DESIGN AND ANALYSIS OF ALGORITHMS

Project Report : Team Baguette and Co

Submitted by

Denis Aira-Benvenuti ES04122

Yanis Guerin Berrabeh ES04033

Thomas Seignour ES04098

Department of computer science

Semester 2 : 2022-2023

Original Scenario :

In the early 80s, games were all 2D. Mario, Metroid, Castlevania... Doing 3D was really difficult, if not impossible, because computers couldn't handle a large number of operations. However, programmers discovered the BSP, which only renders the points that the player can see in front of them, enabling the creation of the first 3D games such as DOOM. In this way, they could simply render the closest point visible to the player and optimise the performance of most computers. So our project is based on BSP technology and makes it possible to find the closest point to the player, corresponding to an obstacle in a video game.

Why finding an optimal solution for this scenario is important :

Finding an optimized solution is important because we need to handle many points, which can result in numerous calculations, particularly during the quicksort process. To ensure scalability and the ability to modify this algorithm, it is crucial to ensure that it can handle a significant amount of data. By optimizing the algorithm's efficiency and reducing unnecessary computations, we can enhance its performance and make it suitable for processing large datasets. This scalability is essential for accommodating growing data sizes and ensuring the algorithm remains efficient even when dealing with substantial amounts of input data.

Suitability of sorting, DAC, DP, greedy and graph algorithms as a solution paradigm for the chosen problem by stating their strengths and weaknesses :

Paradigm	**Strengths**	**Weaknesses**	**Relevance to Program**
Sorting	Organizes data for easier search. - High-performance on large datasets.	No direct solution to the specific problem.- May require additional algorithms.	Not directly applicable
DAC	- Divides problem into manageable subproblems. - Suitable for divisible problems.	Not efficient for all problems. - Requires careful subproblem analysis.	Utilized for problem decomposition and independent subproblem solving in the program.
DP	Solves problems with overlapping subproblems. Stores intermediate results.	Requires significant memory usage.- Complex problem formulation.	Not used in the context of our program.
Greedy Algorithms	Easy to understand and implement.- Solutions close to optimality. - Efficient for certain optimization problems.	No guarantee of globally best solution.- Potential for suboptimal or incorrect solutions.	Not mentioned in our program.
Graph Algorithms	Suitable for graph-based problems.- Powerful algorithms available.	Requires appropriate graph representation.- High complexity for some graph operations.	Not used in our program.

Method used for our program (recurrence and optimisation):

In our Java program, we have employed the Divide and Conquer (DAC) paradigm to solve the problem we are facing. The algorithm we have designed follows the DAC approach, which involves breaking down the problem into smaller subproblems, solving them independently, and then combining the solutions to obtain the final result.

To implement the DAC algorithm in our program, we have structured our code as follows:

1. **Recurrence:**
    - We have incorporated the main recursive step within the **closestPointToPlayer()** function. This function takes a list of points (**set**), the size of the set (**N**), and a straight line (**line**) as input.
    - First, we call the **findPointsOnLine()** function, which identifies the points from the given set that lie on the specified straight line. We iterate over the set and check if the y-coordinate of each point matches the equation of the line (**point.getY() == line.getA() * point.getX()**).
    - If there are no points on the line, we return **null**, indicating that there are no obstacles on the line.
    - However, if there are points on the line, we proceed to perform a recursive call to the **quicksort()** function. This function allows us to sort the points based on their x-coordinate, using the **Lomuto partitioning** scheme.
    - Finally, we return the closest point, which corresponds to the first point in the sorted list.
2. **Optimization:**
    - The optimization aspect of our program lies in the sorting step. By sorting the points on the line based on their x-coordinate, we ensure that the closest point is located at the beginning of the sorted list.
    - This optimization enables us to retrieve the closest point in constant time (**O(1)**) once the sorting is performed. Without the sorting step,

we would have to iterate through the entire list to find the closest point, resulting in a time complexity of **O(N)**.

Algorithm Paradigm / Specifications : Divide and Conquer (DAC)

In our project, we have adopted the Divide and Conquer (DAC) algorithm paradigm.

Here's how our algorithm works:

1. Input : We take as input a set of points (`set`), the size of the set (`N`), and a straight line (`line`).

2. Output : Our goal is to find the closest point on the line from the given set of points.

3. Function: `closestPointToPlayer(set, N, line)`

- Base case: If the set of points is empty, we conclude that there are no obstacles on the line, and we return `null`.

- Recurrence:

    - To begin, we identify the points from the set that lie on the specified straight line. We accomplish this by calling the `findPointsOnLine(set, N, line)` function, which gives us a list of points `pointsOnLine`.

    - If there are no points on the line (i.e., `pointsOnLine` is empty), we return `null` to indicate the absence of obstacles.

    - If points do exist on the line, we proceed with sorting them based on their x-coordinate. We achieve this by making a recursive call to the `quicksort()` function.

    - Finally, we return the closest point, which corresponds to the first point in the sorted list.

4. Function: `findPointsOnLine(set, N, line)`

- We start with an empty list called `pointsOnLine`.
- Next, we iterate over each point `point` in the set:
  - If the y-coordinate of `point` matches the equation of the line (`point.getY() == line.getA() * point.getX()`), we add `point` to the `pointsOnLine` list.
- Once all points have been processed, we return the `pointsOnLine` list.

5. Function: `quicksort(A, l, r)`

- Base case: If the index `l` becomes greater than or equal to `r`, we have a sorted list, and we can return it.
- Recurrence:
  - We employ the Lomuto partitioning scheme to partition the list `A` around a pivot element. The pivot is chosen as the element at index `l`.
  - After partitioning, we make recursive calls to `quicksort(A, l, s-1)` and `quicksort(A, s+1, r)` to sort the elements before and after the pivot index, respectively.
- Once the recursion concludes, we return the sorted list `A`.

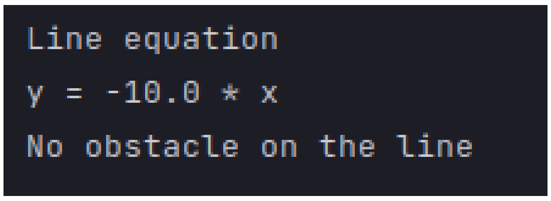
By leveraging the DAC paradigm in our program, we are able to divide the problem of finding the closest point on a straight line into smaller subproblems, solve them independently, and then combine the solutions to obtain the final result. The recursion occurs in the `closestPointToPlayer()` function, while the sorting optimization is achieved through the `quicksort()` function. This approach allows us to effectively and efficiently determine the closest point on the line from the given set of points.

Algorithm correctness :

As we explained it, we have two possible outputs: null and the closest point.

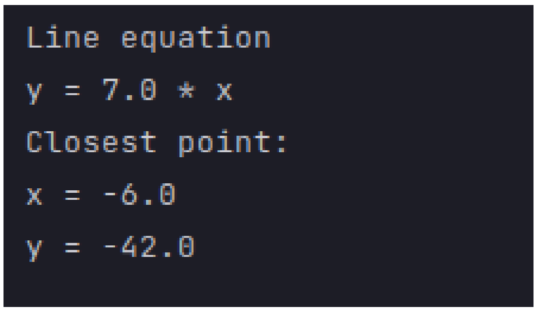
Our implementation makes it easy to see what happens.

Here, no point was found on the line :



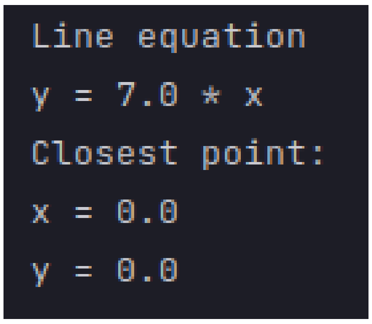
```
Line equation
y = -10.0 * x
No obstacle on the line
```

And here, at least one point was found and we display it :



```
Line equation
y = 7.0 * x
Closest point:
x = -6.0
y = -42.0
```

We often find that the closest point to 0,0 is 0,0 which is normal and can happen :



```
Line equation
y = 7.0 * x
Closest point:
x = 0.0
y = 0.0
```

Time complexity :

The time complexity of our Java program can be understood in terms of the different operations involved. Let's take a closer look :

1. Generating the set of points: When generating `N` random points using the `generateSet(N)` function, the time complexity scales linearly with the size of the set. So, as the number of points increases, the time required for this operation grows proportionally.

2. Identifying points on the line: To find the points on the line using the `findPointsOnLine(set, N, line)` function, we iterate through the set of points once. Checking if each point satisfies the line equation takes constant time. Hence, the overall time complexity for this step remains proportional to the size of the set.

3. Sorting the points: We employ the `quicksort()` function to sort the points based on their x-coordinate. In the best and average cases, the time complexity of quicksort is reasonable, with a complexity of O(N log N). However, in the worst case, when the pivot choices consistently result in unbalanced partitions, the time complexity can become less efficient at O(N^2). It's important to note that the size of the list being sorted depends on the number of points on the line, which can vary.

4. Retrieving the closest point: To find the closest point, we simply access the first element from the sorted list, which takes constant time regardless of the list's size.

Considering all these operations, we can summarize the overall time complexity of our program as follows:

Best case : The best case time complexity is O(N log N), which occurs when the sorting operation is efficient. It can also be O(1) when there is not point on the line.

Average case : The average case time complexity is also O(N log N), which is typically encountered when the input data is randomly distributed.

Worst case : In the worst case, where the sorting operation is less efficient due to poor pivot choices, the time complexity can be O(N^2).

It's important to keep in mind that the sorting step dominates the overall time complexity, as it plays a significant role in finding the closest point efficiently. To minimize the likelihood of worst-case scenarios, choosing an appropriate pivot selection strategy is crucial.

Thank you for reading our report