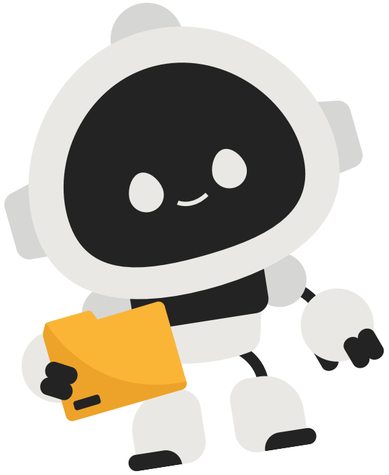
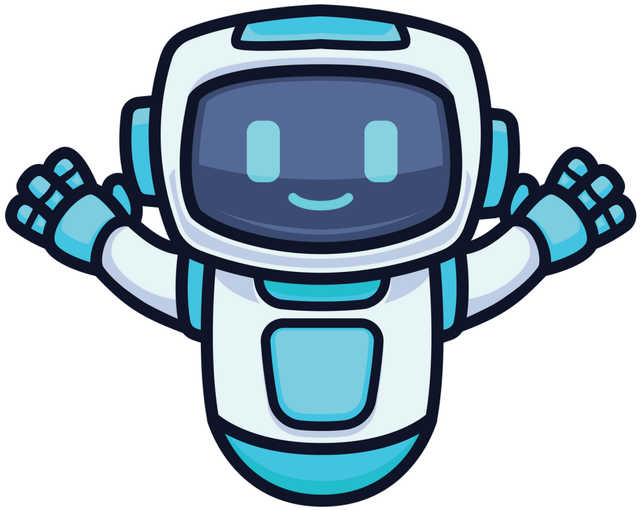
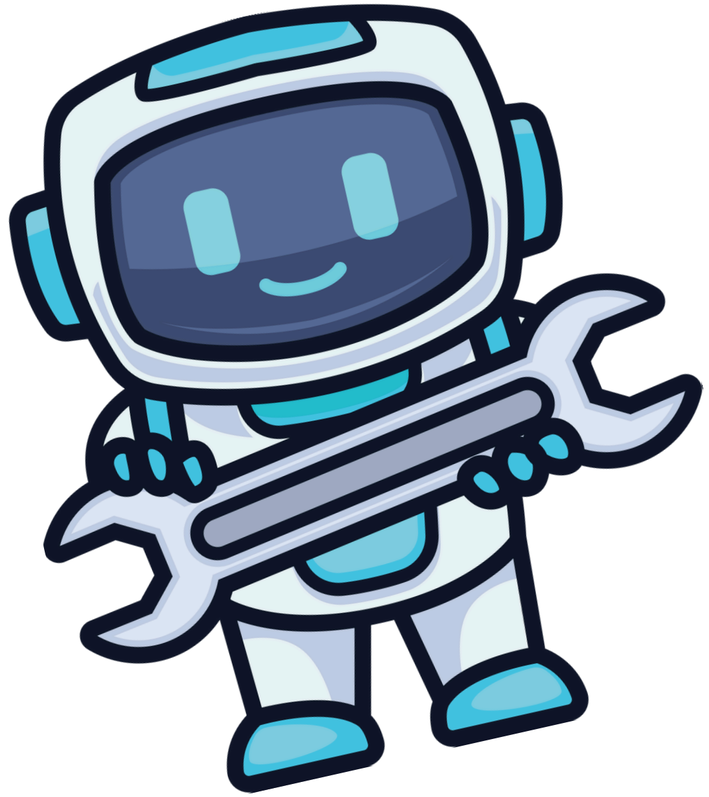
SPORTS TOURNAMENT SCHEDULING VIA : GENETIC ALGORITHMS (GA)





team work



Ahmed Gamal
20220015

Karem Ehab
20220342

Mohamed Ahmed 20220427

Mohamed Mahmoud 20220376

PROJECT OVERVIEW

Organizing sports tournaments, especially those involving multiple teams and venues, is a complex task due to the large number of constraints involved—such as avoiding venue conflicts, ensuring fair rest periods between games, and balancing match timings. This project applies Genetic Algorithms (GAs), a biologically inspired evolutionary optimization technique, to automatically generate optimized tournament schedules that satisfy such constraints. By encoding tournament schedules as individuals in a population, and evolving these through genetic operations like crossover and mutation, the algorithm iteratively improves the quality of solutions based on a fitness evaluation system. The approach is validated by comparing generated schedules with a baseline, demonstrating the efficiency and adaptability of GAs for real-world sports scheduling problems.

Problem Type:

Constrained Optimization Problem

Introduction and Motivation:

🔍Scheduling a sports tournament is not just about assigning matches—it's about doing it fairly and efficiently. With constraints like venue availability, team rest requirements, and the total number of rounds, the challenge becomes computationally intensive, especially as the number of teams increases. Traditional manual or greedy approaches often fail to meet all constraints or take too long to compute.

Why use Genetic Algorithms (GA)?

- GAs are well-suited for combinatorial optimization problems.
- They are adaptive and can explore a large solution space.
- GAs can provide near-optimal solutions in a reasonable amount of time.

PROJECT GOALS

- Represent tournament schedules (e.g., round-robin or knockout) as individuals in the GA population.
- Define multiple objectives, such as:
- Minimize venue conflicts
- Maximize fairness in rest periods
- Balance game times across all teams
- Develop a fitness function to evaluate how well a schedule satisfies the above constraints.
- Apply GA operators like crossover and mutation, customized to tournament structures.
- Validate the GA-generated schedule against a baseline scheduling method.

Chromosome Creation Method

In this scheduling problem, each chromosome represents a complete and valid tournament schedule.
Each gene 🧩 inside the chromosome stands for a single match 🏆 placed in a unique combination of:

📅 Day

⏰ Time slot

🏟️ Venue

Chromosome Creation Method

⚙️ **How It Works (Step-by-Step):**

1. **Generate All Possible Slots** 🗓️
   a. **The function all_possible_slots() creates a full list of possible combinations of days, time slots, and venues.**
   b. **These represent the available spaces for scheduling matches.**
2. **Shuffle the Slots** 🔀
   a. **The slots are randomly shuffled to ensure diversity in the individuals created (so the population doesn't start identical).**
3. **Assign Matches to Slots** 🤝
   a. **For every match:**
   - **The code tries to find a slot that's:**
     - ✅ **Not already used.**
     - ✅ **On a day when neither team is already playing (to avoid overbooking a team).**
   - **Once a suitable slot is found, the match is assigned, and the slot is marked as used.**
4. **Conflict Check** ❌
   a. **If a match cannot be scheduled without violating the constraints (like a team playing twice in one day), an exception is raised to highlight the conflict.**
5. **Build Population** 👥
   a. **The create_population(size) function:**
   - **Creates a number of unique individuals (schedules).**
   - **Ensures that duplicates are avoided to maintain diversity in the genetic pool.**

Refrence Of Representation

Permutation Representation

Ordering/sequencing problems form a special type.

The task is (or can be solved by) arranging some objects in a certain order:

- Example: **Production Scheduling**: the important thing is which elements are scheduled before others (order)
- Example: **Travelling Salesman Problem (TSP)**: the important thing is which elements occur next to each other (adjacency)

These problems are generally **expressed as a permutation**:

– if there are n variables, then the representation is a list of n integers, each occurring exactly once.

57

Evolutionary Algorithms [AI420] An Introduction to Applied Evolutionary Computation and Nature-Inspired Algorithms Lecture 4

🧠 Fitness Function Explanation

This function evaluates how good a given tournament schedule is, by calculating penalties for rule violations and bonuses for balanced distribution.

📊 Inputs

- **schedule: A list of matches, where each match is a tuple like:**
- **(team1, team2, day, time_slot, venue)**

⚖️ Step-by-Step Breakdown

- 🟥 **Penalty: For conflicts or undesirable patterns.**
- 🟩 **Bonus: For fairness and good distribution.**

🧠 Fitness Function Explanation

- ⚠️ Penalties on Team Conflicts
  - If a team plays more than once on the same day ➡️ +50 penalty 😓
  - If a team plays on consecutive days ➡️ +20 penalty 🤦 (not enough rest)
- 🎯 Bonus for Even Distribution
  - If a team plays on multiple days, we compute the variance of their playdays.
  - Higher variance = more spread out = better ➡️ Add bonus ✨
- 🚫 Penalty on Venue Conflicts
  - If multiple matches are scheduled at the same time, same venue ➡️ +100 penalty per conflict 🏟️❌
- 🧮 Final Fitness Score
  - Start from a base of 1000
  - Subtract total penalty, add bonus
  - Ensure it's never negative

preserving diversity

🌍 Island Model Explanation :

The implemented algorithm uses the Island Model (also known as Coarse-Grain Parallel Evolutionary Algorithm) to enhance the performance and diversity of the Genetic Algorithm (GA). This model divides the global population into multiple independent subpopulations, called islands 🏝️, which evolve separately and periodically exchange individuals through a process called migration 🔁.

preserving diversity

⚙️ **Key Concepts in the Implementation:**

1. 🧬 Parallel Evolution
   - The total population is split into 4 separate islands.
   - Each island evolves its own population independently.
   - Standard GA operations (selection, crossover, mutation) occur within each island.
   - This allows each island to explore different regions of the search space in parallel 🧭.

2. 🔄 Migration
   - Migration occurs every 10 generations (MIGRATION_INTERVAL).
   - During migration:
     - The top 2 fittest individuals (MIGRATION_COUNT) from each island are selected 🥇.
     - They replace the worst individuals in the neighboring island 😵.
   - This exchange:
     - Spreads strong solutions 💡.
     - Injects new genetic material 🧪.
     - Prevents premature convergence 🚫.

preserving diversity

⚙️ Key Concepts in the Implementation:

3. 🔁 Ring Topology

- Islands are connected in a ring-like structure:
  - Island 0 → Island 1
  - Island 1 → Island 2
  - Island 2 → Island 3
  - Island 3 → Island 0
- This creates a controlled flow of information between islands 🔗.

💡 Why Use the Island Model?

- 🌈 Maintains Diversity: Each island explores the search space differently, reducing the chance of getting stuck in local optima.
- 🧠 Balances Exploration & Exploitation:
  - Local evolution exploits current solutions.
  - Migration supports global exploration.
- ⚡ Scalable Design: Easily extendable for multi-core or distributed systems.

Tournament Selection 🏆

Purpose:

The function implements Tournament Selection, a popular selection mechanism used in genetic algorithms (GAs). This method selects individuals from the population based on their fitness, but unlike roulette wheel selection, it operates by selecting a subset of individuals randomly (the "tournament"), then choosing the best individual among them. This helps to ensure diversity and prevents premature convergence.

Parameters:

- population: A list of individuals representing the current population. Each individual is assumed to have a fitness attribute or can be evaluated using a fitness function.
- tournament_size (optional, default=3): The number of individuals randomly chosen from the population to participate in the tournament. A larger tournament size means a stronger selection pressure, where individuals with better fitness have a higher chance of winning.

Returns:

The individual that wins the tournament, i.e., the individual with the highest fitness from the selected tournament subset.

Tournament Selection 🏆

Process:

1. 🎲 **Random Selection:**
   a. **A random subset of individuals is selected from the population. The size of the subset is defined by the tournament_size parameter. This randomness introduces diversity, as not all individuals with high fitness will always be selected for the tournament.**
2. 🧠 **Fitness Calculation:**
   a. **The fitness of each individual in the tournament is calculated. Typically, the fitness function is designed to evaluate how "good" or "fit" each individual is in relation to the problem being solved. In this case, the fitness of each individual is evaluated using the fitness() function, which computes the fitness score.**
3. 🏅 **Winner Selection:**
   a. **Once the fitnesses of the tournament participants are calculated, the individual with the highest fitness value is chosen as the winner. This step introduces selection pressure, which favors individuals with better fitness, promoting the survival of the best solutions.**
4. 🏆 **Return Winner:**
   a. **The individual with the highest fitness from the tournament is returned. This individual is then used for the next step of the genetic algorithm, such as reproduction (crossover or mutation) or replacing a member of the population.**

Survivor Selection (Elitism) - Function Documentation 🏆🌟

Purpose 🎯

This function implements **Survivor Selection** using **Elitism** in a genetic algorithm (GA). It ensures that the fittest individuals from the current generation and offspring are selected to form the next generation, promoting the retention of the best solutions found so far. 🌱

Parameters 📋

- `population` (*List*): The current set of individuals (chromosomes) in the GA. These are the solutions that are evaluated in the current generation. 👥
- `offspring` (*List*): The newly generated individuals (offspring) produced by crossover and/or mutation. These are potential solutions that need evaluation. 🔄

Returns 🔄

- **List:** A list of the best individuals selected from the combined population and offspring, ensuring the best solutions are carried to the next generation. This list will have the same size as the original population. 🏅

Survivor Selection (Elitism) - Function Documentation 🏆🌟

How It Works ⚙️

1. **Combining Population and Offspring** 🤝
    - First, the function combines both the current population and the offspring into one larger list. This allows for the evaluation of all individuals before selecting the best ones. 🔗
2. **Sorting Individuals Based on Fitness** 🏋️
    - The function sorts the combined list based on each individual's **fitness**. The `shared_fitness` function calculates the fitness values. The sorting ensures that the individuals with the highest fitness are ranked first. 📊
3. **Selecting the Best Individuals** 🥇
    - The top individuals (those with the highest fitness) are selected, ensuring that the population size stays the same as the original. This guarantees that the best individuals make it into the next generation. 🍒

🧬 ORDER CROSSOVER (OX)

🎯 Purpose

Implements the Order Crossover (OX) operator in the context of sports tournament scheduling.

Its goal is to combine two parent schedules into two offspring, ensuring:

- **All match pairings are preserved uniquely 🆗**
- **The offspring remain valid under scheduling constraints 🗓️**
- **Match order is partly inherited from both parents 🔁**

🟦 Background: What is Order Crossover (OX)?

Order Crossover is a widely-used operator in genetic algorithms, especially for problems involving permutations, such as:

- **Traveling Salesman Problem (TSP) 🚗**
- **Job sequencing 📋**
- **Match scheduling in tournaments ⚽🏀🏐**

OX works by copying a slice of one parent and filling the rest from the other parent while preserving the original order and avoiding duplicates.

🧬 ORDER CROSSOVER (OX)

🧠 Conceptual Workflow

1. 🧪 **Crossover Decision:**
2. **A random value is compared to the crossover_rate.**
    - **If crossover is skipped: return clones of the parents.**
    - **If applied: continue to crossover logic.**
3. 🎲 **Crossover Points Selection:**
    a. **Two random points are selected to identify a slice in each parent.**
4. 📦 **Segment Copying:**
    - **A slice (subsequence of match pairs) is copied from one parent.**
    - **The rest is filled from the other parent, preserving the order and avoiding duplicates.**

🔄 Swap Mutation

🎯 Purpose :

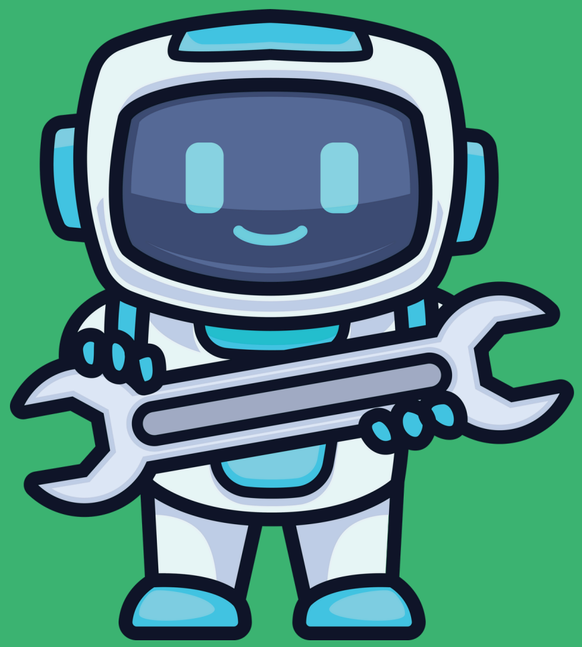
The Swap Mutation function introduces variability into the population by modifying an individual schedule. It helps the genetic algorithm escape local optima and explore new areas of the solution space.

This function is applied probabilistically based on a mutation rate

⚙️ How It Works

- 🧪 **Mutation Trigger:**
  - **A random number is generated and compared with the mutation_rate.**
  - **If the number is less than the mutation rate, mutation is performed.**
  - **Otherwise, the schedule is returned unchanged.**

🔄 Swap Mutation



⚙️ **How It Works:**

- 🔀 **Match Swap:**
  - **Two matches are randomly selected from the schedule and swapped.**
  - **This maintains the permutation property but may alter the overall tournament flow.**
-
- 🛠️ **Slot Mutation:**
- **After swapping, one of the matches is randomly chosen and its time and venue are mutated by:**
  - **Replacing its time slot with a random value from Days ⏰**
  - **Replacing its venue with a random value from Venues 🏟️**
- ✅ **Schedule Update:**
  - **The schedule is scanned for the match being mutated (by team pair), and its entry is updated with the new slot info.**

🌍 Island Genetic Algorithm

🏝️ Step-by-Step Breakdown

1. **Initialization - Creating Islands:**
   - The algorithm starts by creating multiple islands, each representing a separate subpopulation.
   - The function create_islands divides the total population into smaller populations, allowing each island to evolve independently.
   - Why? This approach mimics natural phenomena where isolated populations evolve separately, leading to a broader exploration of the solution space. 🌱

2. **Local Search for Initial Population (Step 1):**
   - After creating the initial islands, each individual in the population undergoes a local search to improve its solution. This is done by the local_search function, which refines the individual with a certain probability (local_search_prob).
   - Why? Local search helps improve the initial population, enhancing the quality of the starting solutions for each island. 🔍

🌍 Island Genetic Algorithm

🏝️ **Step-by-Step Breakdown**

1. **Main Evolution Process (Generations Loop):**
    a. **The algorithm runs for a predefined number of generations (max_generations). Each generation involves several steps:**
        i. **a. Selection (Tournament Selection):**
        ii. **Tournament selection is used to choose parents for reproduction. A pool of individuals is randomly selected, and the best one (based on fitness) is chosen to be a parent.**
        iii. **Why? Tournament selection helps in maintaining diversity while focusing on the fitter individuals. ⚔️**
    b. **b. Crossover:**
        i. **Parents are paired, and a crossover operation is applied to produce offspring. If a random number is less than the crossover rate, the offspring are generated by combining features from both parents (using the order_crossover function).**
        ii. **If crossover does not occur (based on the probability), the offspring are simply copies of the parents.**
        iii. **Why? Crossover allows the algorithm to combine beneficial traits from multiple individuals, aiding in exploring a larger portion of the solution space. 🔄**
    c. **c. Mutation:**
        i. **The offspring undergo a mutation step where small random changes are made. This is performed by the swap_mutation function.**
        ii. **Why? Mutation introduces diversity and ensures that the algorithm doesn't get stuck in local optima. 🧬**
    d. **d. Local Search on Offspring (Step 3):**
        i. **After crossover and mutation, a local search is applied to the offspring with a certain probability (local_search_prob).**
        ii. **Why? This further refines the offspring, enhancing their potential for finding better solutions. 🔍**

🌍 Island Genetic Algorithm

🏝️ Step-by-Step Breakdown

a. Survivor Selection:
   i. After generating the offspring, the survivor selection process decides which individuals remain in the population. It ensures that the best individuals (parents + offspring) are selected to move forward to the next generation.
   ii. Why? Survivor selection ensures that the population evolves over time by retaining high-quality solutions. 🌟

b. Migration Between Islands:
   i. Periodically, migration occurs between islands. Migrants (top-performing individuals) from one island are moved to a neighboring island. This helps spread good solutions across islands and promotes diversity.
   ii. Why? Migration enables information sharing between islands, preventing each subpopulation from becoming too similar and aiding in better global exploration. 🌍

c. Tracking Best Solution:
   i. During each generation, the algorithm tracks the best solution found so far. The fitness of the best individual is updated and recorded in the fitness_history.
   ii. Why? Keeping track of the best solution ensures that the algorithm converges towards an optimal or near-optimal solution over time. 🏆

🌍 Island Genetic Algorithm

🏁 Termination:

- The algorithm terminates after reaching the specified maximum number of generations (max_generations). At this point, the best solution across all islands is returned.
- Why? The termination condition ensures that the algorithm doesn't run indefinitely and provides the final best solution after sufficient evolution. ⏳

📝 Multiple Runs with Seed Storage

🎯 **Implemented Changes:**

1. **Multiple Evolutionary Runs:**
    a. **To assess the consistency and reliability of the solution, the genetic algorithm is now executed multiple times (optimally, 30 runs per setting). This repeated process ensures that the algorithm's outcomes are not dependent on the specific randomness of a single run, and it provides a more comprehensive evaluation of its performance across diverse scenarios.**
2. **Seed Storage:**
    a. **Each run of the genetic algorithm initializes the random number generator with a unique seed, ensuring reproducibility and fairness in testing. The list of seeds used across all runs is now stored and can be retrieved for reference or to replicate specific conditions in future experiments. This approach is critical for reproducibility in research and benchmarking purposes, as it allows the exact same sequence of random numbers to be used when required.**

Multiple Runs with Seed Storage

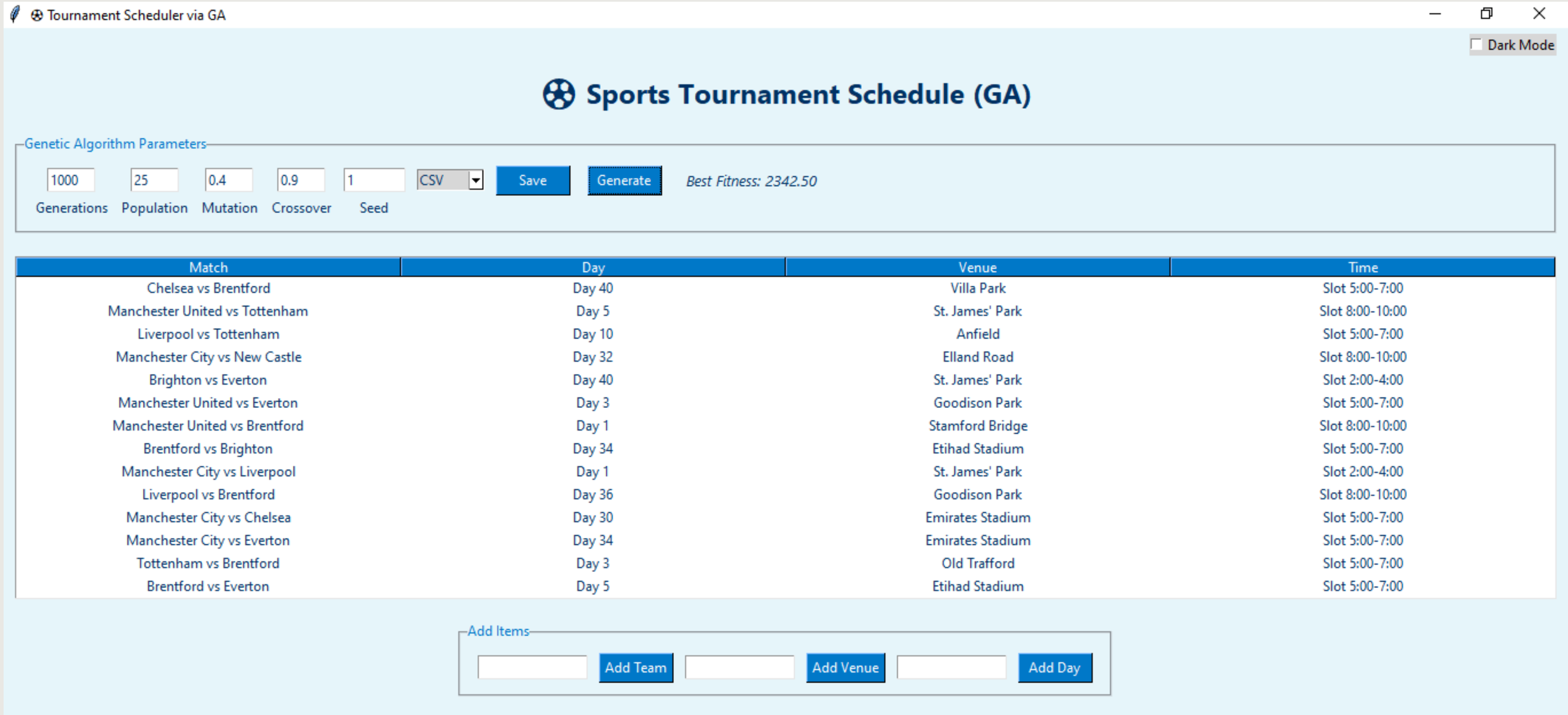
🛠️ How It Was Implemented:

- **Seed Generation: A list of 30 unique seeds is generated at the start of the experiment. These seeds are used to initialize the random number generator before each run of the genetic algorithm. By controlling the randomness in this manner, we ensure that each run starts from a distinct point in the solution space, promoting diverse search paths and preventing the algorithm from getting trapped in suboptimal solutions.**
- **Seed Storage: The seeds are stored in a list and can be printed or saved to a file for future reference. This provides a clear traceability of the initialization process and supports the possibility of re-running specific settings to verify or compare results.**

📚 Purpose of Multiple Runs and Seed Storage:

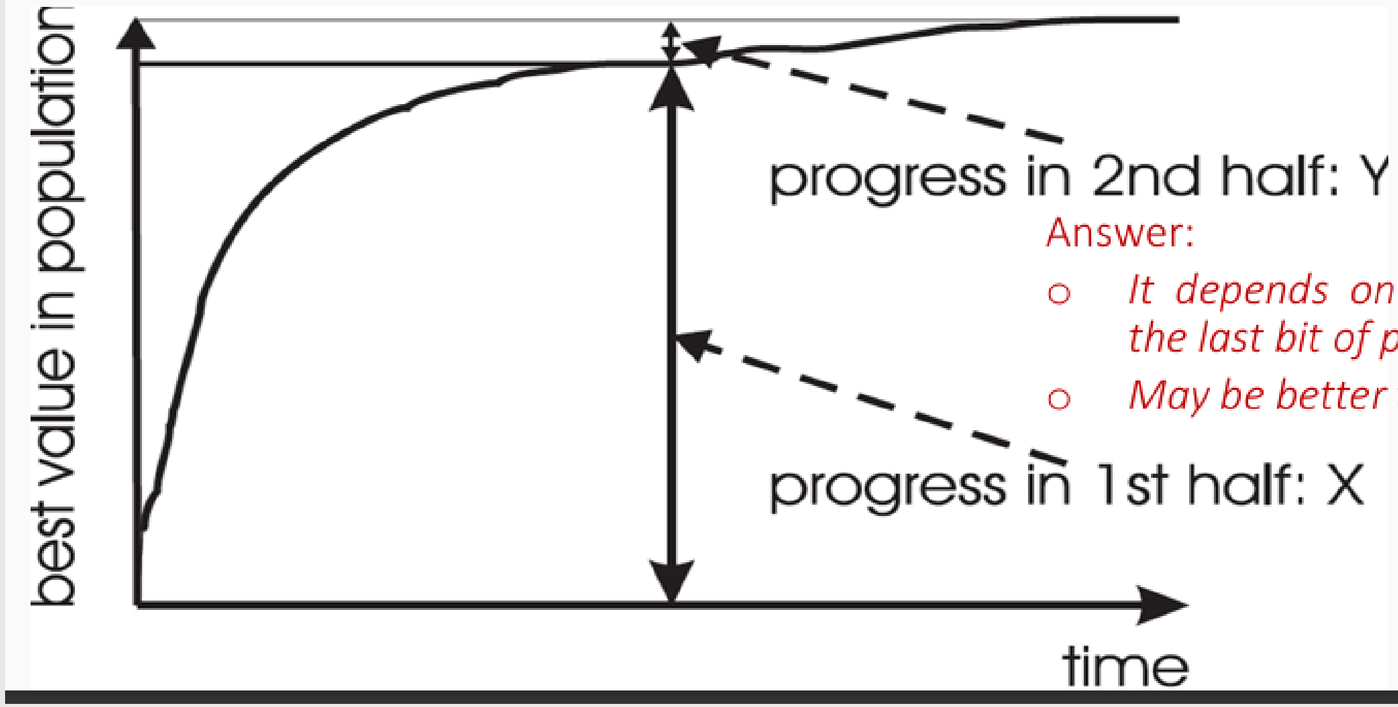
- **Increased Reliability: Conducting 30 runs ensures that any observed result is not an artifact of a single run's randomness. The variability in outcomes is reduced, giving more confidence in the algorithm’s ability to consistently find high-quality solutions.**
- **Reproducibility: By storing the seeds, the exact conditions can be replicated later. This is especially important in scientific research where reproducibility is a cornerstone of validating results.**
- **Statistical Significance: Multiple runs provide a wider data set, enabling the calculation of average performance metrics, such as the mean fitness score and standard deviation, which are useful for evaluating the algorithm's effectiveness and stability.**

EXAMPLE OF SEED



Visualization of Fitness

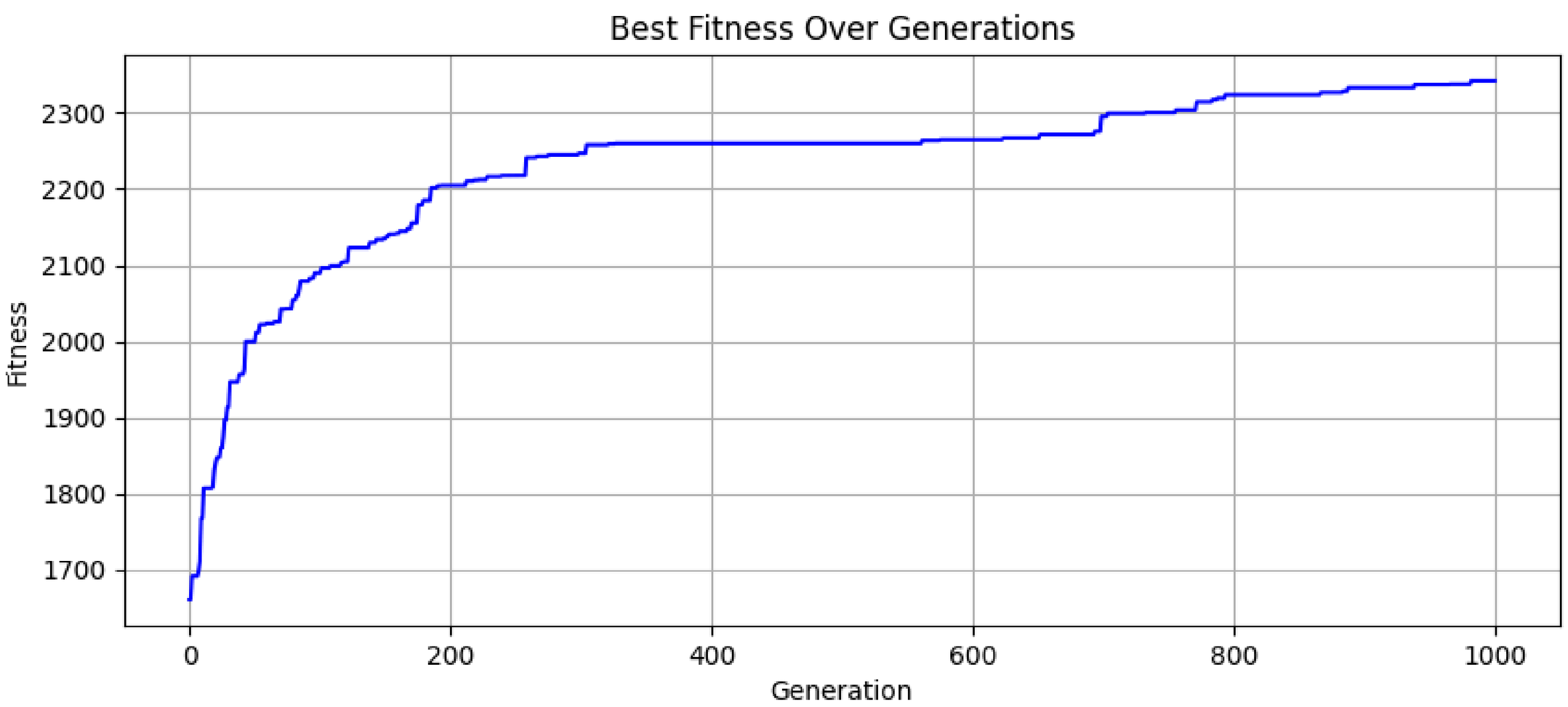
Typical EA Behaviour:
Are Long Runs Beneficial?



Answer:

- *It depends on how much you want the last bit of progress.*
- *May be better to do more short runs.*

Visualization of Fitness



Used Tools 📚 :

1. **tkinter 🎨: A library for building the GUI (Graphical User Interface).**
2. **ttk 🖌️: A module from tkinter for themed widgets like buttons, entries, and labels.**
3. **filedialog 💾: A module from tkinter for opening and saving files.**
4. **messagebox ⚠️: A module from tkinter used to show alerts, warnings, or error messages.**
5. **threading 🔄: Used to run the genetic algorithm in the background to keep the UI responsive.**
6. **random 🎲: Generates random values for tasks like scheduling and mutation.**
7. **itertools 🔢: Provides powerful functions like combinations() for generating matchups.**
8. **collections 📚: A module providing useful containers like defaultdict for managing data.**
9. **csv 📊: A module for reading/writing CSV files, used for saving the tournament schedule.**
10. **combinations 🔄: Used to generate all possible pairs of teams for matchups.**
11. **defaultdict 🤖: A dictionary subclass that returns a default value when a key doesn't exist.**
12. **shuffle 🔀: A method from random to randomize lists (like available slots for scheduling).**
13. **set 🔒: Data structure used to track used slots and days to ensure no conflicts.**
14. **list 📝: Used to create lists for team/venue/day data, and to store the schedule.**
15. **zip 🔗: Combines iterables, used in comparing team matchups in the fitness function.**
16. **Exception Handling 🚨: Used to manage errors (e.g., when there are too many matches for available slots).**
17. **GUI Elements 🖱️: Labels, entries, buttons, and treeviews for user input and displaying data.**
18. **pandas 📊🐼: A powerful library for data manipulation and analysis, helpful for managing large datasets like match schedules.**
19. **matplotlib 📈🎨: A plotting library for creating visualizations, such as graphing the fitness progress or match distribution in the tournament.**

REFRENCEs 📚:

Gopal, G. (2015). Enhanced Order Crossover for Permutation Problems. International Journal of Innovative Research in Science, Engineering and Technology, 4(2), 151-157.

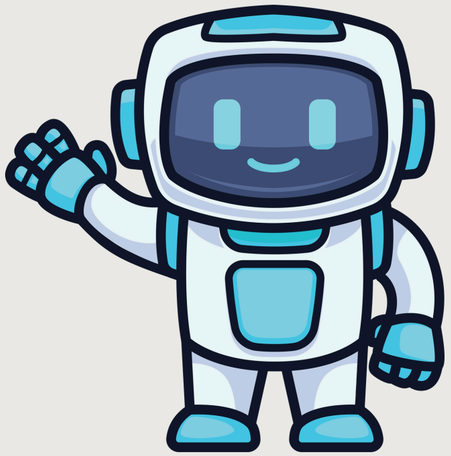
Rutjanisarakul, T., & Jiarasuksakun, T. (2017). A sport tournament scheduling by genetic algorithm with swapping method. arXiv.

Global constraints for round robin tournament scheduling Martin Henz a,*, Tobias Muller € b , Sven Thiel c

Evolutionary Algorithms [AI420] An Introduction to Applied Evolutionary Computation and Nature-Inspired Algorithms by Dr.Amr S. Ghoneim

CI Lectures by Dr.Amr S. Ghoneim

THANK YOU !



FOR YOUR ATTENTION .

DR.AMR