To improve the efficiency of your comparison of publication titles using Levenshtein distance, here are some optimization strategies you could try:

1. **Tokenization and Preprocessing:**

- **Lowercasing and removing punctuation**: Preprocess the titles to make them uniform before applying Levenshtein distance. By converting to lowercase and removing special characters, you'll reduce unnecessary variations.
- **Removing stopwords**: Eliminate common stopwords like "the," "and," etc., as they do not change the essence of the title but could impact the distance calculation.
- **Sort words within titles**: Sorting words within a title alphabetically can help minimize discrepancies caused by word order.

2. **Leverage Approximate String Matching (Fuzzy Matching):**

- Use a more efficient fuzzy matching library like **FuzzyWuzzy** (Python) or **Fuse.js** (JavaScript). These libraries offer optimized methods for string similarity and might perform better than a direct implementation of Levenshtein distance.
- **FuzzyWuzzy** uses ratio scoring methods and pre-processing, which can speed up similarity matching in some cases.

3. **Reduce the Search Space:**

- **Clustering/Blocking**: Instead of comparing every title with every other title, group titles by some heuristic criteria (such as length or hashing). You can compare titles within clusters, dramatically reducing the number of pairwise comparisons.
    - For example, titles that are very short might be unlikely to match long titles, so you could block them by approximate length first.
- **N-grams or Jaccard Index**: You can create n-grams (bigrams or trigrams) from the titles and compare these before applying Levenshtein distance. If n-grams don’t match sufficiently, skip the Levenshtein calculation.

4. **Levenshtein Distance with Threshold Early Stopping:**

- Set a threshold for what counts as a "match" (e.g., if the distance is above a certain level, the strings are too dissimilar). Implement early stopping during the Levenshtein

computation once it becomes clear the threshold will not be met, thus saving computation time.

5. **Parallelization and Optimization:**

- **Parallel Processing**: Leverage multithreading or multicore processing to parallelize the comparisons across multiple CPU cores. This will split the workload and speed up the overall process.
- **Vectorized Libraries**: Depending on your tech stack, consider using optimized libraries that can handle vectorized operations, reducing overhead.

6. **Approximate Nearest Neighbor Search (ANN)**:

- Use approximate nearest neighbor search algorithms such as **LSH (Locality-Sensitive Hashing)**. This method hashes similar strings into the same bucket with high probability, allowing for efficient similarity search.
- **FAISS** (Facebook AI Similarity Search) or **Annoy** can also be adapted for textual data, especially after converting strings to embeddings.

7. **Sorting and Early Termination in Comparison:**

- If your arrays are sorted, and you have some prior knowledge about likely matches, you can break out of the loop early when comparing titles that are clearly not similar based on their sorted order (e.g., lexicographically).

By combining these methods, especially tokenization/preprocessing and blocking, you'll likely see a substantial reduction in time complexity while maintaining accuracy in identifying duplicate titles.