No Need for Search to Solve the Scanner

The ICPC Scanner Problem

David Knöpp
University of Tübingen, Germany
david.knoepp@student.uni-tuebingen.de

ABSTRACT

ACM releases several programming problems for ICPC every year. Problem "5168 - Scanner" provides the depth of a three-dimensional body as input, and demands the discrete reconstruction of the body as output. Exhaustive search fails to solve the problem in reasonable time. We propose a method that greatly reduces the search space for Exhaustive Search by exploiting a spatial property of the wanted body. Applying this method to randomly generated, valid inputs allows to fully avoid searching for $\sim 98\%$ of inputs.

KEYWORDS

ICPC, Scanner, Search

1 INTRODUCTION

The Scanner Problem introduces a scenario in which a three-dimensional body has been scanned for its depth. The task is to reconstruct the body from those depth-values alone. We explain the problem in detail in Section 2.

The intuitive solution to this problem is to search through all possible assignments' discretized matrices, through exhaustive or local search. This approach is not ideal for two reasons.

1. Exhaustive Search has an exponential time complexity, which makes it unusable for bigger matrices (especially the ones featured in the original problem description [1]).
2. Local Search cannot identify invalid inputs.

We cannot eliminate the need for search entirely (see Section 3.2). Still, we can view the problem in a different light.

- The results that we are searching for share a common property, which we call the "chunk property". The chunk property can be exploited to simultaneously identify the values of cells belonging to large groups (see Section 3.3).
- Some inputs may have zero or multiple solutions. We found two conditions that allow for identifying those kind of inputs early (see Section 3.4).
- We performed an experiment to quantize the number of inputs that can be solved in this way. We generated random, valid data and checked the number of inputs where the program did not time-out by performing a time-consuming search. The results show that we can solve $\sim 98\%$ of problems in sub-exponential time (see Section 4).

While we are able to avoid exhaustive search for most inputs, some inputs still require a full search. Thus, further research for faster search algorithms is still required.

We chose Python to implement the algorithm and the experiment, for its popularity and simplicity. The source code can be accessed via GitHub [2]. The code-listings in this paper follow a pythonic pseudocode style, which leans on the source code but has been heavily simplified to improve readability.

2 ENCODING A 3D BODY

To understand the scanner-algorithm, we must first understand the semantics of the input we are given. In this section, we start with a three-dimensional body and encode it step by step, to end up with a set of integer-arrays.

Step one: We lay a three-dimensional grid of dimensions $(h \times w \times n)$ over the body.

Step two: Along the third axis, the body is divided into a finite set of n two-dimensional slices. We then encode every slice independently of the others. All subsequent steps are applied to each slice individually. For the rest of the paper, we focus on one slice only for better understandability.

Step three: The slice is discretized as a grid of $h \times w$ cells. A cell's state is binary-encoded: if the cell contains *any* portion of the body, the cell is encoded as `FULL`. Otherwise, it is encoded as `EMPTY`. See Figure 1.

Step four: The grid of cells is now being measured for its depth along four directions. See Figure 2 for a visualization. The directions are:

- horizontal
- first diagonal (from bottom left to top right)
- vertical, and
- second diagonal (from bottom right to top left).

For each of those directions, the discretized body's depth is measured at all possible locations. For a grid of dimension $h \times w$, this yields four arrays $a_1, ..., a_4$ with $\dim(a_1) = h$, $\dim(a_2) = h + w + 1$, $\dim(a_3) = w$, and $\dim(a_4) = h + w + 1$. For our example, the resulting arrays are shown in Listing 1. Those four arrays make up the encoded slice.

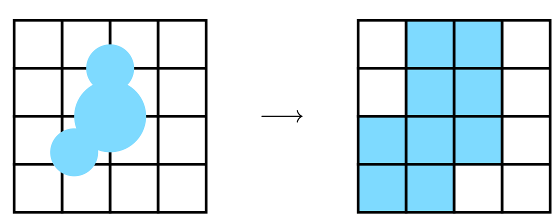

Figure 1: Discretizing an object as a grid of 4×4 cells

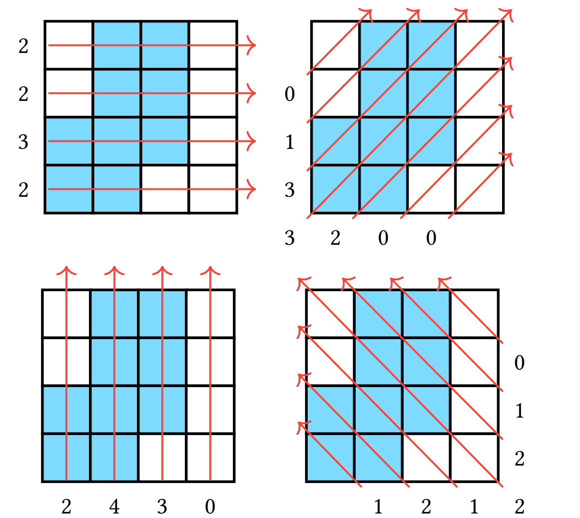

Figure 2: Encoding the object by counting FULL-valued cells along four directions

```
in_horiz = [2, 2, 3, 2], # horizontal
in_diag_lr = [0, 1, 3, 3, 2, 0, 0], #left-right-diagonals
in_vert = [2, 4, 3, 0], # vertical
in_diag_rl = [1, 2, 1, 2, 2, 1, 0] # right-left-diagonals
```

Listing 1: Input-Arrays encoding the 4×4 slice

3 RECONSTRUCTING A SLICE

We are given four integer-arrays representing the depth of the object (see Section 2). We want to reconstruct the discretized image from this data only. In this chapter, we explain

1. the intuitive approach and why it is not ideal (Section 3.2)
2. the *chunk property* (Section 3.3), and how we use it to assign many cells simultaneously
3. how we handle special cases (Section 3.4)

3.1 Interpretation of the input

So far, we interpreted the input as "depth of the object for a certain sub-array". Now, we want to introduce a slightly different interpretation. This will help to keep track of our progress later.

The new interpretation for an input-integer n corresponding to sub-array arr is: n equals the number of cells in arr that are currently `UNASSIGNED` and need to be assigned with `FULL` at some point. In other words: n is the number of `FULL`-valued cells in arr that have yet to be assigned.

This means that every time we assign `FULL` to a cell, one value in each of the input-arrays has to be reduced by one.

3.2 General Approach: Exhaustive Search

The dimension of the resulting matrix is known from the lengths of the horizontal and vertical input arrays. The possible values to fill the matrix with are also known (`EMPTY` and `FULL`). Thus, we can simply try out all possible solutions, calculate the depth of the resulting object at the four given directions, and compare those to the input arrays.

This approach is obviously not optimal, as it has exponential time complexity. However, we will need to incorporate exhaustive search into our solution to guarantee completeness, as we will see in Section 3.4.

3.3 Exploiting the chunk property

Our goal is to reduce the search space such that the slice can be reconstructed in sub-exponential time. To achieve this, we exploit a property of the input. We call this the **chunk property**.

We know that we are recreating images of two-dimensional bodies. The term "body" is interpreted as: **Most of the `FULL`-valued cells of the matrix are located next to each other**. What we do not expect, for example, is a noisy image, where the value of each cell is decided independently of its neighbors. See Figure 3 for an example.

From the chunk property follows that many sub-arrays (verticals, horizontals or diagonals) of the matrix are completely filled with `EMPTY`-values. Such sub-arrays represent the area in which the object **is not** located, like the edges of the matrix. The respective depth for such a sub-array must then be zero. Thus, from finding a value of zero in the input, we can deduct that, in the corresponding sub-array, all unassigned cells must be `EMPTY`-valued.

The chunk property makes no statement about sub-arrays being completely filled with `FULL`-values. Consider an object located in the center of a matrix and surrounded with empty cells. None of the sub-arrays of this matrix is completely `FULL`-valued. However, if we only consider the unassigned cells of a sub-array - let us call the collection of those cells the **assignee** of the sub-array - we get a different picture. The previous paragraph implies that `EMPTY`-valued cells are quickly being identified. What remains in the assignee are the `FULL`-valued cells only.

So we can conclude: From the chunk property follows that, in advanced iterations, many assignees are completely filled with `FULL`-values. Such assignees represent the area in which the object **is** located. The respective depth of the object for this assignee must then be equal to the number of unassigned cells. Thus, from finding a value of maximal depth in the input, we can deduct that, in the corresponding sub-array, all unassigned cells must be `FULL`-valued.

Listing 2 shows the code implementing `compare_and_fill`, the function which fills all cells whose state can be derived logically by applying the chunk property. Its parameters are

- `sensor_data_point`, a single depth-value from the input
- `arr`, the corresponding sub-array of the matrix.

The function does exactly what has been described above: if `sensor_data_point` equals zero, it assigns all unassigned values of `arr` to `EMPTY`. If `sensor_data_point` equals the number of unassigned values of `arr`, it assigns all those values to `FULL`.

As any cell belongs to exactly four sub-arrays (one for each direction), on assignment of `FULL` to one cell, each input-value for all of those four sub-arrays needs to be updated (see Section 3.1). This is what the function call `update_sensor_data` in line 13 does.

3.4 Repeat until we're stuck

To solve the problem, all we have to do now is applying `compare_and_fill` to all pairs of depths and sub-arrays iteratively

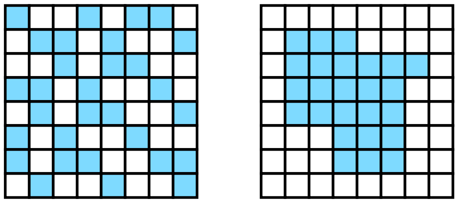

Figure 3: Chunk property examples. The left matrix does not fulfill the chunk property. The right matrix does.

```
def compare_and_fill(sensor_data_point, arr):
    n_of_unassigned = n_of_unassigned(arr)

    if sensor_data_point == 0:
        for cell in arr:
            if cell == UNASSIGNED:
                cell = EMPTY

    elif sensor_data_point == n_of_unassigned:
        for cell in arr:
            if cell == UNASSIGNED:
                cell = FULL
                update_sensor_data(cell)
```

Listing 2: Using the chunk property

until we find a solution, see Listing 3. But how do we know whether a valid solution has been found? Consider the call to `update_sensor_data` in Listing 2. With this call, all relevant input values are being reduced by one after an assignment of `FULL` to a cell. Thus, when a valid solution has been found, the entire input has to be zero. This is the exact condition we need to check in order to find a valid assignment. If, at one point, all cell-entries have been assigned to either `FULL` or `EMPTY`, and simultaneously, not all inputs are zero, then the assignment that has been found is invalid.

However, by simply calling `compare_and_fill` repeatedly, we are not guaranteed to find a solution. This is due to the fact that `compare_and_fill` does not guarantee to fill out all cells. At some point during the iteration, we may get stuck.

This is where we introduce back our exhaustive search approach. Should we, at some point during the execution of `fill_loop()`, get stuck (i.e. no value has been altered during one iteration), we assign one cell of value `UNASSIGNED` by force, and then continue the loop. Listing 4 shows the relevant code: if, at some point during the execution of `fill_loop()`, the matrix does not change, and there are still `UNASSIGNED` cells left, we sequentially assign both values, `EMPTY` and `FULL`, to those cells.

Notice line 12 of Listing 4: As soon as we have to rely on exhaustive search, we are not guaranteed that a valid solution is unique. Thus, we have to

1. Search among all possible assignments of `UNASSIGNED` variables, and
2. Keep track how many solutions have been found. In Listing 4, this is done with the `int`-valued variable `solutions_found`.

We do not accept multiple solutions, which is why we immediately exit the program as soon as two solutions have been found.

```
def fill_loop():
  while(not is_done()):
    diag_lr = get_diagonal_lr(matrix)
    diag_rl = get_diagonal_rl(matrix)

    for i in range(height):
      compare_and_fill(in_horiz[i], matrix[i,:])
    for i in range(height + width - 1):
      compare_and_fill(in_diag_lr[i], diag_lr[i])
    for i in range(width):
      compare_and_fill(in_vert[i], matrix[:,i])
    for i in range(height + width - 1):
      compare_and_fill(in_diag_rl[i], diag_rl[i])

  search_if_stuck()
```

Listing 3: Applying `compare_and_fill`

```
1  def search_if_stuck():
2    if not has_change_occurred:
3      # unassigned cells
4      unassigned_cells = matrix[UNASSIGNED]
5
6      for cell in unassigned_cells:
7        for assignment in [EMPTY, FULL]:
8          stack.push(matrix, input) # save
9          matrix[idx] = value # assign by force
10         fill_loop()
11         matrix, input = stack.pop() # load
12         if solutions_found > 1:
13           # solution is ambiguous -> leave loop
14           return
```

Listing 4: Exhaustive Search

4 THE CHUNK PROPERTY'S RELEVANCE

We have seen in Section 3.4 that we need to resort to exhaustive search for some inputs. Our naive approach has a worst-case time-complexity of $\mathcal{O}(2^{m\times n})$. To research the quality of our algorithm, we want to quantify the fraction of inputs that can be solved in sub-exponential time, meaning, without relying on exhaustive search at such extense that the runtime exceeds a certain threshold.

We approached this question experimentally. This chapter describes this experiment's setup and presents and discusses its results.

4.1 Setup

1. Modify the scanner-algorithm to terminate if it has not found a solution after $T_{\max}$ seconds.
2. Generate $N = 1000$ inputs that satisfy the chunk property.
3. Apply the scanner-algorithm to each input and count the number of terminations.
4. Repeat step 2 and 3 with variating values for the parameter `chance` in `generate_chunk` to find the worst-case result.

```python
def generate_chunk(chance, height, width):
  chunk_matrix = all_empty(height, width)

  # make middle element 1
  chunk_matrix[int(height/2), int(width/2)] = 1

  for _ in range(min(height, width)):
    for cell in chunk_matrix:
      if one_neighbor_of(cell) == FULL and
         random.random() < chance:
        cell = FULL

  matrix_to_data(chunk_matrix, height, width)
```

Listing 5: Generating chunk data

For point 1, the exact value of $T_{\max}$ depends on the machine that is being used. We found most inputs to be solvable in $\sim 0.07s$. We thus chose $T_{\max} = 0.1s$ as an appropriate threshold value.

To generate an input, we developed the function `generate_chunk(chance, height, width)`, see Listing 5. The function creates an `EMPTY`-valued matrix of dimension $(\text{height} \times \text{width})$ and assigns `FULL` to the central cell. Next, it iterates over every cell and assigns `FULL` to a cell with a probability of chance $\in [0, 1]$. This is repeated exactly `min(height, width)` times.

We repeat the experiment for various values of chance $\in [0.15, 0.16, ..., 0.25]$ to find the worst-case result. We furthermore chose `height = 10` and `width = 15`, as those are the values used in the original problem description.

4.2 Results

The results are listed in Table 1 and plotted in Figure 4. The timeout-rate increases with the chance until a maximum between 0.175% and 0.25%, and decreases thereafter. The highest measured timeout-rate is 1.4% for chance $= 0.22\%$.

4.3 Interpretation

Our experiment shows that the timeout-rate t does not exceed $t = 1.5\%$. The measured peak is expected to be the global maximum, as generated matrices with chance $< 0.1\%$ are almost entirely empty, while matrices with chance $> 0.3\%$ are almost entirely full. Such matrices do not require searching, as `compare_and_fill` fills out almost all sub-arrays in the first iteration. From this we can safely conclude that $\sim 98\%$ of inputs can be solved in sub-exponential time.

5 ACHIEVEMENTS AND FUTURE WORK

We identified the chunk property in the inputs of the scanner problem. We were able to show that the search space for 98% of inputs can be reduced dramatically by exploiting this property. Especially the inputs featured in the original problem description do not rely on any search at all. This may justify the provocative title of this paper.

The search algorithm we used is a simple exhaustive search. We put all work into reducing the search space such that exhaustive search has to be used as few times as possible. To further improve the performance for every possible input, future work may focus on finding more efficient search algorithms. One possibility that has been tried during our research is simulated annealing, a local search algorithm which is guaranteed to find the global optimum, given enough time. However, a thorough implementation was beyond the scope of this research project.

Table 1: Experiment Results

chance in %	timeout-rate in %	chance in %	timeout-rate in %
0.10	0.2	0.21	1.0
0.11	0.2	0.22	1.4
0.12	0.0	0.23	0.9
0.13	0.3	0.24	1.1
0.14	0.7	0.25	0.5
0.15	1.0	0.26	0.4
0.16	0.9	0.27	0.7
0.17	1.2	0.28	0.1
0.18	0.8	0.29	0.2
0.19	1.1	0.30	0.1
0.20	1.2		

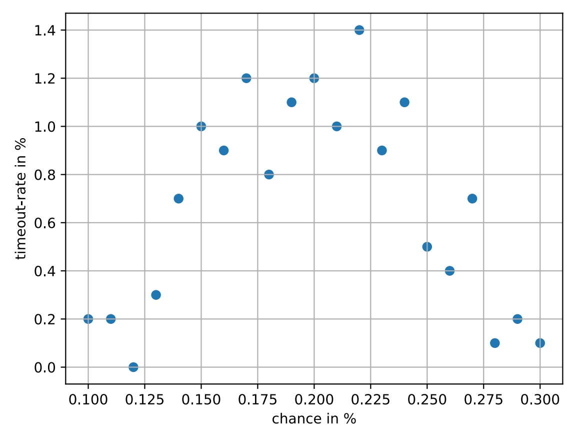

Figure 4: Experiment Results Scatterplot

REFERENCES

[1] https://db.cs.uni-tuebingen.de/staticfiles/ACM-problems/Scanner.pdf.
[2] https://github.com/Vogelmensch/scanner.git.