# Contents



# Large Graph Path Finder (LGPF)

## Table of Contents

- Project Structure
- Future Work

---

## Abstract


The **Large Graph Path Finder (LGPF)** project benchmarks and evaluates multiple pathfinding algorithms on arbitrarily large graphs. It supports uninformed search (BFS, DFS, Depth-Limited), weighted search (Dijkstra, Bidirectional Dijkstra), informed search (Greedy Best-First, A*), and advanced heuristics including **ALT (A* with Landmarks and Triangle inequality)**. For graphs containing negative edge weights, the system falls back to **Bellman–Ford** .

Graphs are generated in JSON format and preprocessed to build fast lookup structures, detect uniform weights, and compute landmark tables for ALT on large graphs. During preprocessing, a **real-time timer and memory monitor** report progress directly to the terminal and log file, warning if runtime exceeds 60 seconds or memory usage exceeds 1 GB. Queries can then be executed from file or interactively, producing exact path costs and performance statistics


---

## Algorithms & Heuristics

### Used Algorithms & Heuristics

Pathfinding in large graphs requires balancing **accuracy** , **speed** , and **scalability** . In `pathfinder.py`, the algorithm is chosen automatically based on graph properties:

- If landmarks exist → **A* + ALT [USING]**
- If negative weights exist → **Bellman–Ford [USING]**
- If graph is very large → **A* (Zero Heuristic) [USING]**
- If graph is unweighted → **BFS [USING]**
- Otherwise → **Dijkstra [USING]**

This adaptive design ensures that queries are solved efficiently and correctly without the user having to manually choose the algorithm.

---

### Uninformed Search

- **BFS (Breadth-First Search) [USING]**

- *Use* : Unweighted graphs.
- *Pros* : Finds shortest path in edge count.
- *Cons* : High memory, ignores weights.

- **DFS (Depth-First Search)**
  - *Use* : Reachability checks.
  - *Pros* : Simple, low memory.
  - *Cons* : Not optimal, may miss goal in deep graphs.
- **Depth-Limited DFS**
  - *Use* : Same as DFS but avoids infinite paths.
  - *Pros* : Bound search depth.
  - *Cons* : Still not guaranteed optimal.

---

**Weighted Graph Algorithms**

- **Dijkstra [USING]**
  - *Use* : Weighted graphs, no negative edges.
  - *Pros* : Guarantees shortest path.
  - *Cons* : Slower for very large graphs.
- **Bidirectional BFS**
  - *Use* : Unweighted graphs, source–goal pairs.
  - *Pros* : Faster in practice than BFS.
  - *Cons* : Only works on unweighted graphs.
- **Bidirectional Dijkstra**
  - *Use* : Large weighted graphs.
  - *Pros* : Can halve search space.
  - *Cons* : Not fully optimized in current code.
- **Bellman–Ford [USING]**
  - *Use* : Graphs with negative edge weights.
  - *Pros* : Correct with negatives.
  - *Cons* : Very slow (O(VE)).

---

**Informed Search**

- **Greedy Best-First Search**
  - *Use* : Uses heuristic only (H).
  - *Pros* : Fast, goal-directed.
  - *Cons* : Not optimal.
- **A* [USING]**
  - *Use* : Weighted graphs with heuristics.
  - *Pros* : Optimal if heuristic is admissible.
  - *Cons* : Needs good heuristic, otherwise same as Dijkstra.

---

**Heuristics**

- **Zero Heuristic [USING]**
  - *Pros* : Always admissible, safe.
  - *Cons* : Reduces A* to Dijkstra.
- **Manhattan**
  - *Use* : Grid-like graphs.
  - *Pros* : Simple, consistent.
  - *Cons* : Only makes sense on grids.
- **Euclidean**
  - *Use* : Spatial graphs.
  - *Pros* : Geometrically intuitive.
  - *Cons* : Less useful on abstract graphs.
- **Chebyshev / Octile**
  - *Use* : Grid movement with diagonals.
  - *Pros* : Models 8-directional movement.
  - *Cons* : Not needed in non-grid graphs.
- **Haversine**
  - *Use* : Geographic (lat/lon).
  - *Pros* : Real-world Earth distance.
  - *Cons* : Costly, only relevant to maps.
- **ALT (A* + Landmarks + Triangle inequality) [USING]**
  - *Use* : Very large graphs.
  - *Pros* : Strong admissible heuristic, massive speedups for queries.

- *Cons* : Preprocessing is expensive (runs many Dijkstras)

**Uninformed Search**

- **BFS (Breadth-First Search)**
  - *Use* : Unweighted graphs.
  - *Pros* : Simple, guarantees shortest path in terms of edges.
  - *Cons* : Inefficient for large weighted graphs.
- **DFS (Depth-First Search, Depth-Limited)**
  - *Use* : Exploratory only, not used in weighted shortest-path.
  - *Pros* : Low memory footprint, useful for reachability.
  - *Cons* : Does not guarantee shortest path.

---

**Weighted Algorithms**

- **Dijkstra**
  - *Use* : Weighted graphs with non-negative edges.
  - *Pros* : Guarantees optimal paths, robust for medium graphs.
  - *Cons* : Slower for very large graphs; expands many nodes.
- **Bidirectional Dijkstra**
  - *Use* : Large graphs with a clear source-target pair.
  - *Pros* : Faster in practice by searching from both ends.
  - *Cons* : More complex implementation; not always optimal in sparse graphs.
- **Bellman–Ford**
  - *Use* : Graphs with negative edge weights.
  - *Pros* : Handles negative edges safely, guarantees correctness.
  - *Cons* : Very slow (O(VE)), not suitable for large graphs unless required.

---

**Informed Algorithms**

- **Greedy Best-First Search**
  - *Use* : Heuristic-based exploration.
  - *Pros* : Very fast, explores promising nodes first.
  - *Cons* : Not guaranteed to find optimal path.

- **A ***
  - *Use* : Weighted graphs with admissible heuristics.
  - *Pros* : Guarantees optimal paths, faster than Dijkstra if heuristic is strong.
  - *Cons* : Requires a good heuristic; fallback is same as Dijkstra if heuristic=0.

---

**Heuristics**

- **Zero Heuristic**
  - *Pros* : Safe, guarantees correctness.
  - *Cons* : Reduces A* to Dijkstra.
- **ALT (A* with Landmarks and Triangle inequality)* *
  - *Use* : Very large graphs (≥10k nodes).
  - *Pros* : Strong admissible heuristic, greatly reduces search space; queries become very fast after preprocessing.
  - *Cons* : Preprocessing cost is high (multiple Dijkstra runs for each landmark).

### Installation

No installation step is required — just ensure prerequisites are installed see Prerequisites and tested Environment & Versions all other OS is risk of the user.

---

### Quick Start

*note: this program is designed to run on Linux/Linux VMs OS, other OS is not fully supported, certain commandline is gear thorse the OS mentioned see Prerequisites*

1. See Usage and its subsection `3. Required files to run` then `4. Run the Path Finder *main program*`

### Prerequisites

The program runs on **Python 3.13+** and does not require installation beyond having the correct dependencies available.

- **Python** : ≥ 3.13 (tested on 3.13.5)
- **pip** : ≥ 25.0 (tested with 25.2)

- **Operating Systems Tested** :
  - Windows 11 Pro for Workstations
  - Ubuntu 25.04 (inside VirtualBox)
- **Required Libraries** :
  - `networkx` (graph handling)
  - `matplotlib` (visualization)
  - `psutil` (timer & memory monitor)
  - `pytest` (testing framework)

Optional but supported:

- `pandas` / `scipy` for extended analysis
- `graphviz` for extra visualization format

---

### Environment & Versions

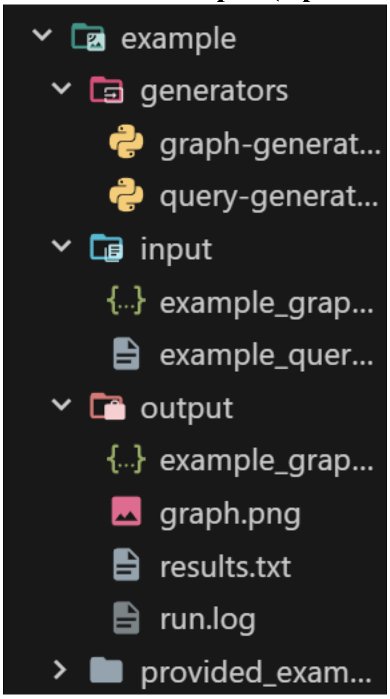
- **Python Version** : 3.13.5
- **pip Version** : 25.2
- **OS** : Windows 11 Pro for Workstations; Ubuntu 25.04 (VirtualBox)
- **Key Packages** :
  - `networkx 3.5`
  - `matplotlib 3.10.6`
  - `psutil 7.1.0`
  - `pytest 8.4.2`
- **Memory Tested** : Preprocessing up to ~1 GB RAM (monitored with psutil)
- **Hardware Used** : Intel Core i9 13980HX(24/32), 96 GB RAM

---

## Usage

### Run Main Program

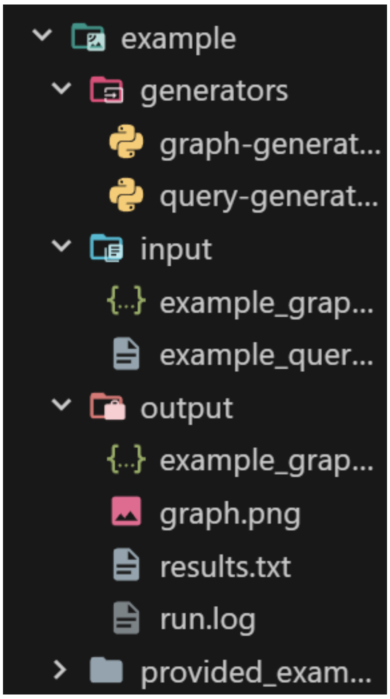
**1. Generate Graphs (Optional, if and only if sub section 3. Required files to run are satisfied)**




Use the provided generator to create test graphs in JSON:

```
# Example: 20k vertices, 60k edges
python3 example/generators/graph-generator.py \
  --num_vertices 20000 \
  --num_edges 60000 \
  --min_weight 1 \
  --max_weight 100 \
  --json_out input/large_graph.json \
  --img_out input/large_graph.png \
  --seed 42
```

**2. Generate Queries** Where to locate generator Python files under program root *example/generators*:




Create source–target pairs to test algorithms:

```
python3 example/generators/query-generator.py \
  --graph input/medium_graph.json
  --out input/medium_queries.txt
  --count 500
  --seed 0
```

or generate nodes that are all reachable *note: good luck on tring to perform 10K node on personal machines. generator will provide node pairs*:

```
python3 example/generators/query-generator.py \
  --graph input/medium_graph.json
  --out input/medium_reach.txt
  --count 500
  --seed 0
  --reachable-only
```

**3. Required files to run** *note: exsample files can be obtained in `exsample/input` or generate see this subsection 1 and 2*

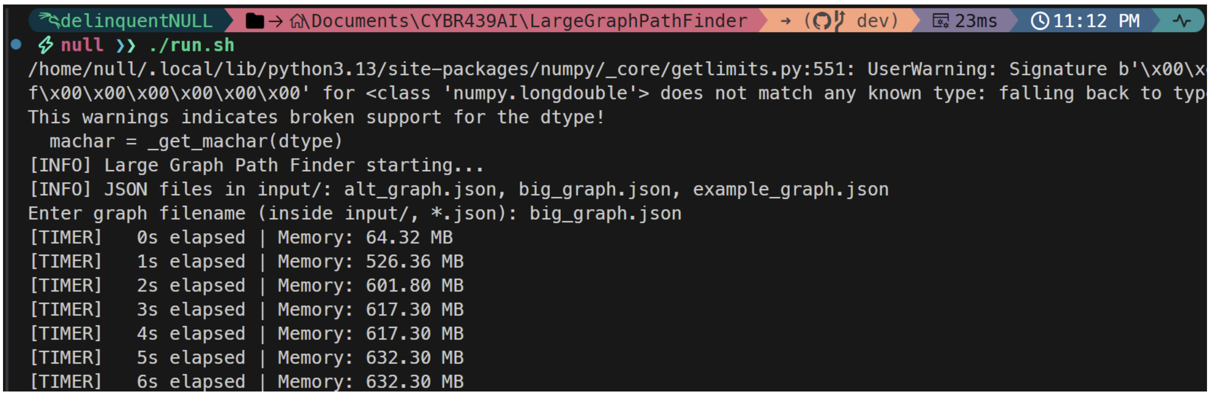
1. `<graph>.json` file required and insert it in `input/`
2. `query.txt` file optional and insert it in `input/`

**4. Run the Path Finder *main program*** Launch the main program *note: this program is designed to run on Linux/Linux VMs OS, other OS is not fully supported, certain commandline is gear thorse the OS mentioned see Prerequisites*:

```
python3 -m core.main
```

or use quick script *note: user must be under root directory*.

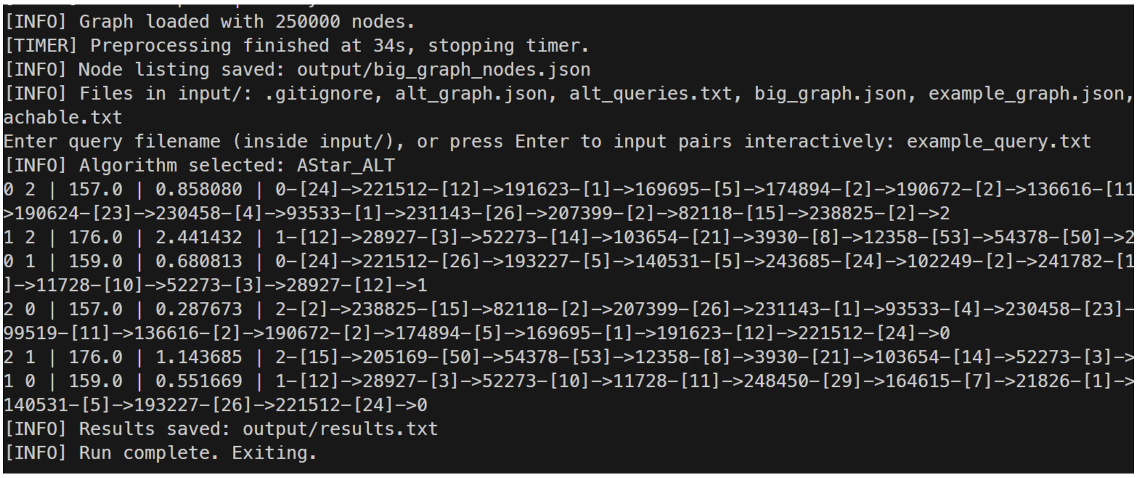
```
./run.sh
```



```
delinquentNULL  → Documents\CYBR439AI\LargeGraphPathFinder  → ( dev)  23ms  11:12 PM
null ›› ./run.sh
/home/null/.local/lib/python3.13/site-packages/numpy/_core/getlimits.py:551: UserWarning: Signature b'\x00\x
f\x00\x00\x00\x00\x00\x00' for <class 'numpy.longdouble'> does not match any known type: falling back to typ
This warnings indicates broken support for the dtype!
  machar = _get_machar(dtype)
[INFO] Large Graph Path Finder starting...
[INFO] JSON files in input/: alt_graph.json, big_graph.json, example_graph.json
Enter graph filename (inside input/, *.json): big_graph.json
[TIMER]   0s elapsed | Memory: 64.32 MB
[TIMER]   1s elapsed | Memory: 526.36 MB
[TIMER]   2s elapsed | Memory: 601.80 MB
[TIMER]   3s elapsed | Memory: 617.30 MB
[TIMER]   4s elapsed | Memory: 617.30 MB
[TIMER]   5s elapsed | Memory: 632.30 MB
[TIMER]   6s elapsed | Memory: 632.30 MB
```

You'll be prompted to select:

- A graph file (e.g. input/large_graph.json)
- A query file `after preprocessing is complete`(e.g. input/large_queries.txt)



```
[INFO] Graph loaded with 250000 nodes.
[TIMER] Preprocessing finished at 34s, stopping timer.
[INFO] Node listing saved: output/big_graph_nodes.json
[INFO] Files in input/: .gitignore, alt_graph.json, alt_queries.txt, big_graph.json, example_graph.json,
achable.txt
Enter query filename (inside input/), or press Enter to input pairs interactively: example_query.txt
[INFO] Algorithm selected: AStar_ALT
0 2 | 157.0 | 0.858080 | 0-[24]->221512-[12]->191623-[1]->169695-[5]->174894-[2]->190672-[2]->136616-[11
>190624-[23]->230458-[4]->93533-[1]->231143-[26]->207399-[2]->82118-[15]->238825-[2]->2
1 2 | 176.0 | 2.441432 | 1-[12]->28927-[3]->52273-[14]->103654-[21]->3930-[8]->12358-[53]->54378-[50]->2
0 1 | 159.0 | 0.680813 | 0-[24]->221512-[26]->193227-[5]->140531-[5]->243685-[24]->102249-[2]->241782-[1
]->11728-[10]->52273-[3]->28927-[12]->1
2 0 | 157.0 | 0.287673 | 2-[2]->238825-[15]->82118-[2]->207399-[26]->231143-[1]->93533-[4]->230458-[23]-
99519-[11]->136616-[2]->190672-[2]->174894-[5]->169695-[1]->191623-[12]->221512-[24]->0
2 1 | 176.0 | 1.143685 | 2-[15]->205169-[50]->54378-[53]->12358-[8]->3930-[21]->103654-[14]->52273-[3]->
1 0 | 159.0 | 0.551669 | 1-[12]->28927-[3]->52273-[10]->11728-[11]->248450-[29]->164615-[7]->21826-[1]->
140531-[5]->193227-[26]->221512-[24]->0
[INFO] Results saved: output/results.txt
[INFO] Run complete. Exiting.
```

**5. Real-Time Timer & Memory Monitor** During preprocessing:

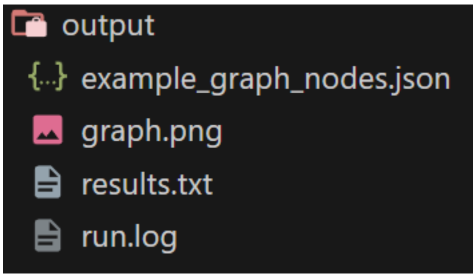
- timer counts up in seconds
- memory usage is displayed
- warnings appear if:
  - Runtime exceeds 60s
  - Memory usage exceeds 1024 MB

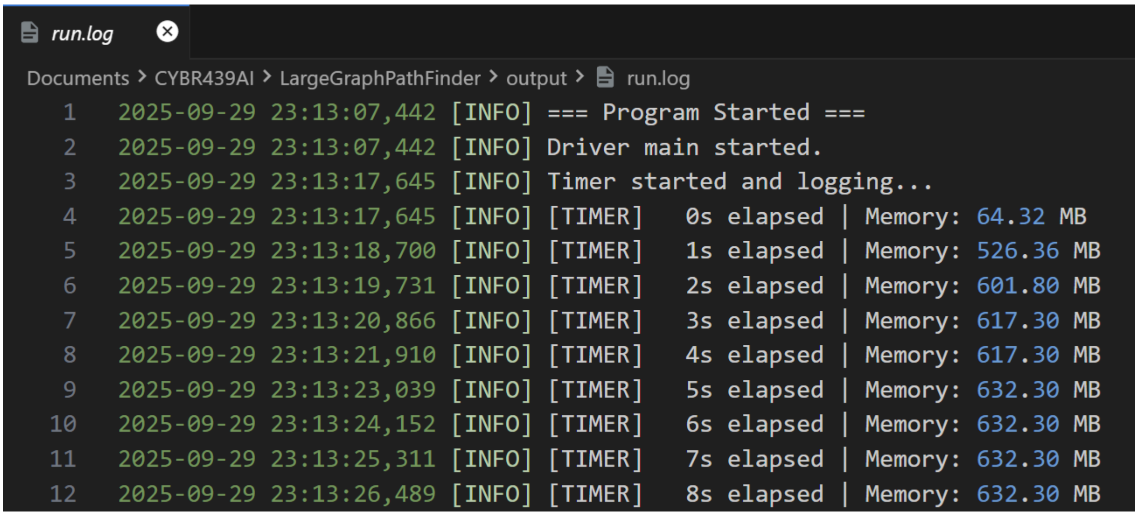
Example:

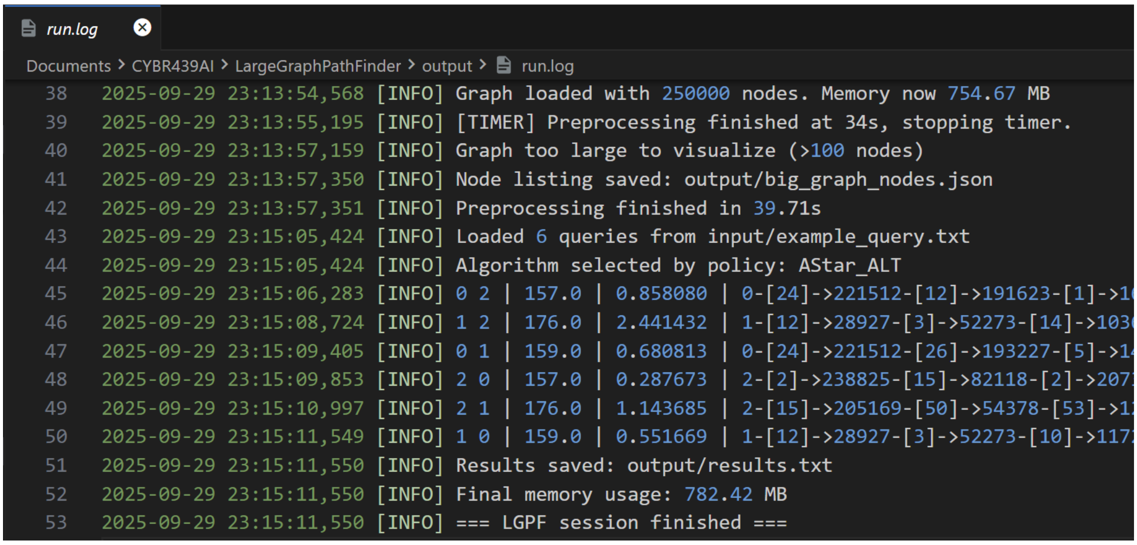
```
[TIMER]    0s elapsed | Memory: 64.31 MB
[TIMER]    1s elapsed | Memory: 531.61 MB
...
[TIMER] Preprocessing finished at 35s, stopping timer.
```

**6. Results** Output contains after a success or partial successful run:

- Query results are printed in terminal and saved to `output/results.txt`. Each line shows source, target, path cost, and runtime. If no path exists, cost will be inf.
- Query outputs are written to `output/results.txt`
- Program logs as ‘run.log
- A viasual graph if and only if node is no more than 1000 nodes.
- A json file showing all the proccessed nodes



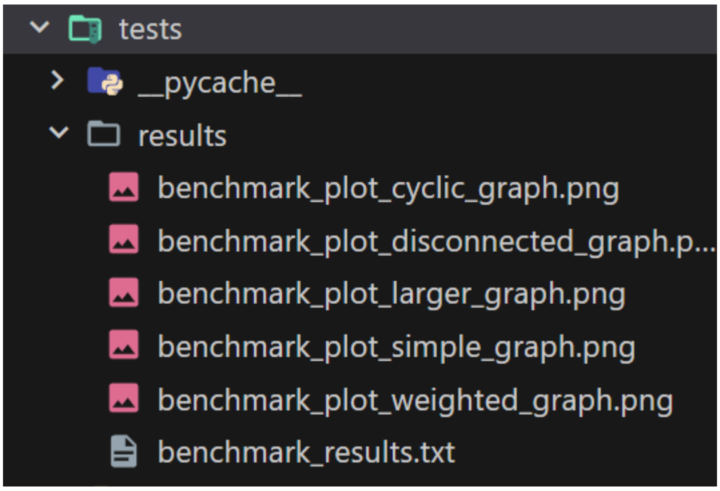



**7. Benchmarking (Optional)** *note: see Benchmarking for deeper details*

1. Run all unit tests and benchmarks it will create `tests/results/benchmark_result.txt` see Testing on how to run tests.
2. or Generate plots at root directory:

```
python3 tests/plot_benchmarks.py
```

3. Charts are saved in `tests/results/benchmark_plot_<name>_graph.png`.




---

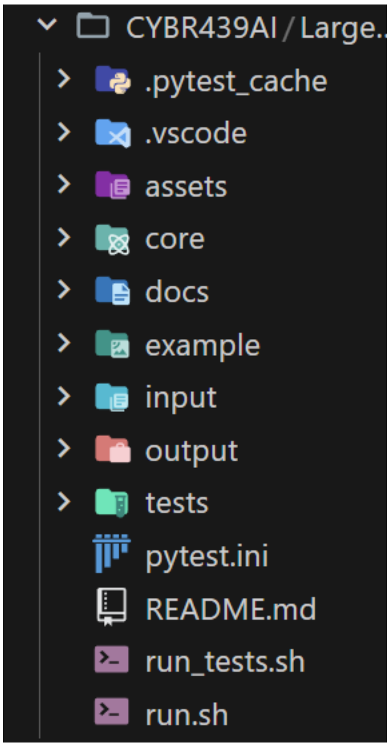
## Testing

Unit tests are provided under `tests/<test_test_file_name>.py`. Run with:

This project uses **pytest** for automated unit tests and benchmarks.

### Running All Tests

Run all unit tests and benchmarks with *note: must be under root folder*:




```
pytest -v
```

or use the helper script:

```
./run_tests.sh
```

This will execute all test cases under *tests/* and run benchmark comparisons across algorithms.

Save benchmark results into *tests/results/benchmark_results.txt*

Benchmark Plotting After benchmarks are generated, you can visualize results see Benchmarking.

---

## Benchmarking

Benchmarking evaluates how each algorithm performs across different graph types. This helps us compare runtime efficiency, path costs, and algorithm suitability.

### Running Benchmarks

Run the benchmarks directly with Python:

```
python3 tests/benchmark.py
```

Or use the convenience script:

```
./run_tests.sh
```

Both commands will execute all algorithms against predefined graphs (simple_graph, weighted_graph, cyclic_graph, etc.) and store results in: *output/benchmark_results.txt*

### Example Results

Below is a snippet of results from benchmark_results.txt:

```
weighted_graph      | GreedyZero           | Time: 0.000005s | Cost: None
cyclic_graph        | GreedyZero           | Time: 0.000004s | Cost: None
larger_graph        | GreedyZero           | Time: 0.000005s | Cost: None
simple_graph        | AStarZero            | Time: 0.000005s | Cost: 1.0
disconnected_graph  | AStarZero            | Time: 0.000005s | Cost: inf
weighted_graph      | AStarZero            | Time: 0.000005s | Cost: 4.0
cyclic_graph        | AStarZero            | Time: 0.000009s | Cost: 4.0
larger_graph        | AStarZero            | Time: 0.000006s | Cost: 4.0
simple_graph        | BidirectionalBFS     | Time: 0.000005s | Cost: None
disconnected_graph  | BidirectionalBFS     | Time: 0.000005s | Cost: None
weighted_graph      | BidirectionalBFS     | Time: 0.000004s | Cost: None
cyclic_graph        | BidirectionalBFS     | Time: 0.000003s | Cost: None
larger_graph        | BidirectionalBFS     | Time: 0.000004s | Cost: None
```

### Benchmark notations and meaning

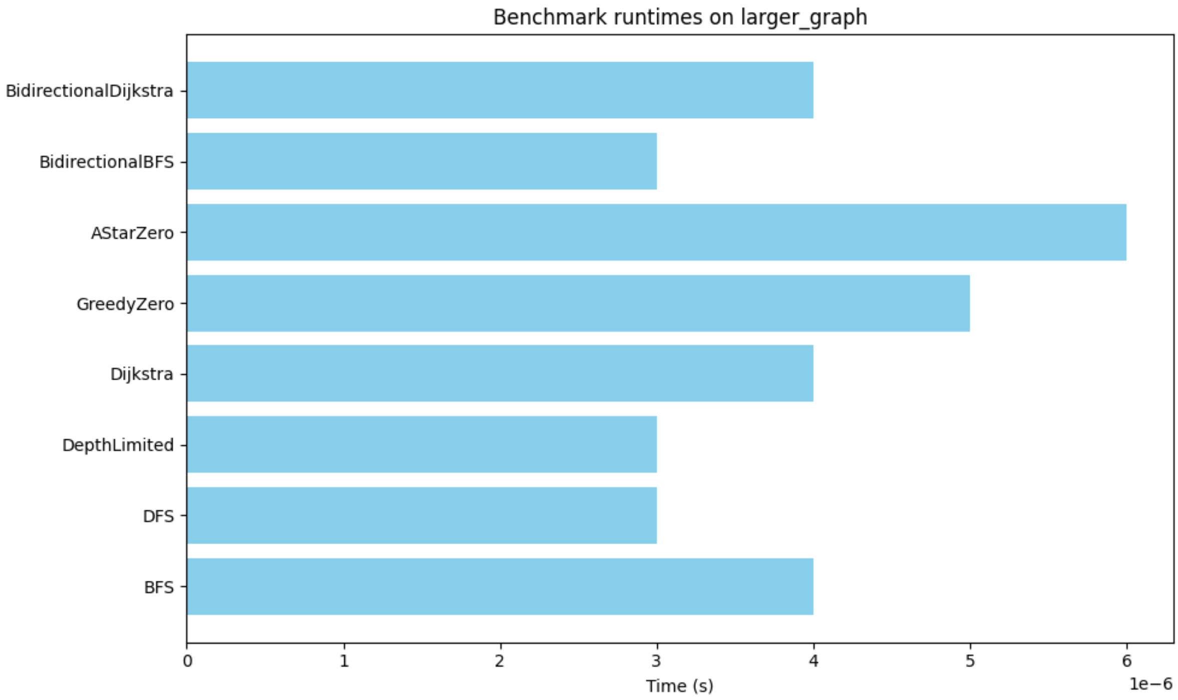
- Time → Runtime of the algorithm on that graph.
- Cost → Path cost from source to target:
- A number (e.g., 4.0) means a path was found.
- inf means no path exists between source and goal.
- None means the algorithm doesn't compute costs in that context (e.g., plain BFS for reachability).

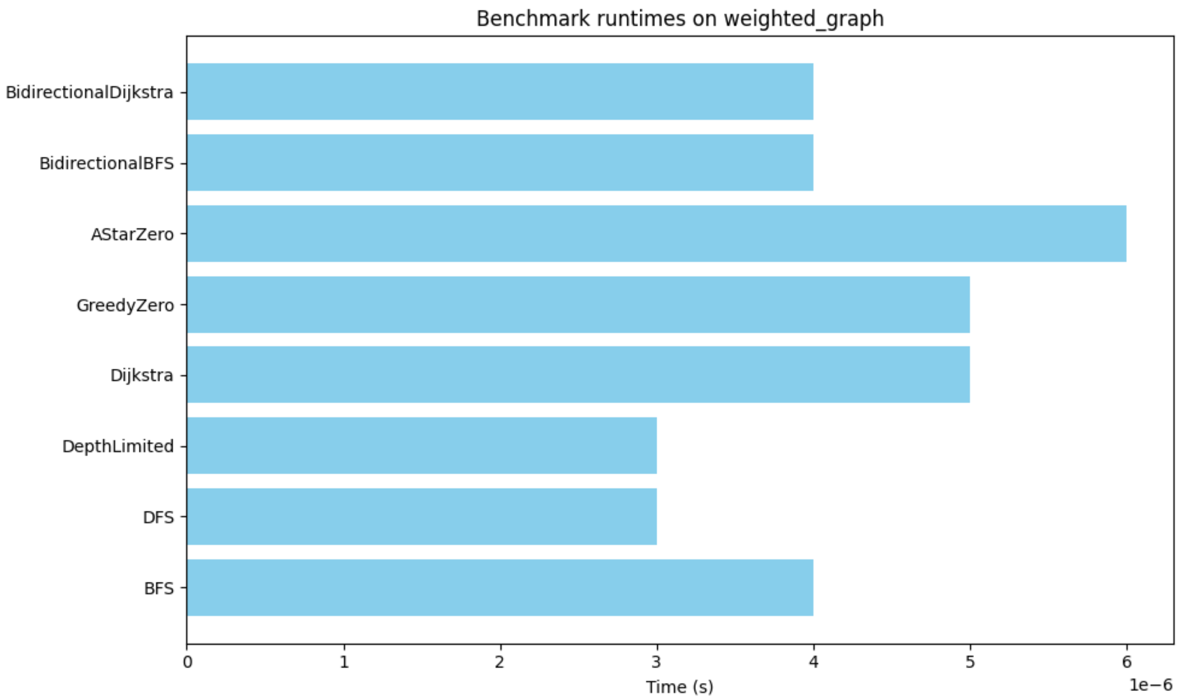
### Visualization

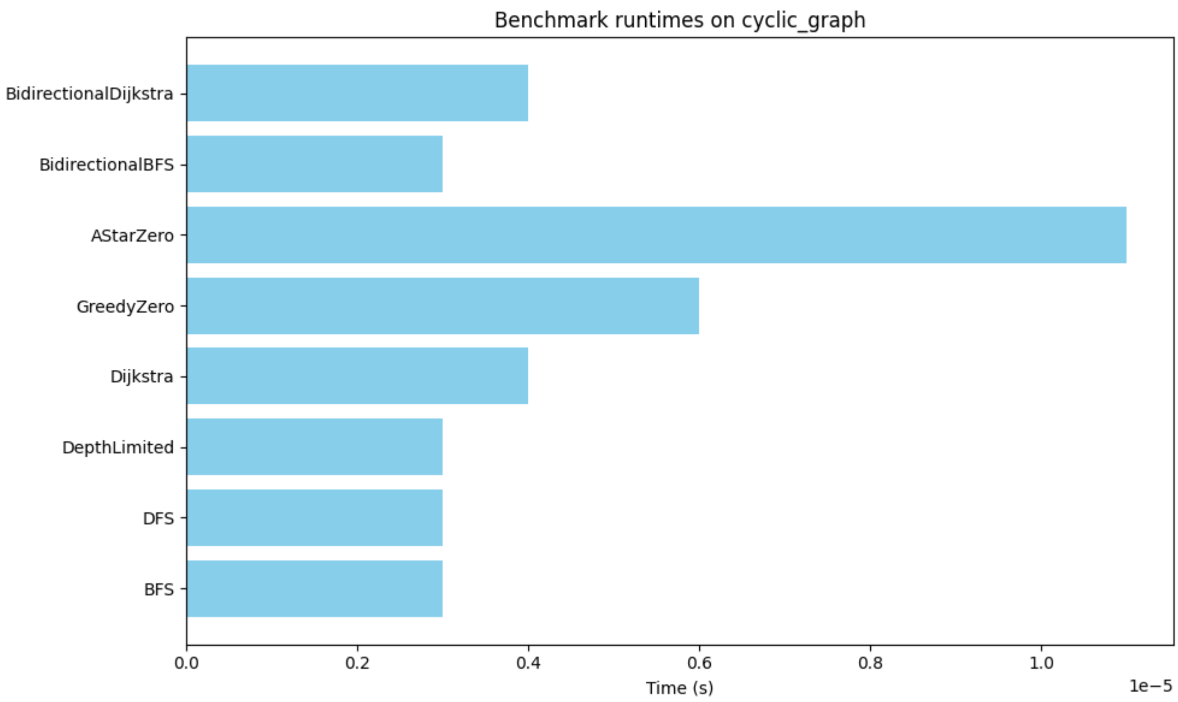
You can visualize algorithm performance (runtime vs. cost) using matplotlib:

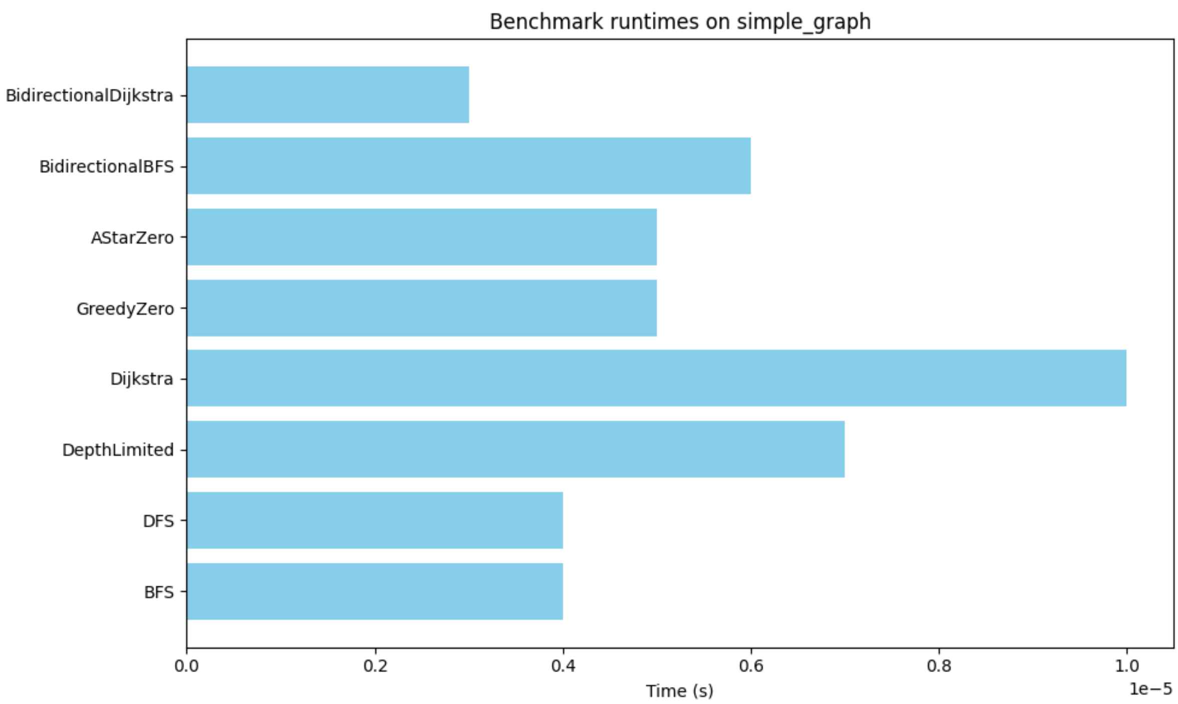
```
python3 tests/plot_benchmarks.py
```

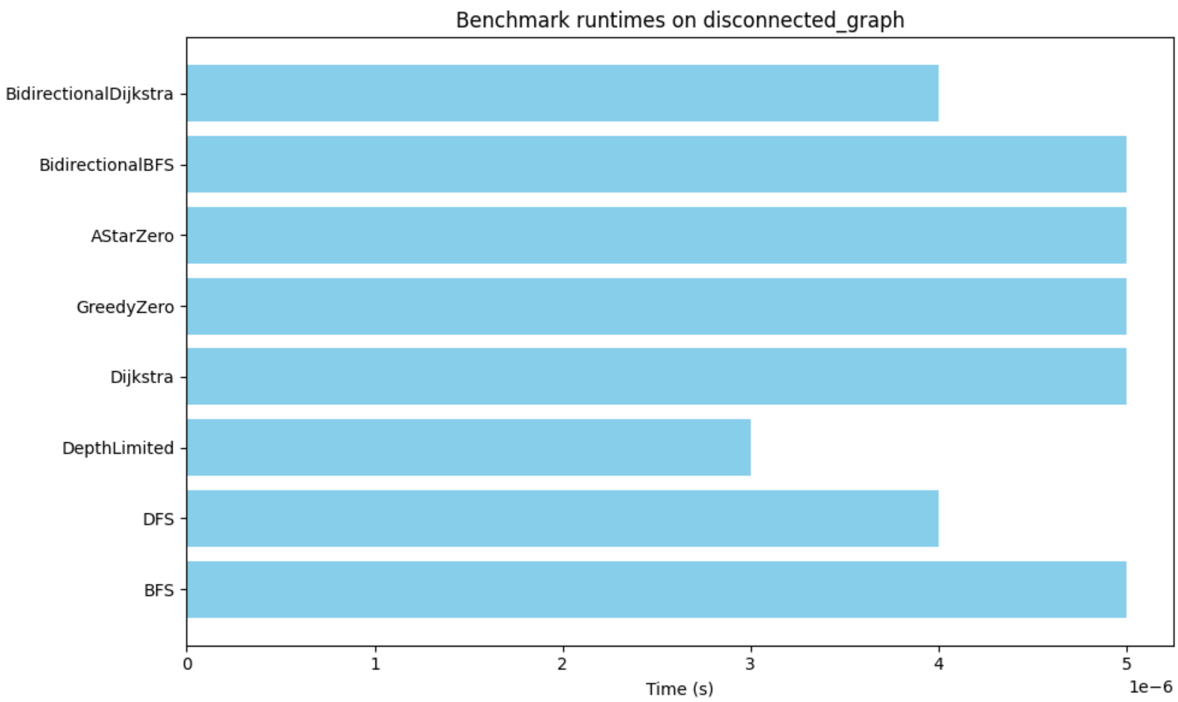
This produces a chart comparing algorithms across test graphs, saved under *tests/results/benchmark_plot.png*.


Benchmark runtimes on larger_graph
BidirectionalDijkstra
BidirectionalBFS
AStarZero
GreedyZero
Dijkstra
DepthLimited
DFS
BFS
0
1
2
3
4
5
6
Time (s)
1e−6


Benchmark runtimes on weighted_graph
BidirectionalDijkstra
BidirectionalBFS
AStarZero
GreedyZero
Dijkstra
DepthLimited
DFS
BFS
0
1
2
3
4
5
6
Time (s)
1e−6


Benchmark runtimes on cyclic_graph
BidirectionalDijkstra
BidirectionalBFS
AStarZero
GreedyZero
Dijkstra
DepthLimited
DFS
BFS
0.0
0.2
0.4
0.6
0.8
1.0
Time (s)
1e−5


Benchmark runtimes on simple_graph
BidirectionalDijkstra
BidirectionalBFS
AStarZero
GreedyZero
Dijkstra
DepthLimited
DFS
BFS
0.0
0.2
0.4
0.6
0.8
1.0
Time (s)
1e−5




---

## Documentation

- Development log: `docs/ai_chat_log.md` Contains incremental notes on implementation choices, code summaries, and next steps.

---

## AI Usage

- AI is used through out the development process. Mostly auto-complete feature with VS Code GitHub Copilot
- For code intergration and advice Open-AI ChatGPT 5 is used see project_root/docs/*ai_chat_log.md* for prompt and scripts.

  20250930_000502_ai_chat_log.md

## Project Structure

```
LargeGraphPathFinder/
├── .pytest_cache/    # Pytest cache
├── .vscode/          # VS Code configuration
├── assets/           # Used to store README photos
```

```
├── core/              # Main source code
├── docs/              # Documentation (ai_chat_log.md, etc.)
├── example/          # input and out generator is located here too
├── input/             # User-provided JSON graphs
├── output/            # Program-generated query files & logs
├── tests/             # Unit tests + benchmarks
├── pytest.ini         # Pytest configuration
├── README.md          # This file
├── README.pdf         # Pandoc converted PDF version of this file
├── run_test.sh        # Launcher script for all tests
├── run.sh             # Launcher script
└── pytest.ini         # Pytest configuration
```

---

**Future Work**

- Increase performance across the board
- Code refactoring and efficenecy
- Scale ALT preprocessing to millions of nodes with smarter landmark selection
- Parallel preprocessing (multi-core Dijkstra)
- Cluster-based heuristics for very large graphs
- Web-based visualization for graphs >10k nodes
- Incremental memory-usage profiling