SILESIAN UNIVERSITY OF TECHNOLOGY PUBLISHING HOUSE

SCIENTIFIC PAPERS OF SILESIAN UNIVERSITY OF TECHNOLOGY 2024
ORGANIZATION AND MANAGEMENT SERIES NO. 214

EVALUATING PERFORMANCE AND MEMORY TRADE-OFFS IN DYNAMIC PROGRAMMING: A COMPARATIVE STUDY OF TABULATION AND MEMOIZATION TECHNIQUES

Vasyl LYTVYN[1], Oliwier SOJDA[2*]

[1] Information Systems and Networks Department, Lviv Polytechnic National University, Lviv, Ukraine; Vasyl.V.Lytvyn@lpnu.ua, ORCID: 0000-0002-9676-0180
[2] WSB Merito University, Poland; oliwier.sojda@nuebyte.com, ORCID: 0009-0008-0316-3215
* Correspondence author

Purpose: This research aims to conduct a diverse quantitative analysis of the two main dynamic programming techniques: Tabulation and Memoization on the Word Break and Minimum Sum Path in a Grid problems. The results should provide insights into the dependencies and entanglements in which one technique evinces superior results or both present regularly similar outputs depending on the input.
Design/Methodology/Approach: The study was carried out using Swift programming language and Xcode IDE to perform a series of tests on Tabulation (bottom-up approach) and Memoization (top-down approach) to solve various computational scenarios of the Word Break and Minium Sum Path in a Grid problem. To recognize the performance of each technique, we analyzed the data using R and RStudio.
Findings: Memoization leverages its caching mechanism that stores the results of previously calculated problem subproblems. In scenarios when the number of overlapping subproblems is significant, it offers a superior execution time over Tabulation. Nonetheless, as the complexity increases and the number of overlapping subproblems lowers, Memoization fails to provide consistent results and may lead to unpredictable memory usage. In those scenarios, Tabulation offers predictable memory usage with expedited execution times, which is beneficial in resource-constrained environments. Figures 1, 2, 3, and 4 present optimized solutions to the Word Break and Min Sum Path in a Grid problems, solved using Tabulation and Memoization. Figures 5 and 6 present memory usage and execution time achieved with our testing scenarios.
Originality/Value: The research gives us a practical understanding of scenarios in which Tabulation or Memoization is a better algorithm optimization technique.
Keywords: Tabulation, Memoization, Dynamic Programming.

1. Introduction

Dynamic Programming relies on decomposing a complex problem into overlapping subproblems, where each subproblem gets solved only once. The main two techniques in Dynamic Programming are Tabulation, known as the top-down approach, and Memoization, known as the bottom-down approach.

The tabulation technique relies on iteratively solving the smallest subproblems and storing their results in a so-called dp table. These results are then used to tackle larger subproblems progressively. The solution to the original problem is obtained from the final entry. The "bottom-up" approach originates from this exact characteristic – working way up to the solution to the original, much more complex problem (Levitin, 2011).

On the other hand, the Memoization technique begins with the original problem and then breaks it down into smaller subproblems. Results of these subproblems are stored in a so-called memo and retrieved in case of needing to stumble upon the same function's parameter values, which would produce an already computed result (Kleinberg, Tardos, 2005; Haftmann, Nipkow, 2020). Such behavior evinces a "top-down" approach as we work down to the smallest subproblems.

It's essential to pick a technique based on the problem characteristics and constraints. We should pay attention to problem structure, resource constraints such as available amount of memory, the efficiency of the chosen programming language with recursion, and initialization requirements such as initializing a dp table. A deep understanding of these characteristics will allow us to leverage these techniques' strengths and provide the well-optimized solutions.

2. Methodologies

Our problems of choice are the Word Break and the Minimum Sum Path in a Grid problems. We assess how each technique will impact memory usage and execution time and determine if the performance is predictable.

The Word Break problem is based on a variable-length substring, which makes it a prominent recursive problem. Meanwhile, the Minimum Sum Path in a Grid problem has a grid-based structure, where subproblems directly depend on neighboring cells. Some number of overlapping subproblems characterize both problems.

The Word Break Problem asks if a given string can be segmented into dictionary-defined words. This problem is a foundation of natural language processing (NLP), language modeling, text parsing, and word segmentation. Their implementation may be found in solutions such as spell checkers, autocomplete systems, or speech-to-text algorithms, where text needs to be broken down into a set of meaningful words.

In the Minimum Sum Path in a Grid Problem, each cell of the 2-dimensional grid has a value associated with it. The objective is to traverse from the upper left corner to the lower right corner and return the minimum required sum of the cell values needed to reach the destination.

In the Minimum Sum Path in a Grid Problem, each cell of the 2-dimensional grid has a value associated with it. Our objective is to traverse from the upper left to the lower right corner and return the minimum required sum of the cell values needed to reach the destination (Cormen et al., 2009; Erickson, 2019). This problem finds its implementation in navigation-related tasks such as optimizing supply chains or designing a network.

We conducted the examinations on a standardized hardware and software setup to ensure consistent and reproducible findings. The hardware consisted of an Apple M3 Max MacBook Pro with 64GB of RAM with MacOS Sequoia 15. The software environment was centered around Xcode 16 and Swift 6 for measuring algorithms' memory consumption and execution time, and R and RStudio were used for data analysis.

3. Test Cases

For the Minimum Sum Path in a Grid Problem, the test cases included various spatial configurations such as small square matrices (30x30), medium square matrices (50x50), large square matrices (100x100), rectangular matrices (50x100), boundary scenarios (30x150), and edge case performance-oriented grids (150x150).

For the Word Break problem, the test cases covered a variety of input lengths and dictionary sizes with differing distributions and lengths of words within the dictionary. They included configurations such as (60,10,500), (160,10,500), (160,20,500), (60,10,1000), (60,20,1000), (160,10,1000), (160,20,1000), (160,10,2000), (160,20,2000), (300,10,500), and (300,20,2000), where the first item represented the size of input string, second one the length of a word in the dictionary, and the third one the size of dictionary.

3.1. Solutions

In the Word Break problem with the Memoization technique, we first convert the dictionary into a set, allowing us for O(1) lookup times. Then, we define the memo, which stores the results of previously computed subproblems. Then, we define the canBreak function, which checks if a string can be segmented into words in the dictionary. Within this recursive function, we first check for a base case – if the start index equals the length of the input string, if it does, the function returns true, indicating that the input string can be segmented. Otherwise, we check if the memo already contains a computed result for the same parameters – the exact start value; in such a case, we return that value to prevent redundant recomputation. If both of these checks

fail, the function enters a for-loop where it iterates over all possible end indices from start +1 to the length of the input string. For each substring, we check if it exists in the dictionary; if it does, and the recursive call to break with the end set as a startIndex also returns true, we store the obtained result in the memo and return true. If the for-loop finishes and we've not found any matching cases, we also store the result in the memo and return false.

```
func wordBreakMemoization(s: String, wordDict: [String]) -> Bool {
   let wordSet = Set(wordDict)
   var memo: [Int: Bool] = [:]

   func canBreak(_ start: Int) -> Bool {
      if start == s.count {
         return true
      }
      if let memoized = memo[start] {
         return memoized
      }
      for end in (start + 1)...s.count {
         let substring = String(s[s.index(s.startIndex, offsetBy: start)..<s.index(s.startIndex,
offsetBy: end)])
         if wordSet.contains(substring) && canBreak(end) {
            memo[start] = true
            return true
         }
      }
      memo[start] = false
      return false
   }
   return canBreak(0)
}
```

Figure 1. Solving the Word Break Problem with Memoization Technique.

In the Word Break problem with a tabulation technique, firstly, we initialize a dp table with a size equal to the size of the input string +1. By default, the table is filled with false values, except for the first value, which we set to true (an empty string can always be segmented). The function then performs an outer loop, iterating over all input indices starting from 1. For each index i, a nested loop checks all possible start indices j from 0 up to i. The function checks if the segment up to j can be segmented (if yes, then dp[j] returns true) and if the obtained substring is present in the dictionary. If both conditions are met, we set the value under the i index in the dp table to true and break out of the inner loop. The solution to the main problem is the last element of the dp table.

In the Minimum Sum Path in a Grid problem with a tabulation technique, we initialize the memo with default values of -1. The memo will store all the minimum path sums for each cell. Then we define a recursive function called dp, which calculates a minimum path sum to reach a specific cell based on the row and column. We perform checks to ensure that row and column don't extend the table's boundaries. If a value has already been computed and is present in the memo, we return it. Then, the function calculates the minimum path sum for the cell above and

to the left from our current coordinates (defined by parameters passed into the function). We pick the smaller one from the obtained results, place it in the memo, and return it.

```
func wordBreakTabulation(s: String, wordDict: [String]) -> Bool {
   let wordSet = Set(wordDict)
   var dp = Array(repeating: false, count: s.count + 1)
   dp[0] = true

   for i in 1...s.count {
      for j in 0..<i {
         let substring = String(s[s.index(s.startIndex, offsetBy: j)..<s.index(s.startIndex,
offsetBy: i)])
         if dp[j] && wordSet.contains(substring) {
            dp[i] = true
            break
         }
      }
   }
   return dp[s.count]
}
```

Figure 2. Solving the Word Break Problem with Tabulation Technique.

```
func minSumPathTabulation(grid: [[Int]]) -> Int {
   guard !grid.isEmpty else { return 0 }
   let rows = grid.count
   let cols = grid[0].count
    var dp = grid
     for col in 1..<cols {
      dp[0][col] += dp[0][col - 1]
   }
   for row in 1..<rows {
      dp[row][0] += dp[row - 1][0]
   }
      for row in 1..<rows {
      for col in 1..<cols {
         dp[row][col] += min(dp[row - 1][col], dp[row][col - 1])
      }
   }
      return dp[rows - 1][cols - 1]
}
```

Figure 3. Solving the Min Sum Path Problem with Tabulation Technique.

In the Minimum Sum Path in a Grid problem with a memoization technique, we initialize our dp table as a grid copy. Then, we fill the first row and column with the values above and to the left of the cell, respectively. Subsequently, the function iterates over the cells and picks the smaller value from the cell above and to the left. The answer to the main problem is computed in the right lower corner cell.

```
func minSumPathMemoization(grid: [[Int]]) -> Int {
   guard !grid.isEmpty else { return 0 }
   let rows = grid.count
   let cols = grid[0].count
   var memo = Array(repeating: Array(repeating:-1, count: cols), count: rows)
   func dp(_ row: Int, _ col: Int) -> Int {
      if row < 0 || col < 0 {
         return Int.max
      }
      if row == 0 && col == 0 {
         return grid[0][0]
      }
      if memo[row][col] != -1 {
         return memo[row][col]
      }
      let left = dp(row, col - 1)
      let up = dp(row - 1, col)
      let minPath = grid[row][col] + min(left, up)
      memo[row][col] = minPath
      return minPath
   }
   return dp(rows - 1, cols - 1)
}
```

Figure 4. Solving the Min Sum Path Problem with Memoization Technique.

4. Results

For the Word Break problem, we meticulously selected cases that would cover various combinations of input lengths of each of the following parameters: word length, dictionary size, and size of a word in the dictionary. The cases are A: (60,10,500), B: (60,10,1000), C: (60,20,1000), D: (160,10,500), E: (160,10,1000), F: (160,20,1000), G: (160,10,2000), H: (160,20,2000), I: (300,10,500), J: (300,20,2000). Tabulation provides a more consistent execution time, showing a platykurtic distribution with fewer extreme values and distribution heavily concentrated around the mean. On the other hand, Memoization shows a leptokurtic distribution, which does not guarantee a repetitive performance due to extreme outliners. Still, it takes significantly less time to solve all cases. The execution time ranges from 0.9ms to 7.2ms (Figure 6).

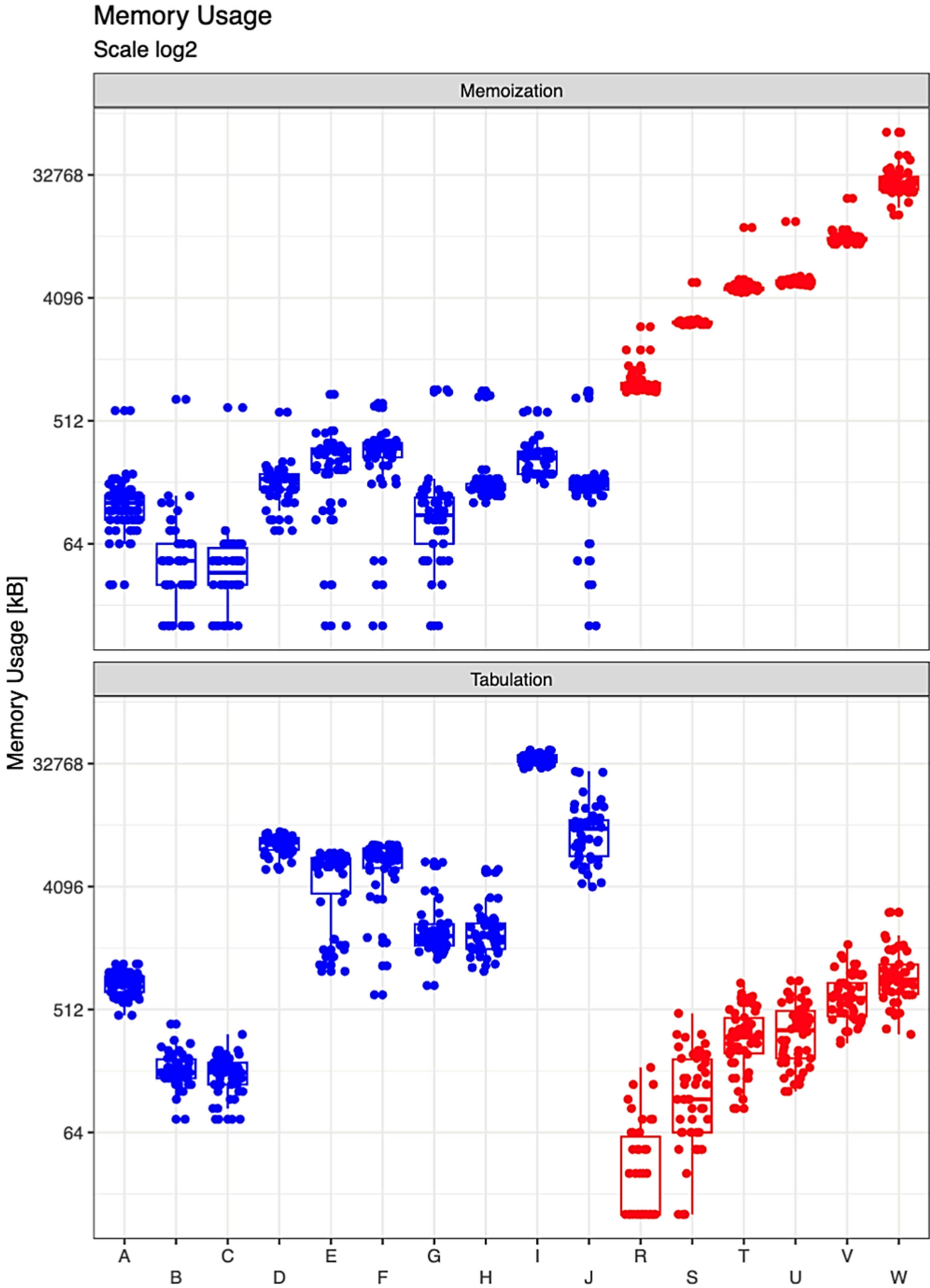

Figure 5. Memory Usage for Word Break and Min Sum Path Problems in kB.

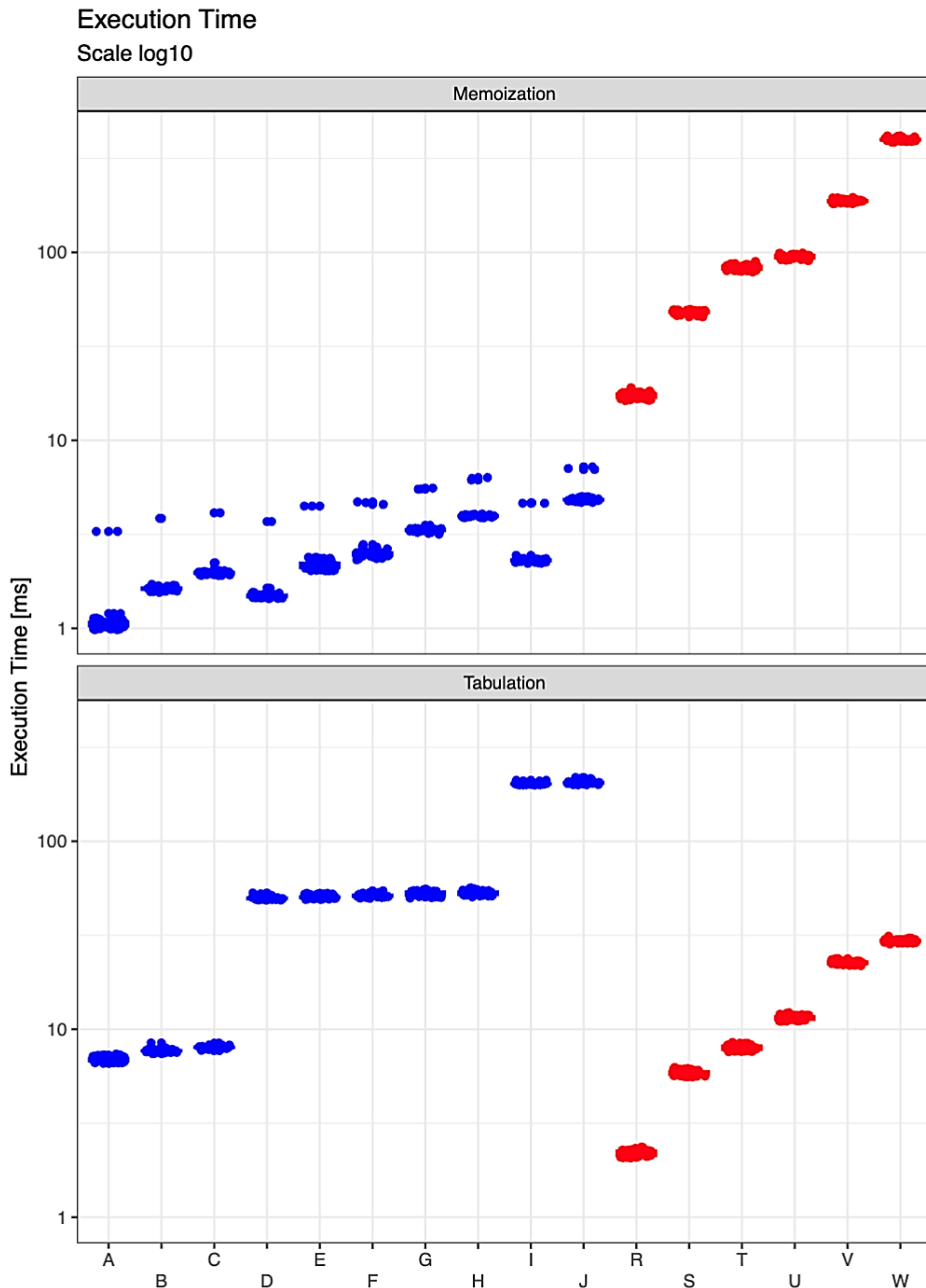

Figure 6. Execution Time for Word Break and Min Sum Path Problems in ms.

Meanwhile, Tabulation requires between 6.5ms and 218ms to compute the result (Figure 6). The increase in execution time is heavily dependent on the length of both input word and words in the dictionary. Dictionary size seems to be a lesser factor as cases with only

different sizes of dictionaries score similarly. Memoization also continues its unpredictability with memory management, as it utilizes as little as 16kB and as much as 864kB when the dictionary reaches a size of 2000 (Figure 5). Despite its unpredictability, Memoization provides substantially superior memory utilization, as Tabulation uses between 80kB and 41232kB across all cases, even though stable and consistent (Figure 5). For Memoization, the main contributing factor is the dictionary size; meanwhile, Tabulation is the size of the word and words in the dictionary.

For the Minimum Sum Path in a Grid problem, we selected test cases of various grid sizes and shapes such as R: (30,30), S: (50,50), T: (30,150), U: (50,100), V: (100,100), W: (150,150). Both Tabulation and Memoization reveal an absence of extreme outliners, providing more consistent execution times. Tabulation is significantly faster across all cases, ranging from 2ms for the most minor test case (30x30) to 31ms for the most complex (150x150) (Figure 6). It takes Memoization significantly longer to complete the same instances, 16ms to 386ms, respectively (Figure 6). For both, the means grow with the grid size. Tabulation appears far more stable and resourceful in terms of memory use. Compared to Memoization, it uses 21 times less memory on average. The results range from 16kb to 2656 kB (Figure 5). Memoization has not only higher memory use but also frequent extreme outliners. The minimum usage is at 832kB, 52 times higher, and the maximum is close to 26 times higher than Tabulation (Figure 5). In both instances, there's a relation, the larger the grid, the more memory must be allocated to solve the problem.

5. Conclusions

Our findings indicate that Memoization is particularly efficient when a problem has many overlapping subproblems due to its caching mechanism. However, it generally requires more memory in complex scenarios when the ratio of overlapping to non-overlapping subproblems lowers. Despite the Tabulation being potentially slower, it offers a more consistent and efficient memory usage due to its iterative approach. That makes it a far more suitable approach for an environment with tight memory constraints.

Future work could explore some hybrid approaches or extend this comparison to other dynamic programming problems to further refine our understanding of these techniques' applications and limitations.

References

1. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2009). *Introduction to Algorithms.*
2. Erickson, J. (2019). *Algorithms*.
3. Haftmann, F., Nipkow, T. (2020). *Verified Memoization and Dynamic Programming*.
4. Kleinberg, J., Tardos, É. (2005). *Algorithm Design*.
5. Leetcode (n.d.). *Problems and Solutions for Word Break and Minimum Path Sum*. Available at: https://leetcode.com/problems/minimum-path-sum/description/; https://leetcode.com/problems/word-break/description/
6. Levitin, A. (2011). *Introduction to the Design and Analysis of Algorithms*.
7. Schmatz, S. (2024). *Dynamic Programming and Memoization. Data Structures and Algorithms*.