COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION SPORTS LEAGUE OPTIMIZATION

23/05/2025

GitHub repository: https://github.com/jm651120/CIFO.git

Group H
João Luís Marques - 20240656
Gustavo Gomes - 20240657
Rodrigo Luis - 20240742

Index

1. Problem Statement and Objectives

We were asked to divide 35 unique professional footballers into 5 evenly skilled teams, while obeying the constraints of:

1. Roster of exactly 1 GK, 2 DEF, 2 MID, 2 FWD;
2. Budget limit of 750€ million per team;

The search space is naturally very big because of all the possible league combinations (5^{35}) so we framed this as an optimization problem, using an evolutionary search (Genetic Algorithm).

The algorithm starts from a smart 'greedy' draft with random viable listings, then it iteratively reshuffles the players until the imbalance between the different teams strength falls below a certain threshold.

Our fitness function is aimed to maximize by the GA, so higher fitness means a more balanced league overall. [1]

What we achieved:

- Our fitness always converged to a value of 0.9459;
- All squads satisfied both salary cap and positional rules;

2. Representation

To represent league configurations, we decided to use a genotype of 35 integers, with each gene encoding the team of one player, for example a list [0,1,2,...] means the player in the first index belongs to team 0, the second to team 1 and so on.

We chose this approach mostly due to its efficiency since the Genetic Algorithm will only operate in the integer space, facilitating the mechanisms of crossover and mutation. It combines computational simplicity and is still able to work with more complex constraints. [2]

To interpret this genotype, we decode it into a list of lists, which contains the team rosters, using a **decode()** helper function. [2][3]

If any of the decoded solutions violates the 1-2-2-2 composition of the team or exceeds the 750€ million salary limit, a **repair()** function is applied to rectify it, which guarantees that every solution remains valid and does not complicate the genetic operators. [2][4]

3. Fitness Function

To drive the genetic search we convert our multi-criterion, partly constrained problem into a single scalar that a maximising genetic algorithm can handle, for any given league configuration we compute the fitness.[1][5]

Going over the terms present in our fitness function denominator, we added the constant 1 in order to cap our fitness maximum at 1 and prevent dividing by zero, we enforce the budget constraint by adding the budget overspends across the league and multiply it by the the coefficient λ which tunes the tradeoff, a high λ makes feasibility crucial, whereas a low value lets the GA explore slight violations. To measure the balanced distribution of talent we use standard deviation(σ)[1] of skill across all teams. Small swaps that tighten the talent spread lower σ slightly and are immediately rewarded, giving the GA a fine-grained signal.
Last but not least, we take into account the number of composition violations multiplied by a coefficient which "tells" the genetic algorithm how rigidly to enforce the constraint.

In order to reflect real-world tolerance for minor budget overruns we use 0.5 as the default value of λ, while a weight of 1 on composition mirrors the non-negotiable rule that teams must field the exact positional lineup.

4. Genetic Algorithm Design

Our Genetic Algorithm features a generational characteristic, made to obey the specific structure and rules of the problem. The population evolves through applications of **selection**, **crossover** and **mutation** operators, backed by the repair mechanism to guarantee feasibility after each genetic adjustment.

4.1. Genetic Operators

4.1.1. Selection Algorithms

- **Stochastic Universal Sampling (SUS):** a probabilistic method to ensure a fair and proportional representation of individuals by placing equally spaced pointers on a cumulative fitness wheel, then selecting the individuals where the pointer lands. [6]

- **Truncation Selection:** a deterministic approach that applies a stronger selective pressure and favouring high-fitness solutions, retaining the top 40% of individuals in each generation. [6]

4.1.2. Crossover Algorithms

- **Team Mask Crossover:** applies a 5 bit random mask that decides, for each team, whether to inherit the entire roster from parent 1 or from parent 2. However, the results may include conflicts regarding the team composition and that's why the child is passed to the repair function to restore feasibility. [7]
- **Uniform Position Crossover:** for each position, the algorithm "flips a coin" to inherit all players in that role from which parent. If the coin lands on the parent 1 side for the 'MID' position, all players from that position will take their team assignments as it is in parent 1. [7]

At the end of each crossover operation, a repair function is applied to ensure there are no duplicate or missing players.

4.1.3. Mutation Algorithms

- **Swap-Only Mutation:** selects two players with the same position but in different teams and swaps their teams, introducing variability when maintaining the roster structure. [8]
- **Kempe Mutation** (Inspired in the Kempe chain concept): Selects two teams and one position and swaps all players of that selected position between the two selected teams. Still respects the team composition and introduces a larger perturbation. [8]
- **Budget-Shave Mutation:** if a team is over-budget, replaces its most expensive player with the least expensive player available with the same position from another team, targeting directly salary limit violations.

Unlike crossover mechanisms, these mutation operators will not break team composition rules since they always swap players from the same position, however the child is still passed through the repair function to prevent any potential issues.

4.2 Main Genetic Algorithm Loop

The main Genetic Algorithm function serves as the main driver of the evolutionary process. Its implementation relies on a generational Genetic Algorithm in which each generation produces a new population regarding the genetic operators algorithms, followed by replacement of the previous population. [9]

Each run generates an initial population with one solution generated by a greedy heuristic that balances player roles across teams and ensures an immediate structured starting point, the remaining individuals are randomly created, which injects the GA with a mix of quality and diversity.
Then, the fitness of each individual is evaluated, proceeding with the following steps until a maximum number of generations:

1. **Elitism**: the top 2 elite individuals regarding fitness are maintained in the following generation, to preserve the best solution until that moment.
2. **Parent Selection**: the selection genetic operator chooses the parents based on their fitness.
3. **Crossover and Mutation:** with 90% and 10% respectively, crossover and mutation can happen to the parents to create the child.
4. **Fitness and reevaluation:** The new generated population replaces the old one and the best fitness in the current population is recorded at each generation.
5. **Early Stopping:** If there are no improvements in fitness over 50 consecutive generations, halt the algorithm.

The run_ga() function returns the best individual and its fitness and convergence history. This flexible and modular design allows the Genetic Algorithm to be easily manageable with different operator combinations and settings

5. Experimental protocol

5.1. Factors and levels

A full 3×3×2×2 factorial design [10] tests the separate and combined effects of population size, mutation mix, selection scheme, and crossover style. Ten independent random seeds are run for every configuration making it 360 runs in total, on each execution we log the best fitness, the implied skills standard deviation, number of generations until the best result was achieved, time and the full convergence curve.

5.2. Sanity-check run

Before the sweep, a minimal setting was executed to verify that the pipeline returns a feasible league and non-degenerate fitness curve, we used a population size of 20 we chose to test in this run the *sus_selection* [6] with the *team_mask_crossover* [7] and mut_swap_same_pos [8]. The run converged in 86 generations with a skills standard deviation of 0.077, validating the instrumentation. [11]

5.3. Batch loop implementation

We implemented a nested loop that iterates over all four experimental factors, selects the corresponding genetic operator configurations, executes the Genetic Algorithm, times the run, and appends results to a pandas frame.
Each configuration is run with 10 random seeds. A typical run lasts 1-8 s depending on population size, so the entire grid completes in < 15 min on the reference machine. We grouped the final results and built a summary dataframe. [12][13]

6. Results & analysis

By visualizing the plot (Best Fitness by Generation [14]) we can understand that the algorithm is effectively optimizing the solution over time. It starts around 0.775 and improves drastically over the next generations reaching approximately 0.950 by the end. This improvement is relatively rapid in the early generations, suggesting a strong initial optimization. The fitness values start to stabilize in later generations (around 20 to 60) approximately with values between 0.925 and 0.950. This plateau suggests the algorithm is nearing the best possible solution for the given problem and that the population diversity might be decreasing making it harder for further improvements.
Through systematic testing the genetic algorithm was optimized to 36 configurations where the key results include:

- **Population size: 40** individuals proved optimal. Larger population sizes did not improve fitness and greatly increased runtime.
- **Selection**: **Stochastic Universal Sampling** was outperformed by **Truncation** (top 40%), achieving the theoretical best fitness (0.9459) and being more reliable and faster (10% faster, 65-70 vs 80-90 generations).
- **Crossover: Team-mask** preserved critical synergies, surpassing or at least matching constantly uniform positional crossover.

- **Mutation: Truncation, Team-mask** jointly used with **Swap** plus **Kempe** mutations was enough. Budget-shave was only needed under SUS(weaker) selection.
- **Elitism**: dropping **Elitism** to 0 cut convergence to 50 but got worse average fitness (0.70) than if keeping the best 2 individuals, which sustained a higher average fitness (0.93) at only a cost of the runtime. This proved Elitism is essential for quality. [17]

This setup balances speed and solution quality, consistently near-optimal fitness. [15]

7. Discussion & justification

7.1. Final configuration choice

For our final configuration we chose to use a population of 40 which achieves the best fitness with a faster convergence than that of population 80. We chose the truncation selection because of its consistent and fast convergence, which reaches the optimal result in 76 generations, and team mask crossover for balancing the team level and cost skill balance more effectively than uniform position crossover.
Only swap and Kempe mutations were retained to maintain structural diversity. The budget-shave mutation no longer changes outcomes under strong selection and was removed. Together these choices give $\sigma \approx 0.057$, a fast and stable route to near perfect balance.[14][15]

7.2. Unexpected findings

Some unexpected patterns surfaced during the grid search. The highest-scoring league emerged from the leanest setup with pop 20, SUS selection, team-mask crossover, Swap-Only mutation [16] which showed that a compact population can sometimes intensify search focus, although the outcome was seldom reproduced across seeds, highlighting the greater robustness of our final configuration. The Budget-Shave operator, introduced as a feasibility safeguard, proved redundant under strong truncation pressure as selection plus repair already kept salaries in check. Conversely, the Uniform-Position crossover, advertised for exploration, consistently lagged behind team-mask because its finer granularity disrupted the cost-skill synergies preserved within intact rosters, leading to lower average fitness. [13]

7.3. Lessons learned & final remarks

Three core insights taken from the experiments were:

- **Truncation-plus-repair**, where aggressive selection is cushioned by an automatic fixer, shortened convergence and outperformed gentler diversity-oriented schemes.
- Roster-level crossover, which swaps entire teams intact, proved superior to fine-grained position shuffles by protecting latent cost-skill synergies under the budget cap.
- Mid-sized populations with predictable pressure, rather than very large pools chasing occasional record scores, delivered repeatable high-quality leagues.

Collectively, these findings show that pairing the right selection pressure with a synergy-preserving crossover yields greater gains than simply enlarging the population or bolting on extra repair operators.

Annexes

$$\text{fitness} = \frac{1}{1 + \sigma + \lambda \cdot \text{budget\_excess} + \mu \cdot \text{composition\_errors}}$$

[1] Fitness Function

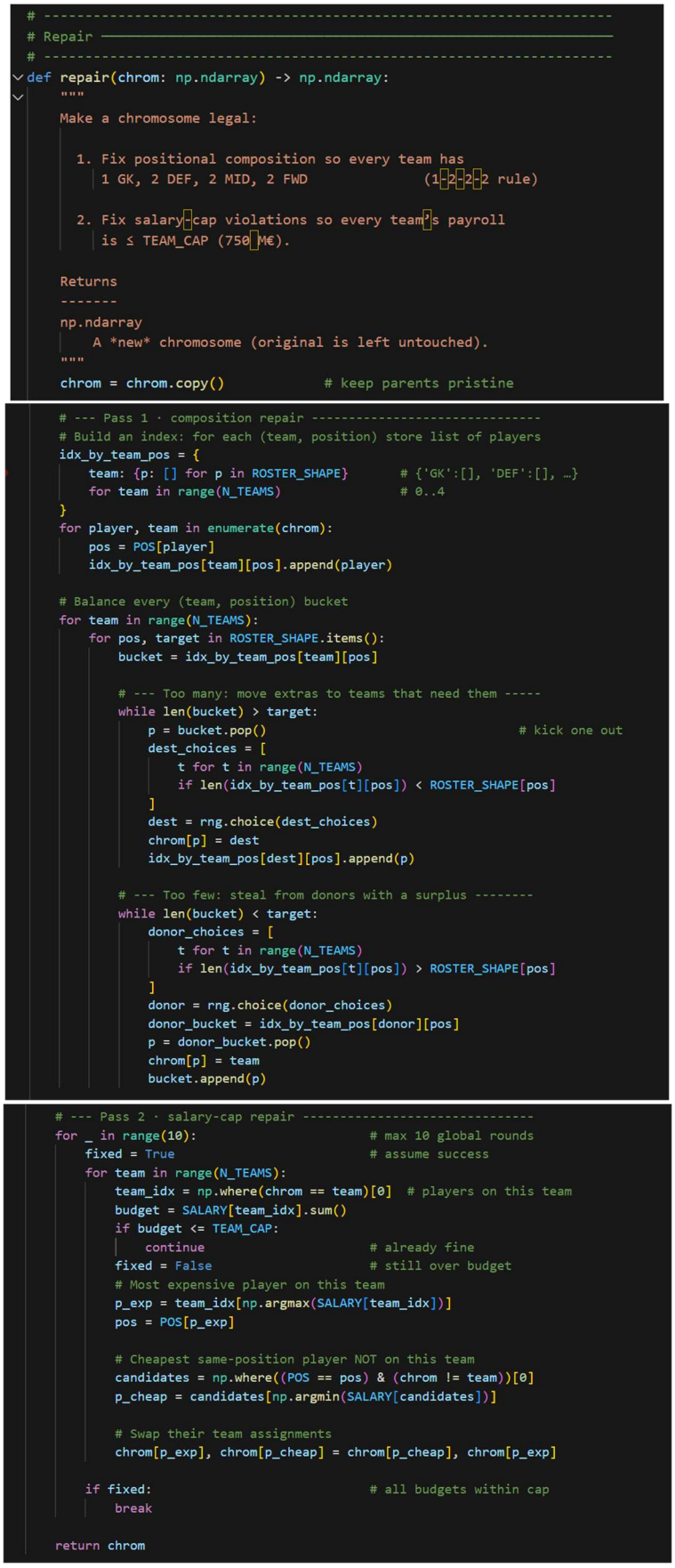
σ = standard deviation of the five teams' average skill (to measure imbalance)
budget_excess = total **budget excess** across all teams, scaled so 100 M€ = 1.0
composition_errors = number of teams violating the **1 GK, 2 DEF, 2 MID, 2 FWD** roster rule
λ=0.5, μ=1.0 are penalty weights

Component	Description	Purpose
Genotype	A vector $T = (t_0,\ldots,t_{34})$ where each gene $t_i \in \{0, 1, 2, 3, 4\}$ assigns player *i* to a team.	Compact, efficient encoding. Keeps all operators (crossover, mutation) in fast integer space.
Decoder	Converts the genotype T into a list of five rosters (phenotype), where each roster contains the indices of its players.	Allows evaluation: we can compute team averages, check rules, and display results.
Repair	After crossover/mutation, adjusts the chromosome to ensure that: • Each team has 1 GK, 2 DEF, 2 MID, 2 FWD • Each team stays under €750M budget	Guarantees all individuals remain **feasible** without complicating the genetic operators.

[2] Chromosome representation, decoder and repair

```python
# ---------------------------------------------------------------------
# 1.  Helper ──────────────────────────────────────────────────────────
# ---------------------------------------------------------------------
def decode(individual, n_teams: int = 5):
    """
    Convert a chromosome (length-35 integer vector) into an explicit league
    representation: a list of five teams, each containing the indices of the
    players that belong to it.

    Parameters
    ----------
    individual : list[int] | np.ndarray[int]
        Genotype where element i ∈ {0,…,4} tells which team player i joins.
        (Player order = the order in players.csv.)
    n_teams : int, optional
        Number of teams in the league (default 5).

    Returns
    -------
    list[list[int]]
        `teams[k]` is a *list* of player indices belonging to team k.
    """
    # Initialise n_teams empty buckets …
    teams = [[] for _ in range(n_teams)]

    # … iterate once over the chromosome and drop each player in its bucket.
    for player_idx, team_id in enumerate(individual):
        teams[team_id].append(player_idx)

    return teams
    # NOTE: this function does *no* feasibility checking by itself;
    #       it is kept lightweight so we can reuse it for many things
    #       (evaluation, visualisation, repairing, etc.).
```

[3] Decode helper function



```
# ---------------------------------------------------------------------
# Repair ──────────────────────────────────────────────────────────────
# ---------------------------------------------------------------------
def repair(chrom: np.ndarray) -> np.ndarray:
    """
    Make a chromosome legal:

      1. Fix positional composition so every team has
         1 GK, 2 DEF, 2 MID, 2 FWD              (1-2-2-2 rule)

      2. Fix salary-cap violations so every team's payroll
         is ≤ TEAM_CAP (750 M€).

    Returns
    -------
    np.ndarray
        A *new* chromosome (original is left untouched).
    """
    chrom = chrom.copy()            # keep parents pristine

    # --- Pass 1 · composition repair ------------------------------
    # Build an index: for each (team, position) store list of players
    idx_by_team_pos = {
        team: {p: [] for p in ROSTER_SHAPE}       # {'GK':[], 'DEF':[], …}
        for team in range(N_TEAMS)                # 0..4
    }
    for player, team in enumerate(chrom):
        pos = POS[player]
        idx_by_team_pos[team][pos].append(player)

    # Balance every (team, position) bucket
    for team in range(N_TEAMS):
        for pos, target in ROSTER_SHAPE.items():
            bucket = idx_by_team_pos[team][pos]

            # --- Too many: move extras to teams that need them -----
            while len(bucket) > target:
                p = bucket.pop()                                # kick one out
                dest_choices = [
                    t for t in range(N_TEAMS)
                    if len(idx_by_team_pos[t][pos]) < ROSTER_SHAPE[pos]
                ]
                dest = rng.choice(dest_choices)
                chrom[p] = dest
                idx_by_team_pos[dest][pos].append(p)

            # --- Too few: steal from donors with a surplus --------
            while len(bucket) < target:
                donor_choices = [
                    t for t in range(N_TEAMS)
                    if len(idx_by_team_pos[t][pos]) > ROSTER_SHAPE[pos]
                ]
                donor = rng.choice(donor_choices)
                donor_bucket = idx_by_team_pos[donor][pos]
                p = donor_bucket.pop()
                chrom[p] = team
                bucket.append(p)

    # --- Pass 2 · salary-cap repair -------------------------------
    for _ in range(10):                       # max 10 global rounds
        fixed = True                          # assume success
        for team in range(N_TEAMS):
            team_idx = np.where(chrom == team)[0]  # players on this team
            budget = SALARY[team_idx].sum()
            if budget <= TEAM_CAP:
                continue                      # already fine
            fixed = False                     # still over budget
            # Most expensive player on this team
            p_exp = team_idx[np.argmax(SALARY[team_idx])]
            pos = POS[p_exp]

            # Cheapest same-position player NOT on this team
            candidates = np.where((POS == pos) & (chrom != team))[0]
            p_cheap = candidates[np.argmin(SALARY[candidates])]

            # Swap their team assignments
            chrom[p_exp], chrom[p_cheap] = chrom[p_cheap], chrom[p_exp]

        if fixed:                             # all budgets within cap
            break

    return chrom
```

[4] Repair function

```python
# ---------------------------------------------------------------------
# Scalar fitness ------------------------------------------------------
# ---------------------------------------------------------------------
def fitness(
    individual,
    skills = SKILL, salaries = SALARY, positions = POS,
    budget_excess_penalty: float = 0.5,   # weight for the budget-excess penalty
    positional_violation_penalty:  float = 1.0    # weight for the positional-violation penalty
):
    """
    Evaluate a chromosome according to project rules:

      • Minimise the standard deviation of team-average skills
      • Penalise budgets over 750 M€
      • Penalise wrong positional mixes        (optional if operators repair)

    The GA will *maximise* this fitness, so we invert the objective by
    placing everything in the denominator +1 to avoid division by zero.

    Returns
    -------
    float
        A higher value = a better, more balanced and feasible league.
    """
    # 1. Decode once -> list of teams
    league = decode(individual)

    # 2. Get league-level numbers
    avg_skill, over_budget, comp_bad = team_stats(
        league, skills, salaries, positions)

    # 3. Components of the denominator
    sigma   = avg_skill.std(ddof=0)          # population std-dev
    penalty = budget_excess_penalty * over_budget + positional_violation_penalty * comp_bad

    # 4. Final score (higher is better)
    return 1.0 / (1.0 + sigma + penalty)
```

[5] Fitness function

```python
def sus_selection(fitness_vals, n_parents, rng=rng):
    """
    Stochastic-Universal Sampling (1 spin of a 'nested' roulette wheel).
    Picks n_parents *indices* from the population list.
    """
    total_fit = sum(fitness_vals)
    # Create equally spaced pointers on the wheel
    pointers = np.linspace(0, total_fit, n_parents, endpoint=False) + \
               rng.random() * total_fit / n_parents
    idx, cum_sum, picks = 0, fitness_vals[0], []
    for p in pointers:
        while cum_sum < p:
            idx += 1
            cum_sum += fitness_vals[idx]
        picks.append(idx)
    return picks            # list of indices


def truncation_random_fill(fitness_vals, n_parents, top_frac=0.4, rng=rng):
    """
    1. Sort indices by fitness, keep the top `top_frac` proportion.
    2. Draw uniformly at random among those survivors to get n_parents.
    """
    pop_size = len(fitness_vals)
    k = max(2, int(top_frac * pop_size))
    survivors = np.argsort(fitness_vals)[-k:]     # top→best
    return rng.choice(survivors, size=n_parents, replace=True).tolist()
```

[6] Selection Algorithms

```python
def team_mask_crossover(p1, p2, rng=rng):
    """
    5-bit mask; bit==1 → take that *whole* team from parent-1,
    else from parent-2.   Returns one child chromosome.
    """
    mask = rng.integers(0, 2, size=N_TEAMS)      # array of 0/1 of length 5
    child = p2.copy()                            # start with parent-2
    for team in range(N_TEAMS):
        if mask[team] == 1:
            child[p1 == team] = team             # copy all players of that team
            child[p2 == team] = np.where(child[p2 == team] == team,
                                         team,  # already correct
                                         child[p2 == team])  # keep others
    return repair(child)


def uniform_position_crossover(p1, p2, rng=rng):
    """
    For each position (GK/DEF/MID/FWD) flip a coin.
    If heads, child gets **all players of that position** from parent-1,
    otherwise from parent-2. Returns one child.
    """
    child = p2.copy()
    for pos in ROSTER_SHAPE:
        if rng.random() < 0.5:                   # take from parent-1
            idx = np.where(POS == pos)[0]
            child[idx] = p1[idx]
    return repair(child)
```

[7] Crossover Algorithms

```python
# Switch two players with the same position from different teams
def mut_swap_same_pos(chrom, rng=rng):
    child = chrom.copy()
    # choose two DISTINCT players with same position
    while True:
        i, j = rng.choice(N_PLAYERS, size=2, replace=False)
        if POS[i] == POS[j] and child[i] != child[j]:
            break
    child[i], child[j] = child[j], child[i]
    return repair(child)

# If a team is over budget, replace its most expensive player with the cheapest player from all the teams
def mut_budget_shave(chrom, rng=rng):
    child = chrom.copy()
    # find team(s) over the cap
    for team in range(N_TEAMS):
        idx = np.where(child == team)[0]
        if SALARY[idx].sum() > TEAM_CAP:
            # swap out the most expensive player
            p_exp = idx[np.argmax(SALARY[idx])]
            pos = POS[p_exp]
            # bring in the cheapest compatible player not on this team
            candidates = np.where((POS == pos) & (child != team))[0]
            p_cheap = candidates[np.argmin(SALARY[candidates])]
            child[p_exp], child[p_cheap] = child[p_cheap], child[p_exp]
            break                                   # one swap is enough
    return repair(child)

# Chose a position and swap all the players between 2 random selected teams.
def mut_kempe_swap(chrom, rng=rng):
    child = chrom.copy()
    pos = rng.choice(list(ROSTER_SHAPE))            # choose position e.g. 'DEF'
    teams = rng.choice(N_TEAMS, size=2, replace=False)
    tA, tB = teams
    # players of that position on each team
    idxA = np.where((child == tA) & (POS == pos))[0]
    idxB = np.where((child == tB) & (POS == pos))[0]
    child[idxA] = tB
    child[idxB] = tA
    return repair(child)
```

[8] Mutation Algorithms

```python
# ──────────────────────────────────────────────────────────
#  GA main driver
# ──────────────────────────────────────────────────────────
def run_ga(pop_size=40, max_gen=300,
           Pc=0.9, Pm=0.1, elite=2,
           select_fn=sus_selection,
           *,                                  # <-- force keyword args below
           crossover_ops=None,
           mutation_ops=None,
           rng=rng, verbose=True):

    # ── defaults if caller omits lists ──────────────────────
    if crossover_ops is None:
        crossover_ops = [team_mask_crossover,
                         uniform_position_crossover]
    if mutation_ops is None:
        mutation_ops = [mut_swap_same_pos,
                        mut_budget_shave,
                        mut_kempe_swap]

    # ----- initialise ---------------------------------------------
    pop = initial_population(pop_size)         # list of chromosomes
    fit = np.array([fitness(ind) for ind in pop])

    best_history = [fit.max()]
    if verbose:
        print(f"Gen 0  best={fit.max():.4f}")

    # ----- evolutionary loop --------------------------------------
    for gen in range(1, max_gen + 1):
        new_pop = []

        # --- elitism: keep best k ---------------------------------
        elite_idx = np.argsort(fit)[-elite:]
        for idx in elite_idx:
            new_pop.append(pop[idx].copy())
```

```python
        # --- create children until population is full -------------
        while len(new_pop) < pop_size:
            # Selection
            parents_idx = select_fn(fit, n_parents=2, rng=rng)
            p1, p2 = pop[parents_idx[0]], pop[parents_idx[1]]

            # Crossover
            if rng.random() < Pc:
                cx_fn = rng.choice(crossover_ops)          # <-- use caller's list
                child = cx_fn(p1, p2, rng)
            else:
                child = p1.copy()

            # Mutation
            if rng.random() < Pm and mutation_ops:         # <-- use caller's list
                mut_fn = rng.choice(mutation_ops)
                child = mut_fn(child, rng)

            new_pop.append(child)

        # --- replace, evaluate ------------------------------------
        pop = new_pop
        fit = np.array([fitness(ind) for ind in pop])

        best_history.append(fit.max())
        if verbose and gen % 20 == 0:
            print(f"Gen {gen:<3} best={fit.max():.4f}")

        # Optional early stop: no progress in 50 gens
        if len(best_history) > 50 and \
           max(best_history[-50:]) - best_history[-51] < 1e-6:
            if verbose:
                print("No improvement in 50 generations—stopping early")
            break

    best_idx = np.argmax(fit)
    return pop[best_idx], fit[best_idx], best_history
```

[9] Main Genetic Algorithm function

Variable	Tested values	Rationale
Population size	20 · 40 · 80	Verify scalability and detect diminishing returns.
Mutation mix	(1) *swap only* · (2) *all three* · (3) *no budget-repair*	Isolate the contribution of the two "smart" mutations (budget-repair & Kempe swap).
Selection	Stochastic Universal Sampling (SUS) · Truncation	Compare high-diversity vs high-pressure selection.
Crossover	*Mask* (team-level) · *Uniform-pos* (role-level)	Contrast exploitation-oriented vs exploration-oriented recombination.

[10] Experimental Protocol

pop	mut	sel	cx	avg_fit	sd_fit	avg_sigma	gens
20	swap_only	sus	mask	0.928936	0.021960	0.077048	86.4

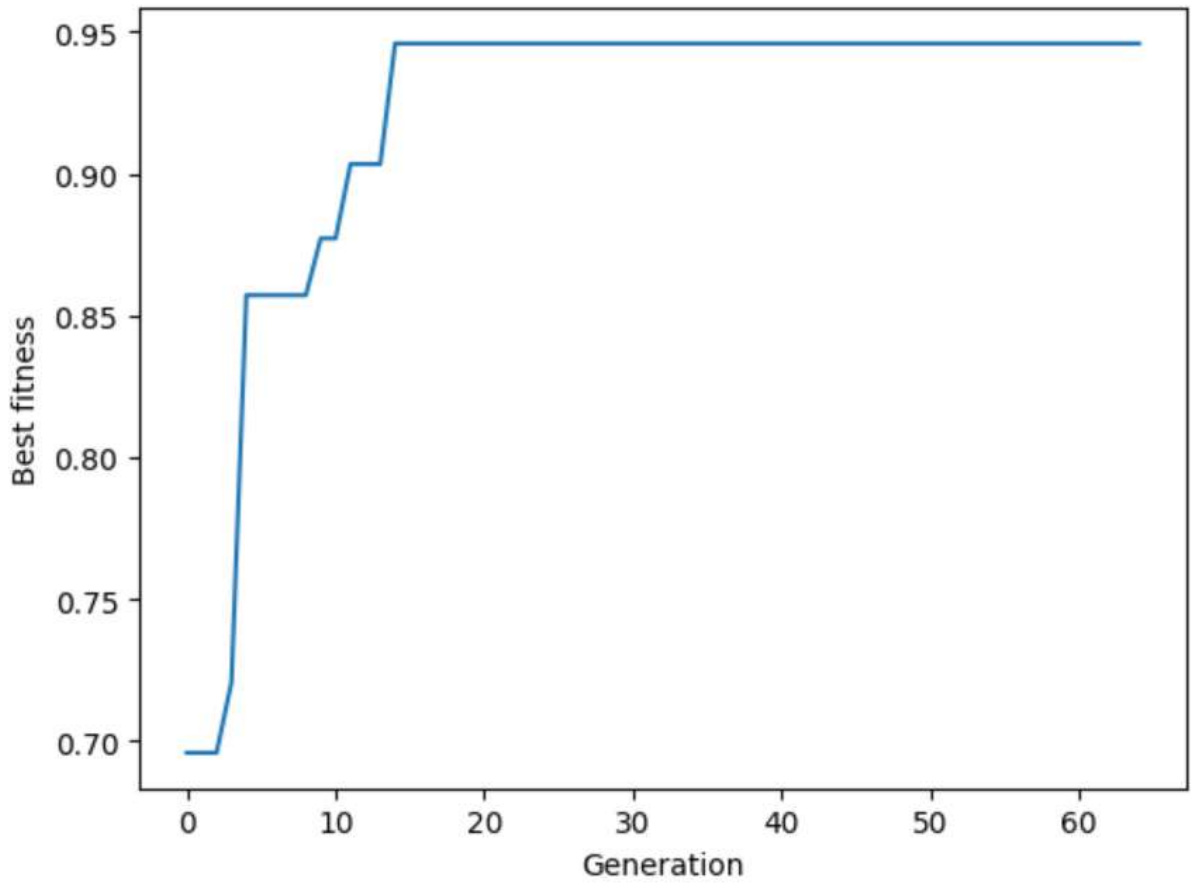
[11] Sanity check result

```
results = []

for pop_size, mut_mode, sel_name, cx_name in itertools.product(
        [20, 40, 80],
        ['swap_only', 'all_three', 'no_budget_shave'],
        ['sus', 'trunc'],
        ['mask', 'uniform']):

    for seed in range(10):                    # 10 runs per setting
        rng = np.random.default_rng(seed)

        # ---- choose selection ----------------------------------
        select_fn = sus_selection if sel_name == 'sus' else truncation_random_fill

        # ---- choose crossover list -----------------------------
        cx_ops = [team_mask_crossover] if cx_name == 'mask' \
                 else [uniform_position_crossover]

        # ---- choose mutation list ------------------------------
        if mut_mode == 'swap_only':
            mut_ops = [mut_swap_same_pos]
        elif mut_mode == 'no_budget_shave':
            mut_ops = [mut_swap_same_pos, mut_kempe_swap]
        else:                                  # all_three
            mut_ops = [mut_swap_same_pos, mut_budget_shave, mut_kempe_swap]

        # ---- run GA --------------------------------------------
        t0 = time.time()
        best_ind, best_fit, hist = run_ga(
                pop_size      = pop_size,
                select_fn     = select_fn,
                Pc            = 0.8,
                Pm            = 0.3,           # ← mutation probability inside GA
                crossover_ops = cx_ops,        # ← lists chosen above
                mutation_ops  = mut_ops,
                rng           = rng,
                verbose       = False)
        elapsed = time.time() - t0

        # ---- log result ----------------------------------------
        results.append({
            'pop'    : pop_size,
            'mut'    : mut_mode,
            'sel'    : sel_name,
            'cx'     : cx_name,
            'seed'   : seed,
            'fitness': best_fit,
            'sigma'  : 1 / best_fit - 1,
            'gens'   : len(hist) - 1,
            'time_s': elapsed
        })

df_results = pd.DataFrame(results)
```

[12] Batch loop implementation

	pop	mut	sel	cx	avg_fit	sd_fit	avg_sigma	gens
0	20	all_three	sus	mask	0.937441	0.017930	0.067095	81.2
1	20	all_three	sus	uniform	0.933188	0.020542	0.072071	87.5
2	20	all_three	trunc	mask	0.937441	0.017930	0.067095	82.4
3	20	all_three	trunc	uniform	0.928936	0.021960	0.077048	82.7
4	20	no_budget_shave	sus	mask	0.937441	0.017930	0.067095	85.9
5	20	no_budget_shave	sus	uniform	0.937441	0.017930	0.067095	90.0
6	20	no_budget_shave	trunc	mask	0.945946	0.000000	0.057143	76.1
7	20	no_budget_shave	trunc	uniform	0.945946	0.000000	0.057143	85.6
8	20	swap_only	sus	mask	0.941693	0.013448	0.062119	85.8
9	20	swap_only	sus	uniform	0.937441	0.017930	0.067095	86.1
10	20	swap_only	trunc	mask	0.941693	0.013448	0.062119	73.9
11	20	swap_only	trunc	uniform	0.941693	0.013448	0.062119	71.7
12	40	all_three	sus	mask	0.941693	0.013448	0.062119	76.8
13	40	all_three	sus	uniform	0.941693	0.013448	0.062119	87.5
14	40	all_three	trunc	mask	0.945946	0.000000	0.057143	70.3
15	40	all_three	trunc	uniform	0.941693	0.013448	0.062119	67.0
16	40	no_budget_shave	sus	mask	0.937441	0.017930	0.067095	78.0
17	40	no_budget_shave	sus	uniform	0.941693	0.013448	0.062119	81.2
18	40	no_budget_shave	trunc	mask	0.945946	0.000000	0.057143	68.7
19	40	no_budget_shave	trunc	uniform	0.945946	0.000000	0.057143	66.2
20	40	swap_only	sus	mask	0.941693	0.013448	0.062119	81.0
21	40	swap_only	sus	uniform	0.941693	0.013448	0.062119	86.9
22	40	swap_only	trunc	mask	0.945946	0.000000	0.057143	67.8
23	40	swap_only	trunc	uniform	0.945946	0.000000	0.057143	73.8
24	80	all_three	sus	mask	0.945946	0.000000	0.057143	88.4
25	80	all_three	sus	uniform	0.945946	0.000000	0.057143	86.9
26	80	all_three	trunc	mask	0.945946	0.000000	0.057143	69.5
27	80	all_three	trunc	uniform	0.945946	0.000000	0.057143	66.1
28	80	no_budget_shave	sus	mask	0.937441	0.017930	0.067095	81.7
29	80	no_budget_shave	sus	uniform	0.945946	0.000000	0.057143	78.6
30	80	no_budget_shave	trunc	mask	0.945946	0.000000	0.057143	67.7
31	80	no_budget_shave	trunc	uniform	0.945946	0.000000	0.057143	63.0
32	80	swap_only	sus	mask	0.945946	0.000000	0.057143	76.5
33	80	swap_only	sus	uniform	0.945946	0.000000	0.057143	80.1
34	80	swap_only	trunc	mask	0.945946	0.000000	0.057143	62.2
35	80	swap_only	trunc	uniform	0.945946	0.000000	0.057143	62.4

[13] All tested scenarios and the results



[14] Best fitness by Generation Plot

Parameter	Adopted value	Justification
Population	**40**	Same average quality as pop 80 but ~45 % less CPU time.
Selection	**Truncation (40 %)**	Converges reliably in ≈ 65–70 generations.
Crossover	**Team-mask**	With truncation, reaches the best fitness in > 90 % of seeds.
Mutation	**swap + Kempe**, *Pm* = 0.10	Maintains structural diversity; *budget-shave* proved redundant in this setup.
Elitism	2	Prevents accidental loss of the best leagues.

[15] Final Configuration

```
best_row = df_results.loc[df_results['fitness'].idxmax()]
print(best_row)
```

✓ 0.0s

```
pop              20
mut        swap_only
sel              sus
cx              mask
seed               0
fitness     0.945946
sigma       0.057143
gens             108
time_s      1.318645
Name: 0, dtype: object
```

[16] Configuration that achieved the best fitness score across all experimental runs (pop = 20, selection = SUS, crossover = team mask, mutation = swap, seed = 0; fitness = 0.945946, σ = 0.057)

elite	avg_fit	avg_gens
0	0.701331	50.0
2	0.933188	70.7

[17] Impact of Elitism on GA performance - average of 10 seeds