1. Building A Wedding Seating Planner

Specification of the work:

- **Objective**: Develop a wedding seating plan optimizer using *Simulated Annealing*, aiming to maximize guest satisfaction based on defined preferences (who they like to sit with and avoid). Balancing table sizes is also a key objective.
- **Methods**: Implemented *Simulated Annealing* with a custom cost function and neighbor generation. The system includes a Pygame-based UI to visualize arrangements and preferences. The code prioritizes creating balanced tables to enhance the feasibility and aesthetics of the seating plan.

Work done by **Group_A1_78**:

Francisco Miguel Pires Afonso (up202208115)
Miguel Moita Caseira (up202207678)
Pedro Trindade Gonçalves Cadilhe Santos (up202205900)

2. Project Structure

Custom Modules

- **ui.py**: Manages the user interface, including menu navigation, drawing seating arrangements, displaying guest preferences, and handling user input.
- **seater.py**: Encapsulates the core logic for seating arrangement generation, cost evaluation, neighbor creation, and the Simulated Annealing algorithm itself.
- **benchmark.py**: Used to evaluate and compare the different algorithms by running them multiple times with the same initial parameters.
- **plotting.py**: Generates a visual representation of each run, with relevant information such as cost evolution over time.
- **file_handler.py**: Handles the input of a .csv file and generates a text file with the results and final table arrangements of each run.

Key Libraries Used

Pygame: Powers the graphical user interface, allowing for interactive visualization of seating arrangements and guest preferences.

random: For generating random numbers and making probabilistic choices during neighbor generation and acceptance.

math: Used for mathematical operations, particularly in the cost function and acceptance probability calculations (e.g., exp, floor, ceil).

copy: Essential for creating deep copies of data structures to avoid unintended modifications during neighbor generation.

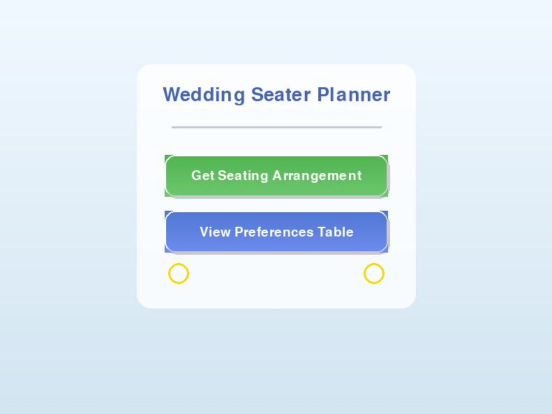
os: Used to generate the output file in a relative location.

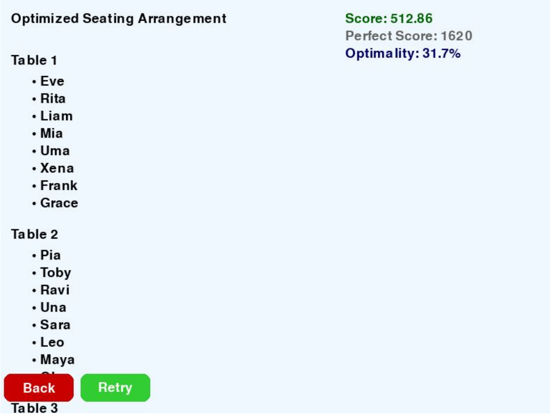
datetime: For naming of the different output files.

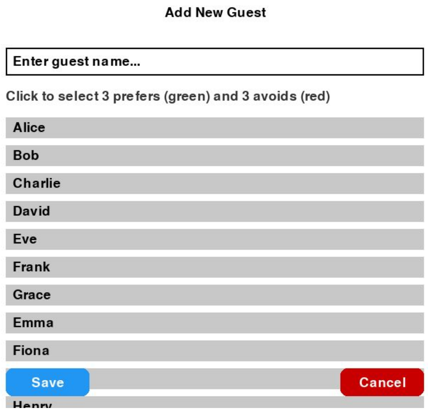
Matplotlib and plotting: Both of these libraries are used to create the visual graphs after each run of an algorithm finishes.

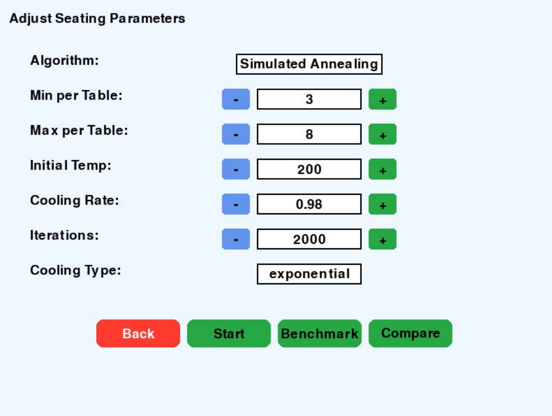
csv: Facilitates the reading of the .csv files in the file_handler.py file.

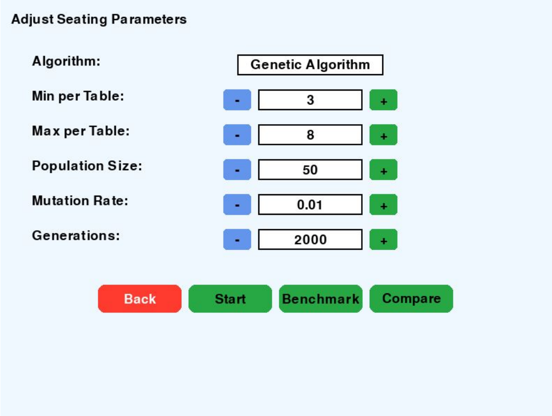
3. Graphical Interface (UI)

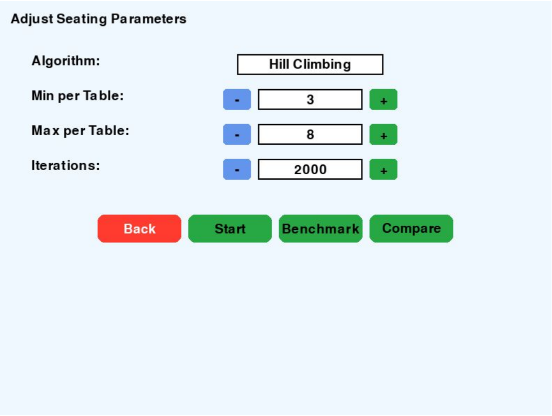












4. Formulation of the Optimization Problem

Solution Representation

A list of **tables**, where each table is a **list of guest names** (*strings*). Example: [["Alice", "Bob"], ["Charlie", "David", "Eve"]]

Cost Function

- ★ **Guest Preferences**: Lower cost for guests sitting with preferred individuals; higher cost for sitting with avoided individuals.

- ❖ **Table Balance**: High penalty for significantly unbalanced tables to encourage fair distribution of guests. Uses *calculate_cost* in *seater.py*.

Neighborhood Function

- ***Swap***: Swaps two guests between different tables.
- ***Move***: Moves a guest from one table to another.
- The function attempts to maintain balanced table sizes.

Hard Constraints

Table Capacity: The *create_balanced_seating* function attempts to create an initial seating arrangement within a *min_per_table* and *max_per_table* range, but this is not strictly enforced after the initial setup. This could be a potential improvement.

Balanced Tables: *create_neighbor* prioritizes swaps and moves that maintain relatively balanced table sizes.

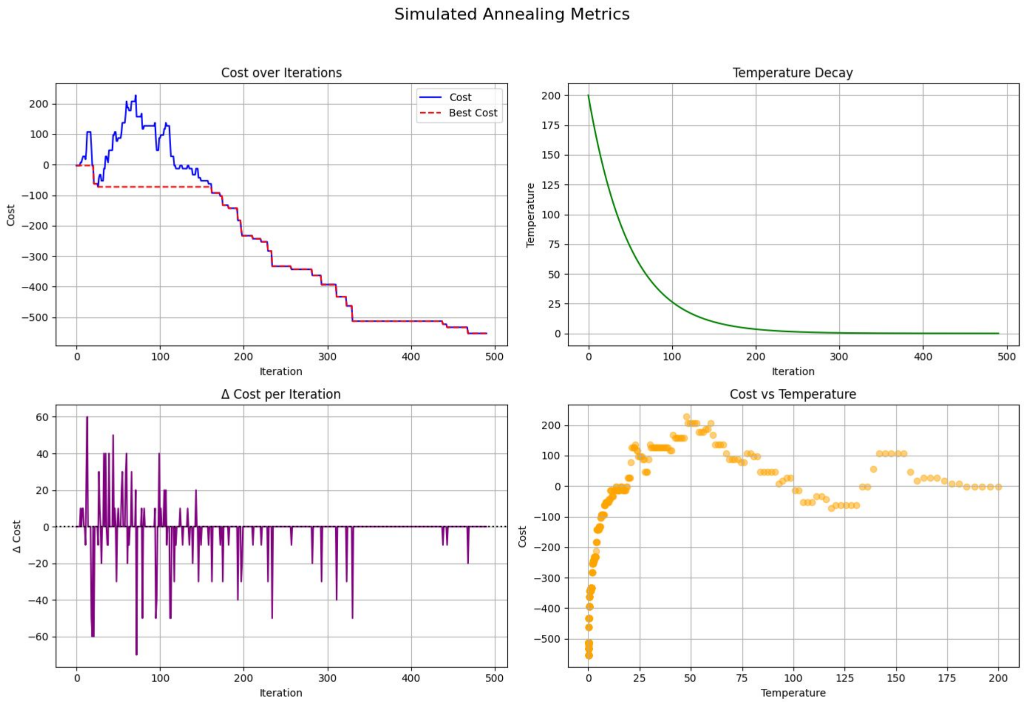
5. Simulated Annealing Implementation

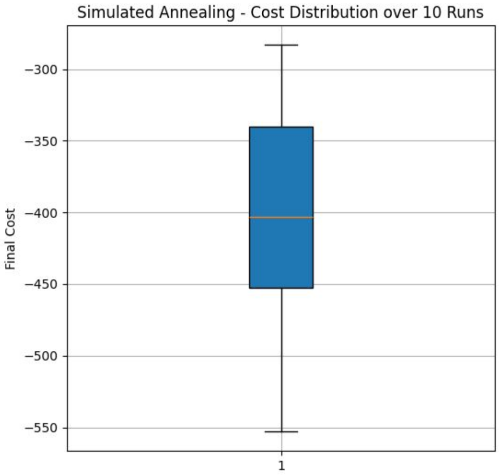
- Algorithm
    - The core Simulated Annealing logic is implemented in the ***simulated_annealing*** function in ***seater.py***.
    - The algorithm begins with an initial seating arrangement generated by ***create_balanced_seating***.
    - It iteratively explores neighboring solutions generated by ***create_neighbor***, accepting better solutions and, with a certain probability, worse solutions to escape local optima.

- Parameters (empirically chosen after multiple runs and benchmarks)
    - ***Min / Max per Table***: 3 and 8 guests respectively.
    - ***Initial Temperature***: Controlled by the *initial_temperature* parameter (default: 200).
    - ***Cooling Schedule***: Default Exponential decay, governed by the *cooling_rate* parameter (default: 0.98).
    - ***Acceptance Probability***: Calculated using the Metropolis criterion: *P = exp(-delta_cost / temperature)*.
    - ***Stopping Criterion***: The algorithm halts after a predetermined number of *iterations* (2000).

- Benchmarks and Comparisons
    - Our system not only generates single optimized seating arrangements, but also supports benchmarks, that runs the selected algorithm 10 consecutive times saving all seating arrangements and a boxplot; and comparisons that allows comparing the 3 algorithms implemented generating side-by-side boxplots.
    - In the next slide, we present a visualization of the optimization process using **Simulated Annealing**, including the evolution of cost, temperature and a benchmark boxplot to illustrate consistency across multiple runs.

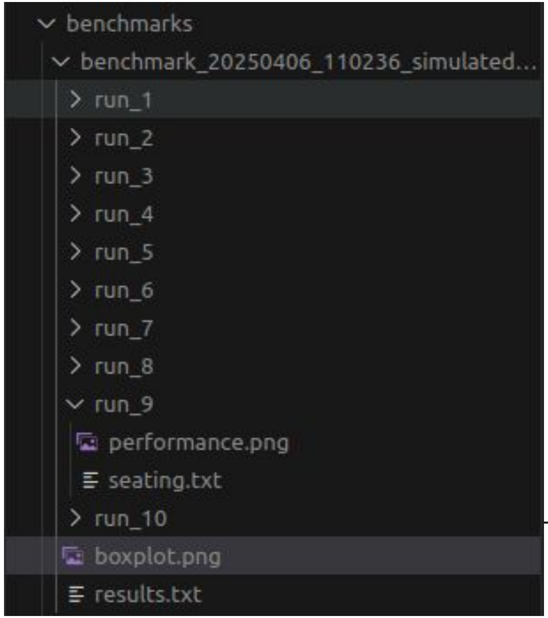
6. Simulated Annealing Results



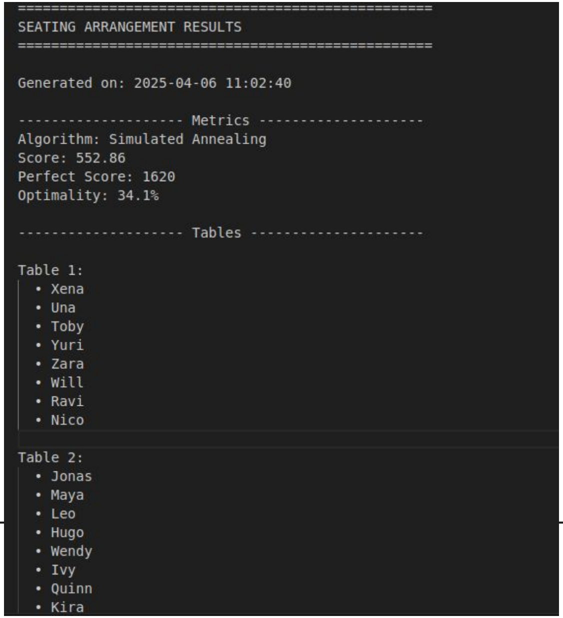


7. Simulated Annealing Outputs

Benchmarks Folder

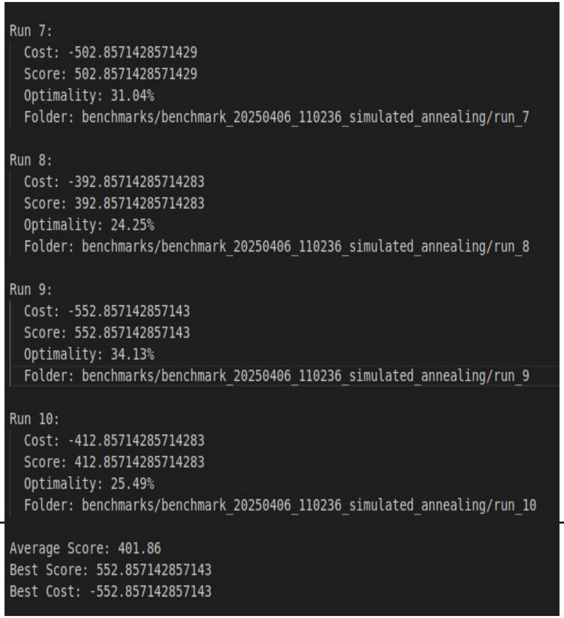


seating.txt (Seating Repres.)



```
==================================================
SEATING ARRANGEMENT RESULTS
==================================================

Generated on: 2025-04-06 11:02:40

-------------------- Metrics --------------------
Algorithm: Simulated Annealing
Score: 552.86
Perfect Score: 1620
Optimality: 34.1%

-------------------- Tables --------------------

Table 1:
  • Xena
  • Una
  • Toby
  • Yuri
  • Zara
  • Will
  • Ravi
  • Nico

Table 2:
  • Jonas
  • Maya
  • Leo
  • Hugo
  • Wendy
  • Ivy
  • Quinn
  • Kira
```

results.txt (Benchmark)



```
Run 7:
  Cost: -502.8571428571429
  Score: 502.8571428571429
  Optimality: 31.04%
  Folder: benchmarks/benchmark_20250406_110236_simulated_annealing/run_7

Run 8:
  Cost: -392.85714285714283
  Score: 392.85714285714283
  Optimality: 24.25%
  Folder: benchmarks/benchmark_20250406_110236_simulated_annealing/run_8

Run 9:
  Cost: -552.857142857143
  Score: 552.857142857143
  Optimality: 34.13%
  Folder: benchmarks/benchmark_20250406_110236_simulated_annealing/run_9

Run 10:
  Cost: -412.85714285714283
  Score: 412.85714285714283
  Optimality: 25.49%
  Folder: benchmarks/benchmark_20250406_110236_simulated_annealing/run_10

Average Score: 401.86
Best Score: 552.857142857143
Best Cost: -552.857142857143
```

8. Other Implemented Algorithms

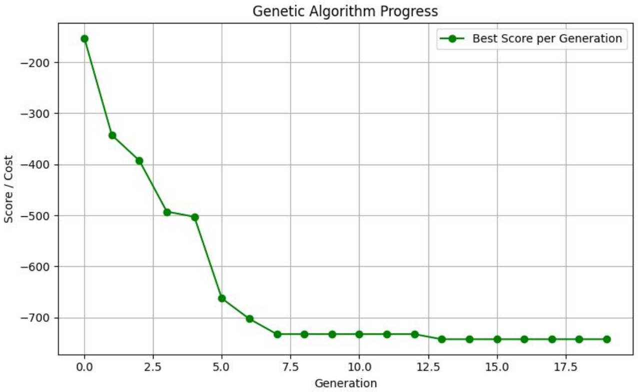
Genetic Algorithm

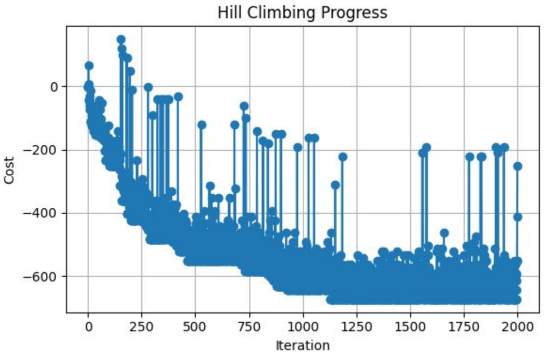
- **Idea**: Inspired by natural selection, this algorithm evolves a population of seating plans using genetic operations.
- **Key Steps:**
    - ❖ Initialization: Generate a population of valid, balanced seatings.
    - ❖ Crossover & Mutation: Combine and mutate individuals to create diversity.
    - ❖ Selection: Choose the best-performing individuals to survive to the next generation.
- **Pros**: Capable of escaping local optima and exploring diverse solutions.
- **Cons**: Slower convergence; requires careful tuning of parameters (e.g., mutation rate, population size).

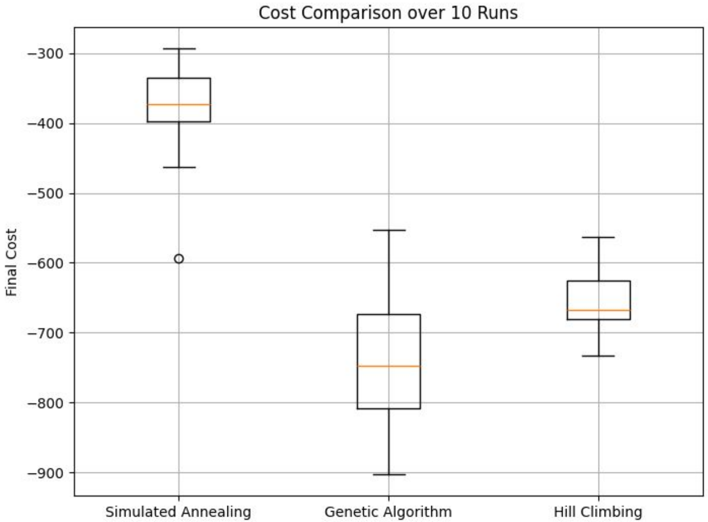
Hill Climbing (Greedy)

- **Idea**: Starts with a solution and iteratively moves to a better neighbor based on cost.
- **Key Steps:**
    - ❖ Generate an initial balanced seating using the same method as Simulated Annealing.
    - ❖ At each iteration, generate a neighbor and accept it only if it improves the cost.
    - ❖ Keep track of the best solution found so far.
- **Pros**: Very fast and easy to implement. Serves as a solid baseline for comparison.
- **Cons**: Highly prone to getting stuck in local optima; does not accept worse solutions to explore other paths (no diversification).

9. Comparison between algorithms







10. Work Conclusions

Our main algorithm, **Simulated Annealing**, proved to be the most balanced approach:

- **Good solution quality** with consistent performance across runs.
- **Ability to escape local optima** by probabilistically accepting worse solutions.
- **Reasonable execution time**, making it suitable for multiple benchmark runs.

The **Genetic Algorithm**:

- Occasionally produced **high-quality results**, but with **significant variability**.
- Its population-based approach introduces diversity, but it **requires a long execution time**, which limits its practical usefulness for fast evaluations.

Hill Climbing, while simple and fast:

- Served as a solid **baseline**, but often got stuck in local optima.
- Results were consistently **worse than the metaheuristics**.

Through **automated benchmarks and visual plots**, we were able to: **Compare algorithms objectively**, **visualize the optimization process** of Simulated Annealing and **justify our final choice** with empirical evidence.

Final takeaway: Simulated Annealing strikes the best balance between performance, stability, and runtime for this problem

Related Works and used Material:

"Solving the Wedding Seating Problem Using Simulated Annealing" by Zhi Jing Eu

"Wedding Seating Optimization using Simulated Annealing" by Linan Qiu

IA Course Moodle Slides