Optimization of Tile Sets for DNA Self-Assembly

Joel Gawarecki
Department of Computer Science
Simpson College
Indianola, IA 50125
joel.gawarecki@my.simpson.edu

Adam Smith
Department of Computer Science
Simpson College
Indianola, IA 50125
adam.smith@my.simpson.edu

Jaris Van Maanen
Department of Computer Science
Simpson College
Indianola, IA 50125
jaris.vanmaanen@my.simpson.edu

Linsey Williams
Department of Computer Science
Simpson College
Indianola, IA 50125
linsey.williams@my.simpson.edu

Abstract

Tile self-assembly systems serve as models of DNA molecules designed to act as four-sided building units that can self-assemble to various shapes. Research on the design of such nanoscale constructs has shown their high potential usefulness in the area of nanotechnology. Given a target shape of tiles, the goal is to find the tile set that will self-assemble in the target shape. The input to our genetic algorithm is a collection of randomly constructed tile sets. The output of the genetic algorithm is a set of tiles that most closely assembles to the target shape. In the paper we discuss the chosen representation of the tile sets and the experiments we made with selection, cross-over, and mutation methods. Finally we present a distributed computational architecture used to speed up the process of obtaining subsequent generations of tile sets.

1 Introduction

Our research began in the fall of 2011 as part of an Introduction to Algorithms course and has continued into the spring of 2012, partially sponsored by NSF grant CCF-1143839. During our research we studied the tile self-assembly problem and explored algorithms for tile self-assembly. Self-assembly is a process by which components may form complex structures autonomously. Due to the natural properties of DNA structures, self-assembly will occur when DNA molecules are allowed to interact. DNA molecules that can assemble into complex structures can be modeled by a tile self-assembly system. In the Wang model of tile self-assembly [1] a tile can represent a DNA structure with four one-stranded "sticky ends". These ends are composed of a finite combination of short nucleotide sequences that will naturally bind with other nucleotide sequences. Each of the tile's four sides is given a value or "color" that represents the binding properties of the tile.

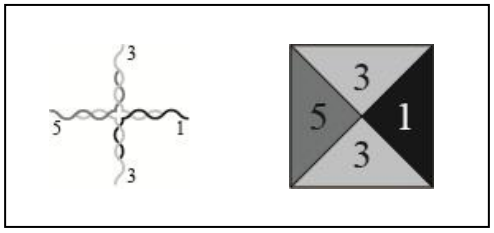


Figure 1: Left: DNA tile as made up of DNA helices. Notice the single stranded "sticky ends". Right: Representational tile model of this structure.

Tiles are placed into a grid where they are allowed to move, interact, and bind based on the compatibility, or binding strength, of their sides. Tiles moving in the grid eventually will bind and form structures that will grow differently based on the tile set on the grid.

We examined the following tile self-assembly problem: given a predefined shape, what tile set can best assemble to that shape? We began by running experiments in the tile self-assembly software (TAS) developed at Iowa State University [3]. In the software, tile sets must be manually designed by the user in order to form specific structures. We found this design process to be time-consuming and inefficient. Looking further into the process of tile assembly, we found work by Terrazas, Gheorghe, Kendall, and Krasnogor, described in [1], [2]. These papers presented a genetic algorithm for tile self-assembly. We were motivated to develop an algorithm that will evolve tile sets instead of going through the process of manually creating and testing tile sets. This evolution would allow an optimal tile set to form based on an original population of randomly generated seed tiles.

In the following sections we outline the genetic algorithm we implemented. We describe the tile set representation and how the fitness evaluation function is computed. We implemented a distributed computational architecture. The results section presents the output our simulated system and shows figures representing this data.

2 Algorithm

A genetic algorithm is an optimization search algorithm. It searches for best "individuals" among a population based on some evaluation criteria, represented by a fitness function. The algorithm starts with a randomly generated initial population. This population is evolved by performing three basic operations in a cycle: parent selection, cross-over, and mutation. The fitness value for each newly-created individual is computed, and this value determines the chances for this individual to be selected for reproduction. A higher fitness corresponds to a higher probability for selection. Thus, over several generations, the quality of the individuals improve to better match the evaluation criteria.

Our program begins by generating a pool of tiles based on the number of unique nucleotide sequences to be used. This pool contains exactly one copy of every possible tile type. From this pool random sets of tiles, or individuals, are generated. An individual consists of a given number of unique tile types with any number of copies for each of these types.

Formally stated, let T be the pool of all possible tiles. An individual in the population is defined by the set

$$T_i = \{(T_{i1}, \alpha_{i1}), (T_{i2}, \alpha_{i2}), \ldots (T_{ik}, \alpha_{ik})\}$$

where $T_{ij} \in T$, and $\alpha_{i1} : \alpha_{i2} : \ldots : \alpha_{ij}$ is the composite ratio of the tiles. In our implementation all individuals have same number of tiles.

Figure 2: Example of one tile set, or individual, made up of five different tile types (indicated by the tile's shading) and 17 total tiles. The composite ratio is 3:2:7:3:2.

These generated individuals are then placed into the total population, which will contain a predetermined number of individuals. This original randomized population will serve as seed for evolving future populations.

For each individual, a Wang tile system simulation is run to determine the shape to which the tile set will assemble. Our simulation allows each tile to move randomly in four cardinal directions on a 50x50 unit grid. If a tile comes into contact with another tile or a group of tiles, the tile may bind and stop moving. Tiles are allowed to bind with each other based on an arbitrarily generated, symmetric binding matrix of existing single-stranded nucleotide sequences that bond at the given strengths. The binding matrix is a numeric representation of the physical binding properties of given nucleotide sequences.

	S_1	S_2	S_3	S_4	S_5
S_1	3	2	9	6	3
S_2	2	5	7	1	0
S_3	9	7	2	5	4
S_4	6	1	5	3	5
S_5	3	0	4	5	1

Table 1: Example of a binding matrix, with S_n representing a unique nucleotide sequence. The values represent the binding strength between a pair of sequences, or edges. Note: two tiles will not bind if the value is below the threshold.

In our tile model, one sticky end of nucleotides is represented as a unique integer, forming one edge of the tile. The compatibility of these edges is stored in the binding matrix, which determines which nucleotide sequences will combine. The sequences needn't be perfectly complimentary to bond however sequences which are not fully compatible will have lower binding strengths, as shown above. Some tiles with a low level of compatibility will never bind, even when occupying adjacent locations on the grid.

Once the simulation for one individual has run through a sufficient number of tile movements, the individual will be assigned a fitness value. After each individual in the population has gone through the simulation and is assigned a fitness value, the population's average and maximum fitness is calculated and stored. To allow for more pressured selection based on high fitness values, the average fitness of the population is subtracted from each individual's fitness.

The fitness of an individual in a given generation is determined by the section of the grid that most closely resembles the target shape. Only locked tiles are considered; movable tiles or empty spaces are not considered. The target shape is a grid of integers where each entry represents the weight of a locked tile in that location. Thus, a positive number in the target shape represents a location in which a locked tile is desired, a negative value represents a location where a locked tile should not be, and a zero represents a space in which the presence of a locked tile is irrelevant. For example, to make a 3x3 square with a hole in the center, the edges of the shape where locked tiles should be found add positively upon a match, and the center location, in which a locked tile should not occur, negatively affect fitness upon match.

1	1	1
1	-1	1
1	1	1

Figure 3: An example target grid for a 3x3 square with a hole in the center.

The program searches every possible section of the grid that could contain the target shape. A record is kept of the highest fitness found during this process for each individual and is used as that individual's final fitness in the generation.

Once the fitness values of the individuals in the population have been computed, roulette wheel selection is used to select parents for cross-over replication. The cross-over method creates two children from two parents by way of a randomly chosen single cross-over point.

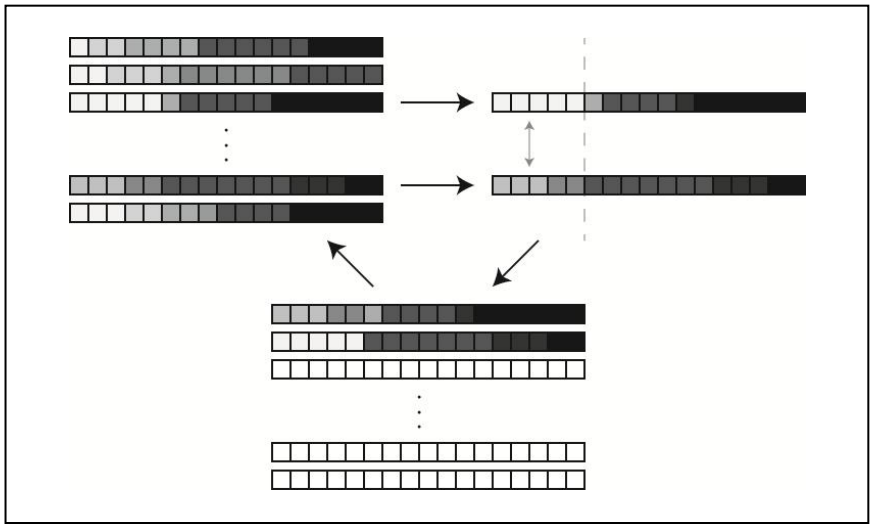


Figure 4: Top Left: Population of individuals. Two parent individuals are selected for cross-over using roulette wheel selection (not shown). Top Right: Crossover occurs. Bottom: The two child individuals are placed into the next generation's population. This process repeats for a given number of generations.

After the cross-over, the children are subjected to a pre-specified rate of tile mutation. A mutation may consist of alteration of the number of tile types within a population or the change of one tile edge to a random value. The resulting children will then populate the next generation, and then the process repeats by selecting two new parents.

To reduce the run time of our program, we implemented a client-server architecture that allows the work to be distributed between multiple computers. Since the majority of the computation takes place in evaluating the fitness, this evaluation runs in the client portion of the program. On the other hand, much of the genetic algorithm is inherently sequential; a new generation cannot be created until the current generation is complete. Additionally, the genetic operations are not very computationally intensive. Therefore, the genetic algorithm runs on the server.

To evaluate individuals for the server, the client requests datasets from the server. These datasets contain all information necessary to evaluate the fitness; that is, all data needed to place a set of tiles on a grid, run the tile motion simulation, and evaluate the results. The primary component of these datasets is the composite ratio of tiles in an individual;

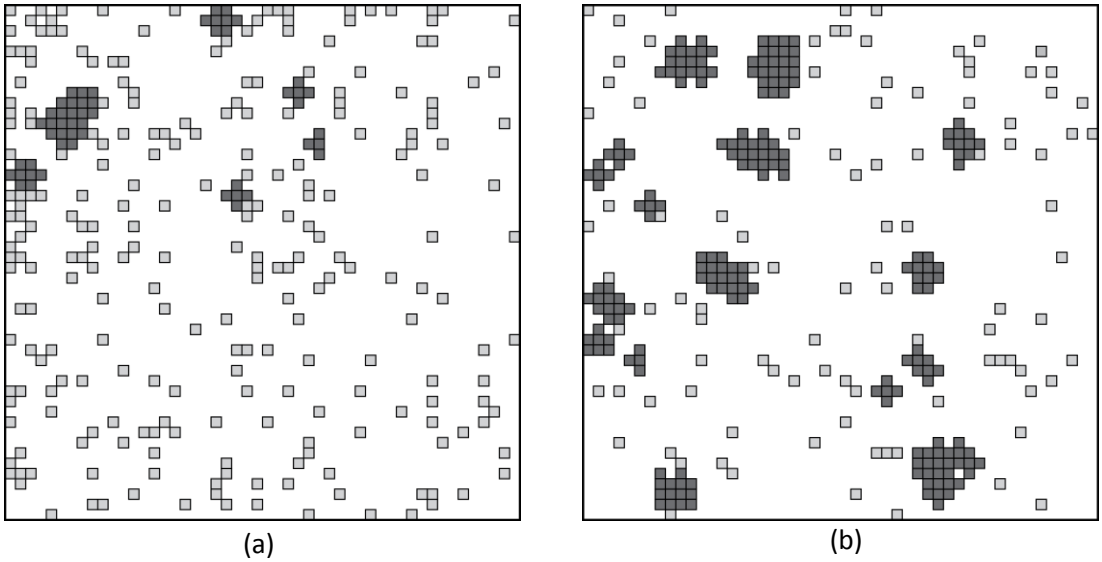
other components include the grid dimensions, the binding matrix, and the target shape. Once a client has computed an individual's fitness, it returns the result to the server. To further increase computation speed, the client process is multithreaded. This allows even a single computer with a multicore CPU to outperform a simple single-threaded implementation in which all fitness values are evaluated sequentially.

The server process handles the genetic algorithm. All selection, cross-over, and mutation take place in a single thread on the server. While this does place a limitation on the potential speedup of the algorithm, the amount of work needed to run the genetic algorithm is small compared to the amount needed to evaluate the fitness. This architecture also has an advantage in that the clients are unaware of anything going on in the server process, except for the data they are sent to evaluate. In this way, we were able to run the genetic algorithm repeatedly with varying parameters, without changing any parameters in, or even restarting, the client processes.

This multithreaded architecture may not have been necessary, but it was convenient. Evaluation time primarily depended on the size of the grid, number of allowed movements, and the number of individuals in a generation. For most of our experiments, we used a grid size of 50x50 tiles, 200 allowed movements per tile, and 100 individuals in a generation. Evaluating a single generation with these parameters took about 10 seconds running on a single client with a single thread. However, with just a few multicore computers this time could be reduced to less than a second. Eventually, the server would become a bottleneck to any further speedup. This was due to displaying and logging a fairly large amount of output.

3 Experimental Results

To test our program, we ran it on various inputs to the genetic algorithm. One of our tests used a simple target shape of a 5x5 grid of ones. We ran this test on a range of other inputs, eventually settling on doing more extensive tests on a 50x50 grid with various numbers of tiles placed on the grid. Figure 5 shows the output of selected generations from one experiment. Our results indicate that the algorithm maximizes the fitness relatively quickly. Very little improvement was observed in either the average or maximum fitness of a generation after the 50^{th} generation (Figure 6). However, such a simple target shape has many solutions. Because a target shape of only positive numbers never penalizes results that are larger than the target shape, this gradually resulted in final grids that consisted of larger and larger groupings of tiles



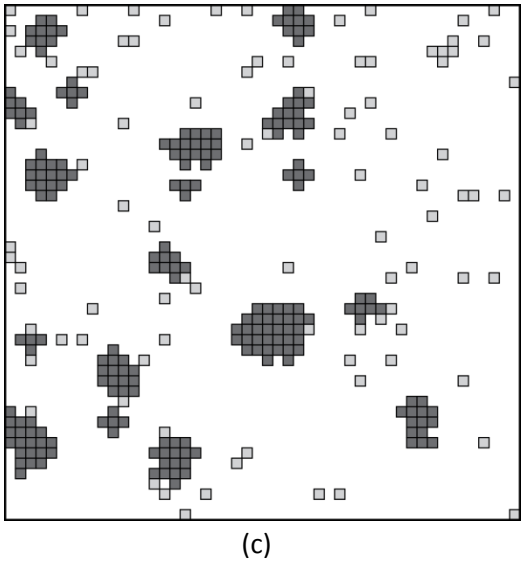


Figure 5: Generations 1 (a), 10 (b), and 25 (c) from a population of 300 tiles and the target shape of a 5x5 square. Dark grey squares represent bound tiles, and light grey squares represent unbound tiles.

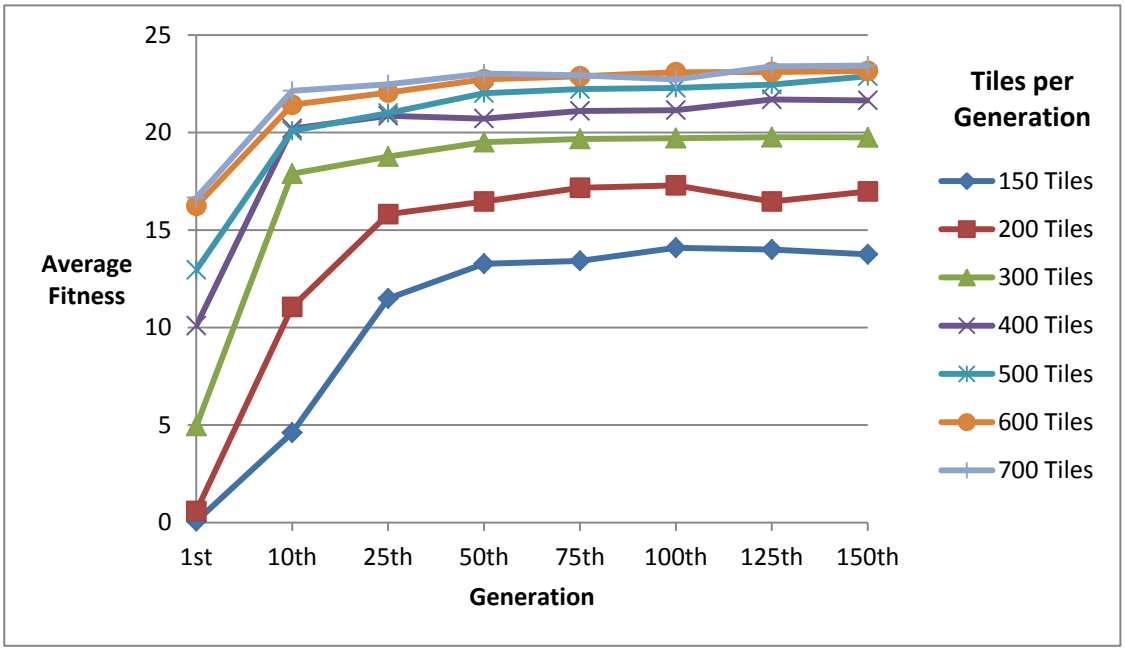


Figure 6: Average fitness for 5x5 solid square target shape.

Another set of experiments also used a target shape consisting of an array of ones, but this time with dimension 1x15. Using this target shape, the algorithm successfully evolved tile sets that produced horizontal strings of tiles, as shown in figure 7.

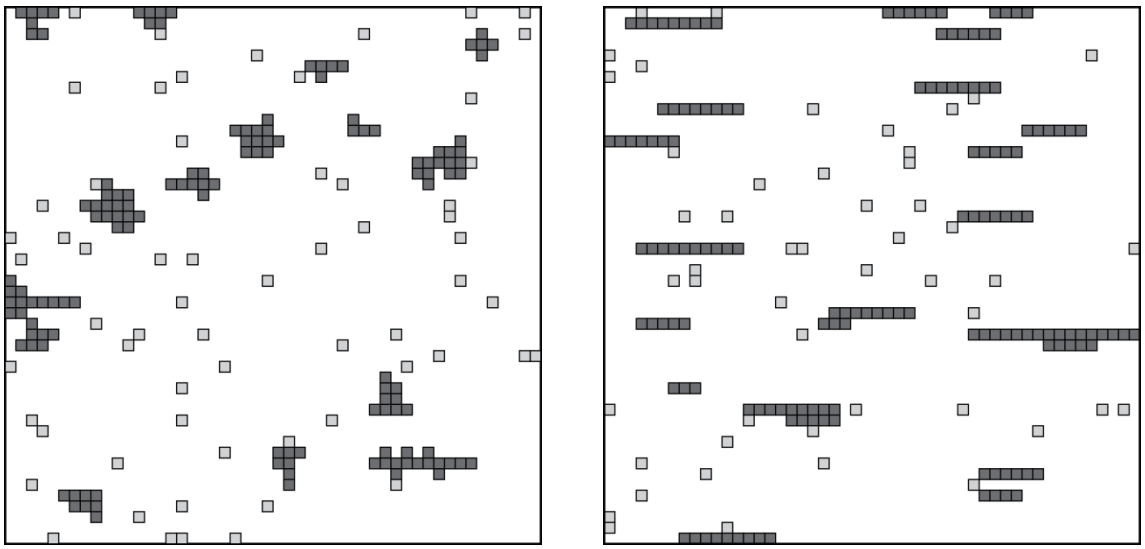


Figure 7: Generations 1 (a), and 25 (b) from a population of 200 tiles and the target shape of a 1x15 rectangle.

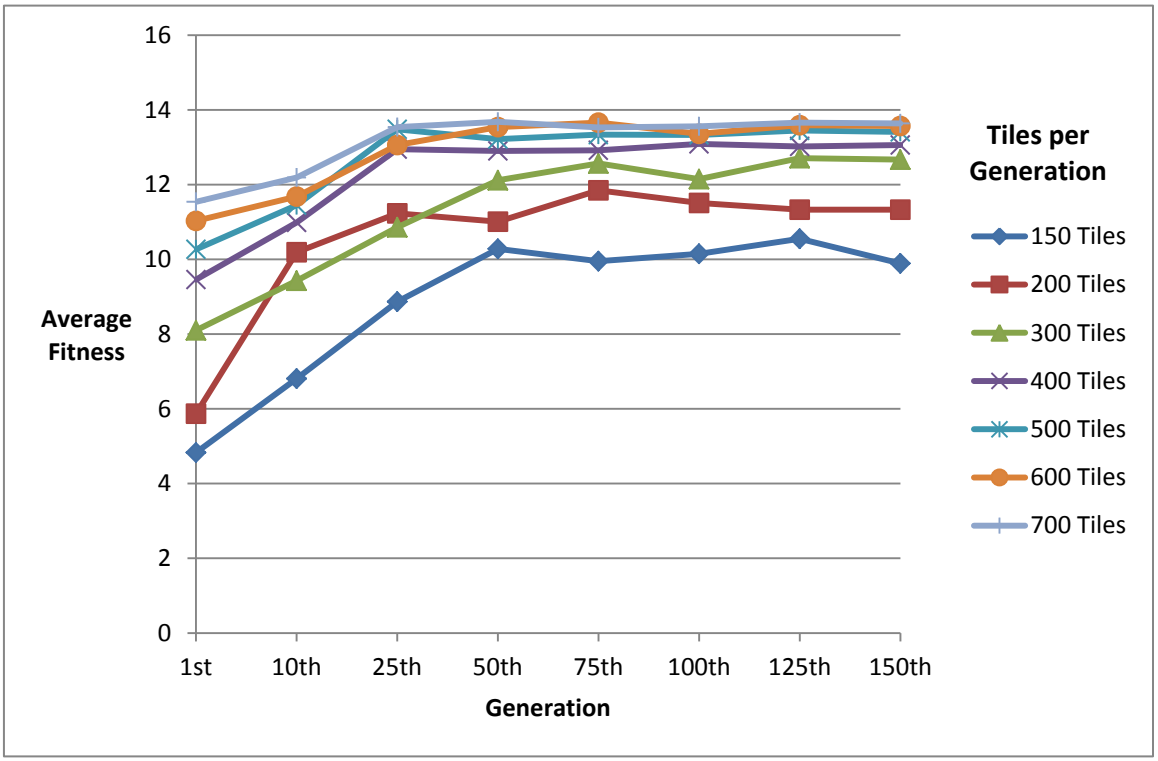


Figure 8: Average fitness for 1x15 target shape.

We have found that the fitness for a population of tiles will level off after approximately 30 generations. We have also found that although increasing the number of tiles in the grid initially allows for better fitness, this tendency has a limit as the grid becomes increasingly full. Because our target shapes in these experiments contained only positive numbers, a grid that is increasingly dense with tiles would only help the fitness. However, target shapes with negative values, that is, areas that should not be filled, would not benefit from an increasingly full grid. This should also prevent overly large groupings of tiles from forming, by assigning them a lower fitness. We have made some preliminary experiments with such shapes, and we are continuing work in this direction.

4 Future Work

There are multiple areas we would like to work on to improve the behavior of the algorithm. Currently, the major disadvantage of our method is that tiles are locked into the grid, and groups of tiles that are bound together are not moved. Additionally, the fitness function treats any tiles locked to the grid the same way it treats a group of tiles that are actually bound to each other.

To improve our program, we would like the tile motion simulation to lock tiles to each other, rather than to the grid itself. In this way, small blocks of tiles are less likely to end up isolated from each other, and could continue to move and bind to other blocks of tiles.

This would be more physically realistic and likely help in forming larger structures on larger grids, without needing as many tiles on the grid.

The other major change we would like to make is to consider groups of bound tiles when calculating the fitness. Currently, two groups of tiles may form next to each other, but not actually have any bonds to each other, or have only weak bonds. However, the current fitness function will treat this exactly the same as one larger group of tiles with the same shape in which all tiles are strongly bound. Ideally, the fitness function would consider groups of bound tiles, allowing two groups to be adjacent while still being considered separately by the fitness function.

Another modification we would like to make to our algorithm is to consider the number of times the target shape, or similar shapes, are produced. The fitness function currently considers a grid with a single occurrence of the target shape just as fit as a grid with many copies of the target shape. We experimented with counting multiple copies, but the fitness function is not able to distinguish groups of tiles, as stated earlier. Because of this, a single group of tiles could count multiple times, and greatly inflate the fitness. However, if the fitness function could be applied to each group of bound tiles, a better overall fitness could be accumulated.

Acknowledgements

This work is partially sponsored by NSF grant CCF-1143839 and the Computer Science Department at Simpson College. We would also like to thank our advisor for this project, Dr. Lydia Sinapova of Simpson College. Finally, we would like to thank the Laboratory for Algorithmic Nanoscale Self-Assembly research team from Iowa State University, in particular Dr. Jack Lutz and Dr. Jim Lathrop, who first introduced us to the field of tile self-assembly, and provided the inspiration for this work.

References

[1] G. Terrazas, M. Gheorghe, G. Kendall, N. Krasnogor. Evolving Tiles for Automated Self-Assembly Design. *Proceedings of the IEEE Congress on Evolutionary Computation*, Swissotel The Stamford, Singapore, 2007.

[2] G. Terrazas, M. Gheorghe, G. Kendall, N. Krasnogor. Automated Tile Design for Self-Assembly Conformations. *Proceedings of the 2005 IEEE Congress on Evolutionary Computation*, Edinburgh, Scotland, 2005.

[3] M.J. Patitz. Simulation of Self-Assembly in the Abstract Tile Assembly Model with ISU TAS. *Presented at CoRR*, 2011.