Estimation-of-Distribution Algorithms
Implementation Report (Tasks 1–6)

1 Introduction

This project implements and empirically evaluates a set of black-box optimizers for pseudo-Boolean functions: the significance-based compact genetic algorithm (sig-cGA), the compact genetic algorithm (cGA), and the (1+1) evolutionary algorithm ((1+1) EA). All algorithms optimize functions $f : \{0,1\}^n \to \mathbb{R}$ by repeated sampling and selection. The sig-cGA is based on detecting statistically significant drifts in per-bit histories and only allows frequencies in $\{1/n, 1/2, 1-1/n\}$, while the cGA performs incremental frequency updates using a parameter K (hypothetical population size). [1]

2 Task 1: Data structures and standard bit mutation

2.1 Individuals

An `Individual` stores:

- the problem size n,
- the bit string as a Python list `bit_values` of length n.

This representation gives $O(1)$ access to any bit and $O(n)$ iteration over the string (e.g., for fitness computation).

2.2 Standard bit mutation implementation

Standard bit mutation flips each bit independently with probability $1/n$. [1] The implementation follows a *two-step* strategy:

1. Sample the number of flips $k \sim \text{Binomial}(n, 1/n)$.
2. Uniformly sample `k` distinct indices and flip those bits.

Why runtime is linear in the number of flipped bits (after copying). After copying the bit array, the algorithm performs:

- sampling k and sampling `k` indices (`random.sample`) and
- flipping exactly `k` positions.

The post-copy cost is therefore $O(k)$ up to constant-factor overhead of sampling, and the loop that performs flips is *exactly* linear in k.

Expected runtime (disregarding the copy step). For $k \sim \text{Binomial}(n, 1/n)$, we have $\mathbb{E}[k] = n \cdot (1/n) = 1$. Hence the expected post-copy runtime is

$$\mathbb{E}[O(k)] = O(\mathbb{E}[k]) = O(1).$$

This matches the project requirement that a no-flip mutation should not cost $\Theta(n)$ after copying. [1]

2.3 Frequency vectors

A `Frequency_Vector` stores:

- n,
- a list `values` of length n, representing $\mathbf{p} \in [0, 1]^n$.

It supports:

1. **Sampling** an individual according to the univariate model: for each i, draw $x_i \sim \text{Bernoulli}(p_i)$ independently (Equation (1) in the sheet). [1]
2. **Updating** a single coordinate p_i with border restriction to $[1/n, 1 - 1/n]$ (Equation (2) in the sheet). [1]

3 Task 2: Significance function and history maintenance

3.1 Significance function

The implementation uses a function `sig(p, H, eps, n)` returning:

$$\texttt{UP} = 1, \quad \texttt{DOWN} = -1, \quad \texttt{STAY} = 0,$$

based on the threshold rule from the project sheet:

$$\text{limit} = mp + \varepsilon \cdot \max\{\sqrt{mp \ln n}, \ln n\},$$

where m is the history length and $|H|_1$ is the number of ones. [1] If $|H|_1$ exceeds the threshold (and $p \in \{1/n, 1/2\}$), we return UP; similarly, if $|H|_0$ exceeds the threshold (and $p \in \{1/2, 1 - 1/n\}$), we return DOWN; otherwise STAY.

3.2 Simplified history

`Simplified_History` stores only the sufficient statistics:

$$(m,\ |H|_0,\ |H|_1).$$

Adding a bit updates counters in $O(1)$ time. This history exposes exactly one subsequence (the full history), consistent with the simplified variant described in the sheet. [1]

3.3 Original history (linked list of blocks)

The original history is approximated via a singly linked list where each node summarizes a block:

$$(\texttt{size},\ \texttt{ones},\ \texttt{next}).$$

The method `get_subsequences()` returns cumulative prefixes, producing the exponentially growing subsequences required by the sig-cGA idea. [1]

3.4 Consolidation: maximum number of merges and a worst-case example

The consolidation procedure merges blocks when three consecutive blocks of equal size appear (as in Algorithm 2 of the sheet). [1]

Maximum number of merges. Each merge *doubles* a block size (from 2^j to 2^{j+1}). Therefore, a single insertion can trigger at most one merge per level j, i.e.,

$$O(\log m)$$

merges in the worst case, where m is the total number of bits summarized so far (equivalently, the total history length).

Instance achieving the maximum (merge cascade). A cascade occurs when, for many levels j, the list contains *two* blocks of size 2^j and an insertion creates the *third* block of size 2^j at the head, triggering a merge to size 2^{j+1}, which in turn creates a third block of size 2^{j+1}, etc. A concrete pattern (head to tail) is:

$$(1,1),\ (1,1),\ (2,2),\ (2,2),\ (4,4),\ (4,4),\ \ldots,\ (2^L,2^L),\ (2^L,2^L),$$

and then inserting a new size-1 block repeatedly until the head reaches size 1 again can trigger a chain of merges across $L = \Theta(\log m)$ levels.

4 Task 3: Full sig-cGA

The sig-cGA implementation follows Algorithm 1 from the sheet: [1]

1. Initialize $\mathbf{p}^{(0)} = (1/2, \ldots, 1/2)$ and empty histories H_i.
2. Repeat:
   (a) Sample two individuals $\mathbf{x}^{(t,1)}, \mathbf{x}^{(t,2)}$ from $\mathbf{p}^{(t)}$.
   (b) Select the better one (swap if necessary).
   (c) For each bit position i:
   - add the winning bit $x_i^{(t,1)}$ to history H_i,
   - scan each subsequence h provided by the history object,
   - apply `sig` and update p_i to $1 - 1/n$ (UP) or $1/n$ (DOWN),
   - reset the history on the first detected significance.

The code supports two history types: simplified and original.

Per-iteration cost (high-level). Sampling costs $O(n)$ for each individual (two individuals per iteration). History updates are $O(1)$ per bit, but scanning subsequences in the original history can add a $\Theta(\log m)$ factor in the worst case, yielding roughly:

$$O(n) \text{ sampling } + O(n \log m) \text{ (worst-case significance scans).}$$

In practice, histories reset often after significance, keeping m moderate.

5 Task 4: Competing algorithms (cGA and (1+1) EA)

5.1 cGA

The cGA implementation follows Algorithm 3. [1] Each iteration:

- samples two individuals,
- chooses the better one,
- updates each frequency:

$$p_i \leftarrow p_i + \frac{1}{K}\big(x_i^{(t,1)} - x_i^{(t,2)}\big),$$

  then clips to $[1/n, 1 - 1/n]$.

Per iteration cost is $O(n)$, dominated by sampling and the n coordinate updates.

5.2 (1+1) EA

The (1+1) EA follows Algorithm 4. [1] It maintains one current individual $\mathbf{x}$, generates one offspring via standard bit mutation, and accepts if fitness improves. Per iteration cost is the mutation cost plus fitness evaluation cost; using the mutation implementation from Task 1, the post-copy mutation overhead is expected $O(1)$ (the copy remains $O(n)$ in the current representation).

6 Task 5: Benchmarks and termination criterion

Three benchmark functions are implemented exactly as in the sheet: [1]

- **OneMax:** $f(\mathbf{x}) = |\mathbf{x}|_1$.
- **LeadingOnes:** number of consecutive ones from the start.
- **Jump$_k$:** plateau at $n - k$ ones and valley before the optimum.

Termination criterion:

- stop if an optimum $\mathbf{1}^n$ is produced, or
- stop after a fixed evaluation/iteration budget.

7 Task 6: Experimental evaluation and interpretation

7.1 Setup implemented in the notebook

The notebook runs multiple independent trials (`n_tries = 10`) using a fixed random seed for reproducibility. Budgets are fixed (e.g., $T = 3000$ total with an internal choice `eval = T/2` for the two-evaluation-per-iteration algorithms). Parameter sweeps include:

- sig-cGA: $\varepsilon \in \{12, 9\}$,
- cGA: K values around theoretical scales (e.g., $K_0 = \sqrt{n \ln n}$ on OneMax/Jump, and $K_0 = n \ln^2 n$ on LeadingOnes). [1]

7.2 Key results (from the notebook summary table)

The notebook reports three main metrics per configuration:

- **Success rate** (fraction of runs that reach the optimum within budget),
- **Mean runtime** (mean iterations/evaluations conditional on success; missing when no successes),
- **Mean best fitness** reached (helpful when success is low).

OneMax. For $n = 100$, both (1+1) EA and cGA reach 100% success within the budget; cGA is faster on average. For $n = 300$, success drops to 10–20% for both (1+1) EA and cGA under the fixed budget, indicating that the chosen budget is too small to reliably scale to $n = 300$.

LeadingOnes. All tested algorithms show 0% success for $n \in \{100, 200\}$ within the chosen budget. Mean best fitness values remain far from the optimum, which is consistent with LeadingOnes typically requiring larger budgets due to its sequential dependency structure (later bits do not matter until earlier ones are fixed). [1]

Jump. Jump is the most discriminating benchmark:

- For $n = 100, k = 2$, cGA reaches up to 100% success (best among tested methods), with mean runtime around a few hundred to ∼600–700.
- For $k = 3$ at $n = 100$, cGA still reaches high success (up to 90%) for larger K values.
- For $n = 300$, success collapses sharply (e.g., around 0–20%), showing strong scaling difficulty under the same fixed budget.

7.3 Interpretation: what we learn (not just what we see)

Budget dominates scaling. A single budget (here $T = 3000$) produces very different outcomes across n and across benchmarks. The sharp success drop from $n = 100$ to $n = 300$ on OneMax and Jump suggests that comparing algorithms fairly requires either:

- budgets that scale with n, or
- reporting performance curves (success vs. budget) instead of a single cut-off.

Parameter sensitivity of cGA. On OneMax and Jump at $n = 100$, cGA succeeds for multiple K values, but the *runtime* varies:

- too small K can cause noisy updates (risk of drift),
- too large K slows adaptation (small steps).

The best-performing K values in the table tend to be in the moderate-to-large range of the tested set, especially for Jump where robustness against the valley is helpful.

sig-cGA results indicate an implementation or configuration issue under the tested setup. In the provided runs, sig-cGA variants show 0% success on OneMax, LeadingOnes, and Jump within the fixed budget, and mean fitness values remain relatively low. Given that sig-cGA is expected to perform competitively on OneMax/LeadingOnes under appropriate conditions (and correct history handling), this outcome strongly suggests that either:

- the budget is far too small for the current implementation overhead, and/or
- the original-history linked list update/consolidation logic does not update the head pointer as intended (a common pitfall in Python when reassigning `self` inside methods), leading to ineffective history growth and thus no significant updates.

This is an important experimental finding: the *concept* may be strong, but correct and careful engineering of the history structure is essential.

8 Conclusion and recommendation (from Task 6 outcomes)

Within the tested budgets and parameter grids:

- **For OneMax at moderate n (e.g., $n = 100$):** cGA is the most efficient among the successful methods (higher speed at 100% success than (1+1) EA).
- **For Jump at $n = 100$:** cGA is clearly the best-performing tested method (high success up to 100% for $k = 2$ and up to 90% for $k = 3$), indicating strong robustness to deceptive landscapes in this regime.
- **For larger n (e.g., $n = 300$):** the fixed budget is insufficient; success rates drop sharply across algorithms, so conclusions should be made from scaled budgets or success-vs-budget curves.
- **For LeadingOnes:** none of the algorithms reached the optimum under the budget; more evaluations are required to compare meaningfully.
- **For sig-cGA:** the current implementation, as exercised in the notebook, does not reach optima within budget and likely needs a careful revision of the original-history data structure handling and/or a larger budget to reveal its intended behavior.

Overall recommendation. For the current codebase and experimental regime, the **cGA** is the best practical choice: it solves OneMax reliably at $n = 100$ and is the only method achieving high success on Jump under the tested settings. For a proper sig-cGA comparison, the next step is to validate the original-history implementation with targeted unit tests (block sizes, merges, and persistence of the updated head) and then re-run experiments with budgets scaling in n.

References

[1] CSC_42021_EP Project Sheet: "Estimation-of-Distribution Algorithms" (sig-cGA, cGA, (1+1) EA, benchmarks, and experimental requirements).