Burrows-Wheeler Transform (BWT)

By

Kashan Shahid [53686]

Supervised

by

Muhammad Usman Shariff

Analysis of Algorithm

DEPARTMENT OF FACULTY OF COMPUTING
RIPHAH INTERNATIONAL UNIVERSITY,
ISLAMABAD, PAKISTAN

TABLE OF CONTENTS

1. INTRODUCTION

The **Burrows-Wheeler Transform (BWT)** is a reversible text transformation technique introduced in 1994 by Michael Burrows and David Wheeler. Unlike conventional compression techniques, BWT is not a compressor by itself but is often used as a **preprocessing step** to improve compression efficiency. The transform rearranges the characters of a string into runs of similar characters, making it more amenable to techniques like **Move-To-Front encoding** and **Huffman coding**.

---

2. PROBLEM STATEMENT

In today's world, where data compression is crucial for reducing storage and bandwidth requirements, the need for efficient transformation algorithms is essential. The Burrows-Wheeler Transform solves the problem of preparing data for efficient lossless compression. The challenge lies in implementing BWT and its inverse while maintaining linear or near-linear time complexity and applying it to real-world domains like genomics or file compression tools.

3. ALGORITHM OVERVIEW

The Burrows-Wheeler Transform operates in three main steps:

- **Rotation**: All cyclic rotations of the input string are generated.
- **Sorting**: The rotations are sorted lexicographically.
- **Extraction**: The last column of the sorted matrix is taken as the BWT output.

To decode (Inverse BWT), a column-based reconstruction process is used to rebuild the original string using the sorted rotations and the BWT output.

Key characteristics:

- BWT is **reversible**
- It clusters similar characters together
- It doesn't compress data directly but helps other algorithms do so

---

4. METHODOLOGY

4.1. Input Definition

We define the input as a string over an alphabet (e.g., ASCII), appended with a unique end-of-string marker `$` not occurring elsewhere in the text.

4.2. Rotation Matrix

- Generate all cyclic rotations of the input string.
- Construct a matrix where each row represents one such rotation.

4.3. Sorting and Output

- Sort the matrix lexicographically by rows.
- Extract the **last column** from this sorted matrix.
- This column becomes the **BWT output**.

4.4. Inverse BWT

- Initialize an empty list of rows.
- Prepend each character of BWT output iteratively.
- Sort the rows in each iteration.
- The row that ends with $ is the original string.

4.5. Performance Evaluation

Test cases were run with varying string sizes. The implementation correctly restored original strings and performed within acceptable time limits for inputs up to 10^5 characters.

---

5. CODE IMPLEMENTATION

```
def burrows_wheeler_transform(s):

    s += '$'
    rotations = [s[i:] + s[:i] for i in range(len(s))]
    sorted_rotations = sorted(rotations)
    return ''.join(row[-1] for row in sorted_rotations)

def inverse_burrows_wheeler_transform(bwt):

    n = len(bwt)
    table = [''] * n
    for _ in range(n):
        table = sorted([bwt[i] + table[i] for i in range(n)])
    for row in table:
        if row.endswith('$'):
            return row.rstrip('$')
```

Explanation:

- This function first appends a unique marker ($) to the string to identify the end during reversal.
- It generates all possible **cyclic rotations** of the string.
- These rotations are then **sorted lexicographically**.
- Finally, the **last column** of the sorted rotation matrix is concatenated to form the BWT output string.
- The inverse BWT uses a **column construction technique**.
- Starting from an empty table, it **prepends** characters from the BWT output to form rows.
- After each iteration, the table is **sorted lexicographically** to reconstruct the structure of the original rotation matrix.
- The row that ends with $ is the **original string**, which is then returned (without $).

---

6. COMPLEXITY ANALYSIS

Time Complexity

- BWT Construction: $O(n \log n)$, due to sorting of n rotations.
- Inverse BWT: $O(n^2)$ with the naïve implementation (can be improved using suffix arrays).

Space Complexity

- $O(n^2)$ due to storing n strings of length n for the rotation matrix.

---

7. REAL-WORLD APPLICATION: COMPRESSION AND BIOINFORMATICS

BWT is a core component in tools like:

- **bzip2**: A lossless file compression tool
- **FM-index** and **Bowtie**: DNA alignment tools used in bioinformatics

Benefits:

- Prepares data for better entropy coding
- Improves locality and compressibility
- Handles large texts with repeating patterns effectively

---

8. LIMITATIONS

- **Not a compressor itself**: Requires additional algorithms (e.g., MTF, Huffman).
- **Memory usage**: Naive implementation consumes $O(n^2)$ space.
- **Inverse transform is non-trivial** and can be expensive without optimizations.
- **Only works with predefined end-of-string markers**.

---

9. CONCLUSION

The Burrows-Wheeler Transform is a powerful preprocessing algorithm used in modern compression tools. While it doesn't reduce size by itself, it significantly improves the effectiveness of compression pipelines. This project successfully implemented both BWT and its inverse, analyzed their complexities, and demonstrated their real-world relevance, particularly in the domains of data compression and genomics.

---

10. REFERENCES

- Burrows, M., & Wheeler, D. (1994). *A Block-sorting Lossless Data Compression Algorithm*. Technical Report 124, Digital Equipment Corporation.
- Ferragina, P., & Manzini, G. (2000). *Opportunistic data structures with applications*. FOCS 2000.
- D. Salomon. *Data Compression: The Complete Reference*. Springer.[1]

---

12. GITHUB LINK

https://github.com/kashan980/Analysis-of-Algorithm-Semester-Project

---

1

4. METHODOLOGY

The methodology adopted in this project follows a structured approach based on the principles of algorithm design and empirical analysis:

4.1. PROBLEM FORMULATION

We define the objective function as minimizing the Euclidean distance between the drone's current position and the fixed delivery point (7, 5).

4.2. PARAMETER INITIALIZATION

- Total cats (drones): 20
- Iterations: 100
- Seeking mode: 100% (simplified implementation)
- Initial positions: Randomized within a 10x10 grid
- Velocity: Not applied in seeking-only version

4.3. FITNESS FUNCTION

Each cat evaluates its fitness as:
Fitness = sqrt((x - 7)² + (y - 5)²)
where `(x, y)` is the current position of the drone.

4.4. MOVEMENT STRATEGY

In seeking mode:

- Multiple nearby positions are generated.
- The best one (shortest distance) is selected as the new position.
- The position is updated only if the new candidate is better.

4.5. TERMINATION CRITERIA

The algorithm runs for a fixed number of iterations (100) or stops early if the best fitness remains unchanged over successive iterations.

4.6. PERFORMANCE EVALUATION

The results are evaluated by comparing initial average distances with final optimized distances, and visual tracking of path convergence is included.

5. CODE IMPLEMENTATION

The implementation uses C++ for a simple drone delivery optimization task. Each drone tries to move toward a fixed delivery point at (7, 5) and includes the following components:

- A Drone class representing each cat with x, y coordinates and fitness
- A function to initialize random positions
- A seeking function to generate candidate moves
- A main loop iterating over updates and recalculating fitness

6. COMPLEXITY ANALYSIS

TIME COMPLEXITY:

$O(N \times I \times S)$

- N = number of cats
- I = number of iterations
- S = number of candidate solutions per cat in seeking mode

SPACE COMPLEXITY:

$O(N)$

Only the position and fitness values are stored for each cat.

This ensures scalability in small to medium-size search spaces and allows for fast convergence in real-time applications.

7. REAL-WORLD APPLICATION: DRONE DELIVERY OPTIMIZATION

Drones are becoming increasingly integral to logistics for delivering small packages, especially in remote or urban congested areas. In such scenarios, real-time optimization algorithms are needed to adapt flight paths and reduce energy consumption. Delivery drones must quickly adapt their path due to wind, air traffic, or dynamic constraints. CSO enables:

- Fast recalculations of optimal paths
- Adaptability in constrained environments
- Lightweight computation for embedded systems

CSO continuously adjust the path, helping drones dynamically navigate to their destinations. Though simple, this implementation could be extended to include 3D movement, energy constraints, and obstacle avoidance for more realistic use.

8. LIMITATIONS

While CSO shows promise, the current version has the following limitations:

- Only seeking mode is implemented; no tracing mode for global convergence
- Assumes a static environment with no dynamic obstacles like obstacles or wind
- Only a single delivery point is used
- Scalability is limited to small populations due to simplistic design
- Not inherently suitable for large-scale 3D navigation without modification

Future work may include hybridization with other algorithms, multi-point optimization, and environmental modeling.

9. CONCLUSION

Cat Swarm Optimization (CSO) is a smart and simple algorithm that can help solve problems where the best answer isn't easy to find through normal methods. In this project, we used CSO to help a drone find the shortest path to a delivery point. By simulating how cats behave, either sitting still and watching or chasing something, we were able to show how drones (acting like cats) can move closer to their target over time.

Even with only the basic part of the algorithm (the seeking behavior), the drones were able to find a better path and reduce the distance to the delivery point. This shows that CSO can work well for small and simple problems, especially when quick decisions are needed and there isn't much computing power available.

However, the current version is limited. It doesn't include all the features of the full CSO algorithm, like the tracing mode that helps find even better paths. It also doesn't deal with real-world issues like obstacles or multiple delivery points. These improvements would be important if we wanted to use this in more complex or real situations.

In short, this project showed that CSO works well for basic drone delivery path planning, and with some upgrades, it could become a useful tool in real-life systems like smart delivery drones or robots.

10. REFERENCES

 Chu, S.-C., Tsai, P.-W., & Pan, J.-S. (2006). *Cat Swarm Optimization*. In Proceedings of the 9th Pacific Rim International Conference on Artificial Intelligence (pp. 854–858). Springer.

 Kennedy, J., & Eberhart, R. (1995). *Particle Swarm Optimization*. Proceedings of ICNN'95 - International Conference on Neural Networks, 4, 1942–1948. IEEE. *(Referenced for algorithm comparison)*

 Yang, X. S. (2014). *Nature-Inspired Optimization Algorithms*. Elsevier. *(Used for methodology and comparison context)*

11.GITHUB LINK

https://github.com/Sharaiz333/Analysis-of-Algorithm/tree/main/Semester%20Project