ECE1387 A2

Nov 6 2023

Troy Denton 1009600932

1. Submission

Submitted on the ECF machine, also available at https://github.com/trdenton/ece1387_a2

1.2 Build instructions

- Clone the repo or extract the source provided
- `cd ece1387_a2`
- `mkdir build`
- `cd build`
- `cmake ..`
- `make`
- `./a2 -h`

example:

```
./a2 -f ../data/cct3
```

You can apply the -i (interactive: UI window displayed) and -s (step algorithm) arguments as well. With the latter, click the "SPEED 1x" button to see the flow routine work. Clicking the "Proceed" button after flow spreading causes the window to re-launch with the post-flow spread cell positions

run unit tests:

```
./unit_tests
```

1.3 Notes on UMFPACK compilation

I've created a configuration that requires no editing for compilation of UMFPACK which may be valuable for future students. It is codified in my *CMakeLists.txt*, but can also be done like so:

- `cd SuiteSparse_config && make UMFPACK_CONFIG="-DNCHOLMOD -DNBLAS" library`
- `cd AMD && make UMFPACK_CONFIG="-DNCHOLMOD -DNBLAS" library`
- `cd UMFPACK && make UMFPACK_CONFIG="-DNCHOLMOD -DNBLAS" library`

I found this to be easier to implement manually (no editing files) and also easier to automate e.g. when building on different systems (ECF vs my laptop).

1.4 Software details

The *circuit* class (circuit.cpp, circuit.h) is the high level class that implements analytical placement.

The *net* and *cell* classes (also in circuit.cpp, circuit.h) provide the connectivity mechanisms needed model the circuit data. Cells and nets contain reference to each other through their labels.

The fabric class (fabric.cpp, fabric.h) implements a model for the 25x25 chip with 8x8 blockage. It contains a 2D matrix of *bin* structs, each which contain coordinates, usage stats in accordance with Darav[2019], and functions to implement the flow algorithm.

psis.cpp and *psis.h* provide functions for implementing Ψ control.

main.cpp, ui.cpp, ui.h contain unremarkable but necessary structure for the *a2* program.

All files ending in *.cc* are unit test programs.

Routines of note

Analytical placement is implemented in circuit.cpp, controlled by the function *circuit::iter(fabric* fab).* It is named as such as I had intended to make it perform one iteration at a time for animating in the UI. With a nullptr provided as *fab,* the program does the baseline analytical placement (i.e. without any flow spreading). The results of a flow spread are provided to the analytical placer by providing a *fabric* object that has undergone the flow spreading algorithm.

The flow algorithm is implemented in fabric::run_flow_step(flow_state* fs). This function is designed such that it is called continuously within a UI loop to view the flow algorithm at work. This was very useful for debugging.

Overall flow works as such:

- AP solver:
    - construct the circuit cells and nets, connectivity in circuit::circuit(string* s)
    - circuit::iter(fabric*) constructs an AP solver equation is built by invoking circuit::build_solver_matrix(fabric*) to build Q and circuit::build_solver_rhs() to build b to solve Qx = b
    - circuit::iter(fabric*) also calls umfpack(enum axis, double*) for each axis (X, Y) to perform the AP matrix solve algorithm
- Spread flow routine:
    - In interactive/step mode, ui_drawscreen() is invoked every 10 milliseconds when running at 1x speed (a modification to easygl). This invokes steps in the fabric::run_flow_step(flow_state*) to advance the algorithm in a way that is easy to animate. View the animation by running a2 with the -s option and clicking the “SPEED 1x” button.
        - In non-interactive mode, it is run to completion the same way in fabric::run_flow()

- Once run_flow_step has concluded there are no overflowed bins (via fabric::get_overused_bins), it return *true* without doing further processing
- When the UI is closed with *Proceed* (or automatically after completing the flow algorithm), there is now a populated *fabric* object. circuit::iter is called with this object as argument, which causes the AP flow to incorporate the solved fixed cells
- The fixed cells are automatically incorporated with the provided spread weight (-z argument) to perform AP flow

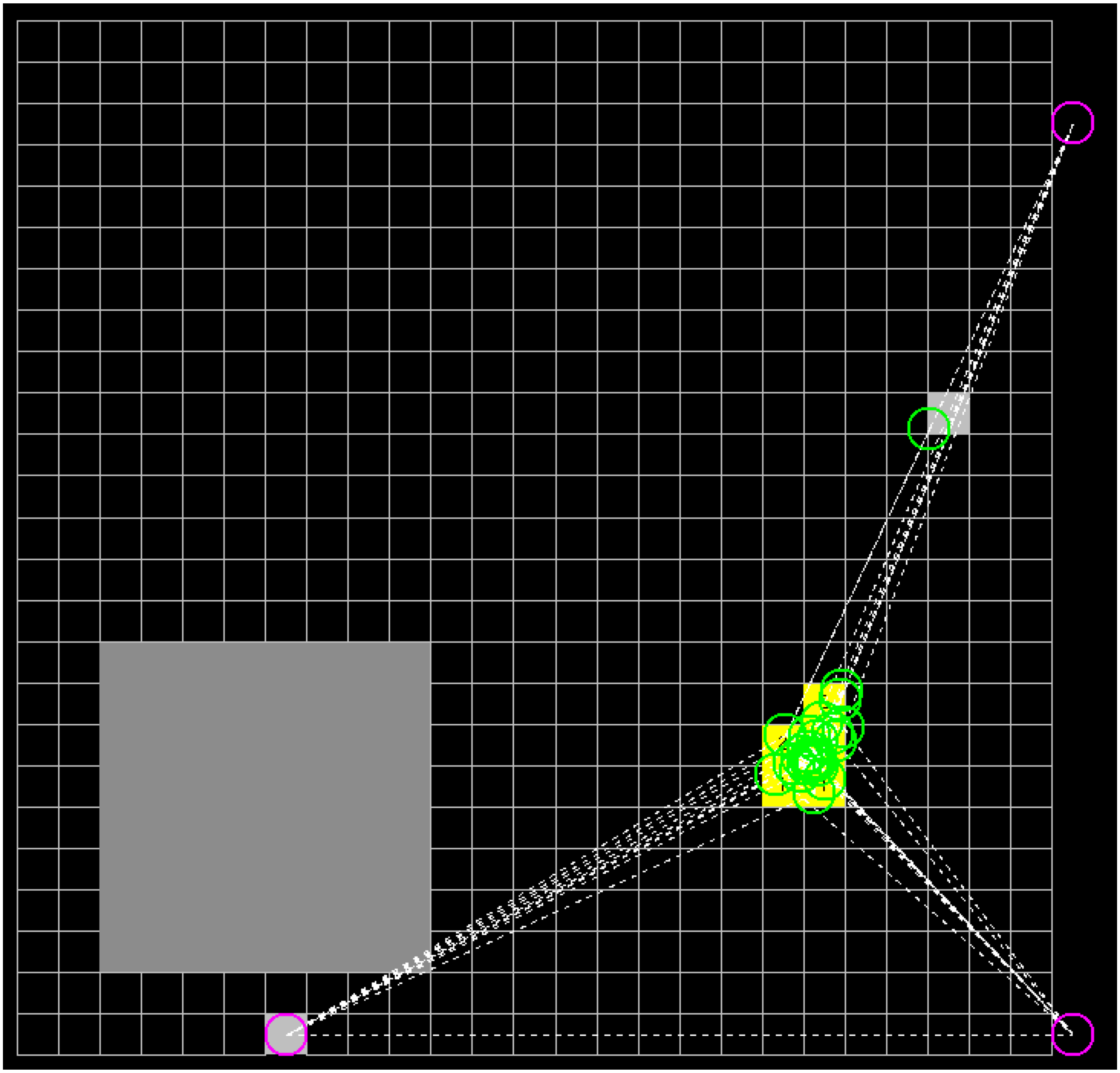
2. Deliverables

2.1 Analytical placement problem with *clique* net model

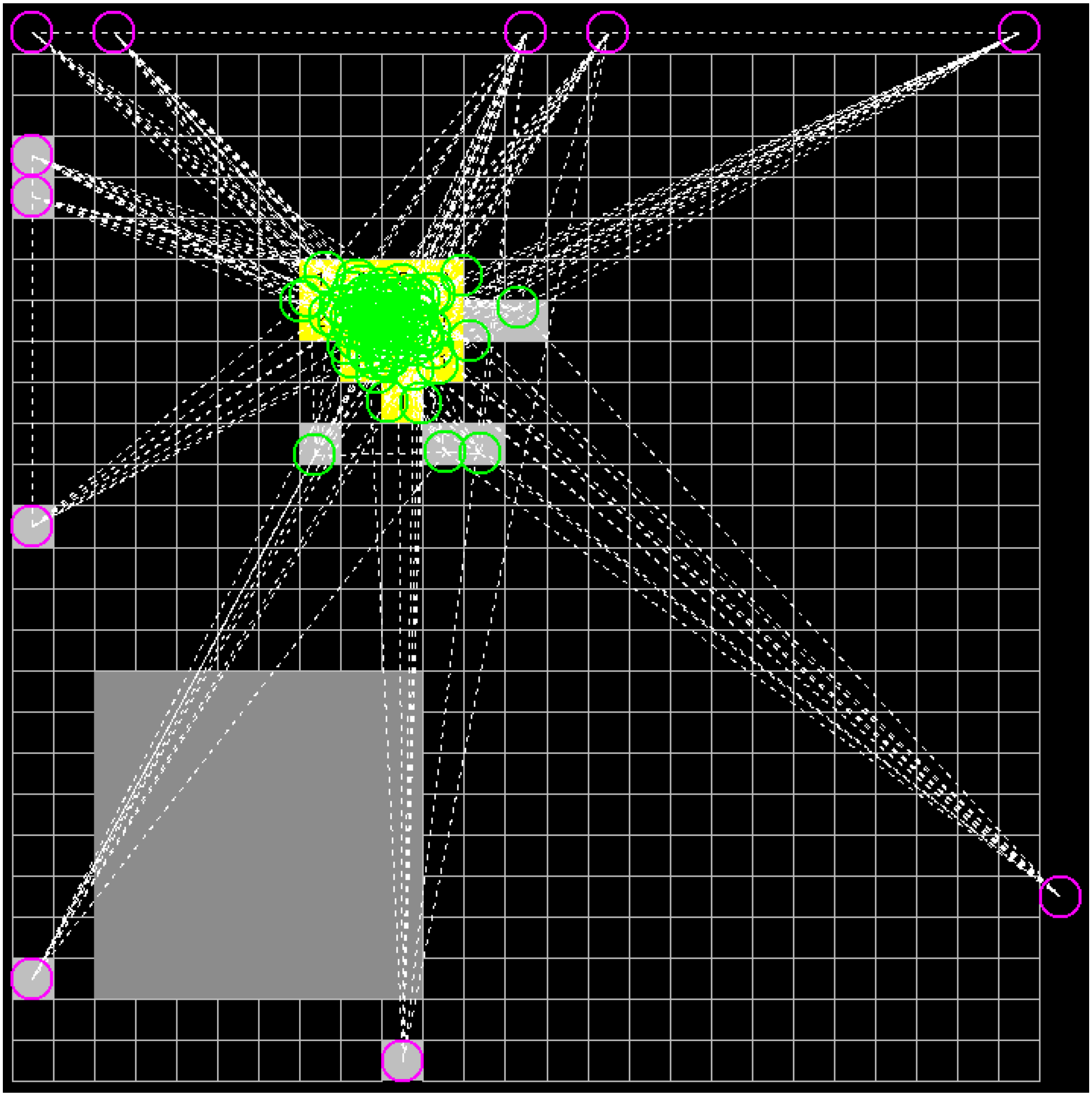
Note: cells are drawn as circles to help make them visually distinct from the bins (the square grid). Green cells are movable, and magenta cells are fixed.

Table of results

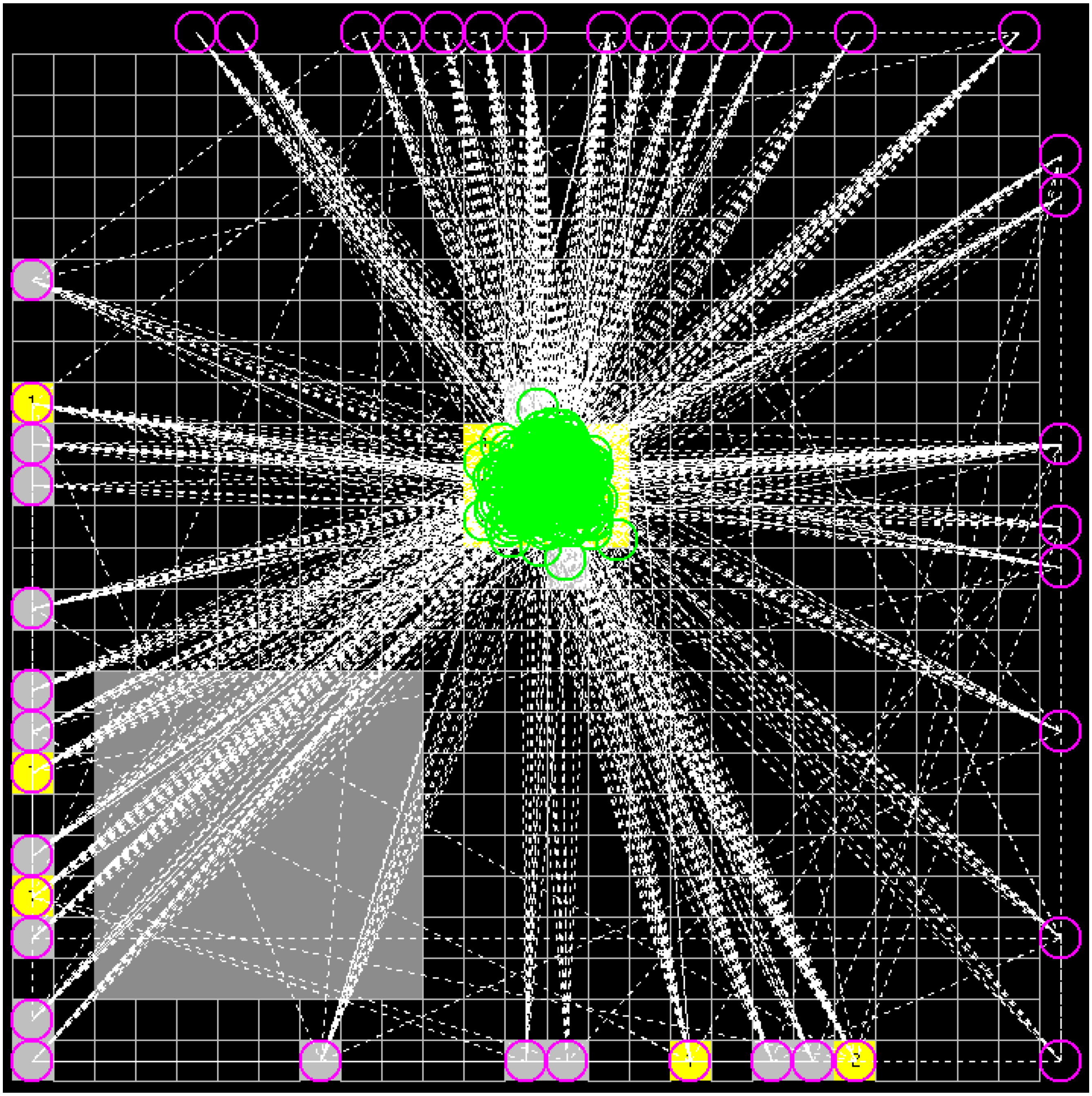
Circuit	HWPL
cct1	12.46
cct2	9.73
cct3	6.85



cct1



cct2



cct3

2.2 Spreading via fixed cell net weight adjustment

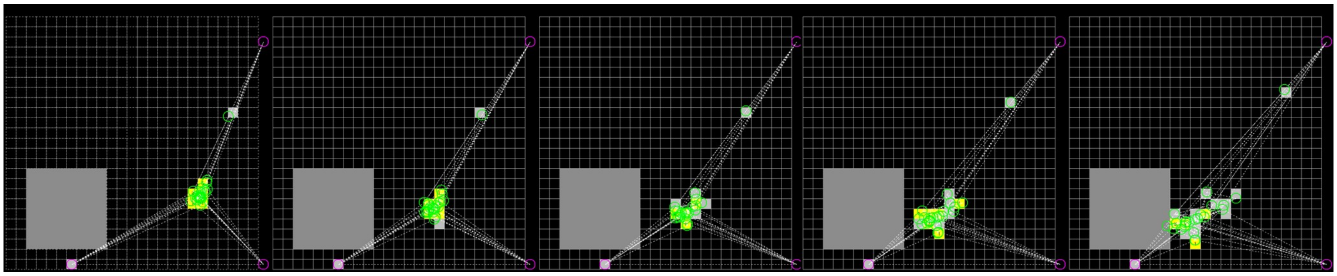
For this method, providing the -w argument to my compiled program increases fixed weight width by the corresponding number of 10ths, e.g. -w 5 increases net weight by 0.5

Circuit	HPWL baseline	HPWL @ -w 2	HPWL @ -w 4	HPWL @ -w 8	HPWL @ -w 16
cct1	12.46	16.02	18.41	22.04	26.49
cct2	9.73	10.60	11.90	14.40	18.3
cct3	6.86	9.7	12.25	16.43	21.62

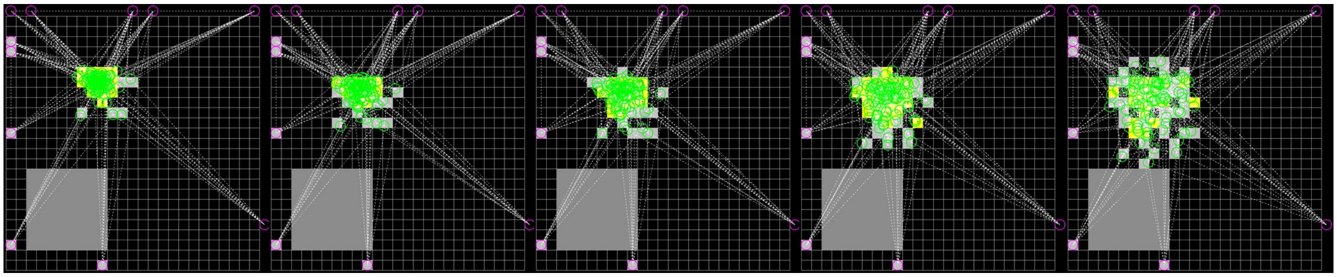
We can see that we do get some spreading with the increased fixed connection net weight in the diagrams below. However, this approach does very little to address overfilled bins – even with a quite extreme bias (-w 16) we see a minimal decrease in the number of overfilled bins. While there is some overlap removal – this approach does nothing to address the gaps created within otherwise contiguous areas of mapped cells (see -w 16 in cct2 for a clear example of these holes). When these contiguous holes go unfilled, it represents a missed opportunity to reduce HPWL.

Overall, we see increased HPWL, and some spreading – but very little overlap removal taking place. For this reason, this heuristic is not a reasonable approach compared to the later technique explored in this assignment.

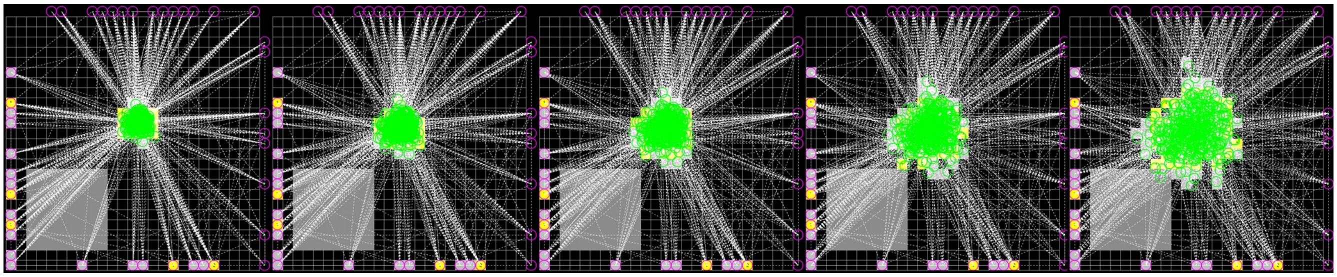
Note: overfilled bins are depicted in yellow.



cct1, from left to right: baseline, -w 2, -w 4, -w 8, -w 16



cct2, from left to right: baseline, -w 2, -w 4, -w 8, -w 16



cct3, from left to right: baseline, -w 2, -w 4, -w 8, -w 16

2.3 Implementation of flow-based spreading

2.3.i Different approaches for Ψ

Approaches taken for managing Psi are linear, quadratic, and cubic growth, with a range of scaling coefficients tested. This is controlled with the -p and -a options to the a2 compiled program. *iter* begins at 0 and increments by 1 after every bin spreading

- Linear: Psi(iter) = A*iter
- Quadratic: Psi(iter) = A*iter*iter
- Cubic: Psi(iter) = A*iter*iter*iter

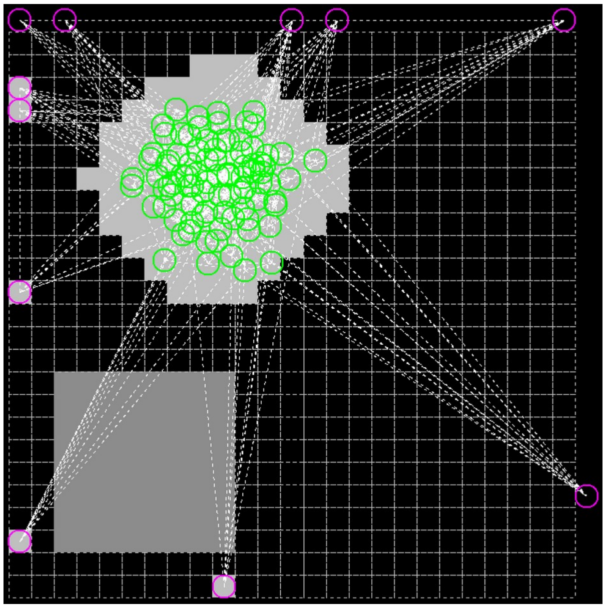
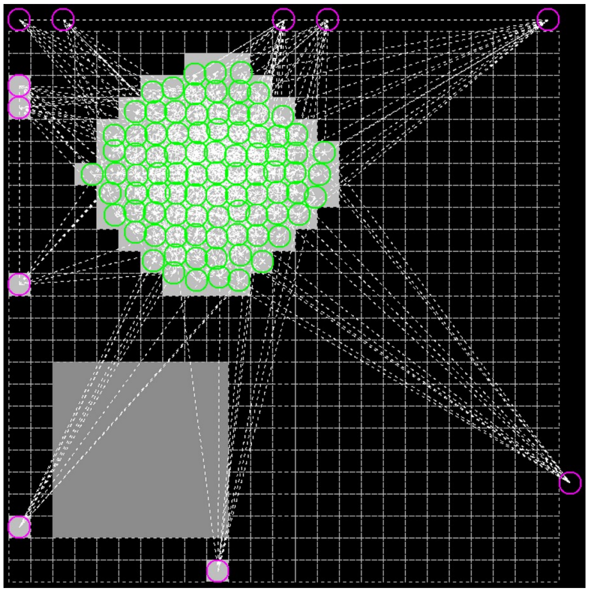
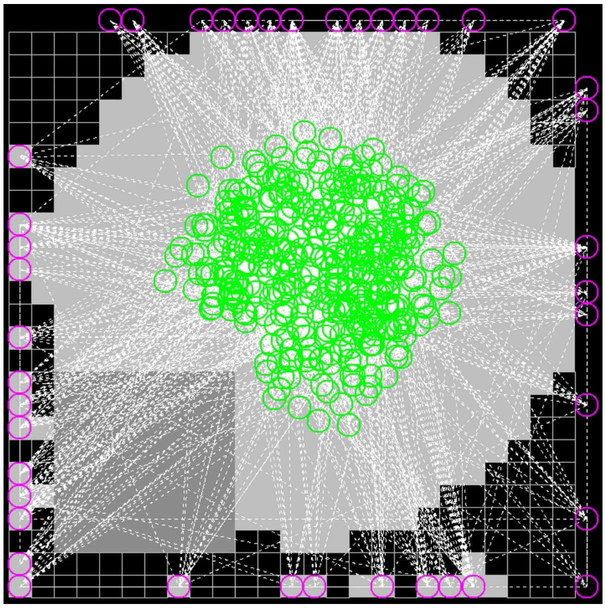
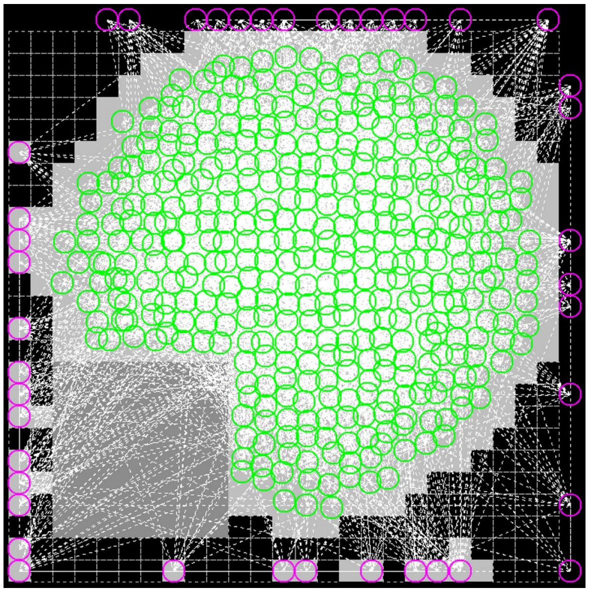
Circuit	Total Cell Displacement								
	Linear			Quadratic			Cubic		
	A=1	A=10	A=100	A=1	A=10	A=100	A=1	A=10	A=100
cct2	291	287	288	289	288	288	291	288	288
cct3	3482	3518	3512	3502	3493	3512	3509	3499	3512

It can be seen that increasing the rate of growth over 10 decades does not have a strong effect on total cell displacement.

2.3.ii Artificial anchors used for spreading adjustment

The spreading coefficient for the virtual fixed nets is specified with the -z option.

Circuit	HPWL Pre-flow Baseline	HPWL with weak spread (-z 10)	HPWL with strong spread (-z 100)
cct2	9.73	15.19	19.92
cct3	6.86	25.68	41.68

Circuit	Weak (-z 10)	Strong (-z 100)
cct2		
cct3		

When the artificial blocks are added, a bias is introduced which tends to pull each cell towards its assigned bin. With a weak anchor weight is used, this bias weight is comparable to the nets from fixed and moveable cells; there is not much movement. With a strong weight applied (z=100) this bin bias weight becomes the dominant term in a cell’s placement, and we get a very nearly regular grid of cells as a result.

An ideal placement would likely lie between these extremes – one approach to achieve the preferred result would be to slowly increase the artificial anchor bias weight until there are no overlapping bins left.