# Machinarium Solver

Hendrik Walter

August 22, 2019

## Contents

# 1 Overview

The problem at hand is to move a token around a square board, divided into fields each of same length, similar to a chess board. However, some of the fields can be preoccupied by obstacles. The token in question needs to be visit every single, unoccupied field without revisiting any. Which in itself does not constitutes a difficult problem, an extra condition makes it significantly harder to solve: A token can only change direction (horizontal to vertical or vice versa), if it hits the boundary of the field, an obstacle or a field already visited in an earlier step. The token can start at any location on the board.

An example is given below. The upper solutions shows a board without an obstacle. The token can move freely around. The lower example shows a board with multiple obstacles and one can see how the token moves between the different obstacles and still manages to find a way to all fields.

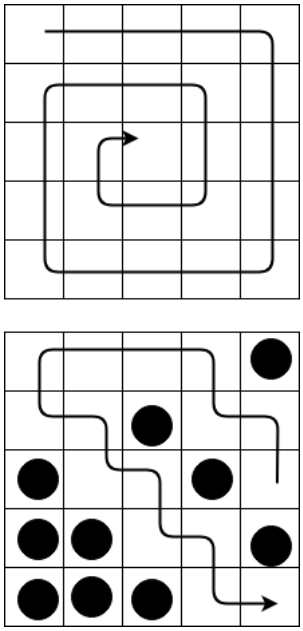

Figure 1: Model 2

This problem is a planning problem, which a known set of steps. As no field can be revisited, the number of steps is equal to the unoccupied fields on the board. A board is unsolvable if this number is not reached or exceeded.

# 2 Basic Model: Variable and Domains

Encoding this model poses a challenge, as the intuitive 2D way humans think about the problem is hard to encode in a machine due to the multiple dimensions to think of. It becomes easier to convert the 2D board into a 1D matrix, such that for a $p * p$ board, every $p$-th entry is the first field of a new row. The size $p$ and the initial grid is given to the program. To describe the problem, I have

defined another three variables and let the program search for another three, making it six in total. The ones defined at the start of the program are

- **n**: n is equal to the square of p and describes to number of total fields in the grid. This is used for further calculation and to define the domain of the search matrix.
- **STEPS**: STEPS is the number of steps necessary to traverse the field. It is equal to n minus the number of obstacles in the array. To obtain this, one can use the command *flatten()* and convert the initial 2D grid into a 1D matrix and the use the command *sum* to count the number of obstacles. This only works due to the fact that the obstacles are defined as 1 and the other fields as 0. STEPS is required to define the size of the search matrix at a later stage.
- **initial_array**: This is the 1D representation of the 2D grid read into the program. This is required to easily check between the search matrix and the initial grid if the space is occupied by an obstacle or not. The process is visualized in Figure 2.

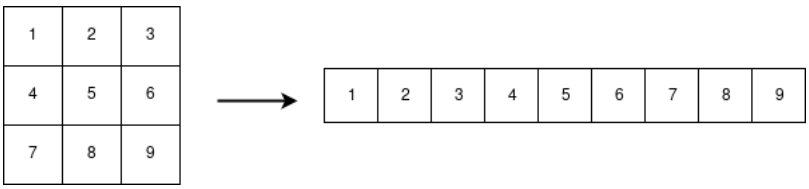


Figure 2: Flattening a 3x3 2D Matrix, left 2D, right 1D representation

Beyond these defined variables, the search is focused on the following three variables:

- **stepMatrix**: This variable constitutes the core of the program and will ultimately contain the solution. The stepMatrix is named as such as it will contain the numbers of the board in which they were visited. This means that the first slot in this one dimensional matrix will contain the number of the starting slot, the second slot will contain the number of the the second step and so forth. Hence, the total length of this matrix is equal to the number of steps, here the previously defined variable STEPS comes into play to define the index. The domain on the other hand is equal to the range between previously defined n and 0, as every number of the board could potentially come up, but 0 serves as a flag for an obstacle. Note that this will never actually occur in a solution matrix as we will see later. An example of this matrix for an empty board is given in Figure 3, note how the length changes in Figure 4. Here, there will not be a one in the final stepMatrix as it is occupied by an obstacle.
- **Direction**: This matrix seems more difficult to comprehend, but is an essential auxiliary variable in the process of finding a solution to all kinds of boards. It is easier to comprehend this matrix starting with the domain, which is (0,1). This matrix describes the change in the fields, which can

either be horizontal (1) or vertical (0). Considering that there are a total of STEPS fields to be visited, there will only be STEPS-1 transitions, hence there will only be STEPS-1 directions involved, which explains the indexing. In figure 4 there is an example given. Note how the direction matrix sits between the different steps, as it describes the transition between them.

- **Movement**: In close conjuncture with direction, the movement matrix helps to fully comprehend the token on the board. With the stepMatrix the field is defined, with the direction horizontal and vertical can be distinguished and with movement it can be decided if the token moved into a positive or negative direction, hence the domain of (-1,1). As there are as many positive or negative movements as directions given, there will also be STEPS-1 entries in this matrix. Please look at Figure 4 to see an example how a movement matrix is placed in the solution.

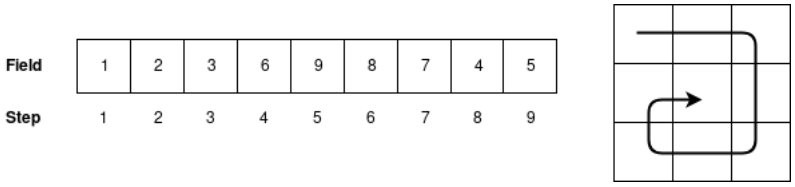


Figure 3: Possible solution for a clear 3x3 Matrix

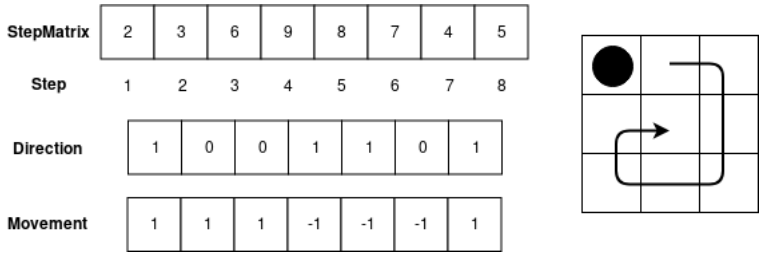


Figure 4: Possible solution for a 3x3 Matrix with one obstacle

## 2.1 Movement and Direction as different variables

Dividing the movement and direction matrix into two different matrices, seems like an additional effort to maintain consistency between the variables. However, splitting these up provides improved usability for the workflow at a later stage. Instead of having one matrix with a domain of size four (to correctly determine all possible directions of movement), one would need to use some equivalent of a switch statement to check on the direction of movements. In this case however, direction can serve as flag, if the difference is $p$ or 1 and then simply multiplying this value with the appropriate value in the movement matrix gives the right transition.

# 3 Basic Model: Constraints

The constraints in my program were developed from the constraints given in the description and follow naturally.

1. *The token can only move into adjacent fields.* This is enforced by the constrains below. The token moves vertically (change of magnitude p) or horizontally (change of magnitude 1). The direction doesn't matter at this stage.

```
stepMatrix[i+1] = stepMatrix[i]+p \/
stepMatrix[i+1] = stepMatrix[i]-p \/
stepMatrix[i+1] = stepMatrix[i]+1 \/
stepMatrix[i+1] = stepMatrix[i]-1,
```

   This brought up the issue that at the end or at the beginning of a row, the token should not move into a field numbered one less or more, as this would mean a jump across the board in 2 dimensions. This created the necessity for the constrains below. If the entry of i is divisible by p, it is the last entry of the row and the change to the next one can't be one. If the entry has remainder 1 if divided by p, the next entry can't be one less. This is the first entry in a row.

```
stepMatrix[i]%p=0 ->
        (stepMatrix[i+1] - stepMatrix[i])!=1 /\
stepMatrix[i]%p=1 ->
        (stepMatrix[i+1] - stepMatrix[i])!=-1
```

2. *A zero represents an obstacle* This can be enforced by a if and only if - structure as seen below:

```
forAll i : int(1..STEPS) .
        initial_array[stepMatrix[i]] = 1 <->
            stepMatrix[i]=0,
```

3. *The direction matrix should represent the direction the token moves..* This is an auxiliary constraint and is enforced straightforward in the following manner by checking the absolute value of the differences in the stepMatrix:

```
forAll i : int(1..STEPS-1) .
        |stepMatrix[i+1]- stepMatrix[i]|=p ->
                direction[i] = 0,
forAll i : int(1..STEPS-1) .
        |stepMatrix[i+1]- stepMatrix[i]|=1 ->
                direction[i] = 1,
```

4. *The movement matrix represents the positive or negative change in direction.* This is an auxiliary constraint as well and is implemented by checking if the difference is positive or negative between two fields in the stepMatrix:

```
forAll i : int(1..STEPS-1) .
        stepMatrix[i+1]- stepMatrix[i]>0 ->
            movement[i] = 1,
forAll i : int(1..STEPS-1) .
        stepMatrix[i+1]- stepMatrix[i]<0 ->
            movement[i] = -1,
```

5. *If there is a change in direction, there must be a valid reason which is one of (a) obstacle (b) the edge of the board or (c) the field has already been visited.* This constraint seems like the most difficult to implement, but is also crucial as it determines that the token is moving in a straight line until hitting an obstacle. This is the constraint which makes this riddle harder, as mentioned in the introduction. As this analyses the change on direction between different steps, it is the difference in the direction matrix which becomes of significance. In this way, two transitions can be considered as each direction describes the change of one. If the difference in direction is 0, there is no change of direction and the case is of no significance for this constraint. If the difference is -1, there was a change from horizontal to vertical and vise versa for -1. Please note how the movement matrix can serve as a flag for the direction, which simplifies the condition. As this takes the difference between the directions, it is required to start from 2 instead of 1.

```
forAll i : int(2..STEPS-1) .
      direction[i]-direction[i-1] = -1 ->
      $(a)
      initial_array[stepMatrix[i]+movement[i-1]*1]=1
      \/
      $(b)
      stepMatrix[i]%p=0 \/
      (stepMatrix[i]-1)%p=0
      \/
      $(c)
      (exists j : int(1..i) .
      stepMatrix[j]=
       stepMatrix[i]+movement[i-1]*1), $(c)

forAll i : int(2..STEPS-1) .
      direction[i]-direction[i-1] = 1 ->
      $(a)
      initial_array[stepMatrix[i]+movement[i-1]*p]=1
      \/
      $(b)
      stepMatrix[i]+p>n \/
      stepMatrix[i]-p<1
      \/
      $(c)
      (exists j : int(1..i) .
      stepMatrix[j]=
       stepMatrix[i]+movement[i-1]*p),
```

6. *No field can be revisited.* This can easily be constraint by the allDifferent constraint.

```
allDiff(stepMatrix)
```

All together these constraints solve any kind of board enforcing the required constraints and delivering a solution back from which the order of the fields visited can be read off.

# 4 Basic Model: Examples

In addition to the schematic examples given in section 2, this section is supposed to give actual outputs from the program and give an indicator that the code solves the problems in an orderly manner. In order to properly analyze the behaviour, the easiest is to start of with a small instance without obstacles, this will be increased step-wise adding new constraints to the test environment. For clarity the output of the program will only be shown as a whole for the 2x2 case as there are too many solutions to bigger boards to present them in a structured way.

## 4.1 2x2 grid

This section is supposed to show that the solver produces a solution in which all steps adjacent and the obstacle is avoided. A good initial grid for this purpose is saved in the file *instance2x2_b.param*.

The grid is shown below as well as the two solutions, a human could identify on this grid.

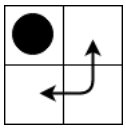

Figure 5: Two possible solutions for 2x2 Matrix with one obstacle

The output of the program is below and it is possible to see that exactly the same result was produced.

**Solution 1**

```
letting direction be [0, 1]
letting movement be [1, -1]
letting stepMatrix be [2, 4, 3]
```

**Solution 2**

```
letting direction be [1, 0]
letting movement be [1, -1]
letting stepMatrix be [3, 4, 2]
```

**Conclusion:** the model can produce a solutions in which the steps are adjacent, obstacles are avoided and the token stays within the range of the board.

## 4.2 3x3 grid

To strengthen the confidence in the implemented model, it needs to be checked that the token moves in a straight line until it hits field already visited or an obstacle. This requires a larger matrix than 2x2, as with every step the token is already on the edge of the board.

The matrix saved in *instance3x3_a.param* provides such conditions. A sample solution can be found below, note that this is the same as presented earlier in section 2.

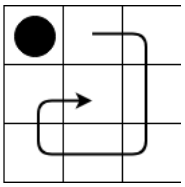

Figure 6: A possible solution for 3x3 Matrix with one obstacle

The solution can be found below as an output of the program:

```
letting direction be [1, 0, 0, 1, 1, 0, 1]
letting movement be [1, 1, 1, -1, -1, -1, 1]
letting stepMatrix be [2, 3, 6, 9, 8, 7, 4, 5]
```

Note that none of the ten solutions found by the solver break any of the conclusions made in this section. In none of the solutions early turns were taken or other constraints broken.

**Conclusion**: The model can produce a solutions in which in addition to the previous conditions a straight line is followed until the token hits the edge or a field already visits or an obstacle.

## 4.3 4x4 grid

A final example will be a 4x4 grid with multiple obstacles to show that the token can also follow more complex patterns. As such the grid in *instance4x4_b.param* was chosen with one possible solution plotted on top.

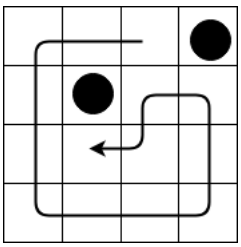

Figure 7: A possible solution for 4x4 Matrix with two obstacles

The solution can be found below as an output of the program:

```
letting direction be
    [1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1]
```

```
letting movement be
    [-1, -1, 1, 1, 1, 1, 1, 1, -1, -1, -1, 1, -1]
letting stepMatrix be
    [3, 2, 1, 5, 9, 13, 14, 15, 16, 12, 8, 7, 11, 10]
```

**Conclusion**: The model can produce a solutions in which in addition to the previous conditions also can avoid multiple obstacles in a complex pattern which require each of the implemented conditions at least once.

# 5 Basic Model: Empirical Evaluation

As required the empirical results are listed for all given examples. This include Solver Node, Solver Solve Time and Savile Row Total Time. Further, the information whether the board was solvable or not has been given.

| | Solver Nodes | Solver Solve Time | Savile Row Total Time | Solvable |
|---|---|---|---|---|
| 2x2 a | 3 | 0.043585 | 0.178 | y |
| 2x2 b | 2 | 0.035464 | 0.163 | y |
| 2x2 c | 0 | 5e-06 | 0.129 | n |
| 3x3 a | 7 | 0.04417 | 0.304 | y |
| 3x3 b | 4 | 0.041217 | 0.315 | y |
| 3x3 c | 0 | 1.5e-05 | 0.266 | n |
| 4x4 a | 181 | 0.048244 | 0.422 | y |
| 4x4 b | 110 | 0.052697 | 0.441 | y |
| 4x4 c | 754 | 0.029015 | 0.401 | n |
| 5x5 a | 7416 | 0.258517 | 0.705 | y |
| 5x5 b | 7600 | 0.245845 | 0.692 | n |
| 6x6 a | 1952 | 0.109324 | 1.041 | y |
| 6x6 b | 3606 | 0.297835 | 1.354 | y |
| 6x6 c | 8873 | 0.430194 | 1.06 | y |
| 6x6 d | 6623 | 0.245413 | 0.813 | y |
| 7x7 a | 2081 | 0.122124 | 1.579 | y |
| 7x7 b | 2047 | 0.140044 | 1.66 | y |
| 7x7 c | 65564 | 3.53965 | 1.309 | y |
| 7x7 d | 85078 | 4.1047 | 1.316 | y |
| 7x7 e | 94249 | 4.36519 | 1.2 | y |
| 7x7 f | 82855 | 3.8837 | 1.374 | y |
| 8x8 a | 124082 | 8.96682 | 2.227 | y |
| 8x8 b | 110716 | 8.0519 | 2.717 | y |
| 8x8 c | 94065 | 6.86809 | 2.151 | y |

Interesting in this regard might be that even though 2x2c and 3x3c indicate that there might be zero nodes required to obtain a result for an unsolvable grid, 5x5b and 4x4c indicate otherwise. Hence, 0 in the former occurrences is just due to the small size of the board. Further, it is worth highlighting that as much as there might be a correlation between grid size and nodes used, the drop of nodes used from 5x5 problems to 6x6 problems indicate that there must be another factor as well, as the nodes used practically half.

# 6 Extensions

To fully understand the different evaluations and the model, it is important to consider the different factors which might impact the performance, they include:

1. The size of the board, $n$ with $n * n$ fields
2. The position of the obstacles, $P = coord_i =< r, c >$ (the coordinates as row or column) or alternatively $p = c + r * n$ for the one dimensional case

Something which should not be listed as a separate parameter is the "solvable", as it follows from 1, and 2. While it seems to be an easy way to categorize the puzzles, it is deemed unsuitable to create an "unsolvable 5x5" board and then check the performance associated with this. This will be done the other way round, if boards vary in size, it should be checked if both are solvable and then worked from there. Similarly, the number of obstacles doesn't need to be a separate parameter as it is simply the magnitude of $P$, $|P|$. The above three parameters are sufficient to define any board.

## 6.1 Impact of the size of the board

To measure the impact of the size of the board on the performance of the solver, an empty board is considered initially, and the size increases from 2 to 15. The number of solutions is expected to skyrocket towards the larger sizes and as such, it it sensible to also consider the size of the board with an obstacle on it, even this already reduces the solution spaces massively. The experiment only runs until it has the first solution found, which limits the search space, but also increases comparison as the internal computations will be similar so the impact of the size can be extracted.

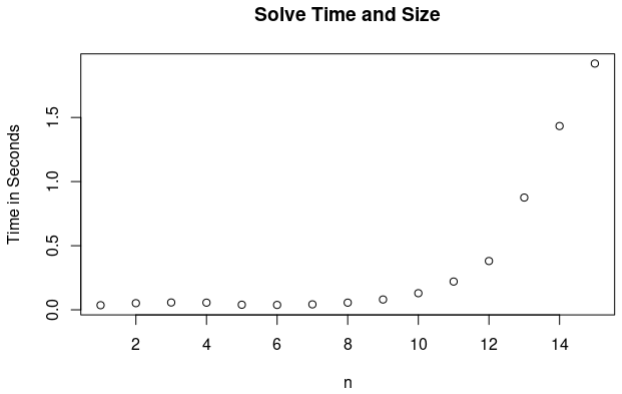


Figure 8: The time for solving an empty grid of size n

From the graph above there seems to be a strong correlation which only starts to pick up as soon as size of about 10 is exceeded. This could be due to the fact that the possible combinations starts to massively increase. This indicates that there is a very strong impact on the size of the board on solver

time, which doesn't come as a surprise as more variables need to be evaluated. Of much more interest is the observation if *-O3* optimization offers any advantage in this case. O3 offers the advantage that it enables tabulation and associative-commutative common subexpression elimination. Before looking at the results, it is worthwhile understanding these two processes. *associative-commutative common subexpression elimination* eliminates syntactically identical expressions. Note that this essentially means that if two expressions are identical after transforming one of them into the other, one of them can be removed. *Tabulation* converts constraint expressions into table constraints to improve propagation.

Using this optimization, we can see that SavileRow performs significantly better on the Solver side of things

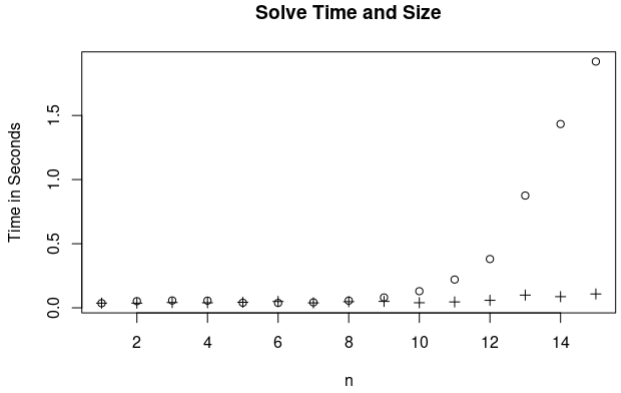


Figure 9: The time for solving an empty grid of size n, + for O3, o for O2

It is strongly noticeable that the computation time is massively decreased for the solver if *O3* is activated.

## 6.2 Impact of position of obstacles

To appropriately check the impact of the positions of obstacles, the coverage needs to include multiple obvious cases and, as shown later, also less obvious cases as some of the slow and fast performances don't follow obvious patterns. The board of chosen size has board length n=5. This has practical reasons, such as the times are long enough that differences are easy to spot, but also leaves enough room to have fields in the middle (as five is an uneven number), fields in the corner and neither of the two. Positioning the obstacles is a different issue as one not only needs to consider a single obstacle in various positions, but as soon as two tokens are included the possibilities become vast. Starting off with one token on different positions on the board.

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 |

Figure 10: Possible locations of one obstacle

For each of the fields the solver solving time is reported below:

| | Solver Solve Time | Solvable? |
|---|---|---|
| 1 | 0.0384 | y |
| 2 | 0.1677 | n |
| 3 | 0.0449 | y |
| 4 | 0.2503 | n |
| 5 | 0.0718 | y |
| 6 | 0.2378 | n |
| 7 | 0.047 | y |
| 8 | 0.228 | n |
| 9 | 0.0671 | y |
| 10 | 0.1952 | n |
| 11 | 0.0687 | y |
| 12 | 0.2291 | n |
| 13 | 0.0334 | y |
| 14 | 0.2192 | n |
| 15 | 0.0526 | y |
| 16 | 0.2334 | n |
| 17 | 0.0352 | y |
| 18 | 0.2564 | n |
| 19 | 0.042 | y |
| 20 | 0.1947 | n |
| 21 | 0.2267 | y |
| 22 | 0.2221 | n |
| 23 | 0.0453 | y |
| 24 | 0.1806 | n |
| 25 | 0.2969 | y |

The data is not well enough displayed to gain any substantial insights from this presentation. As such, there a few hypothesis around the solver, which can be proved or disproved with this data.

**Hypothesis one**: Solveable and unsolvable grids have different times.

This seems to be indeed true, as for above data, the mean time for a solvable grid is $\mu = 0.0823$, while $\mu = 0.2178$ for the case in which the grid is unsolvable. This could be due to the fact, that the solver is only looking for one solution,

which for the other case all possible solutions need to be checked. To further evaluate this all solutions would have to be found. A t.test also yields a p-value of 0.000049, which indicates that the two do not come from the same source. This results is questionable considering the point below. Even though are the solvable grids are faster in processing, the standard deviation is with $\sigma = 0.081$ more than three times the size of the unsolvable other grid with $\sigma = 0.27$, which makes the t.test questionable as the same variance is assumed.

**Hypothesis two**: The position does not have an impact on solving time.

To test this hypothesis, only the solvable nodes are taken. The following plot can be created from this:

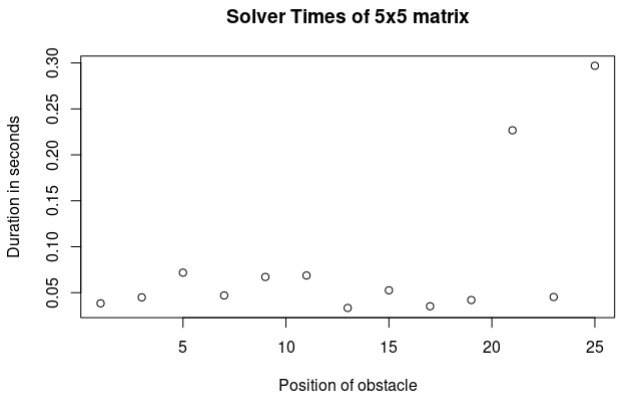


Figure 11: Times in seconds for an obstacle in the location on x

Visually, there can't be any inference made that there are unusual activities, if anything, there might be a small downwards trend for larger indices, only broken by two major outliers, 21 and 25. On another hand, there are small bumps whenever the obstacle is close to the end or beginning of a row, but also this does not give any visual indicator for a difference, especially considering the outliers for 21 and 25.

## 6.3 Other considerations for more obstacles

The case for two or more tokens needs some form of measuring the distribution of the tokens on the board. Simply listing the coordinates won't offer a sufficient insight. The measurements will need to be based on some measurement of dispersion of the obstacles and hence there need to be a measurement of "hardness".