JSONdiff

Wenbo Li

December 2023

1 Introduction

This package is designed for analyzing differences between JSON files. I have utilized 'RapidJSON' as the parser for these files and developed my own differ algorithm. I referred to an online tool, https://www.jsondiff.com/, and made my own improvements by implying the Longest Common Subsequence algorithm for array comparison. In the following introduction, I will also provide examples comparing my tool with that one.

1.1 Definition of differences

1.1.1 Primitive elements

For primitive elements of JSON like strings, numbers, booleans, and null values, both type difference and value difference are regard as the "value_changes" using a function defined in document.cpp. The primitive element must be exactly same or it will be considered as difference.
For value difference:

```
PS E:\Json Diff> .\jsondiff -left '{\"person\":\"Bruce\"}' -right '{\"person\":\"Clark\"}'
Different
value_changes: {"left":"Bruce","right":"Clark","left_path":["person"],"right_path":["person"]}
```

Figure 1: Value_changes

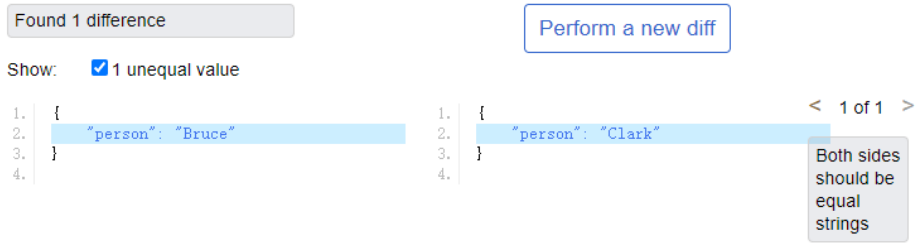

Figure 2: Web-tool: Value_changes

For type difference:

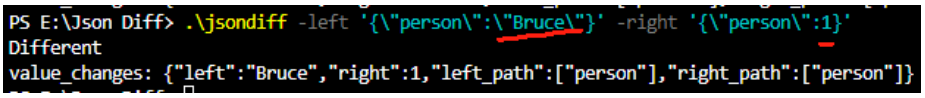

Figure 3: Type_changes (still regard as Value_changes in my tool)

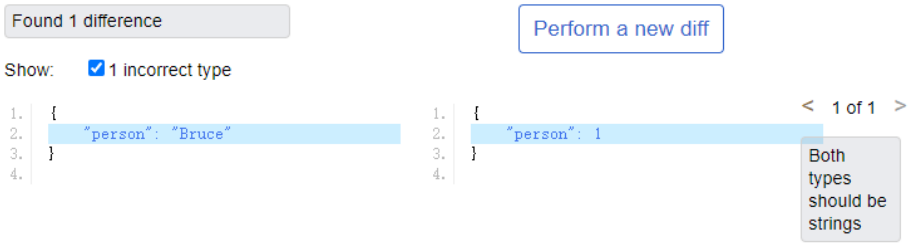

Figure 4: Web-tool: Type_changes

1.1.2 Object

For object, there will be three circumstances:
1. Object:remove: If a key is only contained in an object of left(old) JSON, it will be viewed as an "object:remove" because this element in doesn't exist in the object of the right(new) JSON meaning it is removed.

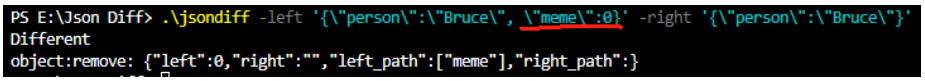

Figure 5: Object:remove

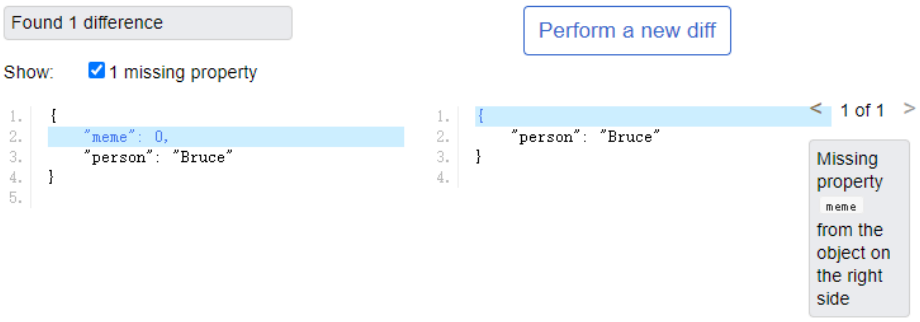

Figure 6: Web-tool: Object:remove

2. Object:add: If a key only exists in the object of right JSON, it will be regard as an "object:add" since this key value pair is not in the object of left JSON meaning that this element is added to the new JSON.

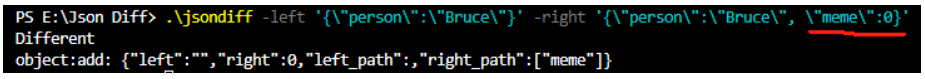

Figure 7: Object:add

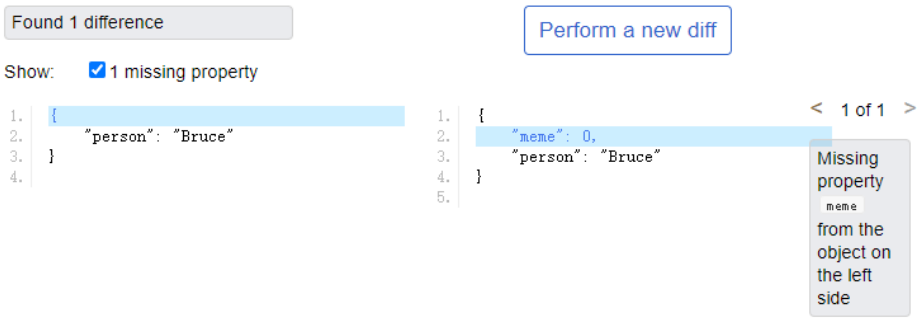

Figure 8: Web-tool: Object:add

3. Recursive comparing: If a key exists in both the left and right JSON files, a recursive difference analysis is performed on its value. In this case, differences are identified based on the nested structure of the value. Therefore, the reported differences might not be simply categorized as 'object:remove' or 'object:add'. Instead, the nature of the difference will depend on the deeper levels of the value's structure, including changes in the attributes of an object, elements of a nested array, as well as variations in primitive elements such as strings, numbers, booleans, and null values.
In JSON files, all elements are encapsulated within the top-level object. Thus, while the JSON structure starts at the object level, differences within are not merely reported as 'object:remove' or 'object:add'. Instead, they are reported according to the specific type and value changes within the individual elements, whether they are primitive elements such as strings and numbers, or complex ones like arrays.

```
PS E:\Json Diff> .\jsondiff -left '{\"person\":\"Bruce\"}' -right '{\"person\":\"Clark\"}'
Different
value_changes: {"left":"Bruce","right":"Clark","left_path":["person"],"right_path":["person"]}
```

Figure 9: Recursive comparing: value_changes

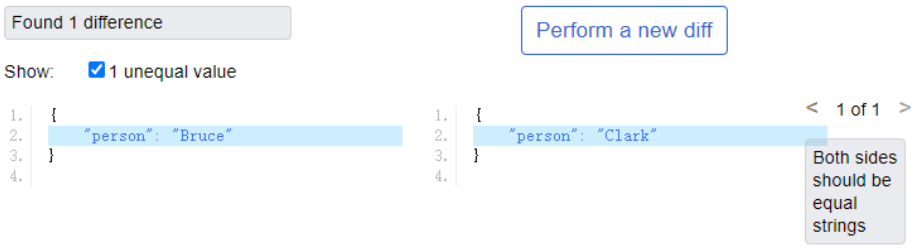

Figure 10: Web-tool: Recursive comparing: Value_changes

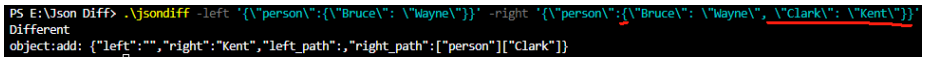

Figure 11: Recursive comparing: Object:add

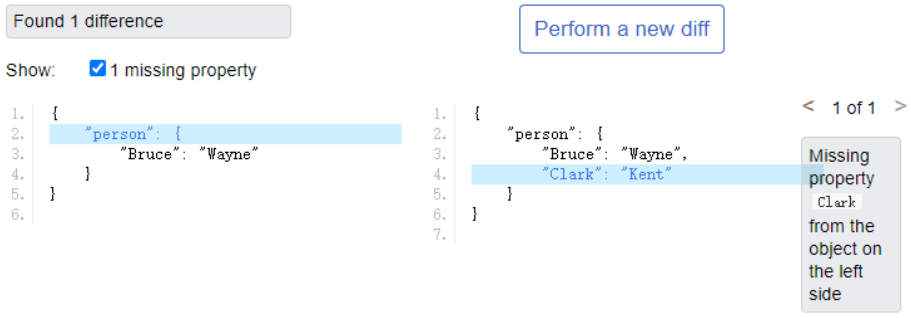

Figure 12: Web-tool: Recursive comparing: Object:add

1.1.3 Array

For array, I designed two way for analyzing the difference between the left(old) and right(new) JSON.
In fast mode (our default mode), there will be two cases:
1. Recursive comparing: Two arrays will be strictly compared one by one based on their indices. Still, elements that are not primitive will undergo recursive difference analysis and the difference between them may not be reported as an "array:remove" or "array:add" because the analysis is happening inside of it.

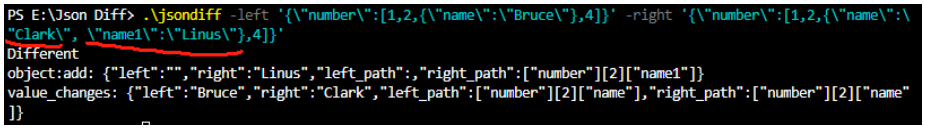

Figure 13: Recursive comparing: Object:add & Value_changes

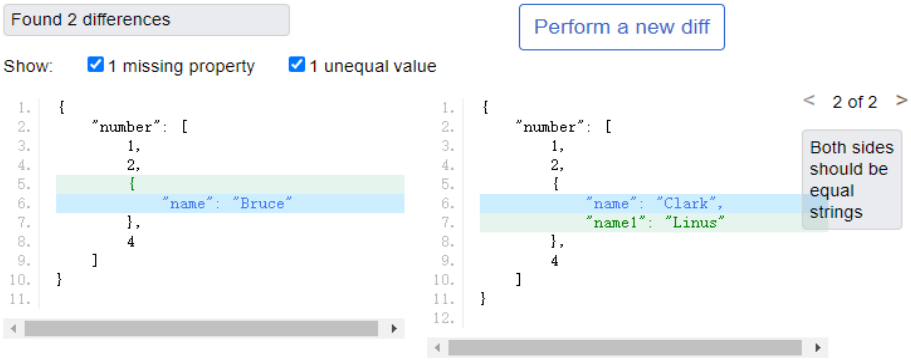

Figure 14: Web-tool: Recursive comparing: Object:add & Value_changes

2. Array:remove or array:add The elements corresponding to the remaining indices of the longer array will all be reported as either "array:remove" or "array:add," depending on which array is longer, whether it's the left or right array.

```
PS E:\Json Diff> .\jsondiff -left '{\"number\":[1,2,3,4,5]}' -right '{\"number\":[1,2,3,4]}'
Different
array:remove: {"left":5,"right":"","left_path":["number"][4],"right_path":}
```

Figure 15: Array:remove

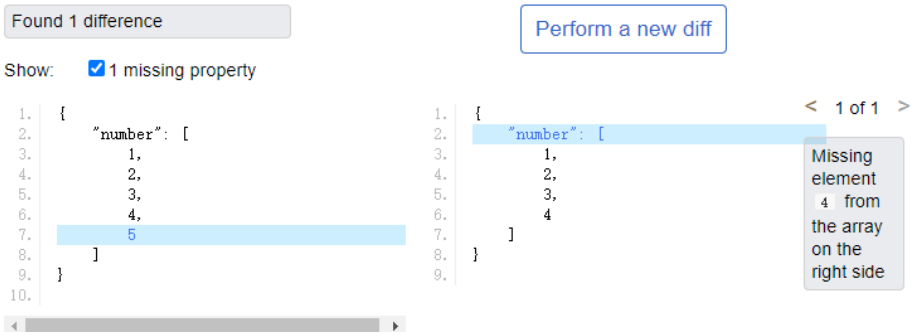

Figure 16: Web-tool: Array:remove

```
PS E:\Json Diff> .\jsondiff -left '{\"number\":[1,2,3,4]}' -right '{\"number\":[1,2,3,4,5]}'
Different
array:add: {"left":"","right":5,"left_path":,"right_path":["number"][4]}
```

Figure 17: Array:add

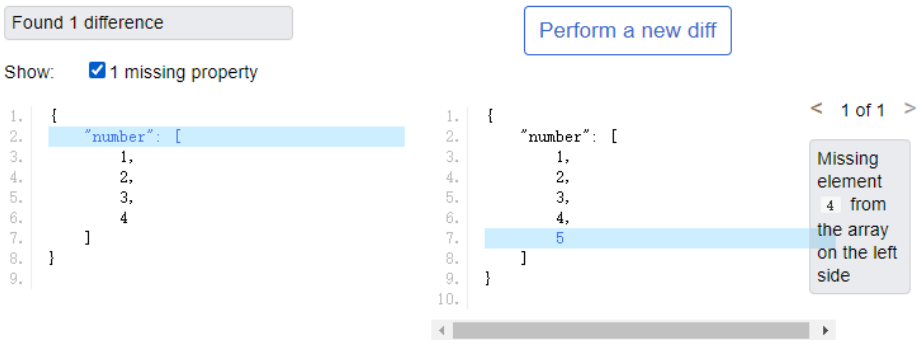

Figure 18: Web-tool: Array:add

In advanced mode, I use the Longest Common Subsequence (LCS) algorithm to match the best array index pairs. Once we have obtained all the best index pairs, we compare them in both arrays, reporting differences based on the internal element comparison. The remaining indices are all considered differences and

are directly reported as "array:remove" or "array:add," depending on whether the indices are from the left JSON or the right JSON.

```
PS E:\Json Diff> .\jsondiff -left '{\"number\":[1,2,3,4,5]}' -right '{\"number\":[1,2,4,5]}'
Different
array:remove: {"left":5,"right":"","left_path":["number"][4],"right_path":}
value_changes: {"left":3,"right":4,"left_path":["number"][2],"right_path":["number"][2]}
value_changes: {"left":4,"right":5,"left_path":["number"][3],"right_path":["number"][3]}
PS E:\Json Diff> .\jsondiff -left '{\"number\":[1,2,3,4,5]}' -right '{\"number\":[1,2,4,5]}' -A
Different
array:remove: {"left":3,"right":"","left_path":["number"][2],"right_path":}
PS E:\Json Diff>
```

Figure 19: Fast mode vs. Advanced mode

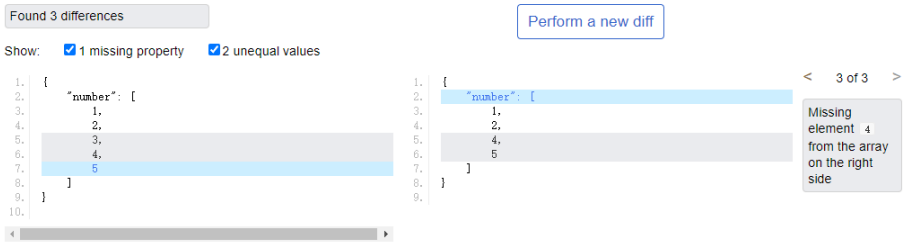

Figure 20: Web-tool (only has one mode: Fast mode)

I also compared some other online tools with my tool. I only list their results here because the only difference between my tool and others lies in the array comparison modes. For comparing primitive elements and objects, the method used by my tool is no different from that of other online tools. The only difference is in the naming of the differences identified; however, the actual differences detected are exactly the same.

My tool provides both 'fast mode' and 'advanced mode' for comparison. The fast mode is consistent with the approach of other tools, but the advanced mode utilizes the LCS algorithm to more accurately identify key differences. This is the distinct advantage of my tool.

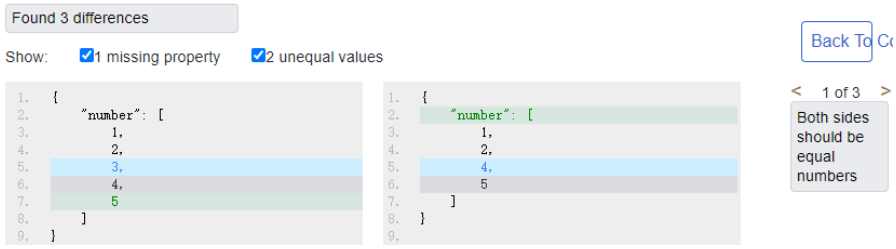

Figure 21: https://jsondiff.org/ (only has one mode: Fast mode)

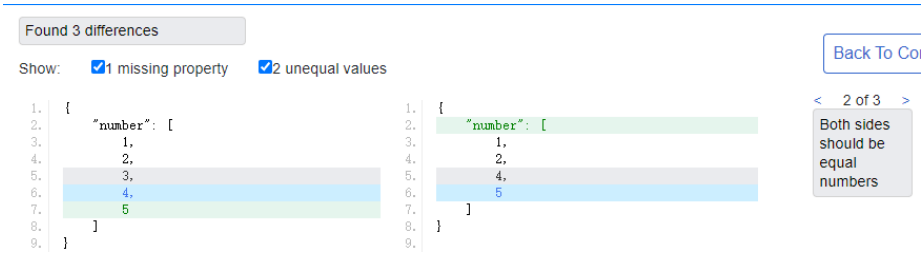

Figure 22: https://jsoncompare.org/ (only has one mode: Fast mode)

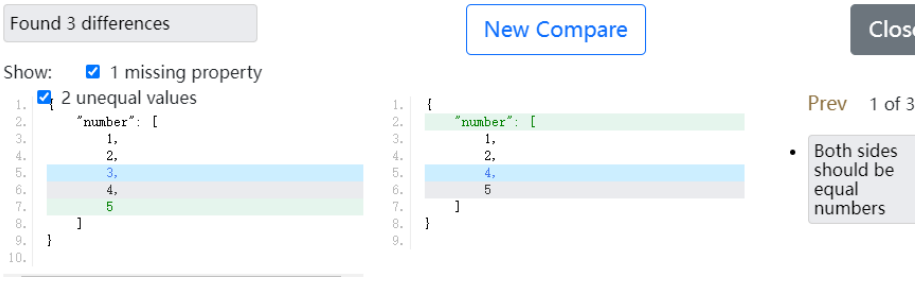

Figure 23: https://jsonformatter.org/json-compare (only has the Fast mode)

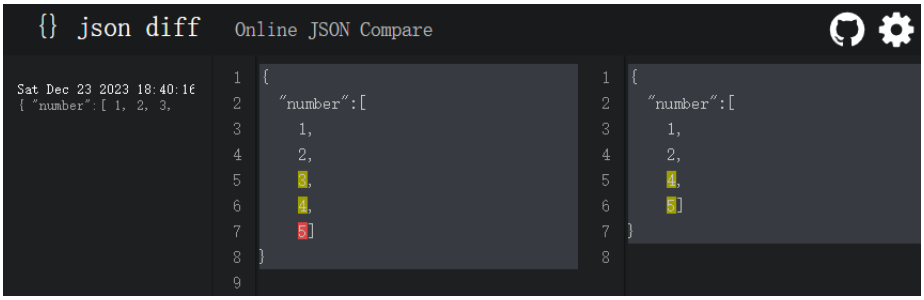

Figure 24: https://json-diff.com/ (only has one mode: Fast mode)

2 Algorithm

2.1 Overall differing algorithm

The overall differing algorithm is based on the types of layers of two JSON parse tree. When the argument "drill" is set to be true in array comparison, the report function will be disabled to avoid outputting worthless result while performing LCS algorithm in order to get best matching pairs. All the following functions with prefix "Jsondiffer.compare_" are listed in the `3.Implementation part`.

Algorithm 1 Difference Level Calculation

```
function _DIFF_LEVEL(level, drill)
    type ← level.get_type()
    if type == "TypeMismatch" then
        return JsonDiffer.compare_primitive(level, drill)
    else if type == "Object" then
        return JsonDiffer.compare_object(level, drill)
    else if type == "Array" then
        return JsonDiffer.compare_array(level, drill)
    else
        return JsonDiffer.compare_primitive(level, drill)
```

2.2 Array comparing algorithms

2.2.1 Fast mode

In the fast mode of array comparison, based on the indices of the left and right JSON arrays, for the common index part, a strict one-to-one comparison is carried out according to the indices, and they are further compared through the diff_level() function; for the parts where the left array or right array have non-common indices, they are simply regarded as array:remove or array:add.

Algorithm 2 Fast Array Comparison

```
function COMPARE_ARRAY_FAST(level, drill)
    len_left ← level.left.Size()
    len_right ← level.right.Size()
    min_len ← min(len_left, len_right)
    max_len ← max(len_left, len_right)
    total_score ← 0
    for index ← 0 to min_len − 1 do
        level_ ← create TreeLevel object with level.left[index], level.right[index], and paths
        score_ ← diff_level(level_, drill)
        total_score ← total_score + score_
    emptyValue ← create empty rapidjson::Value
    emptyRef ← reference to emptyValue
    for index ← min_len to len_left − 1 do
        if not drill then
            level_ ← create TreeLevel object with level.left[index], emptyRef, and left path
            JsonDiffer.report(EVENT_ARRAY_REMOVE, level_)
    for index ← min_len to len_right − 1 do
        if not drill then
            level_ ← create TreeLevel object with emptyRef, level.right[index], and right path
            JsonDiffer.report(EVENT_ARRAY_ADD, level_)
    return total_score / max_len
```

2.2.2 Advanced mode (Single-thread default)

In the advanced mode for array comparison, based on the user-defined similarity threshold or the default threshold (0.5), the LCS algorithm compares each element in the left JSON and right JSON arrays one by one (note: at this point, the comparison results only return scores and do not report differences). If the score obtained from the comparison is higher than the similarity threshold, the current position dp[i][j] will have a score equal to dp[i-1][j-1] plus the comparison score; otherwise, the score at dp[i][j] will be the maximum of dp[i][j-1] and dp[i-1][j]. During the backtracking phase, based on the indices corresponding to dp[i-1][j-1], we compare the elements of the two arrays (still, the comparison results only return scores and do not report differences). If the comparison score is higher than the threshold, we record this pair of indices. In other cases, the process aligns with the traditional LCS algorithm: we observe dp[i-1][j] and dp[i][j-1] to determine which direction to continue backtracking, repeating the previous steps. The pseudo code is given:

Algorithm 3 Longest Common Subsequence (LCS) Algorithm

```
function LCS(level)
    len_left ← level.left.Size()
    len_right ← level.right.Size()
    dp ← create 2D array of size (len_left + 1) × (len_right + 1) initialized to 0.0
    for i ← 1 to len_left do
        for j ← 1 to len_right do
            level_ ← create TreeLevel object with left[i-1], right[j-1], and paths
            score_ ← diff_level(level_, true)
            if score_ > 0 then
                dp[i][j] ← dp[i-1][j-1] + score_
            else
                dp[i][j] ← max(dp[i-1][j], dp[i][j-1])
    pair_list ← create an empty map
    i ← len_left
    j ← len_right
    while i > 0 and j > 0 do
        level_ ← create TreeLevel object with left[i-1], right[j-1], and paths
        score_ ← diff_level(level_, true)
        if score_ > SIMILARITY_THRESHOLD then
            pair_list[i-1] ← j-1
            i ← i − 1
            j ← j − 1
        else if dp[i-1][j] > dp[i][j-1] then
            i ← i − 1
        else
            j ← j − 1
    return pair_list
```

2.2.3 Advanced mode (Multi-thread)

In the multi-thread advanced mode, the performance can be greatly improved by around 4 times than the default single-thread advanced mode. Setting the count of threads as the same as your amount of CPU cores is strongly recommended because this will get the best performance.

- By arranging and combining the indices of the left JSON array and the right JSON array to form index pairs, these pairs are stored in a work queue. This work queue is shared among all threads, thus avoiding redundant calculations.
- The dp table is created and shared by all threads to write the similarity scores of pairs of array elements in.
- Multi-thread parallel computing is applied for computing all the similarity scores of pairs of elements of two arrays and use them to fill the dp table because this is the bottleneck of performance and there is no data dependency between the computing processes of different pairs of array elements.
- The granularity of the multi-threaded parallel computation is set to 6, meaning that each thread reads 6 pairs of array elements from the shared work queue at a time, computes the similarity scores for these 6 pairs, and then writes them back to the shared dp table in one go. This is the optimal granularity determined after multiple tests, striking a good balance between the overhead caused by too frequent thread locking and unlocking with too fine granularity and the load imbalance caused by too coarse granularity. These tests were conducted on JSON files composed of two arrays each containing about ten thousand elements, ‘{”number”:[1,2,3,4, ... ,10000]}‘ and ‘{”number”:[1,2,4,5, ... ,10000]}‘.

Algorithm 4 Parallel Difference Level Calculation

```
1: function PARALLEL_DIFF_LEVEL(work_queue, dp, level, work_queue_mutex, dp_mutex)
2:     ctn ← true
3:     volumn ← 6
4:     while ctn do
5:         task_list ← create vector of pair of unsigned integers with size volumn
6:         count ← 0
7:         lock work_queue_mutex
8:         for i ← 0 to volumn − 1 do
9:             if work_queue.empty() then
10:                ctn ← false
11:                break
12:            task_list[i] ← work_queue.front()
13:            work_queue.pop()
14:            count ← count + 1
15:        unlock work_queue_mutex
16:        score_ ← create vector of doubles with size volumn
17:        for i ← 0 to count − 1 do
18:            level_ ← create TreeLevel object with level.left[task_list[i].first], level.right[task_list[i].second], and paths
19:            score_[i] ← diff_level(level_, true)
20:        lock dp_mutex
21:        for i ← 0 to count − 1 do
22:            dp[task_list[i].first + 1][task_list[i].second + 1] ← score_[i]
23:        unlock dp_mutex
```

```
single thread
Single thread: computing dp table: 63.2834 s
Retreive LCS: 0.0239362 s
Different
array:remove: {"left":3,"right":"","left_path":["number"][2],"right_path":}
Total time: 63.6424 s
Execution time:  63.653387784957886 seconds
multiple thread:  6 threads
Building worklist: 0.434837 s
Allocating threads: 0 s
Parallel computing scores: 15.1879 s
Reconstructing of dp table: 0.227391 s
Retreiving LCS: 0.0219413 s
Different
array:remove: {"left":3,"right":"","left_path":["number"][2],"right_path":}
Total time: 16.1802 s
Execution time:  16.192211151123047 seconds
```

Figure 25: 6-threads vs. Single-thread, almost 4 times faster

- The dp table can be traversed and reconstructed from the position (1,1) according to the rules of the LCS algorithm after obtaining the similarity scores for each pair of array elements.
- The backtracking phase is still as the same as the single-thread advanced mode.

Algorithm 5 Parallel Longest Common Subsequence (LCS) Algorithm

```
1: function PARALLEL_LCS(level)
2:     len_left ← level.left.Size()
3:     len_right ← level.right.Size()
4:     dp ← create 2D array of size (len_left + 1) × (len_right + 1) initialized to 0.0
5:     work_queue ← create empty queue of pair of unsigned integers
6:     for i ← 0 to len_left − 1 do //creation of work queue
7:         for j ← 0 to len_right − 1 do
8:             work_queue.push(make pair(i, j))
9:     work_queue_mutex ← create new mutex //parallel computing
10:    dp_mutex ← create new mutex
11:    threads ← create empty vector of threads
12:    for i ← 0 to num_thread − 1 do
13:        threads.push_back(create new thread(parallel_diff_level, args...))
14:    for each thread in threads do
15:        thread.join()
16:    for i ← 1 to len_left do //reconstruction of the dp table
17:        for j ← 1 to len_right do
18:            if dp[i][j] > 0 then
19:                dp[i][j] ← dp[i - 1][j - 1] + dp[i][j]
20:            else
21:                dp[i][j] ← max(dp[i - 1][j], dp[i][j - 1])
22:    pair_list ← create empty map of unsigned integers
23:    i ← len_left
24:    j ← len_right
25:    while i > 0 and j > 0 do //Backtracking
26:        level_ ← create TreeLevel object with args...
27:        score_ ← diff_level(level_, true)
28:        if score_ > SIMILARITY_THRESHOLD then
29:            pair_list[i - 1] ← j - 1
30:            i ← i − 1
31:            j ← j − 1
32:        else if dp[i - 1][j] > dp[i][j - 1] then
33:            i ← i − 1
34:        else
35:            j ← j − 1
36:    return pair_list
```

3 Implementation

All classes and functions are encapsulated within the `Linus::jsondiff` namespace in `document.h` and `document.cpp`, which helps to avoid naming conflicts with other parts of a program that might use similar names. An additional `jsondiff.cpp` is used for command-line invocation, which takes parameters and produces output results.

3.1 Utility Functions

- `ValueToString()` converts a RapidJSON `Value` to a `std::string`. It is used to serialize JSON values into strings for comparison and reporting.
- `KeysFromObject()` takes a JSON object represented by a RapidJSON `Value` and returns a `std::vector<std::string>` containing all keys in the object. It is used to iterates over keys in a JSON object for comparison.

3.2 `TreeLevel` Class

It represents a level in the JSON tree during comparison. It holds two JSON values, one from each JSON tree being compared, and paths that describe the location of these values in their respective trees.

- `TreeLevel()` is the constructor that initializes a `TreeLevel` object with JSON values and their paths.
- `get_type()` determines the type of JSON values (Object, Array, String, Number, Bool, Null, or TypeMismatch) for comparison purposes.
- `to_info()` serializes the `TreeLevel` information into a JSON-like string format for reporting.
- `get_key()` constructs a key that uniquely identifies the `TreeLevel` based on its paths in the JSON trees.

3.3 `JsonDiffer` Class

The core class that performs the comparison between two JSON values.

- `JsonDiffer()` sets up the two JSON values, the mode for comparison (advanced or fast), and a similarity threshold for array comparisons.
- `report()` records events, including 'add', 'remove', and 'change', during comparison, along with the `TreeLevel` where it occurred.
- `to_info()` returns a map with event types as keys and vectors of stringified JSON differences as values.
- `compare_array()` compares two JSON arrays, with an option for advanced or fast comparison, and returns a similarity score.

- `compare_array_fast()` is a fast array comparison method that iterates over the arrays in parallel and compares corresponding elements.
- `parallel_diff_level()` manages the task distribution for the parallel comparison of JSON array elements. It takes tasks from a shared work queue and processes them in parallel, calculating similarity scores for each comparison and updating a shared dynamic programming table with these scores. It is called by `parallel_LCS()`.
- `parallel_LCS()` implements the LCS algorithm with multi-thread parallel method to find the best matching pairs of elements between two arrays. It will generate work queue and dp table first using the indices of two JSON arrays. It is used by `compare_array_davanced()`.
- `LCS()` implements the LCS algorithm to find the best matching pairs of elements between two arrays, used by `compare_array_advanced()`.
- `compare_array_advanced()` is a detailed and slow method that uses the Longest Common Subsequence (LCS) algorithm to compare arrays. It is in charge of using multi-thread parallel computing (parallel_LCS()) or single-thread (LCS()) computing.
- `compare_object()` compares two given JSON objects, identifying added, removed, or changed keys and values.
- `compare_primitive()` compares two primitive JSON values (i.e., string, number, bool, null) and determines if they are equal or not.
- `_diff_level()` is an internal function that dispatches the comparison to the appropriate method based on the types of the JSON values.
- `diff_level()` caches the results of comparisons and returns a similarity score for a given `TreeLevel`.
- `diff()` initiates the comparison process from the root level of the JSON values and determines if they are identical.

4 How to use

The program can be invoked directly from the command line using the command ‘jsonsdiff -left ”path to your left JSON” -right ”path to your right JSON”‘ or just put your JSON content behind the ’-left’ and ’-right’. The ‘-advanced‘ or ‘-A‘ option allows you to enable the LCS algorithm for array comparison, and the ‘-similarity‘ or ‘-S‘ followed by a double value allows you to customize the similarity threshold. The ’-nthreads’ or ’-N’ option allows you to enable the LCS algorithm for array comparison using multi-thread parallel computing which can improve the performance of advanced mode. All the source code and binary files can be found on my Github: https://github.com/Linus-Lee-1037/JSONdiff.