecting poor convergence and increased variability across runs. These results indicate that the behaviour and performance of DE strongly depend on the characteristics of the objective function. Functions like $f2$ with smoother or less complex landscapes allow DE to explore the search space effectively and converge reliably, whereas functions like $f1$ with steep gradients or more rugged landscapes challenge the algorithm, leading to higher variability and worse solutions. This demonstrates that DE is more effective on functions with regular or multimodal landscapes, while performance deteriorates for functions with narrow valleys and large variations.

2.3.3 PSO: $f1$ & $f2$

For $f1$, the PSO algorithm shows a substantial increase in mean fitness as the number of variables increases from $D = 20$ to $D = 50$. The mean rises from 12406.21 with a standard deviation of 9805.77 at $D = 20$ to $6.2139 * 10^5$ with a standard deviation of $2.6094 * 10^5$ at $D = 50$. This demonstrates that increasing dimensionality enlarges the search space, making it more challenging for PSO to consistently find optimal so-

lutions. The higher variability at $D = 50$ also indicates less stable convergence and greater sensitivity to particle initialization and velocity updates. For $f2$, the mean fitness increases moderately from 0.6380 (STD 0.1733) at $D = 20$ to 1.0947 (STD 0.0269) at $D = 50$. The decrease in standard deviation suggests that PSO maintains more consistent performance on $f2$ despite the higher dimensionality. This indicates that $f2$ is less sensitive to the number of variables, and the algorithm can explore the search space effectively without large fluctuations in results. Overall, these results illustrate that the number of variables strongly affects PSO performance. For complex functions like $f1$, higher dimensionality leads to greater mean fitness and higher variability, reflecting more challenging optimization. In contrast, for smoother or less complex functions like $f2$, PSO remains relatively stable, highlighting that its performance scaling depends on the characteristics of the objective function.

2.3.4 PSO: $D = 20$ & $D = 50$

For $D = 20$, the PSO algorithm performs significantly better on $f2$ than on $f1$. The mean fitness for $f2$ is 0.6380 with a standard deviation of 0.1733, indicating that PSO consistently finds near-optimal solutions. In contrast, $f1$ has a mean fitness of 12406.21 with a high standard deviation of 9805.77, showing that PSO struggles to converge and produces more variable results on this function. For $D = 50$, a similar trend is observed. PSO achieves a mean fitness of 1.0947 with a low standard deviation of 0.0269 on $f2$, demonstrating stable performance even in higher dimensions. Meanwhile, $f1$'s mean fitness rises dramatically to $6.2139 * 10^5$ with a high standard deviation of $2.6094 * 10^5$, reflecting poor convergence and increased variability across runs. These results highlight that the behaviour and performance of PSO strongly depend on the characteristics of the objective function. Functions like $f2$ with smoother or simpler landscapes allow PSO to explore the search space effectively and converge reliably, whereas complex functions like $f1$ with rugged or steep landscapes challenge the algorithm, resulting in poorer performance and higher variability. This demonstrates that PSO is more effective on functions with regular or multimodal landscapes, while its performance deteriorates on functions with narrow valleys and large variations.

3 Estimation of Distribution Algorithm

3.1 Method

3.1.1 Individual

The individuals in this algorithm are represented as arrays of length n, where each element I_n corresponds to an item in the knapsack. An I_n takes the value 0 if the item is excluded and 1 if it is included, as illustrated in Figure 8. This binary representation is ideal because it clearly indicates which items are selected, making it straightforward to evaluate fitness.

Moreover the representation compliments EDA very well which allows for easy probability distribution modelling. For example if item I, has a probability of $p(x_i = 1)$ being 0.7, then it is simple to set the generate to 1 with 70.

$$ind = \begin{bmatrix} I_0 & I_1 & \dots & I_n \end{bmatrix}$$

Figure 8: Individual representation for the knapsack problem

3.1.2 Fitness

The designed fitness function, can be expressed as a pairwise function 9. Since the goal of the knapsack problem is to maximise the Individual's weight, it was only natural to assign it as the fitness if it does not exceed the max capacity. On other hand, if it does go beyond the max capacity, the fitness should not be the same from the previous case, as we are not discouraging it going over weight. Originally the fitness for this case was assigning it a flat $-float('inf')$, this was not appropriate as it provided no measure of how overweight the individual was, and thus making it very random when it tries to find a optimal solution in this space. Instead the final fitness penalised the Individual's value by how over weight it is. This is effective as for example if given an individual that has a great value but is 1 over weight, the GA can still use and iterate from this solution. Finally a constant of 10 was applied so that more overweight penalizations do not dominate ones that are slightly overweight.

$$\text{f(W\_max}, I, x) = \begin{cases} \sum_{i=1}^{n} v_i \cdot x_i & \text{if } W_I \leq W_{\max} \\ \sum_{i=1}^{n} v_i \cdot x_i - (W_I - W_{\max}) * 10 & \text{if } W_I > W_{\max} \end{cases}$$

Figure 9: Fitness Function

3.1.3 Algorithm

EDA employed in this study follows the framework of Population-Based Incremental Learning (PBIL), as outlined in Algorithm 7. The algorithm begins by initializing a probability vector with uniform values of 0.5, representing the distribution from which individual solutions are sampled. In each generation, a population of n individuals is generated, where each genome is constructed according to the current probability distribution. The fitness of all individuals is then evaluated, and the population is sorted accordingly. This ordered population is subsequently used to update the probability distribution: the top N_best individuals reinforce the distribution toward promising regions of the search space, while the worst N_worst individuals push it away from less favourable regions. Finally, the probabilities are clipped within predefined bounds to prevent extreme values and maintain diversity within the search process.

Algorithm 7 Estimation of Distribution Algorithm (EDA)

items, item size, max weight, population size pop_size, generations $gens$, number of best $Nbest$, number of worst $Nworst$, probability limits $pmin$, $pmax$, η
best solution, best value, best solutions per generation, best values per generation
prob = np.full(item_size, 0.5)
best_solution, best_value = None, 0

for generation $\leftarrow 1$ $gens$ **do**
pop = population = (np.random.rand(pop_size, item_size) ¡ prob).astype(int)
calc_fitness(pop)
pop.sort(by_fitness)

for i = 1 $Nbest$ **do** b_best = population[-(i+1)]
prob = prob + η * (b_best - prob)

for i = 1 $Nworst$ **do** b_worst = population[i]
prob = prob - η * (b_worst - prob)
Clip probabilities to $[pmin, pmax]$
return overall best solution, best value, best solutions per generation, best values per generation

The hyper parameters of the EDA 10 can be found and justified.

Parameter	Value	Justification
Population size	200	A population size of 200 was appropriate.
Generation	100	From testing, 100 generation provided more stable fit results.
η	0.01	A small learning rate ensures gradual updates to the probability vector, helping the algorithm converge steadily without overshooting.
n_{best}	8	Selecting 8 top individuals balances exploitation of good solutions with maintaining diversity in the population.
n_{worst}	6	Considering 6 worst individuals allows penalising poor solutions without overly disrupting the probability model.
p_{min}	0.05	Setting the minimum probability at 0.05 prevents any bit from being strongly biased towards 0, maintaining exploration.
p_{max}	0.95	Limiting the maximum probability to 0.95 prevents premature convergence by avoiding overly confident assignments.

Figure 10: Hyper parameters for GA algorithm

3.2 Results & Discussions

This section details the results from 5 seeded runs for each knapsack problem.

3.2.1 Problem - 10_269

Running the EDA on the 10_269 dataset the 5 seeded runs resulted in a mean of 294.4 and a std of 0.489 table 11. The optimal results for this problem however is 295 displaying that the algorithm did not find the most optimal results, but did come very close, with a difference of 0.6. The low standard deviation underscores the robustness of the method, indicating that across multiple seeded runs, the algorithm maintains stability and repeatability.

The convergence graph 12 illustrates that the mean fitness initially exhibited instability around generation 15, before successfully stabilising and converging by approximately generation 20. This behaviour reflects a characteristic strength of EDA: its probabilistic model is capable of rapidly adjusting to the search landscape, allowing the population to explore diverse solution regions before refining towards promising areas. The early fluctuations suggest active exploration, where the algorithm is still sampling broadly and recalibrating its probability distribution. Once sufficient information is gathered, the model shifts towards exploitation, leading to a more stable convergence pattern.

10_269	
Mean	294.4
STD	0.489

Figure 11: Results for 10_269

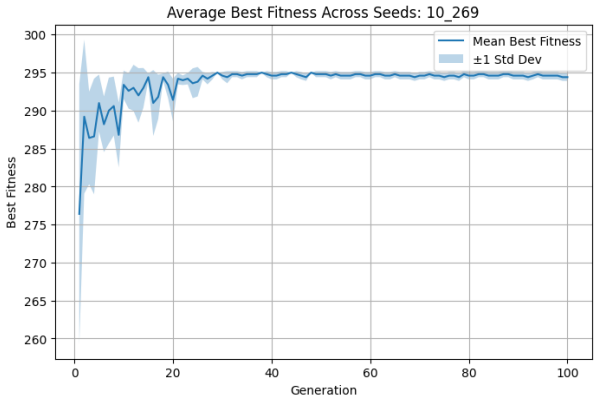

Figure 12: Convergence curve 10_269

3.2.2 Problem - 23_10000

The results of applying EDA to the 23_10000 dataset further demonstrate its effectiveness in addressing the knapsack problem. Across five seeded runs, the algorithm achieved a mean fitness of 9761.4 with a standard deviation of 2.332 (Figure 13). Given that the known optimal solution is 9767, the algorithm consistently produced near-optimal solutions. The relatively low standard deviation reinforces the stability of these outcomes, indicating that the algorithm delivers reliable performance across repeated runs.

The convergence graph (14) provides additional insight into the search dynamics. Here, EDA converges more slowly, stabilising only around generation 60. The ob-

served fluctuations in mean fitness throughout earlier generations reflect a degree of instability, which can be interpreted as the algorithm actively exploring diverse regions of the search space. Such behaviour highlights one of EDA's key advantages: its capacity to maintain exploration pressure before committing to convergence. While slower than in smaller datasets, this exploratory behaviour appears to prevent premature convergence, allowing the algorithm to approach near-optimal solutions consistently.

23_10000	
Mean	9761.4
STD	2.332

Figure 13: Results for 23_10000

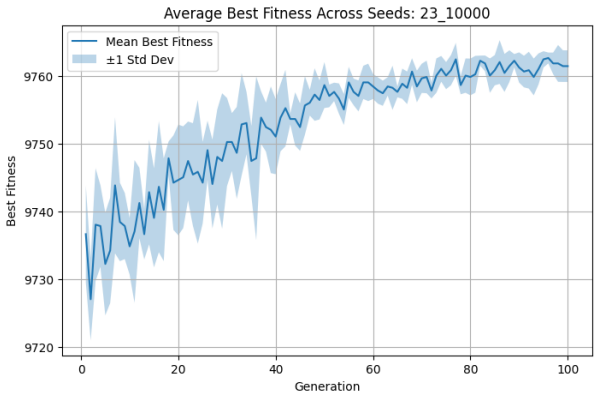

Figure 14: Convergence curve 23_10000

3.2.3 Problem - 100_995

In contrast, EDA performed significantly worse on the 100_995 dataset compared to the other problem instances. Across the five seeded runs, the mean fitness was 1025.8 with a standard deviation of 83.98 (Figure 15). Given that the known optimal solution is 1514, this result highlights the algorithm's difficulty in locating high-quality solutions within such a large and complex search space. The high standard deviation further indicates substantial inconsistency across runs, suggesting that solution quality was heavily influenced by the initialisation of individuals. This is expected, as each individual in this instance may involve up to 100 items, exponentially increasing the size and complexity of the search space.

The convergence graph (16) provides additional insight into this behaviour. For approximately the first 20 generations, the algorithm produced fitness values with large negative scores, reflecting that the best individuals during these early stages were infeasible solutions. While the algorithm eventually converged after generation 20, the quality of the final solutions remained far from optimal. This pattern suggests that in larger, more complex instances, EDA struggles not only with effective exploration but also with maintaining feasible solutions early in the search. Enhancing feasibility handling or incorporating hybrid methods such as repair heuristics or local search may therefore be necessary to improve performance on high-dimensional knapsack problems.

100_995	
Mean	1025.8
STD	83.98

Figure 15: Results for 100_995

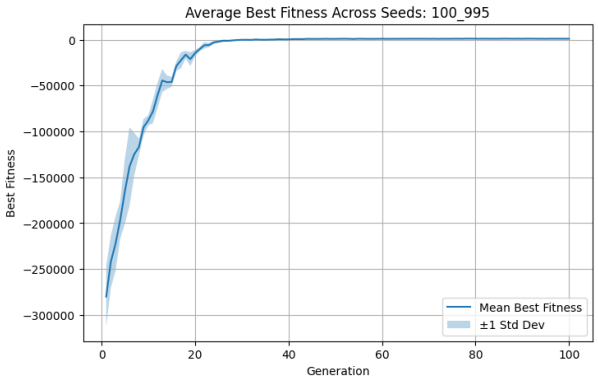

Figure 16: Convergence curve 100_995

3.3 Discussion

The results demonstrate that EDA performs strongly on small- to medium-scale knapsack problems, delivering near-optimal and stable solutions. For the 10_269 and 23_10000 datasets, the algorithm consistently achieved solutions very close to the known optima, with low standard deviations across runs. The convergence behaviour in both cases showed initial fluctuations followed by stable convergence, reflecting EDA's ability to balance exploration and exploitation effectively. This highlights the algorithm's robustness and adaptability when the search space remains reasonably constrained.

However, performance declined considerably in the 100_995 problem, where the algorithm struggled to approach the optimum and produced inconsistent results across runs. The larger search space, coupled with the presence of infeasible individuals in early generations, limited its ability to explore effectively and maintain feasible solutions. While convergence did occur, the final solutions remained far from optimal, underscoring the difficulty EDA faces in high-dimensional settings. These findings suggest that while EDA is effective for smaller problem instances, enhancements such as hybridisation with repair heuristics, local search, or adaptive parameter tuning may be necessary to improve its scalability to more complex knapsack problems.

3.3.1 Conclusion

The results demonstrate that the EDA, implemented using a binary representation and PBIL framework, is highly effective for small- to medium-scale knapsack instances. For the 10_269 and 23_10000 problems, EDA consistently achieved near-optimal solutions with low variance across seeded runs, highlighting its robustness and stability. The probabilistic modelling approach enabled effective exploration of the solution space, followed by steady convergence once promising regions were identified.

However, the performance on the 100_995 instance underscores the limitations of EDA when applied to high-dimensional search spaces. The algorithm struggled to approach the optimal solution, with results varying significantly across runs and early generations dominated by infeasible individuals. This suggests that, while EDA is well-suited for problems of moderate complexity, its ability to scale is constrained without additional mechanisms to manage feasibility and guide the search. Incorporating hybrid strategies such as repair heuristics, local search, or adaptive parameter tuning may help address these challenges and improve performance on larger problem instances.

4 Genetic Programming for Image Classification

4.1 Method

The overall flow of the FLGP follows the structure of a standard Genetic Programming (GP) algorithm, as illustrated in Algorithm 8. The primary distinction between a conventional GP and FLGP is that the individuals in FLGP act as image feature extractors. The fitness of each individual is determined by the accuracy of the features it extracts in predicting the class labels of the images. This process is described in detail in the following section.

The algorithm begins by initialising the population of individuals. The fitness of each individual is then evaluated based on its predictive performance. Over multiple generations, the algorithm iteratively selects parents using tournament selection and applies crossover and mutation operators to generate offspring. This cycle continues until the population is replenished with a new set of individuals, guiding the evolution toward increasingly effective feature extractors.

Algorithm 8 Genetic Programming Algorithm

```
1: function GA(mate_r, mut_r, ngen)
       pop = init_pop()
2:     calcFitness(pop)
3:     for g = 1 to ngen: do
       offspring = select(pop)
       crossover(offspring, mate_r)
       mutate(offspring, mut_r)
       calcFitnessOfIndWith noFitness(offspring)
       pop = offspring
4:     end for
       return pop
5: end function
       =0
```

4.1.1 Fitness function

The fitness function for the FLGP is presented in Algorithm 9. The fitness of an individual is evaluated based on the accuracy of the normalised features it extracts from

the training dataset using a LinearSVC classifier. This approach provides an effective measure of performance, as it directly reflects how well the selected subset of features contributes to accurate predictions.

Algorithm 9 FLGP Fitness Function

procedure EVALTRAIN(individual)
    $func \leftarrow$ compile the GP individual into a callable function
    $train\_tf \leftarrow []$
    for $i = 0$ **to** $len(y\_train) - 1$ **do**
        $output \leftarrow func(x\_train[i, :, :])$
        Append $output$ to $train\_tf$
    end for
    $train\_norm \leftarrow$ MinMaxScaler(train_tf) ▷ Normalize features between 0 and 1
    $model \leftarrow$ LinearSVC(max_iter=100)
    $accuracy \leftarrow 100 \times$ cross_val_score(model, train_norm)
    return $accuracy$
end procedure

4.1.2 Function & Terminals

The function and terminal set used by FLGP is shown by 17 and 18

Function	Justification
Feature Concatenation	
FeaCon2	Concatenates two feature vectors into one.
FeaCon3	Concatenates three feature vectors into one.
Global Feature Extraction	
Global_DIF	Extracts Difference of Intensity Features from the whole image.
Global_Histogram	Computes histogram-based features from the entire image.
Global_HOG	Extracts Histogram of Oriented Gradients globally.
Global_uLBP	Extracts uniform Local Binary Patterns across the image.
Global_SIFT	Extracts global SIFT descriptors.
Local Feature Extraction	
Local_DIF	Extracts Difference of Intensity Features from a region.
Local_Histogram	Computes histogram features from a local region.
Local_HOG	Extracts Histogram of Oriented Gradients locally.
Local_uLBP	Extracts uniform Local Binary Patterns from a region.
Local_SIFT	Extracts SIFT descriptors from a region.
Region Detection Operators	
Region_S	Detects regions based on a sliding window operator.
Region_R	Detects regions based on rectangular coordinates.

Figure 17: Function Set grouped by terminal type.

Terminal	Justification
Image Input	
Grey	Represents the greyscale image input for feature extraction.
Ephemeral Constants	
X	Randomly sampled horizontal coordinate within image bounds.
Y	Randomly sampled vertical coordinate within image bounds.
Size	Random region size sampled uniformly

Figure 18: Terminal Set grouped by type.

Parameter	Value	Justification
Population size	100	A population size of 100 was chosen as a balance between solution quality and stability. Testing showed that 50 individuals yielded less optimal results, while larger populations caused instability.
Generation	50	50 generations were sufficient to reach stable fitness results. Testing with 100 generations showed minor improvements but significantly increased computation time.
Tournament size	5	Provides a balance between selection pressure and maintaining diversity in the population.
Mating strategy	one point	Simple and effective crossover method suitable for evolving tree-based individuals.
Mutation strategy	genFull	Ensures diverse tree structures during mutation, helping explore different feature combinations.
Initial min depth	2	Prevents trivial trees while keeping initial individuals manageable in size.
Initial max depth	6	Allows sufficient complexity in initial trees for feature extraction.
Max depth	8	Limits tree growth to prevent overfitting and maintain computational efficiency.
Mutation probability	0.019	Low probability prevents excessive disruption while allowing occasional structural diversity.
Elitism probability	0.01	Ensures top individuals are preserved with minimal impact on exploration.
Crossover probability	0.08	Allows recombination of good solutions while avoiding excessive disruption of tree structures.

Figure 19: Hyper parameters for ES algorithm

4.1.3 Results format

The pattern was saved to a CSV file, with each column representing an extracted feature and the last column containing the class label. All feature values were normalized

to prevent any single feature from dominating the classifier during training.

4.2 Results and Discussion

4.2.1 Discussion Of Dataset accuracy

To evaluate the extracted features from the best individuals, a GassuianNB() classifier was first trained on the train dataset. The evaluation metric chosen for the pattern files is the accuracy for both train and test dataset. This was to give an accurate metric on how well it accurately predicted the values. table 20, shows the obtained accuracy between the dataset.

The table indicates that $FLGP$ performed exceptionally well in extracting the most relevant features, as reflected by the high accuracies achieved on both FEI_1 and FEI_2. This demonstrates that $FLGP$ is effective at accurately identifying the most informative features for classification.

It is also evident that $FLGP$ achieved significantly higher accuracy on FEI_1 compared to FEI_2, with test accuracies of 97% and 90%, respectively. This result is reasonable, as FEI_1 is a simpler dataset, while FEI_2 introduces greater variation in facial expressions and features, making the classification task more challenging.

The difference between the training (96%) and testing (90%) accuracies on FEI_2 suggests a slight degree of overfitting. Interestingly, for FEI_1, the model achieved even better performance on the test set than on the training set, indicating strong generalization on that dataset.

Dataset	Accuracy
FEI_1	
Train	97%
Test	98%
FEI_2	
Train	96%
Test	90%

Figure 20: Accuracy between f1 and f2

4.2.2 Discussion of GP individuals

The best GP individuals for both FEI_1 and FEI_2 datasets are displayed below.

Listing 1: GP individual for FEI_1

```
Local_SIFT(
    Region_R(Image0, 102, 35, 44, 45)
)
```

Listing 2: GP individual for FEI_2

```
FeaCon3(
    Local_uLBP(Region_R(Image0, 96, 19, 48, 22)),
    Local_uLBP(Region_R(Image0, 99, 19, 22, 22)),
    FeaCon3(
        Local_uLBP(Region_R(Image0, 81, 9, 49, 22)),
```

```
        FeaCon3(
            Local_SIFT(Region_R(Image0, 100, 19, 48, 22)),
            Local_SIFT(Region_R(Image0, 4, 90, 48, 24)),
            Local_SIFT(Region_S(Image0, 100, 61, 48))
        ),
        Global_HOG(Image0)
    )
)
```

The GP trees extract both local and global image features that are highly discriminative for classification. Local features include `Local_SIFT`, which captures distinctive keypoints and descriptors from specific regions (`Region_R` and `Region_S`), and `Local_uLBP`, which encodes micro-textures that are robust to lighting and minor facial variations. The global feature `Global_HOG` captures overall gradient and shape patterns across the entire image. Additionally, the `FeaCon3` operator allows multiple features to be combined, aggregating complementary information from different local regions and global structure. These features provide a rich representation of the images, ensuring that both fine-grained textures and holistic facial structures are captured.

The difference in complexity between the FEI_1 and FEI_2 GP individuals highlights how dataset characteristics influence feature selection. FEI_1 achieves good accuracy with a single `Local_SIFT` feature, suggesting that for this smaller or less diverse dataset, one highly discriminative local descriptor is sufficient. FEI_2, being larger and more diverse, requires a more complex tree combining multiple `Local_uLBP` and `Local_SIFT` features along with `Global_HOG` to handle variations in lighting, pose, and facial expression. The combination of local and global features in the FEI_2 GP individual provides complementary cues, making the classifier robust and enabling high classification accuracy across a wider range of images. This demonstrates that the GP effectively adapts feature selection and combination based on dataset complexity and size.