Contents

AI Chat Log – Large Graph Path Finder (LGPF)

This log summarizes the development of the Large Graph Path Finder project. Each section documents major features added, the corresponding code files, and purpose.

---

1. Project Setup

- Created initial folder structure:

```
Proj_1/
├── core/ (main code)
├── input/ (user-provided graph/query files)
├── output/ (generated logs/results)
├── run.sh (launcher script)
└── README.md
```

- Added `.vscode/launch.json` for debugging in VS Code.

---

2. Timer & Memory Monitor

File: `core/resource.py`

- Added a **60-second countdown timer**.
- Tracks system memory usage each second using `psutil`.
- Runs in a background thread so preprocessing continues.

---

3. Graph Parser

File: `core/parser.py`

- Reads graphs generated by `graph-generator.py`.
- Converts JSON format into an **adjacency list**: `python {0: [(2, 93)], 1: [(2, 24)], 2: [(0, 93), (1, 24)]}` - Handles **undirected and directed graphs**.
- Raises custom `GraphParseError` on invalid files.

---

4. Main Program Driver

File: `core/main.py`

- **Prompts user** for a graph filename inside `input/`.
- Starts the **countdown + memory tracker** only after a valid file is chosen.
- Loads the graph with `parser.py`.
- Creates a companion query file `output/xxx_node.json` containing: `json { "graph_file": "example_graph.json", "num_nodes": 3, "nodes": [0, 1, 2] }` - Verbose output confirms steps (`[INFO]`, `[WARN]`, `[ERROR]`).

---

5. Run Script

File: `run.sh`

- Ensures the program always runs from project root: `bash python3 -m core.main`

---

6. VS Code Debugging

File: `.vscode/launch.json`

- Configured to run `core.main` as a Python module.
- Ensures `core/` imports (`core.parser`, `core.resource`) resolve correctly.

Next Planned Steps

- Extend `xxx_node.json` to include **query pairs** (manual or random).
- Implement **pathfinding algorithms** (`dijkstra`, later `bidirectional_dijkstra`).
- Process queries and log results in `output/results.log`.
- **Allow user to choose whether to print (`cat`) the log at the end.**

7. Algorithms Module

File: `core/algorithms.py`

- Implemented consistent `(path, cost)` return format across all algorithms.
- Added algorithms: - Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Depth-Limited DFS
- Dijkstra's Algorithm
- Greedy Best-First Search (with heuristics)
- A* Search (with heuristics)
- Bidirectional BFS
- Bidirectional Dijkstra (currently falls back to normal Dijkstra for directed graphs).

---

8. Heuristics Module

File: `core/heuristics.py`

- Split heuristics from algorithms for modular design.

- Implemented: - Zero heuristic
- Manhattan distance
- Euclidean distance
- Chebyshev distance
- Octile distance
- Haversine (for lat/lon geographic graphs)
- File structured for easy extension with new heuristics.

---

9. Testing

Files: `tests/test_algorithms.py`, `tests/conftest.py`

- Created pytest-based unit tests.
- Graph fixtures: - Simple graph
- Disconnected graph
- Weighted graph
- Cyclic graph
- Larger graph
- Parameterized tests run all algorithms against all graph types.
- Extended to test A* and Greedy under multiple heuristics (zero, Manhattan, Euclidean).

---

10. Benchmarking

File: `tests/benchmark_test.py`

- Benchmarks all algorithms on all sample graphs.
- Logs results to `tests/results/benchmark_results.txt`.
- Provides execution time and path cost for comparison.

---

11. Documentation

File: `README.md`

- Added structured README with abstract, installation, prerequisites, environment info, usage, testing, benchmarking, documentation pointer, and project structure.

- Includes Table of Contents for Pandoc PDF conversion.
- Points to detailed development log (`docs/ai_chat_log.md`).

Next Planned Steps

- Add more algorithms for completeness and large-graph scenarios:
  - Iterative Deepening DFS (IDDFS)
  - Bellman-Ford (handles negative weights)
  - Floyd–Warshall or Johnson's (all-pairs shortest path)
  - Bidirectional A* (fast for sparse graphs)
- Add support for landmark-based heuristics.
- Expand benchmarking to measure memory usage.

Submission Notes

- **Completeness:** input → preprocessing → query file created.
- **Correctness:** robust parsing + memory/timer enforcement.
- **Readability:** clear code structure, logging, and error handling.
- **Files included:**
  - `core/main.py`
  - `core/parser.py`
  - `core/resource.py`

- `run.sh`
- `README.md`
- `.vscode/launch.json`
- sample `input/example_graph.json`

12. Preprocessing & Query Policy

File: core/pathfinder.py

preprocess_index(adj) builds:

weight_map[u][v] for fast per-edge weight lookups (pretty printing + totals).

Weakly-connected components (reachability pruning).

Basic stats: node_count, edge_count, uniform_weights.

choose_algorithm(adj, index):

Small & unweighted → BFS (fast, hop-optimal).

Weighted → Dijkstra (optimal).

Very large → A* with zero heuristic (safe default).

find_path checks reachability cheaply; then runs the chosen algorithm.

13. Result Formatting (new)

File: core/graphing.py

Converts a path like [1,2,3,4] into:

1-[36]->2-[1]->3-[45]->4

and computes total weight (e.g., 82.0).

format_result_line emits:

src dst | total_weight | time_sec | path_with_edge_labels

write_results writes a header + all result lines to output/results.txt.

14. Interactive Queries (new UX)

If no query file is provided, the program allows interactive pair entry:

src dst (empty to finish):

15. Current Status

Program runs end-to-end:

Prompts for JSON → preprocess (≤60s target) → queries (file or interactive) → results/logs/artifacts.

Output artifacts:

output/results.txt – per-query results (weight, time, path).

output/run.log – detailed log (timer, memory, choices, warnings).

output/graph.png – small-graph preview.

output/_nodes.json – node reference.

16. Next Big Feature: Landmark Heuristics (ALT)

Planned files: core/landmarks.py, updates to core/pathfinder.py

ALT (A, Landmarks, Triangle inequality)* for large graphs:

Pick K landmarks (e.g., 8–16) quickly (degree/farthest selection).

Precompute distances: d(L, v) and d(v, L) (via Dijkstra and reversed graph).

Query uses ALT heuristic:

h(s,t) = max_L { |d(L,t) - d(L,s)|, |d(s,L) - d(t,L)| }

Heuristic is admissible (A* remains optimal when desired).

Major speedups on dense/huge graphs while staying within the 60s preprocessing budget (tune K).

Policy (pathfinder):

If node_count >= 10k and ALT tables exist → use A* + ALT.

Else keep current BFS/Dijkstra/A*0 strategy.

Future: add clustering (e.g., “Tokyo center”) for coarse routing + local refinement.

17. How to Run

./run.sh # then follow prompts: # 1) Enter graph JSON in input/ (e.g., example_graph.json) # 2) Enter query file in input/ (e.g., example_query.txt) or input pairs interactively

Query file format (input/example_query.txt):

0 2 1 2 0 1 2 0 2 1 1 0

18. Roadmap to MVP

Implement core/landmarks.py (selection + tables + ALT heuristic).

Integrate ALT in pathfinder.preprocess_index and choose_algorithm.

Add a large-graph test/benchmark to validate speedups.

Document ALT mode in README (memory/time trade-offs, K tuning).