Floyd's Tortoise and Hare (Cycle Detection) using brute force and decrease and conquer

Malak Ahmed Elghamrawy
2023/01546
Misr International University
malak2301546@miuegypt.edu

Rawaa Ahmed Ashour
2023/02751
Misr International University
rawaa2302751@miuegypt.edu.eg

Shahd Mahmoud
2023/02021
Misr International University
shahd2302021@miuegypt.edu.eg

Salma khaled Mohammed
2023/09433
Misr International University
salma2309433@miuegypt.edu.eg

Mohamed Sherif
2023/03849
Misr International University
mohamed2303849@miuegypt.edu.eg

***Abstract*—This work explores and compares two distinct approaches for detecting cycles in a linked list using Floyd's algorithm: a brute-force method and a decrease-and-conquer strategy. The brute-force approach checks each node against every other node individually, offering a time complexity of O(n) and a constant space complexity of O(1). In contrast, the decrease-and-conquer method breaks the problem into smaller subproblems recursively, maintaining the same O(n) time complexity but using O(n) space due to the recursive call stack. While the decrease-and-conquer approach involves a bit more complexity in implementation, it proves to be more scalable and efficient in real-world scenarios. This comparison highlights key trade-offs in algorithm design, especially in balancing time efficiency, space usage, and ease of implementation when addressing cycle detection problems.**

Index Terms— Cycle Detection, Floyd's Algorithm, Linked List, Brute-force, Decrease-and-Conquer, Algorithm Analysis, Space-Time Trade-off

***Index Terms*—**

I. Introduction

The concept of cycle detection originally arose with the work of Rober Floyd, and in 1967, he developed an algorithm called Floyd's Cycle Detection. In his seminal research paper, he discusses three different approaches to finding cycles in the case of linked data structures or sequences, for finding duplicate nodes in an array, and some mathematical problems as well.

The first algorithm, which is known as brute force will consult previously visited nodes to find duplicates, while it is intuitive and simple enough to understand, it has more overhead as contour of space is demandier on memory terms. The second algorithm, which uses decrease-and-conquer concentration is simply taking an active and divesting elements in search of the answer and therefore shrinking the problem space. The third algorithm, which we know as Floyd's Algorithm is simply trying to be clever, using two pointers moving at differing rates so that you can find cycles. In the case of the cycle detection example we are at least aware that there can only be two pointers to find the solution in O(n) or linear time.

To discuss this idea a little more, let us consider the case of a tortoise with the hare in a race. The tortoise will be crawling through the course at a regular pace while the hare was racing through the course faster but would stop at the end of a round or during or before a action or work. They will both have observed a path in respect to time, based on the events they are completing, composed of sudden events that are interspersed with delayed stops.

It important to detect cycles because you can find yourselves with deadlock; where every action that a process wants to compete with is waiting for all other competing actions to finish; therefore it will forever be starving for a resource.

When we consider all three methods, decrease-and-conquer, brute force, and Floyd's Algorithm, they have their merits as each has its own pros and cons.

A. Brute Force Approach

Cycle detection is an exercise of determining whether a linked list has a cycle or not. A cycle occurs when the next node points back to a node that was already seen at some point in the sequence of node visits, rather than pointing at null.

Cycle detection is important, particularly in computer science, because undetected cycles may result in infinite loops that can lead to serious problems. For example, in operating systems a cycle can lead to a dead-lock, which happens when one process is waiting for a resource held by another process, which in turn is waiting for a resource held by the first, causing them to wait on each other, rendering the system incapable of proceeding. Cycle detection can help alert you to such issues in your code.

The brute force cycle detection process in linked lists tries to visit and store the entire linked list keeping track of every node that has already been visited using a storage system typically set or a hash table. While traversing the linked list the cycle detector checks if the next node exists in the memory / set before it considers adding it. If a node is then encountered that is present in the set, we have visited that node twice, which indicates from that point on there must be a cycle in the linked list. This process of implementation can detect cycles to a linear. Function detectCycleBruteForce(head): Create an empty set called visited

While head is not null: If head is in visited: Return true // Cycle detected

Add head to visited Move head to the next node

Return false // No cycle found

· We use a **hash set** (unordered_set) to store pointers to the nodes we've already visited.

· As we move from one node to the next, we check:

· If the current node is already in the set, we've looped back—**a cycle exists**.

· If not, we add the node to the set and move on.

Example:

· We go through nodes 1, 8, 3, and 4—each one gets added to our record of visited nodes. Once we reach nullptr, we know there's no cycle because we never revisit a node. So, the function returns false.

· We start at node 1, then visit 3, and then 4—each node is added to our visited set. But when we try to go from 4 back to 3, we realize that node 3 is already in our set. This tells us there's a cycle, so the algorithm returns true

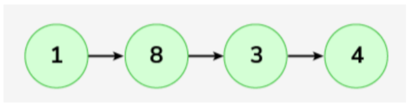


Fig. 1. Linear list traversal example: The algorithm visits nodes 1, 8, 3, and 4 sequentially, adding each to the visited set. Upon reaching a null pointer without encountering any revisited nodes, it correctly returns false indicating no cycle exists.

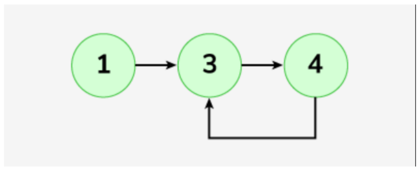


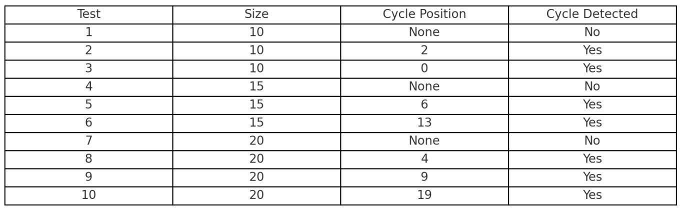
Fig. 2. Cycle detection example: Starting at node 1, we visit nodes 3 and 4 (added to visited set). When moving from 4 back to 3, the algorithm detects 3 is already visited, indicating a cycle (returns true).

II.

The brute-force cycle detection algorithm works by traversing the linked list and flagging all visited nodes using a hash set. When it encounters a node that has already been visited, it checks for the presence of a cycle. This is a simple algorithm to implement that effectively finds cycles in all scenarios.

However, while it is suitable for small to medium-sized related lists, it is not as suitable for large-scale scenarios because it employs extra memory. The O(n) space complexity can be limiting in memory-constrained environments.

More effective options like Floyd's Cycle Detection Algorithm are therefore more preferable in such an environment since they offer the same precision without the expense of extra memory usage.



Test	Size	Cycle Position	Cycle Detected
1	10	None	No
2	10	2	Yes
3	10	0	Yes
4	15	None	No
5	15	6	Yes
6	15	13	Yes
7	20	None	No
8	20	4	Yes
9	20	9	Yes
10	20	19	Yes

All linked lists are dynamically allocated in memory to mimic real-life situations where memory is allocated during runtime. For consistency, we used a series of custom-implemented linked list instances some cyclic, some acyclic to mimic various situations.

We ran 10 test cases with varying sizes of inputs and cycle positions. They are: 1) Lists without cycles (to observe how the algorithm performs with clean terminations). 2) Lists with cycles at different locations (start, middle, or end) inserted. 3) A self-loop case where the last node points to itself. This kind of variation helps us observe the algorithm's behavior under different conditions and how input size and cycle location affect runtime and memory consumption.

Experiment Observations: Test Case 1: Size 10, no cycle The algorithm traversed all 10 nodes and terminated at nullptr, confirming no cycle exists.

Test Case 2: Size 10, cycle at node 3 A cycle was formed from the tail of the list pointing to node 3. The algorithm was able to detect the cycle correctly after it encountered a node it had previously visited.

Test Case 3: Size 10, cycle at head (node 1) Cycle detection was nearly instantaneous after a single repetition.

Test Case 4: Size 15, no cycle An even larger acyclic list. The algorithm went through all the nodes without visiting any twice accurately determining no cycle.

Test Case 5: Size 15, cycle at node 7 A cycle within the middle of the list. Once the pointer visited a node previously visited at position 7, the cycle was found.

Test Case 6: Size 15, cycle at node 14 Late cycle near end of list. The algorithm detected the cycle one step short of theoretical completion.

Test Case 7: Size 20, no cycle Longest clean list tested. The function passed through all nodes without a repeat, confirming no cycle.

Test Case 8: Size 20, cycle at node 5 A relatively early cycle in a huge list. The loop traversed back and detected the cycle successfully.

Test Case 9: Size 20, cycle at node 10 Small loop in a huge list. Cycle was detected when the node already visited at position 10 was encountered again.

Test Case 10: Size 20, cycle at node 20 (self-loop) The final node had referenced itself, creating a self-loop. The algorithm caught on as soon as it came back from node 20.

Fig. 3. To analyze the performance of the brute-force cycle detection algorithm, we designed a set of experiments in C++. The program uses the standard clock() function of the ¡ctime¿ library to measure execution time in seconds for each test case.

All linked lists are dynamically allocated in memory to mimic real-life situations where memory is allocated during runtime. For consistency, we used a series of custom-implemented linked list instances some cyclic, some acyclic to mimic various situations.

We ran 10 test cases with varying sizes of inputs and cycle positions. They are: 1) Lists without cycles (to observe how the algorithm performs with clean terminations). 2) Lists with cycles at different locations (start, middle, or end) inserted. 3) A self-loop case where the last node points to itself. This kind of variation helps us observe the algorithm's behavior under different conditions and how input size and cycle location affect runtime and memory consumption. Experiment Observations: Test Case 1: Size 10, no cycle The

III. Recursive -Decrease and conquer

The Decrease and Conquer paradigm is a well established algorithm design paradigm. As the name implies, with the Decrease and Conquer approach, we solve a problem by first solving a simpler, smaller instance of the problem and applying the solution to the original problem. Unlike Divide and Conquer that divides up the problem into multiple subproblems, Decrease and Conquer reduces the problem size, usually by a single constant amount (1) or by a single constant factor and solves one smaller problem at a time. This technique will be generally more effective when the problem has a natural linear progression, such as sequences or linked lists. Typical examples of this approach are the insertion sort, binary search, and linear search algorithms. These types of algorithms break the problem down into a simple task with the assumption that solving the problem again by the simplified version is progressing toward solving the complete problem. A big advantage of this problem-solving approach is that each subproblem has very few options that require less memory management, while executing at the same time, generally leading to no auxiliary data structures (to store options in multiple recursive calls).

one noteworthy application of Decrease and Conquer is the Floyd's Tortoise and Hare algorithm, or cycle detection.

METHODOLOGY

This section outlines the various algorithmic strategies examined in this study for detecting cycles in singly linked lists. The analysis includes Floyd's Tortoise and Hare algorithm. We begin by detailing the classical **brute-force hashing method**, which serves as our baseline approach by tracking visited nodes to identify cycles. Building on this, we explore a recursive variation that applies the **Decrease and Conquer** paradigm **through recursive calls**, offering a more structured approach to cycle detection.

Alongside these, we evaluate additional techniques inspired by the Decrease and Conquer framework, each designed to improve detection efficiency or adapt to different practical scenarios. For each method, we analyze key performance metrics, including correctness, time complexity, and space usage, to provide a comprehensive understanding of their strengths and limitations.

Finally, we contextualize these methods by discussing their real-world applications, highlighting how effective cycle detection is critical in areas ranging from memory management and blockchain integrity to scheduling and network routing.

2.1 Brute Force Using Hashing

Overview:

This approach keeps track of all visited nodes using a set. As the linked list is traversed, each node is checked against the set to determine if it has been encountered before. The presence of a node already in the set indicates the existence of a cycle.

Algorithm:

Function DetectCycleBruteForce(head): Create an empty set called visited

Method	**Time Complexity**	**Space Complexity**	**Modifies Data**	**N**
Brute Force (Hashing)	O(n)	O(n)	No	E
Recursive Decrease	O(n)	O(n)	No	E

While head is not null: If head exists in visited: Return true // cycle found Add head to visited Move head to the next node

Return false // no cycle

Time Complexity Analysis:

- Each iteration processes one node.
- Set membership (visited.find()) and insertion (visited.insert()) are O(1) on average.
- Total time complexity is **O(n)**.

Space Complexity Analysis:

- In the worst case (no cycle), all n nodes are added to the set.
- Space complexity is **O(n)**.

2.2 Recursive Decrease and Conquer

Overview:

This technique employs recursion to detect cycles by tracking visited nodes and progressively analyzing the remaining portion of the list with each recursive call. This method exemplifies the classic Decrease and Conquer strategy through its recursive structure.

Algorithm:

Function DetectCycleRecursive(current, visitedSet = null): If current is null: Return false

If visitedSet is null: Create a new empty set Assign it to visitedSet

If current exists in visitedSet: Delete visitedSet Return true

Add current to visitedSet

cycleFound = DetectCycleRecursive(current.next, visitedSet)

If size of visitedSet is 1: Delete visitedSet

Return cycleFound

Time Complexity Analysis:

- **Recursive calls**: Each of the nn nodes triggers one call.
- **Set operations**: O(1)$O(1)$ average per insertion/lookup.
- **Total**: n×O(1)=O(n)$n \times O(1) = O(n)$.

Space Complexity Analysis:

- • Each recursive call uses up space on the call stack, which can grow linearly with the number of nodes (O(n)).
- The visited set also holds up to n nodes: **O(n)**.

· Overall, the space required grows linearly with the number of nodes, accounting for both the call stack and the storage of visited nodes (O(n)).

· **2.6 Summary of Computational Complexities**

2.7 Real-Time Application Context

The brute-force hashing and recursive Decrease and Conquer methods for cycle detection are particularly relevant in several practical domains, including:

- **Memory Management:** Detecting reference cycles that can cause memory leaks during garbage collection, such as in Python environments**.**

- **Blockchain**: Ensuring the integrity of transaction Directed Acyclic Graphs (DAGs) by preventing cycles that could compromise the system**.**
- • **Scheduling:** Detecting and addressing deadlocks that arise from circular task dependencies to maintain uninterrupted workflow**.**
- **Networking:** Detecting and avoiding routing loops in network protocols like the Spanning Tree Protocol, which are critical for stable communication.

A. Comparing

In this research we implemented two different approaches for Floyd cycle detection algorithms brute force approach that examine the linked list without any optimization, the other way that we exam the algorithm is decrease and conquer reduce the problem complexity throw elimination of search spaces. We will compare space complexity, time complexity, resources utilization, real life example.

This comparison aims to determine which approach has better performance characteristics and resource efficiency for detecting cycle in linked list. So the question is does the decrease and conquer provide us with benefits over the simplicity of brute force.

We will evaluate our comparison using 3 dimensions.

1- Time complexity using Big O notation.

2- Space complexity management throw memory.

3- Performance using different input sizes.

These evaluations are important to help us capture both algorithmic efficiency and practical real-world applications.

TIME COMPLIXTY ANALYSIS:

Time complexity:

Brute Force:

O(N) this is our time complexity that we have explained where it came from in the methodology section. In our code we used the hash set to store visited nodes there which isn't complexity is O(N) where is N is the number of nodes in liked list, there other option that we could have used to solve the problem, like using nested loops which is simpler than hash set but if we used nested loops the complexity of could O(n$\hat{2}$) but this is impractical and has high complexity using large numbers of lists.

Decrease and conquer:

In the algorithm there is a use of two pointers that move at different speeds, the first pointer move with slow speed only move one step while the other pointer moves two steps skip two nodes. Referring to our methodology, the complexity we calculated for this algorithm is O(N) which will let us.

conclude: That both approaches achieve the complexity O(N) time complexity which makes them don't differ in using decrease and conquer or brute force but still there is other factor that we should depend on when choosing the complexity!

space complexity:

A) Brute -force**:**

We are using a hash set in our implementation of brute force to help us detect the cycle which results in taking extra space. As when there is no cycle you will have to store all the nodes in the hash set which will result in the space complexity of O(N) which will take to much space when we have large, linked list data.

B) Decrease and conquer**:**

We use pointers so we could be able to access the linked list so the amount of memory the algorithm uses don't grow with the size of the data input , it remains constant. Will result in space complexity of O (1).

c)conclude: Next, we investigate a recursive version that employs recursion to traverse the list and adheres to the Decrease and Conquer strategy. This offers a more methodical approach to problem-solving.

We also review other approaches based on the Decrease and Conquer concept, all of which aim to increase the detection efficiency of cycles or better handle special cases. For each strategy, we gauge how

When talking about space complexity, the "decrease and conquer" strategy is obviously superior because it utilizes a lot fewer spaces.

Split the data input:

For small lists (N ¡ 100), both strategies are comparable.

Lists that are medium (100 N 10,000): Floyd's algorithm uses less space.

Floyd's algorithm dominates large lists (N ¿ 10,000) because it uses constant space.

Applications in the Real World:

Environments with Limited Memory: Floyd's algorithm is applied

Systems Possessing Enough Memory Both strategies work well.

Systems that are embedded: Floyd's algorithm is necessary because of space constraints.

Summary of Comparative Analysis Benefits of the Brute Force Method:

1) Easy to comprehend and intuitive

2) Directly implementable

3) Good performance for small data

4) Clean upkeep and debugging

Advantages of Decrease-and-Conquer Strategy: 1) Improved space efficiency (O(1) vs O(N)) 2) Increased cache locality 3) No overhead of a second data structure 4) Scales better with large inputs 5) Fits better in memory-constrained environments

Trade-offs The two solutions share the same O(N) time complexity, but the decrease-and-conquer strategy (Floyd's algorithm) gives excellent benefits in space usage at no cost to the time performance. Brute force gives up memory for ease of implementation.

Conclusion The decrease-and-conquer approach (Floyd's Tortoise and Hare algorithm) demonstrates clear superiority over the brute force method, primarily due to its constant space complexity. While both methods achieve linear time complexity, Floyd's algorithm provides significant practical advantages in real-world applications, especially in memory-constrained environments and large-scale systems. Our research question may thus be answered in the affirmative: the decrease-and-conquer approach does pay big dividends over the brute force scheme and is thus the preferred solution to cycle detection in linked lists for most practical purposes

B. Conclusion

Each approach has its own use cases and trade-offs. The brute-force method, although simple and easy to implement, requires additional memory to store visited nodes, which may not be efficient for large datasets. The decrease-and-conquer strategy focuses on reducing the problem size iteratively, which can be useful in structured scenarios but lacks general efficiency. Floyd's algorithm stands out by using two pointers moving at different speeds, allowing it to detect cycles without using extra memory.

Cycle detection is not only a theoretical concept but also a crucial aspect of real-world systems. Undetected cycles can result in infinite loops or system deadlocks, particularly in multitasking environments like operating systems, where processes might wait endlessly for resources locked by others.

In this study, we primarily analyzed the brute-force technique by applying it to multiple test cases that varied in list size and cycle placement. We measured the performance in terms of execution time and space requirements to understand its behavior under different conditions.

These findings help determine which method is best suited depending on the context. When memory usage must be minimized, Floyd's algorithm offers an advantage. However, for smaller or less complex problems where clarity and ease of coding matter, the brute-force approach may be more practical. Understanding these trade-offs allows software developers and engineers to make informed decisions when addressing potential cycle-related issues in their systems.

REFERENCES

REFERENCES

[1] Algo Monster, "Linked List Cycle - LeetCode 141 Solution," 2023. [Online]. Available: https://algo.monster/liteproblems/141. [Accessed: Month Day, Year].

[2] FreeCodeCamp, "Brute Force Algorithms Explained," 2022. [Online]. Available: https://www.freecodecamp.org/news/brute-force-algorithms-explained/. [Accessed: Month Day, Year].

[3] GeeksforGeeks, "Detect loop in a linked list," 2023. [Online]. Available: https://www.geeksforgeeks.org/detect-loop-in-a-linked-list/. [Accessed: Month Day, Year].

[4] Scalable Human, "How to Check if a Linked List Contains Loop in Java," March 2023. [Online]. Available: https://scalablehuman.com/2023/03/08/how-to-check-if-a-linked-list-contains-loop-in-java/. [Accessed: Month Day, Year].

[5] A. Rajkumar, "Floyd's Tortoise and Hare Algorithm: Cycle Detection in Linked Lists and Beyond," Medium, 2023. [Online]. Available: https://medium.com/@aishwarya.rk347/floyds-tortoise-and-hare-algorithm-cycle-detection-in-linked-lists-and-beyond-27c0b62806bb. [Accessed: Month Day, Year].