Dynamic LEC with Voronoi Diagrams

Overview

This project provides an interactive Python GUI application for visualizing and computing the Largest Empty Circle (LEC) in the presence of dynamically moving obstacles and user-defined Voronoi sites. It leverages Tkinter for the interface and Matplotlib for real-time plotting, supporting dynamic animation of obstacles, Voronoi diagram generation, convex hull visualization, and efficient LEC computation.

Features

- Interactive GUI: Add, select, and manage Voronoi sites and obstacles via mouse clicks and control buttons.
- Dynamic Obstacles: Define obstacle paths and animate their movement at adjustable speeds.
- Visualization Options: Toggle overlays for Voronoi edges, convex hull, and LEC.
- Optimized LEC Computation: Efficiently computes the largest empty circle avoiding all sites and current obstacle positions.
- Statistics Panel: Real-time display of site and obstacle counts, and animation time.
- Clear and Reset: Easily clear all data or reset animation time.

How to Use

- Add Sites: Select "Add Sites" mode and click on the canvas to place Voronoi sites (red dots).
- Add Obstacles: Switch to "Add obstacle" mode. Click to define obstacle paths; each click adds a waypoint. Obstacles are animated along these paths.
- Animation Controls: Use "Play" to start/stop obstacle animation and "Reset" to return to the initial state.
- Display Options: Toggle Voronoi Diagram, Convex Hull, and Largest Empty Circle overlays.
- View Statistics: See the number of sites, obstacles, and current animation time in the statistics panel.

- Clear All: Remove all sites and obstacles to start fresh.

Architecture

Main Components

- DynamicLECVoronoi Class: Manages state, GUI, and all core logic.
- Tkinter GUI: Arranges controls, statistics, and the Matplotlib canvas for visualization.
- Matplotlib Plotting: Renders sites, obstacles, paths, Voronoi diagram, convex hull, and LEC in real time.
- Obstacle Animation: Uses threading for smooth, non-blocking updates.
- LEC Computation: Employs an optimized approach using Delaunay triangulation, boundary checks, and nearest-neighbor bisector intersections.

Core Algorithms and Their Time Complexity

The following table summarizes the time complexity of each major algorithm, based on the latest optimizations and the referenced time-complexity analysis[1]:

Algorithm/Function	Time Complexity	Description/Remarks
compute_voronoi (scipy)	$O(n\log n)$	Fortune's algorithm via scipy.spatial.Voronoi for planar diagrams.
compute_convex_hull	$O(n\log n)$	Graham scan; optimal for 2D convex hull.
get_current_obstacle_positions	$O(m)$	Linear in the number of obstacles; path interpolation per obstacle.
compute_lec (optimized)	$O(n\log n + nk)$	Delaunay triangulation (incremental or sampled), boundary/bisector checks; k is a small constant (e.g., 8). Much faster than previous $O(n^2)$ or $O(n^3)$ approaches.
draw	$O(n+m)$	Iterates through all sites and obstacles for plotting; dominated by subroutine complexities.
on_canvas_click	$O(1)$	Constant time to add a site or obstacle waypoint.
update_stats	$O(1)$	Simple string formatting and label update.
clear_all/reset_time	$O(1)$	Resets state variables; no iteration over data.

- n: Number of sites
- m: Number of obstacles
- k: Number of nearest neighbors (constant, e.g., 8)

Key Points

- Scalability: The optimized LEC computation is significantly more scalable and practical for large numbers of sites compared to brute-force methods.
- Practical Performance: For dozens to hundreds of sites and obstacles, all algorithms perform efficiently in practice.
- Rendering: Plotting is efficient but may become a bottleneck for very large scenes.
- Further Optimization: For extremely large datasets, spatial partitioning (e.g., k-d trees) can further accelerate nearest-neighbor queries in LEC computation.

Detailed Functionality

1. Voronoi Diagram Generation

- Uses `scipy.spatial.Voronoi` (Fortune's algorithm) for efficient computation.
- Plots Voronoi edges in blue (toggleable).

2. Convex Hull Computation

- Implements Graham scan for optimal 2D convex hull calculation.
- Plots the hull in green (toggleable).

3. Dynamic Obstacles

- Obstacles move along user-defined paths, interpolated in real time.
- Each obstacle has a radius and speed; movement is smooth and reversible along the path.

4. Largest Empty Circle (LEC) Computation

- **Optimized Approach:**
    - Generates candidate centers using Delaunay triangulation (incremental for small n, sampled for large n), boundary points, and intersections of perpendicular bisectors with boundaries.
    - For each candidate, computes the maximum possible radius avoiding all sites and obstacles.

- Early termination and duplicate elimination for efficiency.

- **Complexity:**
  - Candidate generation: $O(n\log\ n)$
  - Candidate evaluation: $O(nk)$ where k is a small constant.

5. Animation and Interaction

- Animation is handled in a separate thread for responsiveness.
- All user actions (adding sites/obstacles, toggling overlays, clearing/resetting) are immediately reflected in the visualization.

Example Usage

1. Start the application.
2. Add sites by clicking on the canvas in "Add Sites" mode.
3. Switch to "Add obstacle" mode and click to define paths for moving obstacles.
4. Press "Play" to animate obstacles.
5. Toggle overlays to view Voronoi diagram, convex hull, and LEC.
6. Monitor statistics and reset or clear as needed.

Dependencies

- Python 3.x
- Tkinter (standard library)
- Matplotlib
- NumPy
- SciPy

Notes

- The project is designed for educational and demonstrative purposes, focusing on clarity, interactivity, and algorithmic efficiency.
- For very large numbers of sites or obstacles, further optimizations (such as spatial data structures) may be beneficial.