Practical Work 5: The Longest Path

Doan Dinh Khai - 22BA13167

December 24, 2025

MapReduce Implementation Choice

For this practical work, we implemented the Longest Path problem using a Python-based MapReduce framework, following the same approach as previous MapReduce implementations. This allows us to process file paths from multiple laptops and find the longest path(s) across all input files.

Operating System and Setup: The implementation was developed on Windows 10 using Python 3.x. The system processes multiple input files, where each file contains file paths from a different laptop (one path per line). This design allows parallel processing of paths from different sources.

Input Format: Each input file represents paths from one laptop. Paths can be in different formats:

- Unix-style paths: `/home/user/documents/file.txt`
- Windows-style paths: `C:\Users\Documents\file.txt`
- Mixed formats across different input files

Paths are generated using commands like `find /` on Unix systems or directory traversal on Windows systems, with each full path written on a separate line.

Output Format: The output contains the longest path(s) found across all input files. If multiple paths have the same maximum length, all of them are included in the output. Each line contains: `length\tpath`.

Advantages:

- Simple and lightweight implementation using Python
- Handles multiple input files from different laptops
- Compatible with Hadoop Streaming for distributed execution
- Efficient processing using MapReduce pattern
- Handles both Unix and Windows path formats

Limitations:

- Single-machine execution in standalone mode
- Path length is measured in characters, not filesystem depth
- No validation of path existence or accessibility

Mapper and Reducer Design

Mapper Logic: The mapper (`mapper.py`) processes each input file independently:

1. Reads each line from standard input (one path per line)
2. Strips whitespace and validates non-empty lines
3. Calculates the length of each path (number of characters)
4. Emits key-value pairs in the format `(length, path)`

The mapper outputs pairs where:

- Key: Path length (integer)
- Value: The full path string

This design allows the MapReduce framework to sort all paths by length during the shuffle phase, making it easy for the reducer to identify the longest paths.

Intermediate Key-Value Pairs: The mapper emits pairs in the format `length\tpath` (tab-separated), for example:

```
45      /home/user/documents/file1.txt
67      /opt/very/long/path/to/some/deeply/nested/directory/file.ext
```

Shuffle and Sort Phase: The MapReduce framework automatically sorts all key-value pairs by key (path length) in descending order. This ensures that the reducer receives paths sorted by length, with the longest paths appearing first.

Reducer Logic: The reducer (`reducer.py`) identifies the longest path(s):

1. Reads sorted key-value pairs from standard input (sorted by length, descending)
2. Maintains state for the maximum length encountered
3. Collects all paths that match the maximum length
4. Since input is sorted, the first path(s) encountered will have the maximum length
5. Outputs all paths with the maximum length

The reducer algorithm:

- Initializes `max_length` to -1 and `longest_paths` as an empty list
- For each input pair `(length, path)`:
  - If `length > max_length`: Update max length and reset the list with current path
  - If `length == max_length`: Add current path to the list
  - If `length < max_length`: Skip (since input is sorted, no longer paths will follow)
- Output all paths in `longest_paths` with their length

Data Flow Diagram: The MapReduce longest path process follows this flow:

1. **Input**: Multiple files, each containing paths from one laptop
   - File 1: `laptop1_paths.txt` (paths from laptop 1)
   - File 2: `laptop2_paths.txt` (paths from laptop 2)

- File N: Additional laptop path files

2. **Map Phase**: Each mapper processes one input file
   - Mapper 1 reads laptop1 paths, emits `(length, path)` pairs
   - Mapper 2 reads laptop2 paths, emits `(length, path)` pairs
   - Each mapper runs independently and in parallel
3. **Shuffle/Sort**: Framework collects all pairs, sorts by length (descending)
4. **Reduce Phase**: Single reducer receives sorted pairs
   - Processes pairs in descending length order
   - Identifies maximum length
   - Collects all paths with maximum length
5. **Output**: All longest path(s) with their length

Figure: MapReduce Data Flow for Longest Path
The following diagram illustrates the data flow:

```
Input Files:
  laptop1_paths.txt          laptop2_paths.txt
  /home/user/file1.txt       C:\Users\file.txt
  /opt/long/path/file.ext     C:\Very\Long\Path\file.ext

        |                            |
        v                            v
    Mapper 1                     Mapper 2
        |                            |
        | Emit (length, path)        | Emit (length, path)
        v                            v
        45  /home/user/file1.txt    25  C:\Users\file.txt
        67  /opt/long/path/file.ext 67  C:\Very\Long\Path\file.ext

        |                            |
        +-----------+----------------+
                    |
                    v
              Shuffle & Sort
        (by length, descending)
                    |
                    v
        67  /opt/long/path/file.ext
        67  C:\Very\Long\Path\file.ext
        45  /home/user/file1.txt
        25  C:\Users\file.txt

                    |
                    v
                Reducer
        (finds max length = 67)
                    |
```

```
                v
    Output (longest paths):
67  /opt/long/path/file.ext
67  C:\Very\Long\Path\file.ext
```

Implementation Details

Programming Language and Libraries: The implementation uses Python 3.x with standard library modules:

- `sys`: For reading from stdin and writing to stdout
- `subprocess`: For orchestrating mapper and reducer processes
- `os`: For file system operations and path handling
- `tempfile`: For temporary file management when combining multiple mapper outputs

Running the Job: The system can be executed in multiple ways:
Using the orchestrator script (recommended):

```
python longest_path.py laptop1_paths.txt laptop2_paths.txt output.txt
```

This script handles multiple input files automatically, runs mappers in parallel, combines their outputs, sorts the results, and feeds them to the reducer.

Manual execution (single input file):

```
cat laptop1_paths.txt | python mapper.py | sort -rn | python reducer.py
   > output.txt
```

The `sort -rn` command sorts numerically in reverse order (descending), simulating the shuffle/sort phase.

Manual execution (multiple input files):

```
cat laptop1_paths.txt laptop2_paths.txt | python mapper.py | sort -rn |
   python reducer.py > output.txt
```

Code Structure:

Listing 1: Mapper implementation

```
import sys

for line in sys.stdin:
    path = line.strip()
    if not path:
        continue

    path_length = len(path)
    print(f"{path_length}\t{path}")
```

Listing 2: Reducer implementation

```
import sys

max_length = -1
longest_paths = []

for line in sys.stdin:
```

```
    line = line.strip()
    if not line:
        continue

    parts = line.split('\t', 1)
    if len(parts) != 2:
        continue

    try:
        path_length = int(parts[0])
        path = parts[1]
    except ValueError:
        continue

    if path_length > max_length:
        max_length = path_length
        longest_paths = [path]
    elif path_length == max_length:
        longest_paths.append(path)

for path in longest_paths:
    print(f"{max_length}\t{path}")
```

Key Implementation Features:

- **Multiple input file support**: The orchestrator script can process multiple laptop path files simultaneously
- **Parallel mapper execution**: Each input file is processed by a separate mapper process
- **Efficient reducer**: Takes advantage of sorted input to find maximum length in a single pass
- **Multiple longest paths**: Correctly handles cases where multiple paths have the same maximum length
- **Error handling**: Validates input format and handles malformed lines gracefully

Optimizations:

- **Sorted input**: Reducer assumes sorted input, allowing early termination optimization (though not implemented, could skip after finding max)
- **Streaming processing**: Both mapper and reducer process data line by line, minimizing memory usage
- **Parallel mappers**: Multiple input files are processed concurrently

For production use with very large datasets, additional optimizations could include:

- **Distributed execution**: Use Hadoop or Spark for true distributed processing
- **Combiner**: Local reduction in mapper phase to reduce network traffic
- **Custom partitioner**: For better load balancing if using multiple reducers
- **Path validation**: Check if paths exist and are accessible before processing

Group Work

Choice and Implementation of MapReduce Framework: The team chose to use the same Python-based MapReduce pattern as in Practice 4 for consistency and simplicity. The implementation extends the previous word count example to handle the longest path problem, demonstrating the versatility of the MapReduce programming model.

Implementation of Mapper and Reducer: The mapper was designed to extract path length as the key, enabling efficient sorting and reduction. The reducer implements a single-pass algorithm that identifies the maximum length and collects all paths matching that length. The implementation correctly handles edge cases such as empty files, duplicate paths, and multiple paths with the same maximum length.

Experiments and Testing: Testing was performed using sample path files:

- Created test files with Unix-style paths (`laptop1_paths.txt`)
- Created test files with Windows-style paths (`laptop2_paths.txt`)
- Tested with paths of varying lengths
- Verified correct identification of longest paths
- Tested cases with multiple paths having the same maximum length
- Validated output format and correctness

The implementation correctly handles:

- Multiple input files from different laptops
- Paths in different formats (Unix and Windows)
- Empty lines and malformed input
- Multiple paths with the same maximum length
- Very long paths (tested with paths exceeding 200 characters)

Performance testing showed that the implementation processes files efficiently, with linear time complexity relative to the number of input paths. The parallel mapper execution provides good performance when processing multiple input files.

Writing and Formatting of the LaTeX Report: The report was written in LaTeX following the same structure and formatting style as previous practical works. Code snippets were formatted using the `listings` package for syntax highlighting. The report includes a detailed data flow diagram in ASCII art format, explaining how paths from multiple laptops are processed through the MapReduce pipeline to identify the longest path(s). The document covers implementation choice, mapper and reducer design with figures, implementation details, and group responsibilities.