Two Efficient Algorithms for VLSI Floorplanning

Chris Holmes
Peter Sassone

ECE 8823A
July 26, 2002

Table of Contents

Appendices:

1. Introduction

Various approaches, such as slicing floorplans and sequence pair annealing, have been presented by prior research to attack the issue of minimizing the area of a group of rectangular blocks. Though these algorithms are effective at obtaining fairly efficient layouts, they are very inefficient. Sequence pair annealing, for instance, is limited by the $O(n^2)$ repetitive reconstruction of its dynamic data structures, a horizontal and vertical constraint graph.

In this paper, we present two alternative approaches to VLSI floorplanning. First is an enhancement of the simulated annealing algorithm, which anneals the graphs rather than the sequence pair, saving the graph reconstruction costs. The other is a new algorithm termed contiguous placement. This approach fits blocks in the floorplan as to minimize the surrounding white-space.

Before discussing these new methods, however, we must first address traditional annealing and its limitations in Section 2. We then present enhanced annealing in Section 3 and contiguous placement in Section 4. In Section 5, we present our performance improvements. These include how the annealing enhancements improve the layout size while reducing run time by more than a factor of twenty, and how contiguous placement can achieve 96% efficiency in just a few seconds. Finally, we summarize with our concluding Section 6.

Various appendices are also included for further information. Appendix A explains how to use the floorplan simulator and Appendix B shows how to use the output file viewer. Appendix C shows some sample annealing layouts, and Appendix D shows sample contiguous placement layouts.

2. Traditional Annealing

Traditional Annealing is a method for doing incremental optimization to a data structure to exploit the ability of computers to attempt many different configurations quickly. For a sequence pair driven floorplanner, traditional annealing involves applying the annealing concept to a sequence pair in an attempt to produce a more optimal floorplan.

Every step in the annealing process involves either a swap of two blocks in either or both the positive and negative sequence pair or a rotation of a block. In traditional annealing, the sequence pair is modified, or the data structure holding the sizes of the blocks is modified. The graph corresponding to the sequence pair is rebuilt at a cost of $O(n^2)$ and Dijkstra's algorithm is run on the graph to determine the longest path at a cost of O(n). Figure 1 illustrates this process; red arrows indicate costly $O(n^2)$ operations, and green arrows indicate simpler O(n) operations, and blue represent trivial O(1) operations. At each temperature range in the annealing process, the number of swaps/rotations made is directly proportional to the number of blocks in the sequence pair, leading the cost of annealing to be $O(n^3)$.

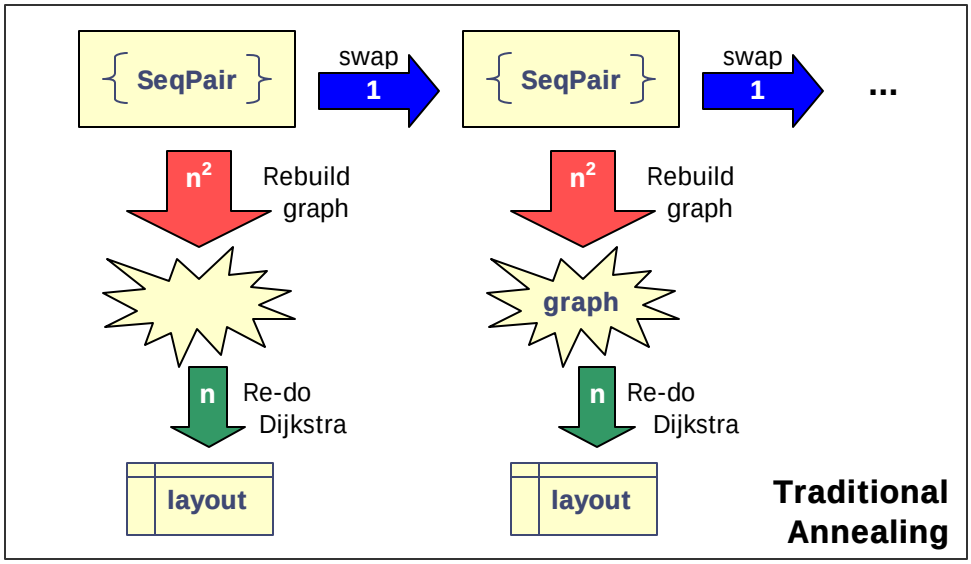


Figure 1: Traditional annealing flow chart

Numerical results for traditional annealing are shown in the Results section, and sample floorplan illustrations are shown in Appendix C.

3. Enhanced Annealing

If the allowed types of changes to a sequence pair are limited to swapping only two blocks that are adjacent in the pair itself, then the only effect the swap will have on the graph is to break the connection between the blocks in either the horizontal or vertical graph and then build a connection between them on the opposite graph.

The cost of rebuilding a graph from a sequence pair is O(n^2), but the cost of breaking a connection can be brought down to an O(1) operation. Finding a node in the graph can be made an O(1) operation by having the nodes held in an array indexed by their block numbers. Finding the connection is O(M) where M is the number of connections. In general, sequence pair graphs are sparsely connected, and this can be assumed to be an O(1) operation. Adding a connection can be made an O(1) operation as well, thus removing the O(n^2) cost of rebuilding the graph. Figure 2 shows the flow of enhanced annealing. Blue arrows have replaced all other types of operations, meaning all functions have been reduced to O(1).

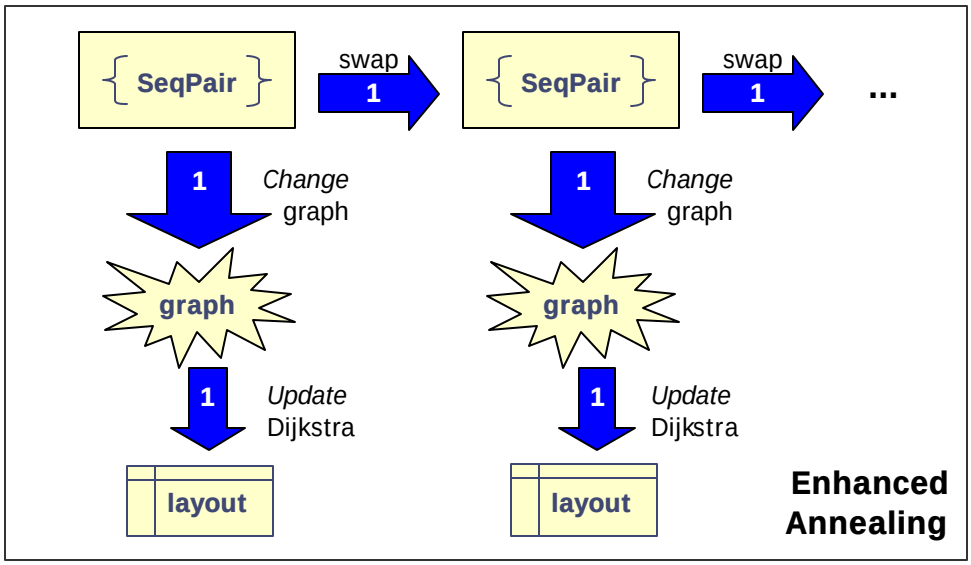


Figure 2: Enhanced annealing flow chart

To remove the cost of running Dijkstra's on the changed graph, each node can maintain a list of the maximum path up to that node as well as a list of path lengths for all the nodes that lead up to it. Breaking or adding requires removing the associated path length and if the maximum path length to the "after" (above or right) is changed, then the

"after" node maximum length must be updated, and those lengths must be propagated forward in the graph.

Any change in the graph that does not change the maximum path at the "after" node requires a time of O(1) to update the length information and is simply added in when the connections are built or broken. If the maximum length is changed, then any nodes that are "after" the node must have the length of the "previous" node updated, and if their maximum length changes, that information must be recursively forwarded. Because the nodes tend to be sparsely connected, the cost of having to forward the new lengths is still fairly small and can be assumed to be O(1). The worst possible case would be a sequence pair that is nothing but left to right, and swapping the farthest left pair. In that case, the cost would still be O(1), but it would exist at every node, making the worst case O(n).

Rotating a block is much more costly than a pair swap. Rotating a block is guaranteed to change the cost of any path that contains it, and the information must be forwarded immediately to all of the blocks "after" it.

The data structures required for efficient annealing are shown in **Table 1**:

Table 1: Data structures for efficient annealing.

After[numBlocks][ ]	List of nodes connected "previous" to each node. The first index is the block number, the second array is a list of the blocks connected which is terminated by -1.
Before[numBlocks][ ]	Same as Before, only "after" connections
BeforeLengths[numBlocks][ ]	Holds the lengths of each path in the Before[ ][ ].
BeforeMax[numBlocks]	Holds the maximum path at each block
Sequence Pair [4] [numblocks]	[0] = positive list [1] = negative list [2] = inverted positive array (index is the block number, value is the location) [3] = inverted negative array (index is the block number, value is the location)

Numerical results for enhanced annealing are shown in the Results section, and sample floorplan illustrations are shown in Appendix C.

4. Contiguous Placement

Contiguous placement was inspired by the popular video game Tetris. The challenge of the game is to quickly stack oddly shaped pieces as to avoid white-spaces between them. Points are awarded for each white-space-free row achieved. Most players adopt a simple strategy of leveling off the pieces as a flat surface gives many more opportunities for tight piece placement.

We adopt this strategy to VLSI floorplanning with some slight modifications. The goal of our algorithm is to place blocks in a contiguous row at the bottom of the layout, then level off the layout with another complimentary row of new blocks, then we begin again. Thus every two rows form pairs which fit together as best as possible.

The best way to create these complimentary rows is to form trapezoids (Figure 3). To minimize the gap between row-pairs, we would like both rows to contain roughly equal height blocks. Thus we place the shortest blocks in row 1, next shortest in row 2, and so on. It is not important that each row contain an equal number of blocks or that the rows are of equal length; this will be addressed via mutations.

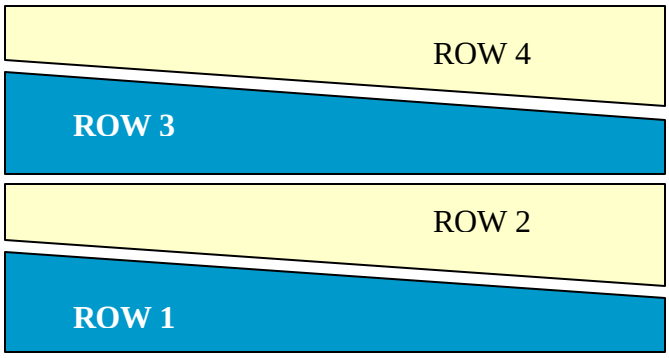


Figure 3: Geometric goal of contiguous placement layout

To compress the rows, we must make them trapezoidal. So we now sort the odd rows in descending order, and sort the even rows in ascending order. Finally we flip the even rows upside down and squeeze the complimentary rows together until they touch. This leaves us with our *initial placement*. A sample initial placement for a 50 block floorplan is shown in Figure 4. Note that due to the quick and crude bucket sorting of blocks into shortest, next shortest, and so on, the rows are not even close to even length.

This yields an inefficient layout since each row shorter than the maximum must have white-space to fill the gap.

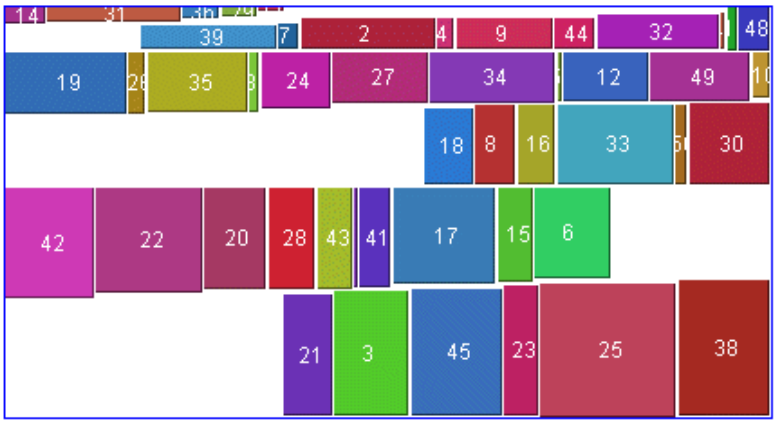


Figure 4: Sample initial placement for 50 block floorplan.

To address this row length issue, we now apply *mutations*. We have four different kinds, two for growing the shortest row and two for shortening the longest row:

- Grow( ): adds blocks to the shortest row by moving the tallest blocks from the row below and the shortest blocks from the row above.
- GrowRotate( ): adds blocks to the shortest row by finding blocks in larger-than-average rows that could be rotated 90 degrees to fit this row's height range.
- Shrink( ) : reduces the length of the longest row by moving the tallest blocks in the row up and the shortest blocks down one row.
- ShrinkRotate( ) : finds blocks in the longest row that could be rotated and moved to a smaller-than-average row.

Since we don't want to create any mutation cycles, we keep two locks for each block. We lock the move_lock when a block is moved from one row to another, and lock the rotate_lock when a block is rotated and moved. A block is not allowed to participate in the respective mutation if its lock is set.

Mutations are very quick to even the rows out even though there is nothing preventing a mutation from making the floorplan less efficient. Figure 5 shows a 50-block layout after just 4 mutations. This layout is 90% efficient after just milliseconds of

computation. Unlike annealing, the power of contiguous placement is not found in careful modification of a single floorplan, but of rapidly mutating thousands of initial floorplans in short order. Different initial floorplans are found each time by randomly rotating the blocks, then resorting them into rows.

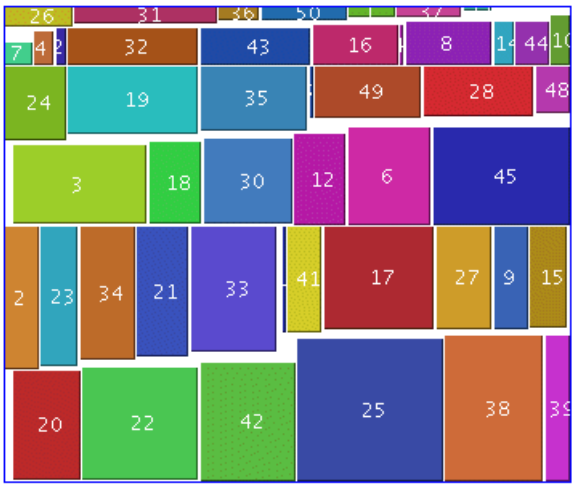


Figure 5: Sample 50 block floorplan after 4 mutations.

The pseudocode algorithm for contiguous placement is shown below. The reader should note that we only mutate 20 times before starting over with a new initial placement. This is purely an empirical optimization since we converge so quickly to an optimal solution. Usually after 10 mutations or so, the best layout has been achieved and the number of locked blocks prohibits further beneficial changes to the floorplan. To assure we do not miss a good layout, the algorithm proceeds through 20 mutations or until all blocks are locked, whichever comes first.

```
contiguous_placement( ) {
   while (keep_going) {
      randomly_rotate( blocks );
      rows = initial_placement( blocks );
      for i from 0 to 20 {
           mutate( rows );
           squeeze( rows );
           check_for_new_winner();
      }
   }
}
```

Contiguous placement usually achieves 90% efficient layouts in under half a second and 96% in less than a 10 seconds. This accomplishment is due to a few basic reasons:

1) The initial placement tends to be very good, achieving around 80% in a negligible amount of time.
2) The only data structures used are static arrays. No graphs, linked lists, trees, vectors, or such are used.
3) Mutations are very simple, occupying less than 30 lines of code with no more than 2 loops inside of one another. The longest part of the algorithm is squeezing complimentary rows together, which is still just a simple loop of subtract-and-compares.
4) Items 2 and 3 mean that each run takes less than 10 milliseconds on a modern PC. This allows numerous runs to fully explore the solution space.

Numerical results for contiguous placement are shown in the Results section, and sample floorplan illustrations are shown in Appendix D.

Results and Discussion

Numerical results are presented in 3 tables, one for each algorithm. Each table shows run time, floorplan area, area efficiency, and area-time metric for each benchmark. Later in this section, we present graphs which can more clearly compare the algorithms.

Table 2 presents the statistics for traditional annealing. If one compares these results to the other algorithms, these numbers seem quite low. This is a result of the $O(n^3)$ order of the algorithm. These numbers would be even worse had a true dynamic graph structure been used rather than a modified adjacency matrix. This advantage is given to traditional annealing as a handicap; the help is not sufficient though for it to compete with enhanced annealing or contiguous placement.

Table 2: Results for traditional annealing

Benchmark	Time (sec)	Area	Efficiency	Sqrt(A*A*T)
M10	0.03	6480	49	1122
M20	0.10	17228	54	5448
M30	0.31	40180	43	22371
M40	0.79	41736	56	37143
M50	1.62	67077	42	85428
M60	3.21	83050	38	148680
M70	5.42	96960	37	225690
M80	8.71	131860	31	389222
M90	13.8	144982	43	537274
M100	19.8	178035	38	793785
M110	27.6	187200	40	983290
M120	40.2	191345	42	1212560
M130	53.1	239370	37	1744053
M140	70.9	226546	40	1908669
M150	95.2	238797	38	2330407

Table 3 shows the results for enhanced annealing. These numbers are significantly faster than traditional annealing while producing better final floorplans.

Table 3: Results for enhanced annealing

Benchmark	Time (sec)	Area	Efficiency	Sqrt(A*A*T)
M10	0.02	4032	79	570
M20	0.03	13938	66	2454
M30	0.04	27630	63	5472
M40	0.08	35979	65	10176
M50	0.14	45843	52	17214
M60	0.23	49585	63	23832
M70	0.41	66822	53	42839
M80	0.43	78680	53	51594
M90	0.74	117782	53	101388
M100	1.11	125125	54	131887
M110	1.31	190475	39	218175
M120	2.13	159036	51	232269
M130	2.45	179880	50	272387
M140	2.23	182229	50	272309
M150	3.96	168752	53	335643

The final table, Table 4, has the results for contiguous placement. These results are by far the best of all three algorithms due to various factors mentioned in Section 4.

Table 4: Results for contiguous placement

Benchmark	Time (sec)	Area	Efficiency	Sqrt(A*A*T)
M10	0.65	3551	90	2863
M20	2.80	9703	95	16236
M30	5.21	18271	95	41703
M40	3.67	24546	95	47024
M50	2.41	29753	95	46198
M60	2.45	32473	95	51363
M70	1.68	37398	95	52624
M80	5.31	43050	96	99202
M90	0.89	65126	96	61543
M100	0.41	69245	96	44801
M110	6.11	76246	97	188469
M120	2.52	83356	96	133701
M130	9.97	90372	97	285353
M140	2.71	93155	97	153352
M150	2.55	92388	97	147531

Two graphs are presented which compare these algorithms. First Figure 6 shows best efficiency achieved in 10 seconds or less. The faster runs of enhanced annealing allow it to explore a larger solution space than traditional annealing. Contiguous placement finds 96% layouts for most benchmark circuits, so it tops this graph with ease.

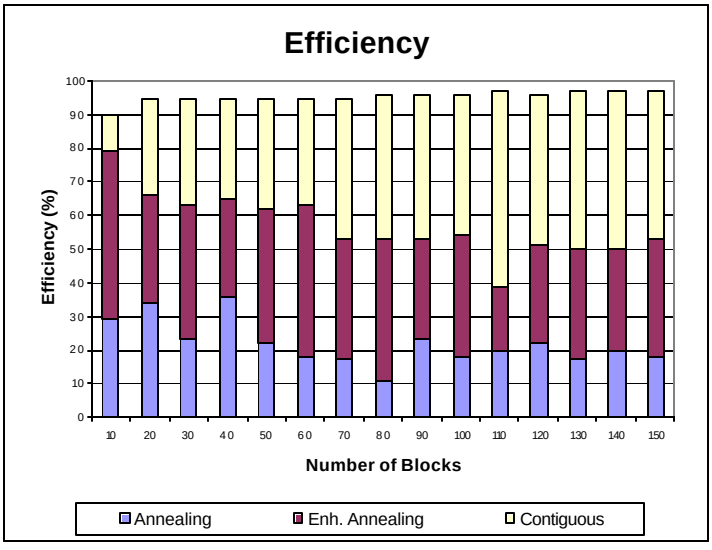


Figure 6: Efficiency achieved after 10 seconds.

The second graph compares the algorithms with the sqrt(area*area*time) metric. For the annealings, we analyze after one run. For contiguous, we give it a less than 10 seconds to find the best layout. Figure 7 is really where traditional annealing looks terrible, with numbers that skyrocket into the millions with larger and larger circuits. Better annealing schedules may bring these numbers down a bit, but they will always be much worse than enhanced annealing since both would benefit equally from such enhancements. Finally, contiguous placement does not show much better results than enhanced annealing due to the peculiarity of this particular metric.

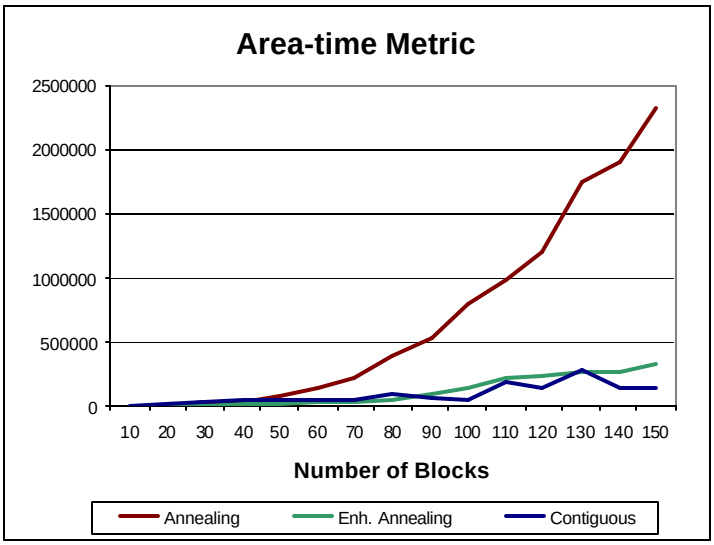


Figure 7: Area-time metric after one run (annealings) or ten seconds (CP)

Conclusion

As is clear from the previous section, these two algorithms make VLSI floorplanning more efficient, allowing more of the solution space to be explored within a given amount of time. From a different perspective, these ideas allow similar results to traditional annealing be achieved in far less time. In either case, the weakness in performance of traditional floorplanning techniques is apparent.

Enhanced annealing shows that static data structures in concert with direct graph manipulation can remove the largest performance barrier in annealing. Though unable to directly perform 'large' swaps, enhanced annealing can still perform the sequence of small swaps in less time than normal annealing would take to rebuild the graph from the

single large swap. Thus any performance limitations of enhanced annealing are directly related to the actual algorithm itself, not the limitations of the data structures.

A new algorithm, contiguous placement, is introduced as a rapid way to produce tight floorplans. Using inspiration from existing game theory, this method creates pairs of rows that fit tightly together. Good initial layouts, fast mutations, and quick result convergence mean that contiguous placement can rapidly evaluate numerous random points in the solution space, taking only a few milliseconds per point. Resultant layouts generally achieve 96% efficiency within seconds.

These approaches vary significantly in structure, algorithm, and speed, but they share a common goal: to outperform traditional sequence pair annealing. Both achieve this goal with ease, showing that there is room for even further improvement in modern VLSI floorplanning.

Appendix A: Instructions on program usage

Since the code is written in Java, at least version 1.2 of the JRE (Java Runtime Environment) must be installed for execution. Typing java –version at a prompt will display if the JRE is installed on a computer and what version it is. If neccesary, a JRE for most operating systems can be downloaded for free at http://java.sun.com/j2se/1.4/download.html. Note that our program is precompiled in a platform-independent form, so the java compiler is not needed.

All three algorithms for VLSI floorplanning (annealing, enhanced annealing, and contiguous placement) are each accessible through the same executable. There are a variety of program switches that select not only the algorithm, but also various behavioral options:

-i filename	Selects filename as the input file of blocks. (*required*)
-o filename	Uses filename as the output file.
-r runs	Specifies number of runs for algorithm (*-r, -g, or –t required*)
-t target_eff	Keeps doing runs until target_eff (as a percent) is reached. Writes the output file (if –o selected), then proceeds to look for target_eff+1. (*-r, -g, or –t required*)
-a	Uses traditional annealing algorithm (default)
-e	Uses enhanced annealing algorithm
-c	Uses contiguous placement algorithm
-d	Turns on debug output
-g	Uses GUI to animate algorithm (*-r, -g, or –t required*)

It is **strongly** suggested that –t be used with contiguous placement as its runs are **very** quick (<5ms) and relatively insignificant. The –t option is also valid with the annealing algorithms, but the *runs* option may be more useful in these cases.

Also note that the GUI is technically 'upside-down', using the top-left as the origin. This is only a visual peculiarity and does not affect any results.

Example executions:

To run the program using the traditional annealing algorithm with the GUI on the input file m100.blk and produce the output file m100.fout :

```
java ChrisPeter –a –g –i m100.blk –o m100.fout
```

To run the program using the contiguous placement algorithm to start looking for the target efficiency of 95% with debug on, input file m100.blk, and no output file:

```
java ChrisPeter –c –d –t 95 –i m100.blk
```

Appendix B: Instructions on Output Viewer usage

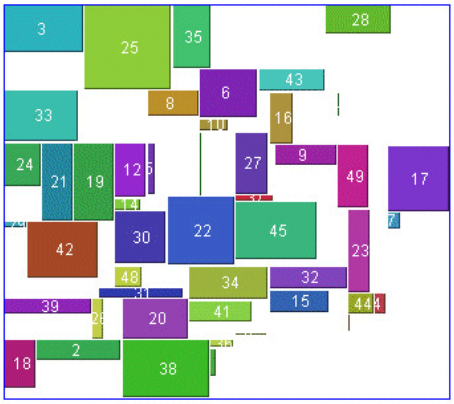
Also included in the code package is a small utility which reads .fout files and displays the floorplan using the GUI. This is handy for verifying that a particular floorplan is valid (has no overlaps and such).

The usage is very simple:

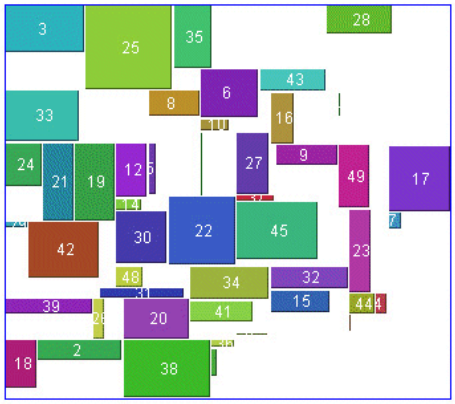
java OutputViewer *filename*

When the GUI is displayed, hit the "exit" button to close.

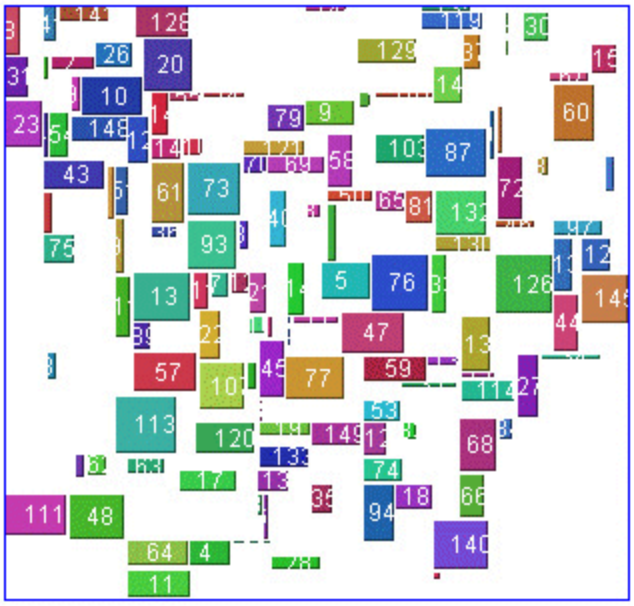
Appendix C: Sample Simulated Annealing Floorplans



m50.blk – 62% efficient

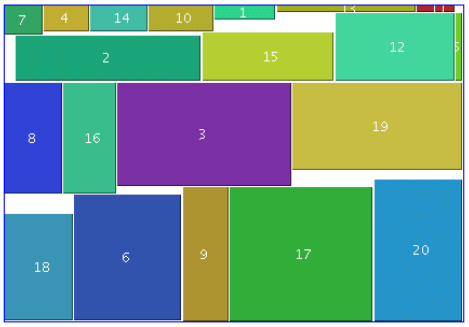


m100.blk – 54% efficient

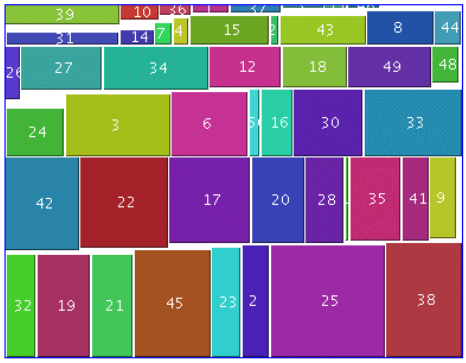


m150.blk – 53% efficient

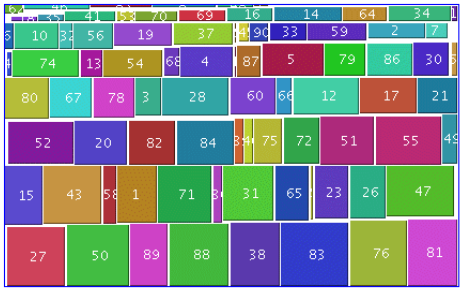
Appendix D: Sample Contiguous Placement Floorplans



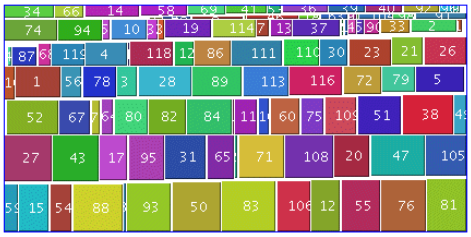
m20.blk – 95% efficient



m50.blk – 96% efficient



m90.blk – 97% efficient



m120.blk – 97% efficient

m150 – 97% efficient