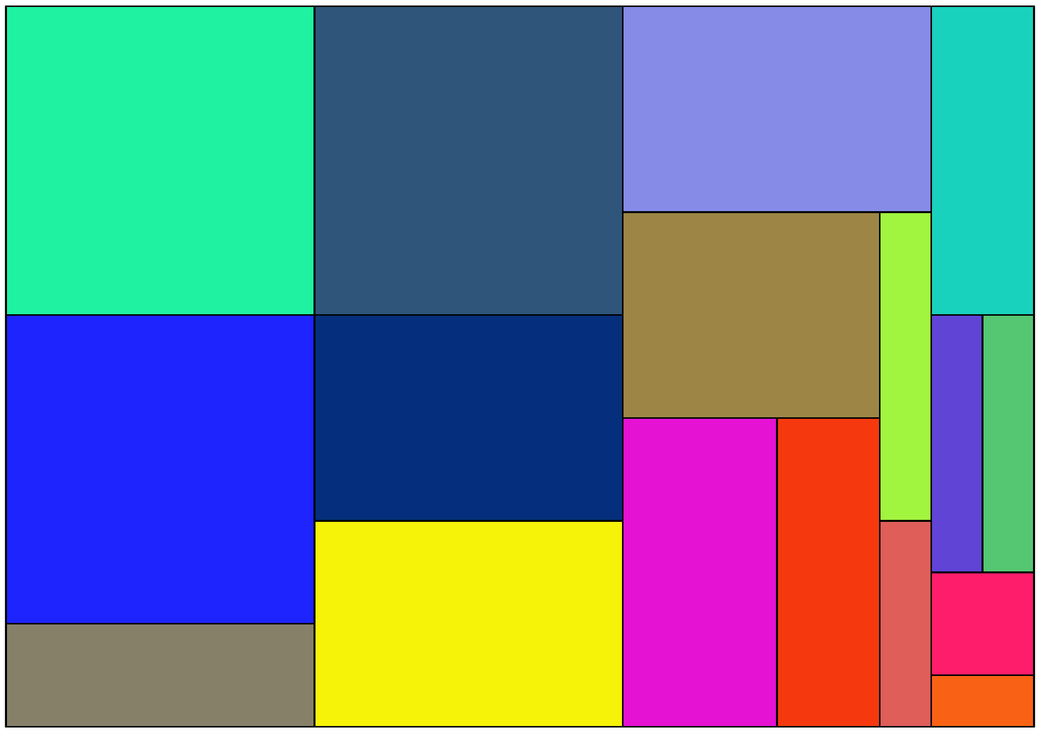
The Tiling Problem

Report of the use of heuristics on algorithms for solving the tiling problem

Vrije Universiteit Amsterdam
05/02/2017



Heuristics Group 5
Roos Bakker - 2554469
Laura Ham - 2556564
Pavan Shahani – 2527281

Table of Contents

Introduction

Algorithms can be useful to solve or help solve many kinds of problems. Algorithms can be implemented/programmed in an environment and perform calculations to solve a problem. However, algorithms on their own are often not sufficient to solve a problem due to the complexity and/or context of a problem. Therefore, heuristics are needed to aid algorithms in acquiring a solution. Heuristics do not aim to optimize an algorithm, but rather satisfice[1]. Heuristics can help an algorithm to find a ‘good-enough’ solution if a solution cannot be found or possibly allow the algorithm to find a solution faster.
In this report a research is conducted on the tiling problem and will be discussed. The tiling problem consists of a given set of fields, with each their own specific set of tiles, where each field needs to be solved by placing each tile on the field in such way that the whole field is covered and there is no overlap amongst the tiles. For each field, there is at least one solution. Given for this assignment are 125 tile set configurations. This is a total of 25 configurations of sets of approximately 15 to 55 tiles, with increments of 10 tiles. This report will focus on implementing a chosen type of algorithm and research on the impacts of heuristics on the algorithm's capabilities on solving the tiling problem.

Research Question

To stay in line with the heuristics aspect of this research, we will design and use a single algorithm and use heuristics to improve this algorithm. The algorithm will be a constructive algorithm and based off existing algorithms. Since the goal of the algorithm is to solve all the fields, our research focus will be on finding the right set of heuristics and the implementation of these heuristics on the algorithm. Therefore, the research question can be formulated as: ‘*Can we improve a constructive algorithm for solving the tiling problem by implementing different heuristics?*’

Hypothesis

Generally speaking, the use of heuristics is supposed to improve the efficiency of an algorithm[1]. So for us it is interesting to see if this also the case for an algorithm which we will program for the tiling problem. The hypothesis is that heuristics will improve the algorithm’s solving efficiency. This implies that the answer to the research question is expected to be positive.

Related Work

The tiling problem is an example of the exact cover problem, which is an NP-complete decision problem to determine if an exact cover exists. The goal of such problems is to satisfy all constraints exactly once. An approach to solve the exact cover problem is Donald Knuth’s algorithm X[2], which is a recursive, non-deterministic, depth-first, backtracking algorithm. The constructive algorithm, which forms the basis for testing our heuristics, is based on algorithm X. The constructive algorithm will however be deterministic and uses hill-climbing instead of depth-first search.

Algorithm description

The constructive algorithm that we will use is a recursive hill-climbing algorithm, which uses backtracking when no solution is found at a final climbing state. This algorithm is deterministic, and will always find the same result when running on a specific field. This means the algorithm is correct, but incomplete. Different heuristics will be implemented in a basic constructive tiling algorithm, and the differences in effectiveness or improvements will be analysed.

The constructive method works by extending an empty solution by adding tiles on the field where possible. If the field is not solved, but not all the tiles are placed, the algorithm will use backtracking. It will restructure tiles from the current state towards the beginning state.

Experimental Setup

Algorithm Version 1

This version of the algorithm is based on the basic constructive algorithm. There are two heuristics used to create this algorithm: tile ordering based on surfaces and positioning in the field. The biggest tiles are places first, the smaller ones at the end. The tiles will be placed from top to bottom, from left to right.

Algorithm Version 2

The second version of the algorithm is an improvement of the previous version and is built on top of the first version. There is one added heuristic, the tile placing is determined by scores of different possible places for a tile. Scores are calculated for every possible coordinate position for a tile to be placed. The score of a possible place for a tile is the number of coordinates of the border of the tile that are adjacent to another tile or the border of the field. The higher the number of adjacent coordinates of a tile on a certain place, the better the position and the higher the score will be. A score is calculated for every possible place for a tile and the tile will be placed on the place with the highest score. All possible positions of tiles are kept in memory, even when a tile is placed. When the algorithm starts backtracking, it will move tiles to the next best position.

Algorithm Version 3

The last version of the constructive algorithm in this experiment is built on top of algorithm version 2. The improvement in this version is that the algorithm now determines isolated white spaces. Isolated white spaces are small, empty areas of the field surrounded by tiles and/or borders of the field. Small isolated white spaces can be problematic if they are created in the beginning of the filling of a field. If these white spaces cannot be filled by tiles, the algorithm will not come to a solution and will need a lot of backtracking before finding a solution. Therefore, a white space checker has been included in the third version of the algorithm. Theoretically, this will save a lot

of time and minimize recursions. When a tile is placed, according to the heuristics of the first two versions, the algorithm will check if there are any white spaces. The biggest white space is not considered, as this stands for the rest of the field which will be filled using the regular recursive function. If smaller white spaces are found by the checker, these will be filled with tiles, if possible. If all the white spaces are filled successfully, the algorithm will continue by placing tiles in the right order on the rest of the field. If not all white spaces have been filled, the algorithm will backtrack the steps to fill the empty spaces, as well as the tile which created the white spaces as this will never lead to a correct solution.
This algorithm includes the implementation of a white space parameter. This is a percentage parameter which will be set so that if a certain percentage of field is filled with tiles, the algorithm will not attempt to determine isolated white spaces anymore. The idea behind this heuristic is to allow the algorithm to speed things up and not require to make unnecessary calculations for white spaces that are small and might not even fit a tile that is yet to be placed.

The flowchart in figure 1 visualizes the simplified procedure of algorithm version 3.

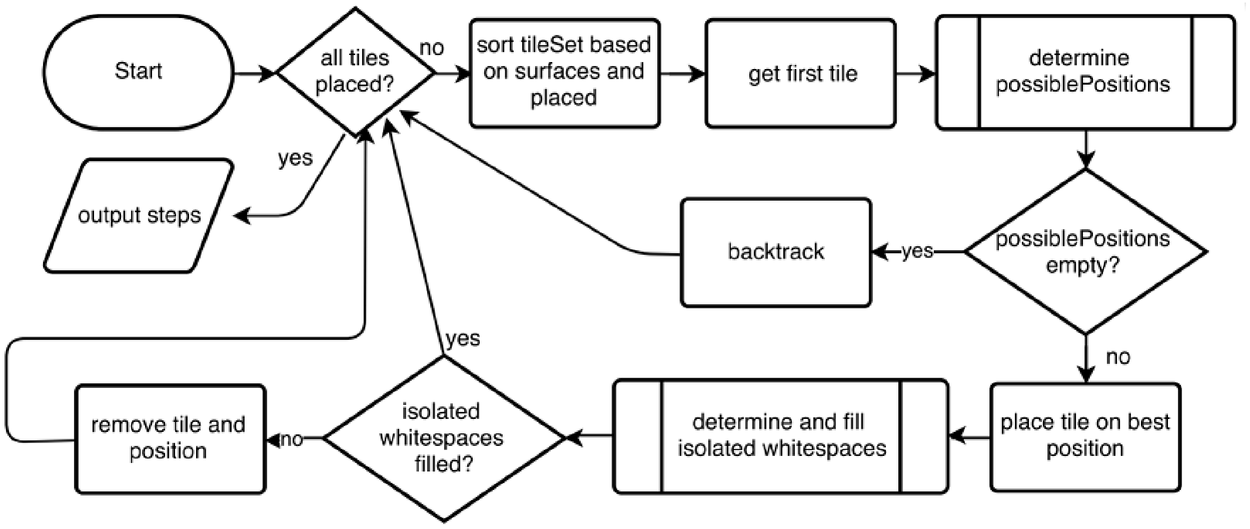

Start
all tiles placed?
no
yes
sort tileSet based on surfaces and placed
get first tile
determine possiblePositions
output steps
backtrack
yes
possiblePositions empty?
no
yes
isolated whitespaces filled?
no
remove tile and position
determine and fill isolated whitespaces
place tile on best position

Figure 1 Flowchart Algorithm Version 3

The pseudocode of algorithm version 3 is displayed below. The underlined parts indicate parts of code that are added to create version 2 and 3 on top of version 1.

```
Steps = recursivePlacing()
def recursivePlacing():
        if field.solved(): return steps
        sort tileSet and get first tile #based on tile surfaces and placed boolean
        determine possiblePositions(tile)
        if (no PossiblePositions) then
                get lastPlacedTile
                remove current Position from possiblePositions
                remove lastPlacedTile from Field
                while (no possiblePositions of lastPlacedTile)
                        get another lastPlacedTile
                        remove current Position from possiblePositions
                        remove lastPlacedTile from Field
                steps += 1
                recursivePlacing()
        sort possiblePositions(tile)
        for position in possiblePositions
                if (placeTile(tile, position)) then
                        if not isolatedWhitespace() then
                                remove current Position from possiblePositions
                                remove tile from Field
                steps += 1
                recursivePlacing()

def possiblePositions(tile):
        if tile is not placed then: add possible position and score
        if no possiblePositions then: flip tile and add possible positions and score

def isolatedWhitespace():
        determine white spaces
        for white space in whitespaces: fill white space
        if whites paces is empty or all filled then: return True
```

Testing

Some of the testing were done during the development of the algorithm version and some after all versions were complete. During development, we had in mind to improve the first algorithm to a version that would solve all field sets in the least amount of time and steps. We were hoping to achieve this solely by the implementation of different (set of) heuristics. All the testing was done on a single computer so that the difference in computation time correlates only to the algorithm version and the heuristics it uses and the field it is trying to solve. During the testing phase, we were interested to see how many fields the algorithm versions can solve and in how much time and recursion steps. The maximum recursion steps were set to 20.000, due to Python’s limitations. We are also interested to see how close the algorithm came to solving the fields that it was not able to solve. The results are shown in the next section.

Experimental Results

After the implementation of the white space parameter, tests were done to see how these parameters would affect the computation time of version 3 of the algorithm. First we looked at the different number of fields solved with the different parameter settings, because the percentage of solved field is more important than the time it takes. The table below shows the impact the white space parameter had in how well the algorithm solves field sets of size 15. An increase is noticeable in the number of fields solved, up to 90%. 90% and 100% have the same number of fields solved.

Table 1: Fields solved with different white space parameters

White space parameter	Fields solved (≈15 tiles)
70%	84%
80%	88%
90%	92%
100%	92%

Because parameter 90% and 100% had the same number of fields solved, we had to look further to determine a parameter that is the most efficient in solving the fields. So, we looked at the average steps taken and the average seconds needed to solve the fields. Figure 2 shows the result of this test.

- Parameter 70%: average number of steps is 53 and average time is 0.112 seconds
- Parameter 80%: average number of steps is 990 and average time is 0.727 seconds
- Parameter 90%: average number of steps is 140 and average time is 0.306 seconds
- Parameter 100%: average number of steps is 199 and average time is 0.381 seconds

For further testing, as shown in figure 3, parameter 90% is used for version 3. This parameter is the most efficient setting.

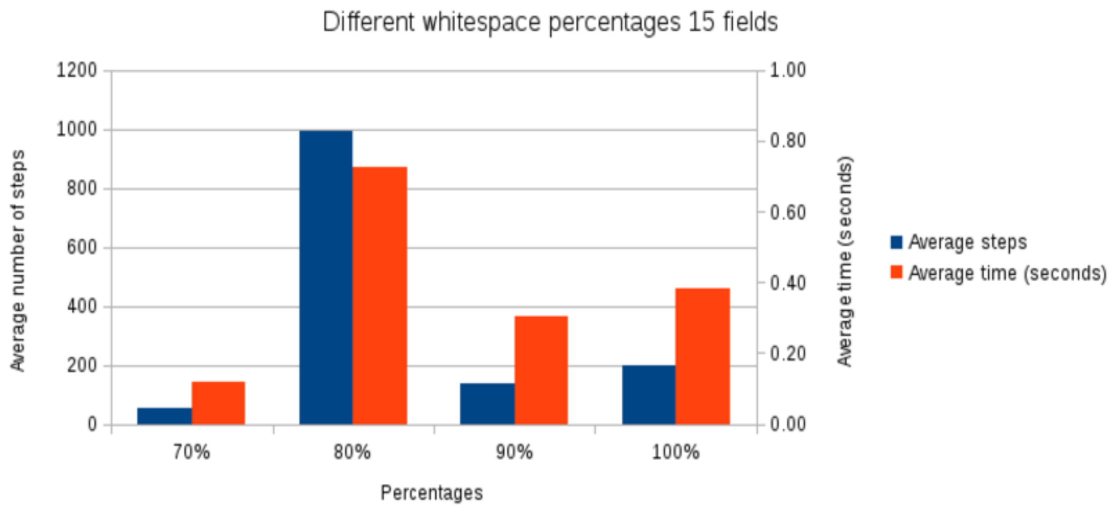


Figure 2 White space parameter percentages

In figure 3 you can see the percentage solved fields for each version.

- Algorithm version 1 solved 76%, 8%, 4%, 0% of field sizes 15, 25, 35, 45, respectively.
- Algorithm version 2 solved 84%, 32%, 12%, 4% of field sizes 15, 25, 35, 45, respectively.
- Algorithm version 3 solved 92%, 36%, 16%, 8% of field sizes 15, 25, 35, 45, respectively.

Fields with ≈55 tiles are not included in the graph, with the reason that none of the versions solved any field configuration of this field size.

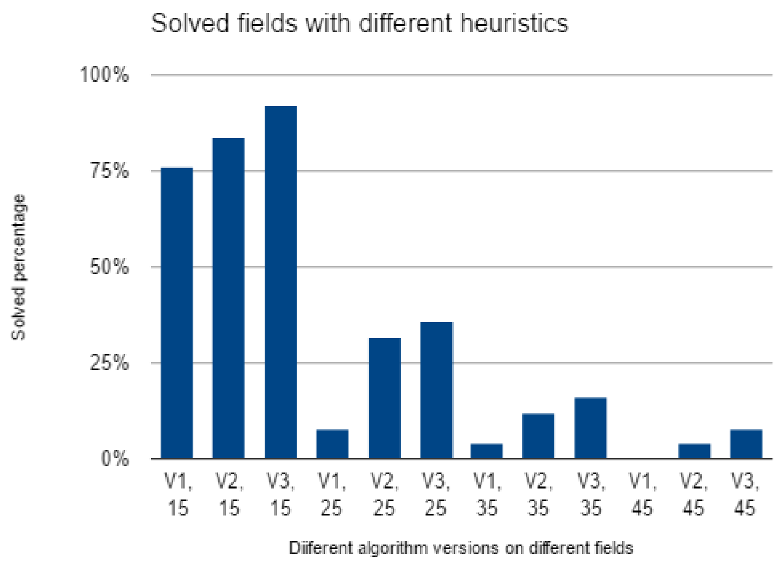


Figure 3 Solved percentages with different heuristics

We have mostly been focusing on the fields that the algorithm could solve and how well it is doing on the fields it can solve, but what about the ones that it cannot solve? To better understand this, we have determined for all field sets, with exception of 55, the percentage of the field covered by a tile with no overlap and taken the average of the field sets per algorithm version. Figure 4 is a graphical representation of this result.

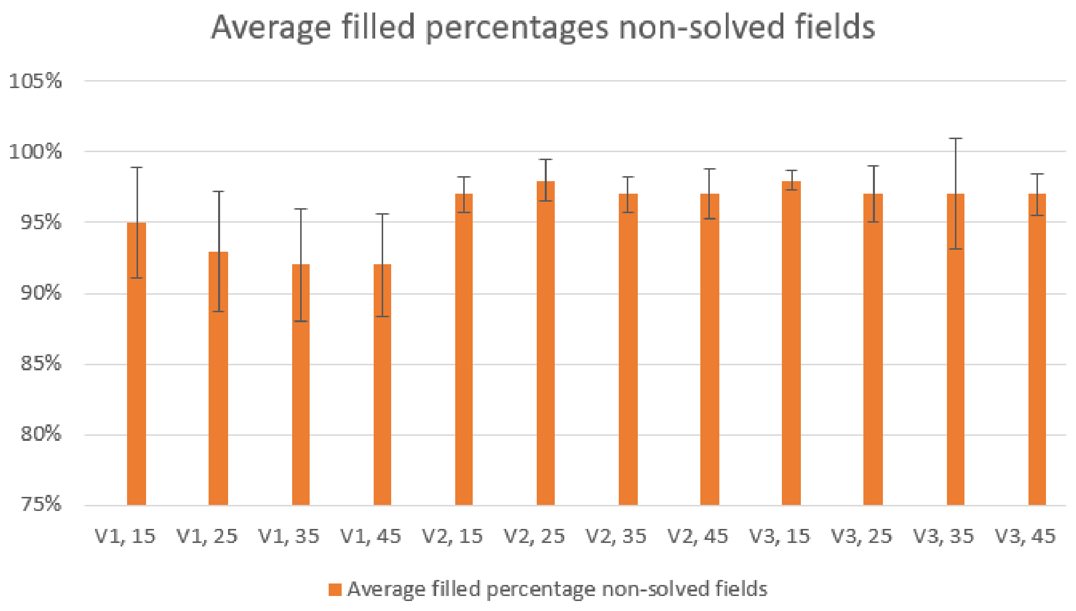


Figure 4 Average filled percentages non-solved fields

Analysis and Discussion

As mentioned earlier it is noticeable from figure 3 that our algorithm (all versions) has difficulties with field sizes larger than 15. Bigger field sizes translate to a problem with a higher complexity. This graph also shows that the improved version was indeed doing better than the initial version, but they are still unable to solve all field sets. Because of the recursion limit, if the algorithm is not able to solve a field set within this limit, it will end up as unsolved and the 'solved percentage' is printed. Bigger field sets with a more complex problem have a bigger search tree and our algorithm needs more time to search the whole tree. In some cases, the solution could be all the way in the end of the tree and the algorithm would take too much time to get there and thus not solve it. This is why we ended up focusing more on field size 15, because in the end the goal of this research is to use heuristics to improve the algorithm. And figure 3 shows that heuristics have, without doubt, improved the algorithm. Figure 4 indicates that the versions with implemented heuristics are closer to a completely filled field and thus a solution.

The white space parameter helped a lot with reducing computation time. Reducing the amount of time the algorithm needs to come up with a solution is also an improvement. The white space parameter table shows that both 90% and 100% parameter have the same result in terms of fields solved. Figure 3 illustrates that the parameter of 90% is the ideal choice because it takes less time than 100% for the same result. It is unclear why there is such a big outlier in the 80%. We think this is a coincidence with the field size and the sizes of the tiles. We think that during backtracking the filled field percentage is around 80%, so removing a tile might cause the filled space to be under 80%. As 90% has proven to be the best choice from this selection, it might be interesting to see how the parameters values of 85 and 95 affect the algorithm.

Version 3 of the algorithm is able to solve the most fields on all field sizes. For all unsolvable field configurations, version 3 was still the version to be the closest to solving the tiles. So even the unsolvable fields have gotten closer to being solved. As expected our hypothesis claim is true and the algorithm designed was improved by implementing different heuristics. This also means that the answer to our research question is positive.

Due to limitations of Python, we could not run the algorithm versions for more than 20.000 recursions. The cause of Python crashing was not clear. Use of memory space and recursions are kept as low as possible. The possibilities of Python or improvements of the codes has to be investigated in order to let the algorithms run longer. All versions of the algorithm will lead to solutions for every field, but the time it takes now is too long.

Due to the time span available for this research, we are unable to continue this research and see if adding or changing heuristics will eventually lead to a perfect version of this algorithm which can solve all fields. Since we were indeed able to prove that our hypothesis is true, we are forced to believe that this is technically possible. This also means, as expected, that the answer to our research is indeed positive.

Conclusion

Looking back at the research conducted, we have designed a constructive algorithm which can solve some of the field configurations. By adapting the heuristics used, we were able to create two improved versions of the initial algorithm. Version 2 was the first improvement and version 3 an improvement on the second. The addition of heuristics has indeed resulted in improved algorithm versions which were able to solve more field configurations in less time (on an average). This has proven that our hypothesis is true and therefore that we are able to improve a constructive based algorithm to better solve the tiling problem by implementing different heuristics.

Bibliography

[1] Gigerenzer, G. , “Why Heuristics Work” , Perspectives on Psychological Science, Vol 3, Issue 1, pp. 20 - 29 (2008)

[2] Knuth, D.E., “Dancing Links”, Millenial Perspectives in Computer Science: 187--214 (2000)