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 $\sim 98.5\%$ 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 participants' 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 of `FULL` and `EMPTY` for the 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 features in the original problem description).
2. Local Search may find solutions, but cannot identify invalid inputs.

We cannot eliminate the need for search entirely (see Section 3.1). Still, we can view the problem at 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 assign large groups of cells at once with absolute certainty (see Section 3.2).
- Some inputs may have zero or multiple solutions. We found two conditions that allow for handling those kind of inputs early (see Section 3.3).
- We performed an exeriment 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, i.e. where it stuck at local search for too long. The results show that we can solve $\sim 98.5\%$ 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 better search algorithms is still required.

2 ENCODING A 3D BODY

To understand the scanner-algorithm, we must first understand the semantics of the code we are solving. 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: Along the vertical axis, the body is divided into a finite set of two-dimensional slices. Each slice is viewed as constant in depth along the vertical axis, i.e. having a discrete vertical depth of one. 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 two: 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 three: 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 with h, $h + w + 1$, w, and $h + w + 1$ entries respectively. For our example, the resulting arrays are shown in Listing 1. Those four arrays make up the encoded slice.

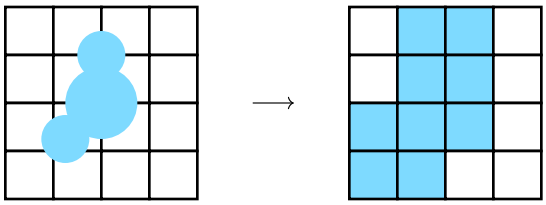

Figure 1: Discretization of an object

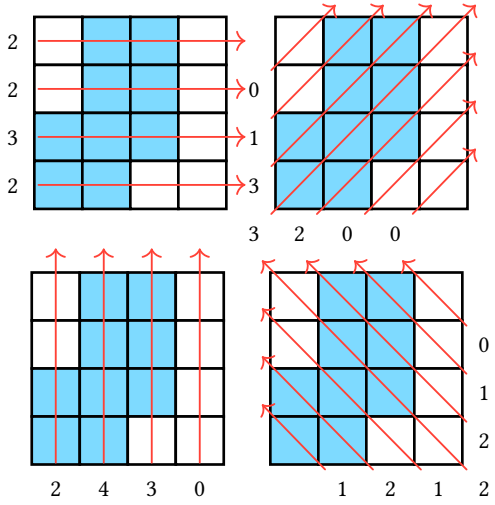

Figure 2: Encoding the object by scanning its depth

```
[2, 2, 3, 2], # horizontal
[0, 1, 3, 3, 2, 0, 0], #left-right-diagonals
[2, 4, 3, 0], # vertical
[1, 2, 1, 2, 2, 1, 0] # right-left-diagonals
```

Listing 1: Arrays encoding the slice

3 RECONSTRUCTING A SLICE

We are given two integer-arrays of lengths m and n, and two integer-arrays of lengths $m + n + 1$, all representing the depth of the object in the four possible directions. We want to reconstruct the discretized image from this data only. In this chapter, we explain the algorithmic approach we found to be most effective.

<say what is being done in this chapter lol. Like that we build the solution bottom up.

- Introduce terminology: Matrix, Cells-Values `FULL`, `EMPTY` and `UNASSIGNED`
- Dimensionality of the matrix from the input length

>

3.1 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 (0 and 1). 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 later.

3.2 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 goal, we exploit a property our resulting matrices have. 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. What we do not expect, for example, is a noisy image, where the value of a cell is decided randomly.

From the chunk property follows that some sub-arrays (verticals, horizontals or diagonals) of the matrix may be completely filled with `EMPTY`-values. The respective depth for this sub-array must then be zero. Searching for zeros in the input thus leads to complete knowledge of all values in the respective sub-array.

On the other hand, from the chunk property follows that some sub-arrays may be completely filled with `FULL`-values. The respective depth for this sub-array must then be equal to the length of the sub-array. Searching for values of maximal depth in the input thus leads to complete knowledge of all values in the respective sub-array as well.

Listing 2 shows the code implementing `compare_and_fill`, the function which fills all cells whose state we can derive logically by 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 depth-value for all of those four sub-arrays need to be updated. This is what the function call `update_sensor_data` in line 13 does.

```
1  def compare_and_fill(sensor_data_point, arr):
2      n_of_unassigned = n_of_unassigned(arr)
3
4      if sensor_data_point == 0:
5          for cell in arr:
6              if cell == UNASSIGNED:
7                  cell = EMPTY
8
9      elif sensor_data_point == n_of_unassigned:
10         for cell in arr:
11             if cell == UNASSIGNED:
12                 cell = FULL
13                 update_sensor_data(cell)
```

Listing 2: Using the chunk property

3.3 Iterate until we're done

How do we know whether we found a valid solution? Consider the call to `update_sensor_data` in Listing 2. With this call, all relevant input values are being updated 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 `FULL` or `EMPTY`, and simultaneously, not all inputs are zero, then the assignment that has been found is invalid.

To solve the problem, all we have to do now is applying `compare_and_fill` to all pairs of depths and sub-arrays iteratively until we found a solution, see Listing 3. However, we are not guaranteed to find a solution just yet. 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 Listing 3, 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 Listing 3, the matrix does not change, and there are still `UNASSIGNED` cells left, we assign both values, `EMPTY` and `FULL`, to this cell sequentially. Notice line 10 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.

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

```
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])
```

Listing 3: Applying `compare_and_fill`

```
1  # ... inside fill_loop()
2  if not has_change_occured:
3    # indices of unassigned cells
4    indices_of_unassigned = np.argwhere(matrix.cell
   == UNASSIGNED)
5
6    for idx in indices_of_unassigned:
7      for assignment in [EMPTY, FULL]:
8        # assign value to matrix[idx]
9        search_in_branch(idx, value, matrix)
10       if solutions_found > 1:
11         # the solution is ambiguous -> leave loop
12         return
```

Listing 4: Exhaustive Search

4 ANALYSIS

We have seen in Chapter TODO that we need to resort to exhaustive search algorithms for some inputs. Our naive approach has a worst-case time-complexity of $\mathcal{O}(2^{m\times n})$ for obvious reasons. To research the quality of our algorithm, we want to quantify the fraction of all inputs that can be solved in sub-exponential time, meaning, without having to resort to exhaustive search.

We approached this question experimentally. This chapter describes this experiment (TODO: genauer wenn Kapitel fertig)

4.1 Setup

1. Modify `scanner.py` to terminate if it has not found a solution after $T_{\max}$ seconds.
2. Generate $N = 1000$ inputs that satisfy the chunk-property using Listing 5.
3. Apply `scanner.py` to each input and count the number of terminations.
4. Repeat step 2 and 3 with variating values for `chance` in Listing 5 to find the worst-case result.

For point 1, the exact value of $T_{\max}$ depends on the machine that is being used. We have 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 `generate_chunk(chance, height, width)`, see Listing 5. The function iterates over every cell and turns it to a `FULL` cell with a probability of `chance`. This is repeated `min(height, width)` times. The value of `chance` is variable, as we repeat the experiment for various values of chance $\in [0.15, 0.16, \ldots, 0.25]$ 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.

```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

4.2 Results

Table 1:

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

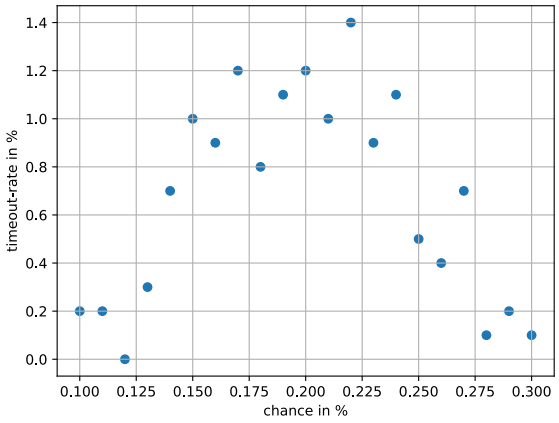

Figure 3:

4.3 Interpretation

Our experiment shows that the timeout-rate t does not exceed $t = 1.5\%$. From this we can conclude that $\sim 98.5\%$ of inputs can be solved in sub-exponential time.

5 FUTURE WORK

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 better search algorithms. One possibility that has been tried during our research is simulated annealing. However, a thorough implementation was beyond the scope of this research project.