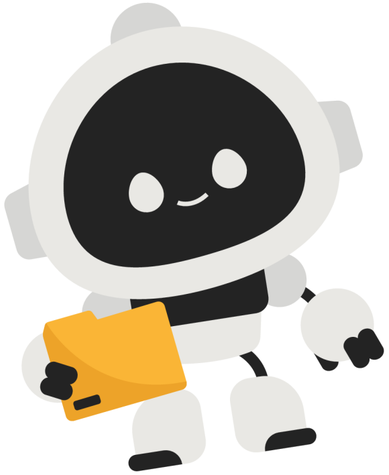
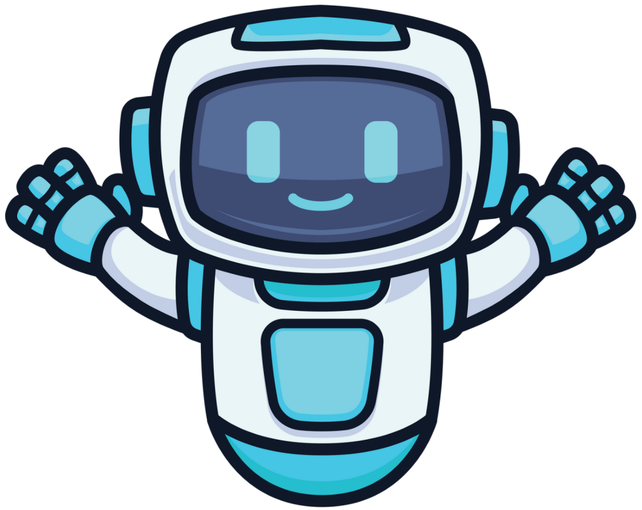
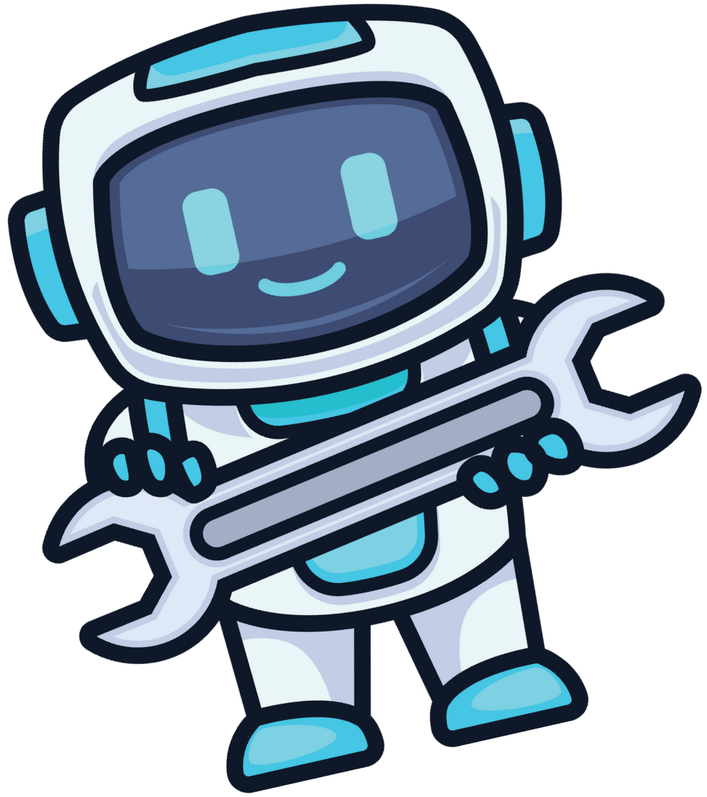
SPORTS TOURNAMENT SCHEDULING VIA : GENETIC ALGORITHMS (GA)





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.

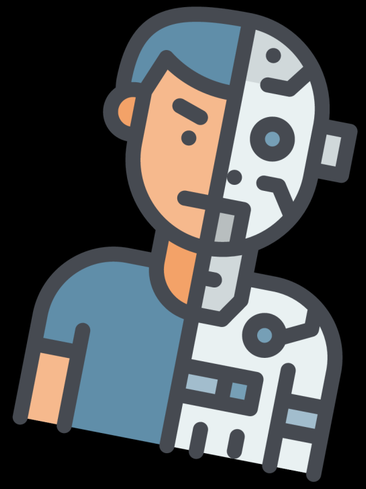
Why use Genetic Algorithms (GA)?

- 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.

Constraints



🔒 Hard Constraints (Must be strictly satisfied):

1. No Repeated Matches Between the Same Teams
   - Explanation: Each pair of teams should only play once.
   - Implementation: In is_valid_schedule, if a team pair is found more than once, the schedule is considered invalid.
2. No Team Plays More Than One Match at the Same Time
   - Explanation: A team cannot participate in more than one match in a single round.
   - Implementation: In is_valid_schedule, each round is checked to ensure no team appears in more than one match.
3. All Matches are Between Two Distinct Teams
   - Explanation: No match should be between the same team (e.g., A vs A).
   - Implementation: Checked during schedule creation and validation to ensure that team1 != team2

Constraints

🌿 Soft Constraints (Preferred but not mandatory, used to evaluate fitness):

1. Even Distribution of Matches Over Rounds
    - Explanation: Matches should be evenly spread to ensure fairness (not too many games clustered together for any team).
    - Implementation: In the fitness function, repeated team pairs are penalized, encouraging an even distribution.
2. Avoiding Repetitions and Clashes in Mutation
    - Explanation: Mutation should avoid creating invalid schedules.
    - Implementation: Mutation swaps matches and then checks validity using is_valid_schedule

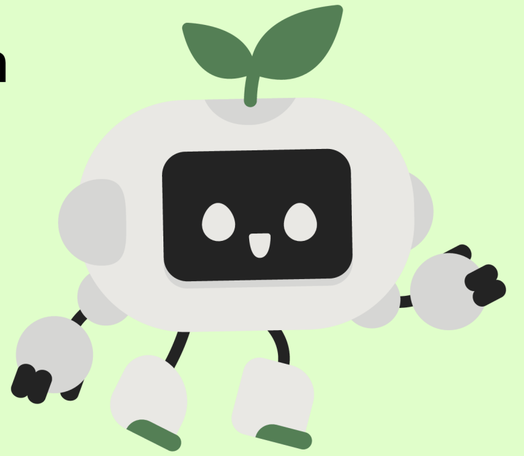
Constraints

⚙️ How Constraints Are Handled in the Genetic Algorithm:

1. During Initialization:
   - generate_initial_population() ensures only valid schedules are added to the initial population.
2. During Crossover:
   - Crossover only accepts a child schedule if is_valid_schedule(child) returns True.
3. During Mutation:
   - After mutation, the new individual is only accepted if it’s still a valid schedule.
4. Fitness Function:
   - Penalizes repeated team matchups and potential conflicts, guiding the algorithm toward soft constraint satisfaction.

Representation

In the sports tournament scheduling problem, we use a permutation-based representation to model how matches are scheduled. This means that each individual (or solution) in our population represents a complete schedule for the tournament. The goal is to allocate teams to matches and assign them to specific days, time slots, and venues, all while ensuring that the constraints of the tournament are satisfied



Representation

Key Features of the Representation

1. **Match Representation ⚔️:**
    - **Each match is represented as a tuple with the following format: (team1, team2, day, time_slot, venue):**
        - **team1 and team2: The two teams that will play in the match 🤾.**
        - **day: The specific day when the match is scheduled 📅.**
        - **time_slot: The time of day when the match will take place ⏰.**
        - **venue: The location where the match will happen 🏟️.**
2. **Chromosome Structure 🧬:**
    - **Each individual (chromosome) in the population is a list of all the matches in the tournament 📝. Each element of the list corresponds to a scheduled match.**
    - **For example, in a tournament with 10 matches, the individual is a list of 10 tuples, where each tuple contains the match details.**

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

preserving diversity

🔄 Fitness Sharing Function 📊

The Fitness Sharing Function is used to adjust the fitness of an individual in a population by penalizing solutions that are too similar to others. This helps preserve diversity in the population, ensuring that the algorithm explores a wider range of solutions rather than converging too quickly to local optima. 🌍

🔍 Similarity Calculation 🤝

- **Purpose: The function calculates how similar an individual is to others in the population.**
- **Method: The similarity between two individuals is determined by comparing their solution components (in this case, the first 3 elements) to see how many match.**
- **Output: A similarity score between 0 and 1, with 1 indicating exact similarity. 🧩**

preserving diversity

👫 Penalty Calculation 🚫

- **Purpose**: To apply a penalty when an individual is similar to others in the population, thus encouraging diversity.
- **Method**: If the similarity score between two individuals is less than a threshold (`sigma_share`), a penalty is applied. The penalty increases with the number of similar individuals, controlled by the parameter `alpha` . ⚖️
- **Penalty Function**: The penalty is calculated as:

$$\text{penalty} = 1 - \left(\frac{\text{similarity}}{\sigma_{\text{share}}} \right)^{\alpha}$$

- **Effect**: This decreases the fitness of similar individuals, promoting diversity in the population. 🔄

preserving diversity

🎯 Adjusted Fitness Calculation 💪

- **Purpose:** To calculate the individual's adjusted fitness by considering both its raw fitness and the penalties for similarity.
- **Method**: The raw fitness is calculated using a predefined fitness function. Then, the adjusted fitness is determined by dividing the raw fitness by the sum of penalties:

$$\text{adjusted fitness} = \frac{\text{raw fitness}}{1 + \text{penalty sum}}$$

- **Effect**: This ensures that individuals who are too similar to others receive a lower fitness, preserving diversity in the population. 🌱

preserving diversity

⚙️ Parameters 📐

- `sigma_share` (default: 0.3) 📊: Controls the threshold for similarity. If individuals are more similar than this value, penalties are applied.
- `alpha` (default: 1) 🔺: Controls the steepness of the penalty function. A higher value increases the penalty for similar individuals.

🎯 Key Benefits 🌟

- **Diversity Preservation** 🌍: Ensures that the population doesn't converge prematurely to a single solution, which can help in finding global optima.
- **Adaptability** 🔄: Allows the algorithm to explore different regions of the solution space by adjusting the fitness based on the similarity between individuals.

Reference Of Fitness Sharinf

$f(i) \equiv f_{original}(i), while\ f'(i) \equiv f_{shared}(i)$

Approaches *for* Preserving Diversity

Explicit Approaches | Fitness Sharing

• Restricts the number of individuals within a given niche by sharing/scaling their fitness to allocate individuals to niches in proportion to the niche fitness.

– In other words, Fitness sharing reduces the fitness of individuals within a given niche by scaling/sharing their fitness in proportion to the niche's size. Thus, it controls the number of members in the niche since individuals are allocated to niches in proportion to the niche fitness.

– Need to set the size of the niche σ_{share} in either genotype or phenotype space.

– Run EA as normal, but after each generation set:

$$f'(i) = \frac{f(i)}{\sum_{j=1}^{\mu} sh(d(i,j))} \qquad sh(d) = \begin{cases} 1 - d/\sigma & d \leq \sigma \\ 0 & otherwise \end{cases}$$

15

Evolutionary Algorithms [AI420] An Introduction to Applied Evolutionary Computation and Nature-Inspired Algorithms Lecture 6.0

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:**
2. **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.**
3. 🧠 **Fitness Calculation:**
4. **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 shared_fitness() function, which computes the fitness score.**
5. 🏅 **Winner Selection:**
6. **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.**
7. 🏆 **Return Winner:**
8. **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:**
4. **Two random points are selected to identify a slice in each parent.**
5. 📦 **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.**
6. 🧩 **Slot Reassignment:**
    - **After creating a valid list of team matchups, new time, day, and venue slots are assigned.**
    - **Constraints like "one match per team per day" and "no duplicate slot usage" are enforced**

🔐 Constraints Enforced

✅ **No team plays more than once on the same day**

✅ **Each time slot (day, time, venue) is used only once**

✅ **Every team pair plays exactly once**

✅ **The full schedule remains valid and conflict-free**

🔄 Swap Mutation

🎯 **Purpose :**

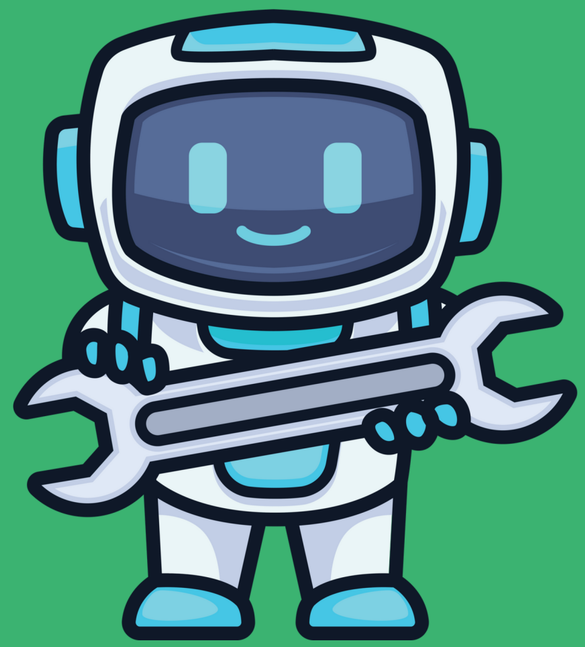
The Swap Mutation function introduces variability into the population by modifying an individual schedule. It helps the genetic algorithm escape local optima and explore new areas of the solution space.

This function is applied probabilistically based on a mutation rate

🔐 **Constraints Enforced**

- Only one match's time and venue are mutated per mutation operation.
- 🎯 The swapped schedule maintains the same set of matchups (no duplicates or deletions).
- ⛔ Slot conflicts (e.g., same team playing twice on the same day or venue overlap) are not explicitly checked, so this mutation assumes subsequent repair or validation is applied later.

🔄 Swap Mutation



⚙️ **How It Works**

1. 🧪 **Mutation Trigger:**
2. **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.**
3. 🔀 **Match Swap:**
4. **Two matches are randomly selected from the schedule and swapped.**
5. **This maintains the permutation property but may alter the overall tournament flow.**
6. 🛠️ **Slot Mutation:**
7. **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 time_slots ⏰**
   - **Replacing its venue with a random value from venues 🏟️**
8. ✅ **Schedule Update:**
9. **The schedule is scanned for the match being mutated (by team pair), and its entry is updated with the new slot info.**

Run Genetic Algorithm

🎯 **Purpose:**

The run_genetic_algorithm function executes the genetic algorithm to find the optimal schedule based on the fitness of individuals (teams' matchups). It iteratively evolves the population over several generations, applying crossover and mutation operations to produce offspring, and selects the best individuals through fitness evaluation. The process continues until certain stopping criteria are met.

⚙️ **Parameters:**

- population_size (int): Defines the number of individuals (schedules) in the population. Default is 10.
- max_generations (int): Specifies the maximum number of generations (iterations) to run the algorithm. Default is 100.
- convergence_threshold (float): The threshold for convergence. If the best fitness doesn't improve by more than this amount in consecutive generations, the algorithm stops. Default is 1e-3.
- max_no_improvement (int): The number of generations without fitness improvement before termination. Default is 10.
- mutation_rate (float): The probability of applying mutation on an individual. Default is 0.1.
- crossover_rate (float): The probability of performing crossover between two individuals. Default is 0.7.

Run Genetic Algorithm

⚙️ **How the Algorithm Works:**

1. **Initial Population:**
    a. **The population is initialized using the create_population function, which generates an initial set of random schedules.**
2. **Fitness Calculation:**
    a. **For each individual in the population, the fitness is evaluated using shared_fitness function, which calculates how well each schedule adheres to the constraints and objectives (e.g., minimizing clashes or optimizing rest days).**
3. **Termination Conditions:**
    a. **The algorithm tracks the best fitness in each generation. If the best fitness does not improve significantly (based on convergence_threshold) over a certain number of generations (tracked with generations_without_improvement), the algorithm terminates early.**
    - **The algorithm also stops if max_no_improvement generations have passed without improvement.**
4. **Dynamic Mutation Rate Adjustment:**
    a. **If the algorithm is not improving for several generations, the mutation rate is increased (up to 1.0), which allows for more exploration. If improvements are being made, the mutation rate is gradually reduced (down to 0.01).**

Run Genetic Algorithm

⚙️ **How the Algorithm Works:**

- **Selection and Reproduction:**
  a. **Selection: Two parents are selected for crossover using tournament selection (tournament_selection).**
  b. **Crossover: The order_crossover function is applied to produce two offspring.**
  c. **Mutation: The offspring undergo mutation using the swap_mutation function, which swaps two matches and randomly changes their time slot and venue.**
- **Survivor Selection:**
  d. **The next generation is formed by selecting the best individuals from the current population and offspring using survivor_selection. This ensures that the fittest individuals are carried forward.**
- **Result:**
  e. **After the algorithm terminates (either by meeting the maximum generation limit or the termination conditions), the best individual (schedule) and its fitness are returned.**

📝 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

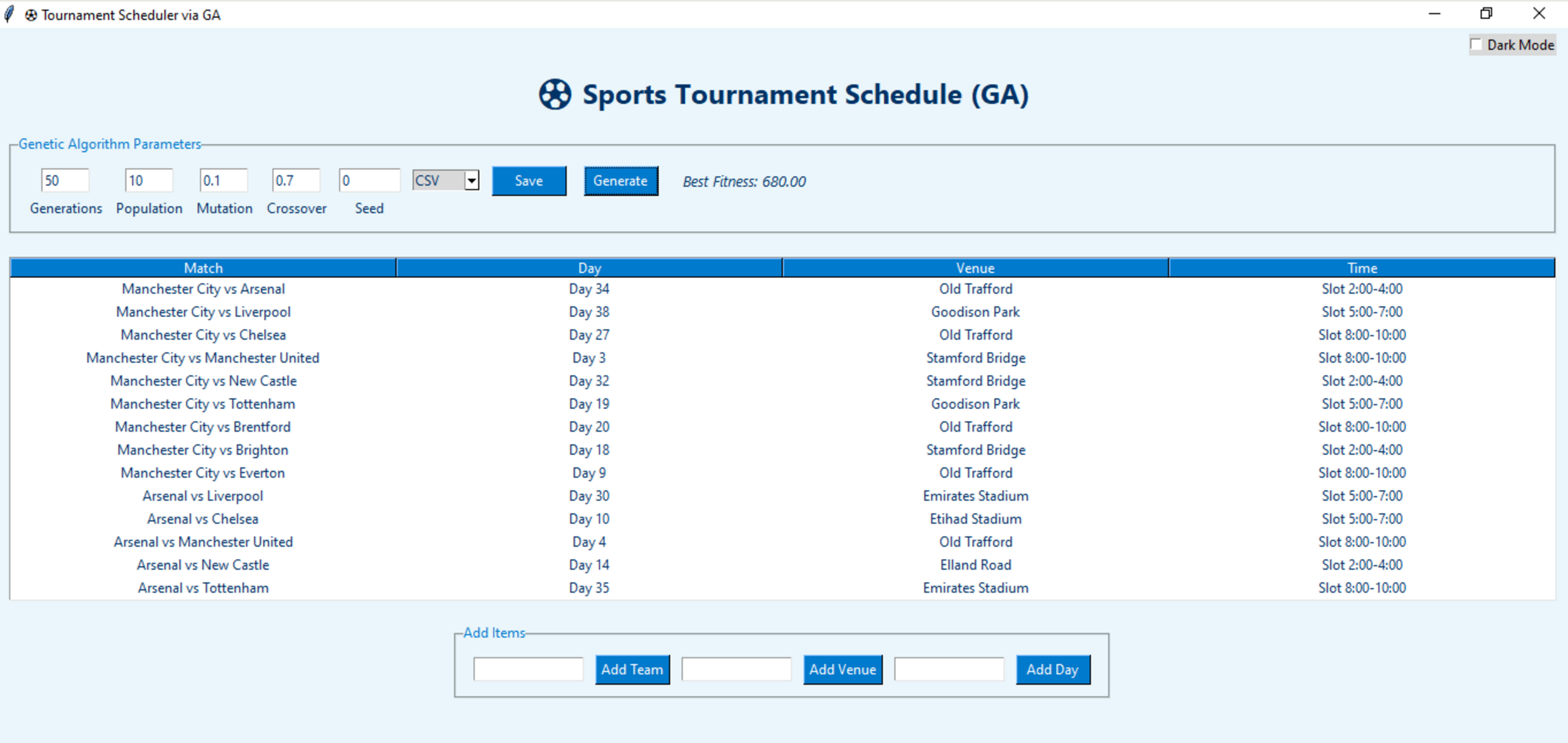
🛠️ How It Was Implemented:

- **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.**

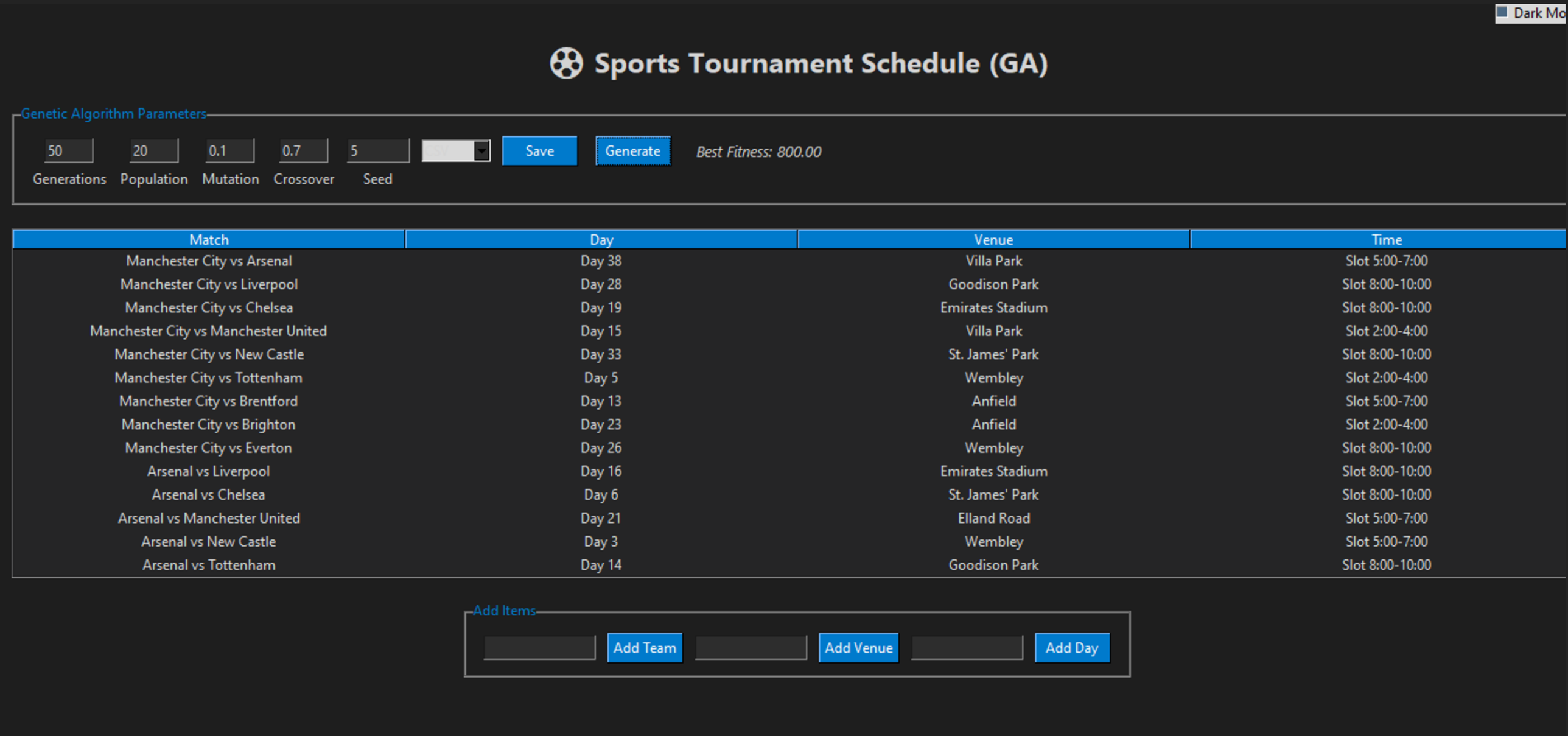
📚 Purpose of Multiple Runs and Seed Storage:

- **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.**

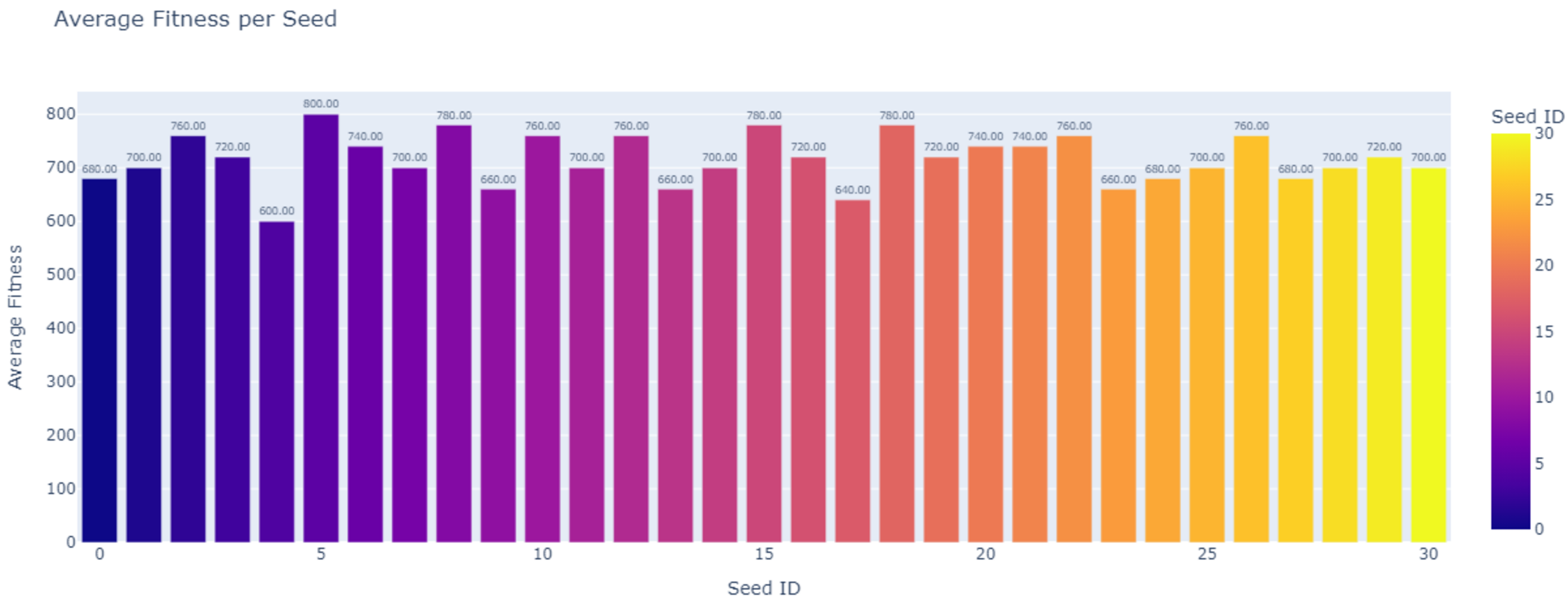
EXAMPLE OF SEED



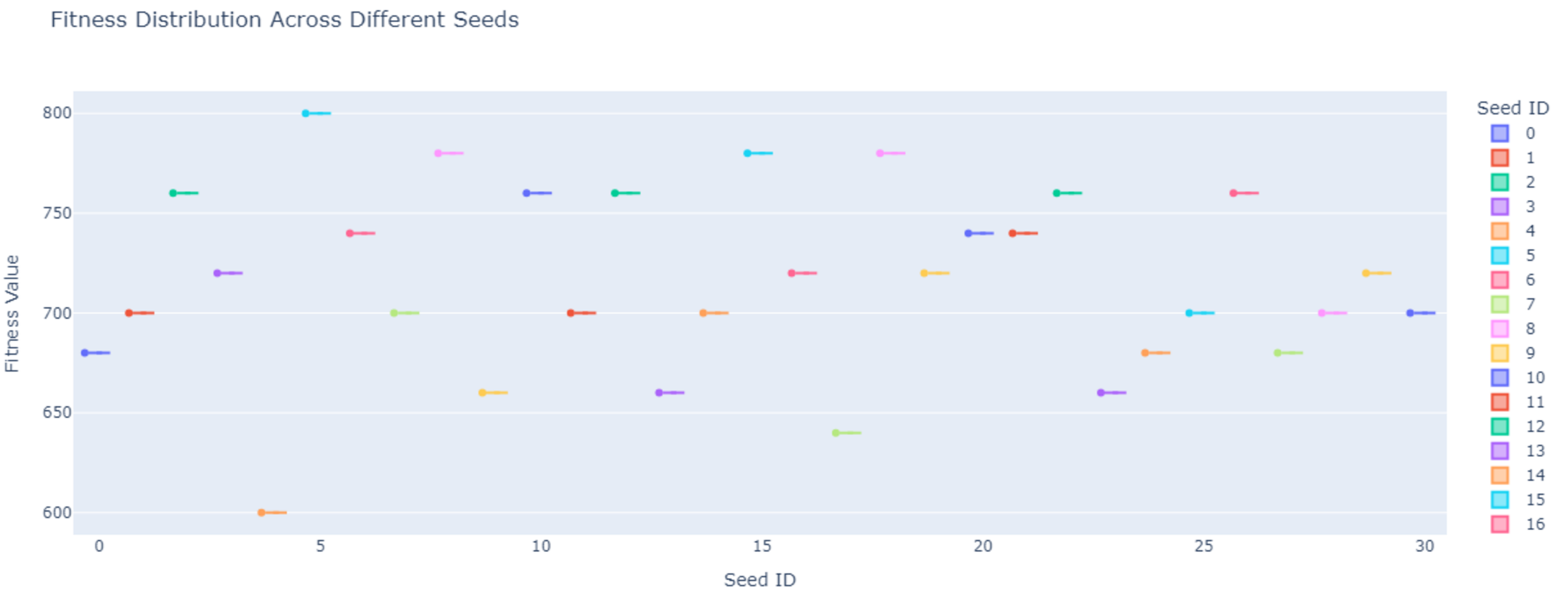
EXAMPLE OF SEED



Visualization of Fitness



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 📚:

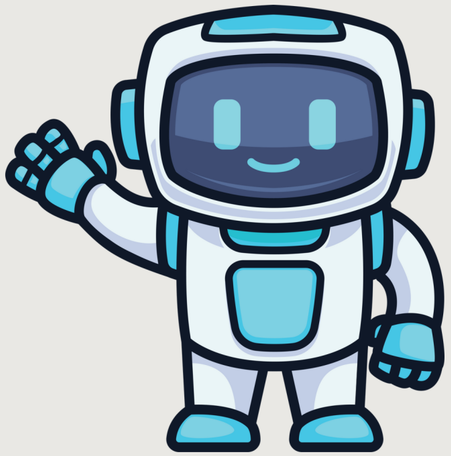
Gopal, G. (2015). Enhanced Order Crossover for Permutation Problems. International Journal of Innovative Research in Science, Engineering and Technology, 4(2), 151-157.

Rutjanisarakul, T., & Jiarasuksakun, T. (2017). A sport tournament scheduling by genetic algorithm with swapping method. arXiv. Gopal, G. (2015). Enhanced Order Crossover for Permutation Problems. International Journal of Innovative Research in Science, Engineering and Technology, 4(2), 151-157.

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

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

THANK YOU !



FOR YOUR ATTENTION .

DR.AMR