# Scanner-Paper

A subtitle is purely optional

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

## ABSTRACT

Start with a structured abstract as a tiny text. That should **not** become your final version of an abstract without alteration.

An abstract (for a short paper like yours) should comprise about 100-150 words. Please, write a minimum of 80 and a maximum of 200 words.

## KEYWORDS

Keywords, To, Increase, Discoverability

## 1 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. 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 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`.

Step three: The grid of cells is now being measured for its depth along four directions:

- 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.

Those four arrays make up the encoded slice.

## 2 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

>

### 2.1 Naive Local 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 local search into our solution to guarantee completeness, as we will see later.

### 2.2 Exploiting the chunk property

Our goal is to reduce the search space of local search 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 1 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 1: Using the chunk property**

## 2.3 Iterate until we're done

How do we know whether we found a valid solution? Consider the call to `update_sensor_data` in Listing 1. 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 2. 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 local search approach. Should we, at some point during the execution of Listing 2, 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 3 shows the relevant code: if, at some point during the execution of Listing 2, 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 3: As soon as we have to rely on local 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 2: 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 3: Local Search**

# 3 ANALYSIS

# 4 FURTHER STUFF

- local search algorithms: maybe faster, but cannot find double solutions or no solution