Final Project Report – Universal TSP Analysis Tool

1. Project Overview

The goal of this project is to explore and analyze *heuristics for the Universal Traveling Salesman Problem (UTSP)* on the Euclidean plane, with a focus on identifying configurations that maximize the discrepancy between heuristic orderings and the optimal TSP solution.

The main challenge lies in reproducing, in practice, *backtracking sets*, which are identified in Lower Bounds For The Universal TSP On The Plane, authored by Cosmas Kravaris as a fundamental obstruction to achieving a low competitive ratio. In that work, Kravaris proves a theoretical lower bound of $C\sqrt{\log n / \log \log n}$ on the competitive ratio for any linear ordering of an n-point set in $[0,1]^2$. Our goal is to empirically confirm this lower bound by explicitly constructing and visualizing backtracking sets.

As linear orders (heuristics), we adopted space-filling curves like Hilbert and Z-order (Z-curve), as well as other ordering heuristics like the Platzman-Bartholdi order (based on Sierkpinski).

The project includes tools for:

- Generating point configurations on a grid.
- Applying various linear orders (heuristics) on a set of points.
- Applying the LK Heuristic for “optimal” TSP computation on a set of points.
- Computing the cost ratio between the linear order and the optimal TSP tour.
- Visualizing the results interactively.
- Detecting specific structural patterns such as backtracks and building sets with such patterns.

This work was supervised by Prof. Manor Mendel, with whom I had regular discussions to clarify the theoretical goals, validate algorithmic directions, and ensure that the final user interface is practical.

2. Interaction with Supervisor

Prof. Manor Mendel guided the research and helped refine the problem definition around the Universal TSP and the role of orderings.

Throughout the project, Manor provided feedback on the mathematical formulations, and we iterated together on the expected outputs (such as graphical features to include and clear “API” definition of a heuristic). The implementation was designed so that he could later run experiments and add heuristics easily.

---

3. Project Structure

The project is organized into two main components: a **computational core**, responsible for generating point sets, computing linear orders, running experiments, and evaluating costs; and an **interactive viewer**, dedicated to visualization and exploratory analysis of the results.

The directory structure is as follows:

```
.
├── launch_viewer.sh
├── README.md
├── results/
│   └── *.npz          # Experimental result files
├── results_cosmas/
│   └── *.npz          # Cosmas experiment result files
└── tsp_analysis/
    ├── cosmas.py
    ├── experiment.py
    ├── geometry.py
```

```
├── heuristics.py
├── plotting.py
├── tsp_solver.py
├── utils.py
└── viewer_tk.py
```

Core Modules:

- `experiment.py`: Implements the generic experimental logic. This module generates random point sets, applies linear ordering heuristics, computes path costs, evaluates competitive ratios, and stores results in serialized `.npz` files for later inspection.

- `heuristics.py`: Defines all linear ordering heuristics used in the project (Hilbert order, Z-order/Morton order, and a Platzman–Bartholdi approximation). Each heuristic maps a set of 2D points to a linear order via a one-dimensional code. All new ordering heuristics must be implemented in this module and registered in the central `heuristics_registry`

- `geometry.py`: Provides geometric utilities used throughout the project, including grid generation, basic geometric primitives (midpoints, centroids), and the construction of approximate Sierpiński curve support points. These tools are notably used to implement the Platzman–Bartholdi ordering heuristic and to reason about dyadic or hierarchical spatial decompositions.

- `tsp_solver.py`: Acts as a wrapper around an external Lin–Kernighan–Helsgaun (LKH) heuristic solver. It handles TSPLIB file generation, invocation of the solver, reconstruction of the resulting tour as a permutation of the original points, and computation of the total Euclidean path cost. This module provides a near-optimal baseline for comparison with heuristic linear orders.

- `cosmas.py`: This module implements a multiscale experimental realization of the Cosmas lower bound, including:
    - Dyadic decomposition of the unit square at multiple scales
    - Detection of local backtracking configurations
    - Selection of pivots of detected backtracks at each dyadic scale, lying in strip of random (local) line

    - Random (global) line drawing and selection of pivots lying in the line neighbourhood
    - Induced linear order on the selected set
    - Exact TSP computation via LKH

    - Fully interactive visualization:
        - global order vs TSP
        - mouse hover to inspect ranks and scales
        - click to open local dyadic backtrack views

- `utils.py:` Contains auxiliary utilities used by the experimental pipeline, such as randomized subset sampling and heuristic rules for allocating computational budgets depending on problem size.

- `plotting.py`: Provides lightweight visualization utilities based on Matplotlib. This module supports both static plots and simple interactive plots (via checkboxes) for displaying grids, point sets, and multiple paths simultaneously. It is primarily used for exploratory analysis and debugging.

- `viewer_tk.py`: Implements the main interactive graphical user interface using Tkinter, embedding a Matplotlib figure. The viewer allows users to browse available experimental results, select grid sizes, heuristics, and subset sizes, and interactively visualize heuristic paths alongside near-optimal TSP paths. Additional functionality includes toggling path visibility, recomputing paths after interactively modifying point positions, and generating new experiments directly from the interface.

- `launch_viewer.sh`: Convenience script to start the viewer from the command line.

---

5. API for Adding New Ordering Heuristics

The `heuristics.py` module is designed to be extensible and allows new linear ordering heuristics to be added in a uniform way. A *heuristic* is any algorithm that maps a finite set of two-dimensional points to a one-dimensional linear order, typically via a space-filling curve or an equivalent ordering rule.

Each heuristic must be implemented as a Python function with the following conceptual signature:

```
def my_heuristic(points: np.ndarray, **optional_parameters):
    ...
    return indices, codes
```

Input

- `points` (`np.ndarray` of shape `(N, 2)`):
  An array of `N` two-dimensional points, where each point is given by its Cartesian coordinates `(x, y)`.
  In the current project, points are assumed to lie in the unit square [0,1]^2.

- Optional parameters:
  A heuristic may define additional optional parameters if required by the algorithm (e.g. the order of a Hilbert curve or the number of bits for a Morton code). These parameters must have default values so that the function can be called with `points` alone.

Output

Each heuristic must return a tuple `(indices, codes)`, where:

1. `indices` (`np.ndarray` of shape `(N,)`):
   A permutation of the integers `{0, …, N-1}` representing the linear order induced by the heuristic.
   Applying this permutation to the input points, `points[indices]`yields the points sorted according to the heuristic.

2. `codes` (array-like, length `N`):
   A one-dimensional code associated with each input point (e.g. Hilbert distance, Morton code, or index on a curve support).
   The `codes` array is aligned with the **original input order** of `points` (one code per input point) and is typically used for debugging, visualization or reproducibility of the ordering.

Constraints and conventions

- `len(indices) == len(points)`
- `indices` contains each integer from `0` to `N-1` exactly once
- `codes` has length `N` and corresponds to the input points, not the sorted points

- The heuristic must handle the case `N = 0` gracefully by returning empty arrays

Registration

To make a new heuristic accessible to the rest of the system, it must be registered in the central dictionary:

```
heuristics_registry = {
    "Hilbert": hilbert_order,
    "Z-order": zcurve_order,
    "Platzman": platzman_order,
    "MyHeuristic": my_heuristic}
```

The dictionary key is the user-facing name of the heuristic, and the value is the corresponding function.
This architecture makes it easy to plug in and compare new approaches.

6. Interactive Visualization of Cosmas Results

The Cosmas experiment results are presented through an interactive global visualization that allows intuitive inspection of the induced linear order. When the user hovers the mouse over a pivot point, the visualization dynamically highlights the relative position of this point within the linear ordering: the hovered point is displayed in **yellow**, all points that **precede it in the linear order** are colored **blue**, and all points that **follow it** are colored **red**. This color coding provides immediate visual feedback on the structure of the ordering and makes local backtracking phenomena clearly visible.

Clicking on a pivot point opens a dedicated local view of the corresponding dyadic square, showing the associated backtracking configuration in detail. This local view focuses on the dyadic square in which the pivot was detected and shows the corresponding geometric structure: the pivot point, the local reference line, the two rectangular regions defining the backtrack, and the selected backtracking triple.

---

7. Maintenance Guidelines (For Prof. Mendel)

- To launch the GUI, simply run:
  `bash launch_viewer.sh`
- All random point sets are saved in `.npz` format in the `results/` folder.
- All cosmas backtracking set results are saved in `.npz` format in the `results_cosmas/` folder
- The code has been tested on Python 3.9 and uses only common scientific libraries (`numpy`, `matplotlib`, `scipy`, `tkinter`). The `requirements.txt` file includes all dependencies.

Important note about the `lk_heuristic` dependency

In order to properly run experiments with the `solve(...)` function from the `lk_heuristic` library, we use a **modified version of the original library** to expose the internal `best_tour` directly. I forked

Why this is needed

The default `solve(...)` function in `lk_heuristic` does not return the best tour directly. For our research/experimentation purposes, we need access to this information for further analysis and visualizations.

What was changed

In the file `lk_heuristic/utils/solver_funcs.py`, we added at the end of the `solve(...)` function:

```
return best_tour
```

See the commit
https://github.com/chaisarfati/lk_heuristic/commit/6613e01d115939698b9860dfef0d451981a5c040

This allows users to get the optimized TSP tour directly in their code.