Exercise 3: Algorithmic Design Choices

Hai, An and Seth

November 2025

1 Single-Objective EA

1.1 Algorithm 1

I chose a single-objective, population-based classical GA under the assumption that successive generations can produce increasingly fit offspring. Parents are selected with `EliteNeighbour()`, which picks the candidate with the highest fitness $f(x)$, keeping this elite solution as a breeder accelerates convergence, as confirmed by our graph showing the fastest convergence when `EliteNeighbour()` is used. For variation, `KMutation()` flips a contiguous segment of bits of random length k where $1 \leq k \leq n/2$ (with n the solution dimension), and the segment's position is chosen uniformly at random. I prefer this over `UniformMutation()` (bitwise flips with probability p) because stronger, structured perturbations help the search escape local optima. To maintain feasibility, a `Repair()` operator is applied both when generating the initial population X^n and after each `KMutation()`. `Repair()` also evaluates fitness, preventing the algorithm from wasting evaluations in infeasible regions—as happens with (1+1)EA or RLS when repair is absent. For survivor/parent selection, `TournamentNeighbour()` introduces diversity by choosing the best among a random subset of candidates. I used a relatively large tournament size (10). On a population of 10, this is effectively elitist, on 50, it did not significantly change outcomes, suggesting our GA already converges very quickly due to elitism. While the implementation could be faster such as caching fitness values, the design intentionally prioritises rapid convergence through elitism and tournament-based selection.

1.2 Algorithm 2

I chose a single-objective, population-based Fast GA under the assumption that variable-strength mutations and fast repairs can efficiently navigate the search space of monotone submodular problems. Parents are selected with `tournament_select_fast()`, which picks the feasible candidate with the highest fitness $f(x)$ (or least infeasible if none feasible), this introduces controlled selection pressure via a tournament size of 3, balancing exploration and exploitation without full population sorting. For variation, `uniform_crossover()` mixes bits randomly from two parents, which is preferable for graph problems over one-point crossover to preserve local structures, and `mutate()` applies power-law distributed k-bit flips (with $\beta = 1.5$ biasing small k), enabling both fine refinements and occasional larger jumps to escape local optima, I prefer this over uniform mutation for its adaptability. To maintain feasibility, a `quick_repair()` operator is applied selectively during offspring generation only if infeasible and budget allows, removing 20% of selected nodes in one shot this minimal intervention prevents wasting evaluations on iterated fixes, as seen in slower algorithms like classical GA without early repair. For survivor selection, a lexicographic sort on feasibility then descending fitness ensures elitism by carrying over the top pop_size from combined P+Q, on populations of 20, this converged fastest in our graphs when paired with sparse initialization (4–6 ones to start feasible under low B). While the implementation could incorporate more graph-specific heuristics such as degree-based removal in repair, the design intentionally prioritises speed through power-law mutation, uniform crossover, and vectorized operations for rapid convergence within tight budgets.

The Fast GA was developed alongside the classical GA for comparison purposes. It runs more efficiently than the classical version, allowing experiments on larger budgets such as 100K evaluations. If the Fast GA already outperforms the classical GA under a smaller budget (10K), it can be expected to perform even better when given a higher budget (100K).

2 Multi-Objective EA